Inst*QL*: A Query Language for Virtual Institutions 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. Institutions provide a mechanism to capture and reason about "correct" and "incorrect" behaviour within a social context. While institutions can be studied in their own right, their real potential is as instruments to govern open software architectures like multi-agent and service-oriented systems. Our domain-specific action language for normative frameworks, Inst*AL* aims to help focus designers' attention on the expression of issues such as permission, violation and power but does not help the designer in verifying or querying the model they have specified. In this paper we present the query language Inst*QL* which includes a number of powerful features including temporal constraints over events and fluents that can be used in conjunction with Inst*AL* to specify those traces that are of interest in order to investigate and reason over the underlying normative models. The semantics of the query language is provided by translating Inst*QL* queries into $AnsProlog$, the same computational language as Inst*AL*. The result is a simple, high-level query and constraint language that builds on and uses the reasoning power of ASP.

1 Introduction

Institutions [21, 23, 6], also known as normative frameworks or organisations in the literature, are a specific class of multi-agent systems where agent behaviour is governed by social norms and regulations. Within institutions it is possible to monitor the permissions, empowerment and obligations of participants and to indicate violations when norms are not followed. The change of the state over time as a result of these actions provides participants with information about each others behaviour. The information can also be used by the designer to query and verify normative properties, effects and expected outcomes in an institution. The research on institutions such as electronic contracts, and rules of governance over the last decade has demonstrated that they are powerful mechanism to make agent interactions more effective, structured and efficient. As with human regulatory settings, institutions become useful when it is possible to *verify* that particular properties are satisfied for all possible scenarios.

Answer set programming [3, 14], a logic programming paradigm, permits, in contrast to related techniques like the event calculus [19] and $\mathcal{C}^+$[11], the specification of

* This work has been supported in part by the European Commission, project FP7-215890 (ALIVE).

J. Dix, M. Fisher, and P. Novák (Eds.): CLIMA X, LNAI 6214, pp. 102–121, 2010.

both problem and query as an executable program, thus eliminating the gap between specification and verification language. But perhaps more importantly, the specification language and implementation language are identical, allowing for more straightforward verification and validation.

In [6], we introduced a formal model for institutions, which admits reasoning about them by mapping to $AnsProlog$, logic programs under answer set semantics. To make the reasoning process more accessible to users, in [7] we developed an action language named Inst*AL* that allows a developer to design an institution in a more straightforward manner. Inst*AL* is then translated into $AnsProlog$, resulting in the same program as the formal description would have provided. While Inst*AL* allowed the designer to specify the institution, it provided little to no support for verifying the institution and its design—indeed, as it stands queries must be written directly in $AnsProlog$, thereby undoing most of the benefits of specifying in Inst*AL*.

In this paper, we present InstQL: a query language designed to complement Inst*AL*. Its semantics is provided by ASP and it is used together with a description of an institution either in Inst*AL* or $AnsProlog$. InstQL can be used in two ways: as a tool to select certain transitions in the state space of the institution or to model-check a certain path. For temporal queries we describe how queries expressed in the widely used temporal logic LTL may be expressed (via simple transformations) in our query language. A brief summary of the InstQL language appears in [17]. In this paper we provide an extended account of the language, illustrations of its capabilities and applications and situate it firmly in the context of multi-agent systems.

2 Answer Set Programming

In *answer set programming* ([3]) a logic program is used to describe the requirements that must be fulfilled by the solutions of a certain problem. Answer set semantics is a model-based semantics for normal logic programs. Following the notation of [3], 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 an 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 [8], SMODELS [20] and CLASP [13].

3 Institutions

In this section, we give an informal description of institutions and their mapping to ASP. A more in-depth description can be found in [6, 7].

The concept of normative systems has long been used in economics, legal theory and political science to refer to systems of regulation which enable or assist human interaction at a high-level. The same principles could be applied to multi-agent systems.

The model we use is based on the concept of *exogenous events* that describe salient events of the physical world—"shoot somebody"—and *normative events* that are generated by the normative framework—"murder"—but which only have meaning within a given social context. While exogenous events are clearly observable, normative ones are not, so how do they come into being? Searle [18] describes the creation of a normative state of affairs through *conventional generation*, whereby an event in one context *counts as* or *generates* the occurrence of another event in a second context. Taking the physical world as the first context and by defining conditions in terms of states, normative events may be created that count as the presence of states or the occurrence of events in the normative world.

Thus, we model an institution as a set of *normative states* that evolve over time subject to the occurrence of *events*, where a normative state is a set of *fluents* that may be held to be true at some instant. Furthermore, we may separate such fluents into *domain* fluents, that depend on the institution being modelled and *normative fluents* that are common to all specifications and may be classified as follows:

- **Permission:** A permission fluent captures the property that some event may occur without violation. If an event occurs, and that event is not permitted, then a *violation event* is generated.
- **Normative Power:** This represents the normative capability for an event to be brought about meaningfully, and hence change some fluents in the normative state. Without normative power, the event may not be brought about and has no effect; for example, a marriage ceremony will only bring about the married state, if the person performing the ceremony is empowered so to do.
- **Obligation:** Obligation fluents are modelled as the dual of permission. They state that a particular event must occur before a given deadline event (such as a time-out) and is associated with a specified violation. If an obligation fluent holds and the necessary event occurs then the obligation is said to be satisfied. If the corresponding deadline event occurs then the obligation is said to be violated and the specified violation event is generated. Such a violation event can then be dealt with perhaps by a participating agent or the normative framework itself.

Each event, being exogenous or normative, when generated could have an impact on the next state. For example, the event could trigger a violation or it could result in permissions being granted or retracted (e.g. once you obtain your driving licence, you obtain the permission to drive a car, but, if you are convicted of a driving offence you lose that permission). The effects of events are modelled by the consequence relation.

Thus we represent the normative framework by these five components: (i) the initial state—the set of fluents which are true when the institution is created, (ii) the set of fluents that capture the essential facts about the normative state, (iii) the set of events (both exogenous and normative) that can occur, (iv) the conventional generation relation, and (v) the consequence relation.

All state changes in a system stem from the occurrence of exactly one exogenous event. When such an event occurs, the transitive closure of the conventional generation function computes all empowered normative events that are directly or indirectly caused by the occurrence of the underlying event. This may include violations for unsatisfied obligations or unpermitted events. The consequences of each of these events with respect to the current state is computed using the consequence relationship. The combination of added and deleted fluents results in the new normative state. The semantics of this framework are defined over traces of exogenous events. Each trace induces a sequence of normative states, called a model or scenario.

In [6], it was shown that the formal model of an institution could be translated to $AnsProlog$ program such that the answer sets of the program correspond exactly to the traces of the institution. A detailed description of the mapping can be found there.

The mapping uses the following atoms: `ifluent(P)` to identify fluents, `evtype(E,T)` to describe the type of an event, `event(E)` to denote the events, `instant(I)` for time instances, `final(I)` for the last time instance in a trace, `next(I1,I2)` to establish time ordering, `occurred(E,I)` to indicate that the event happened at time I, `observed(E,I)` that the event was observed at that time, `holdsat(P,I)` to state that the institutional fluent holds at I, `initiated(P,I)` and `terminated(P,I)` for fluents that are initiated and terminated at I.

When modelling traces, we need to monitor the domain over a period of time (or a sequence of states). We model time using `instant(I)` and an ordering on instances established by `next(I1,I2)`, with the final instance defined as `final(I)`. Following convention, we assume that the truth of a fluent $F \in \mathcal{F}$ at a given state instance I is represented as `holdsat(F,I)`, while an event or an action $E \in \mathcal{E}$ is modelled as `occurred(E,I)`.

In [5] we developed Inst*AL*, an action language inspired by action languages such as $\mathcal{C}^+$ and $\mathcal{A}$ [11]. The use of the action language makes generating the $AnsProlog$ code less open to human coding errors, and perhaps more importantly, easier to understand and create by narrowing the semantic gap without losing either expressiveness or a formal basis for the language.

Institutions specifications 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. Through Inst*QL*, we aim to offer the designer the same sort of abstraction for queries as is provided by Inst*AL* for the specification.

4 The Dutch Auction: A Motivating Example

4.1 The Case Study

As a case study we will look a fragment of the Dutch auction protocol with only one round of bidding. Protocols such as this have been extensively studied in the area of

agent-mediated electronic commerce, as they are particularly suited to computer implementation and reasoning.

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. Single Bid: 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/institutional event suffixed by `dl` indicating a deadline in the protocol. The differentiation between time-out and deadline events allow a finer and more abstract control structure. While we do not want or can restrict the behaviour of an external clock (time-out) we can control the behaviour of the institution to the occurrence of the these events.

`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

When the auctioneer violates the protocol, an event `badgov` occurs and the auction dissolves.

Figure 1 gives the Inst*AL* specification of the third phase of the protocol. The excerpt shows how internal events are generated and how fluents are initiated or

```
annsold(A,B) generates sold(A,B);                                          (DAR-1)
annunsold(A,B) generates unsold(A,B);                                      (DAR-2)
annconf(A,B) generates conf(A,B);                                          (DAR-3)
biddl terminates pow(bid(B,A));                                            (DAR-4)
biddl initiates pow(sold(A,B)),pow(unsold(A,B)),
       pow(conf(A,B)), pow(notified(B)),perm(notified(B));                 (DAR-5)
biddl initiates perm(annunsold(A,B)),perm(unsold(A,B)),
       obl(unsold(A,B),desdl,badgov) if not havebid;                       (DAR-6)
biddl initiates perm(annsold(A,B)),perm(sold(A,B)),
        obl(sold(A,B), desdl, badgov) if havebid, not conflict;            (DAR-7)
biddl initiates perm(annconf(A,B)),perm(conf(A,B)),
        obl(conf(A,B), desdl, badgov) if havebid, conflict;                (DAR-8)
unsold(A,B) generates notified(B);                                         (DAR-9)
sold(A,B) generates notified(B);                                           (DAR-10)
conf(A,B) generates notified(B);                                           (DAR-11)
notified(B) terminates pow(unsold(A,B)), perm(unsold(A,B)),
 pow(sold(A,B)), pow(conf(A,B)), pow(notified(B)),
 perm(sold(A,B)), perm(conf(A,B)), perm(notified(B)),
 perm(annconf(A,B)),perm(annsold(A,B)),perm(annunsold(A,B));               (DAR-12)
desdl generates finished if not conflict;                                  (DAR-13)
desdl terminates havebid,conflict,perm(annconf(A,B));                      (DAR-14)
desdl initiates pow(price(A,B)), perm(price(A,B)),
 perm(annprice(A,B)), perm(pricedl),pow(pricedl),
 obl(price(A,B),pricedl,badgov) if conflict;                               (DAR-15)
```

Fig. 1. A partial Inst*AL* specification for the Dutch Auction Round Institution

terminates depending on the current state and the occurrence of events. Normative fluents of power, permission and obligation are represented as `pow`, `per`) and `obl` respectively. The full specification can be found on [5]. Figure 2 shows the translation of the first seven Inst*AL* specification rules of Figure 1 translated in $AnsProlog$ and grounded for one auctioneer and one bidding agent. The entire program contains about 1500 rules. Although the program can be written by hand, we believe that this process is rather tiresome and error prone.

Figure 3 shows the state transition diagram for an auctioneer and a single bidder. Every path in the graph is a valid trace.

4.2 Queries

To guide the development of our query language Inst*QL* for institutional models written in Inst*AL*, five types of existing queries which were directly encoded in $AnsProlog$ were considered.

The first case is a simple constraint involving event occurrence. An example would be a query to obtain those traces in which the auctioneer violates the protocol. 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 can occur at **any** time.

```
bad ← occurred(badgov, I), instant(I).
 ⊥ ← bad.                                                                  (Q1)
```

```
occured(sold(a,b),I) :-
    occured(annsold(a,b),I),holdsat(pow(dutch_auction_round,sold(a,b)),I),instant(I).
occured(unsold(a,b),I) :-
    occured(annunsold(a,b),I),holdsat(pow(dutch_auction_round,unsold(a,b)),I),instant(I).
occured(conf(a,b),I) :-
    occured(annconf(a,b),I),holdsat(pow(dutch_auction_round,conf(a,b)),I),instant(I).

terminated(pow(dutch_auction_round,bid(b,a)),I) :-
    occured(bidl,I),holdsat(live(dutch_auction_round),I),instant(I).

initiated(pow(dutch_auction_round,sold(a,b,b)),I) :-
    occured(bidl,I),holdsat(live(dutch_auction_round),I),instant(I).
initiated(pow(dutch_auction_round,unsold(a,b)),I) :-
    occured(bidl,I),holdsat(live(dutch_auction_round),I),instant(I).
initiated(pow(dutch_auction_round,conf(a,b)),I) :-
    occured(bidl,I),holdsat(live(dutch_auction_round),I),instant(I).
initiated(pow(dutch_auction_round,ntified(b)),I) :-
    occured(bidl,I),holdsat(live(dutch_auction_round),I),instant(I).
initiated(perm(alterted(b)),I) :-
    occured(bidl,I),holdsat(live(dutch_auction_round),I),instant(I).

initiated(perm(annunsold(a,b)),I) :-
    occured(bidl,I),not holdsat(havebid,I),holdsat(live(dutch_auction_round),I),instant(I).
initiated(perm(unsold(a,b)),I) :-
    occured(bidl,I),not holdsat(havebid,I),holdsat(live(dutch_auction_round),I),instant(I).
initiated(obl(unsold(a,b),desdl,badgov),I) :-
    occured(bidl,I),not holdsat(havebid,I),holdsat(live(dutch_auction_round),I),instant(I).

initiated(perm(annconf(a,b)),I) :-
    occured(bidl,I),holdsat(havebid,I),holdsat(conflict,I),
    holdsat(live(dutch_auction_round),I),instant(I).
initiated(perm(conf(a,b)),I) :-
    occured(bidl,I),holdsat(havebid,I),holdsat(conflict,I),
   holdsat(live(dutch_auction_round),I),instant(I).
initiated(obl(conf(a,b),desdl,badgov),I) :-
    occured(bidl,I),holdsat(havebid,I),holdsat(conflict,I),
    holdsat(live(dutch_auction_round),I),instant(I).

occured(notified(b),I) :-
    occured(unsold(a,b),I),holdsat(pow(dutch_auction_round,alterted(b)),I),instant(I).

occured(notified(b),I) :-
    occured(sold(a,b,b),I),holdsat(pow(dutch_auction_round,notified(b)),I),instant(I).

occured(notified(b),I) :-
    occured(conf(a,b),I),holdsat(pow(dutch_auction_round,alterted(b)),I),instant(I).
```

Fig. 2. The first seven DAR-Inst*AL* specification rules translated into $AnsProlog$ and grounded for one auctioneer and bidding agent

Similarly, the second query involves 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. As an example, we have a query that selects those traces in which a conflict occurs, i.e. more than one bidder submits a timely bid.

```
hadconflict ← holdsat(conflict, I), instant(I).
          ⊥ ← not hadconflict.                                   (Q2)
```

In the third case, the query condition is for an event to occur **at the same time** as a fluent holds. Again, only answer sets in which the condition is satisfied should be included. An example of such a query would be selecting those traces in which at the occurrence of the `desdl`-event we also have a conflict between two or more bidders.

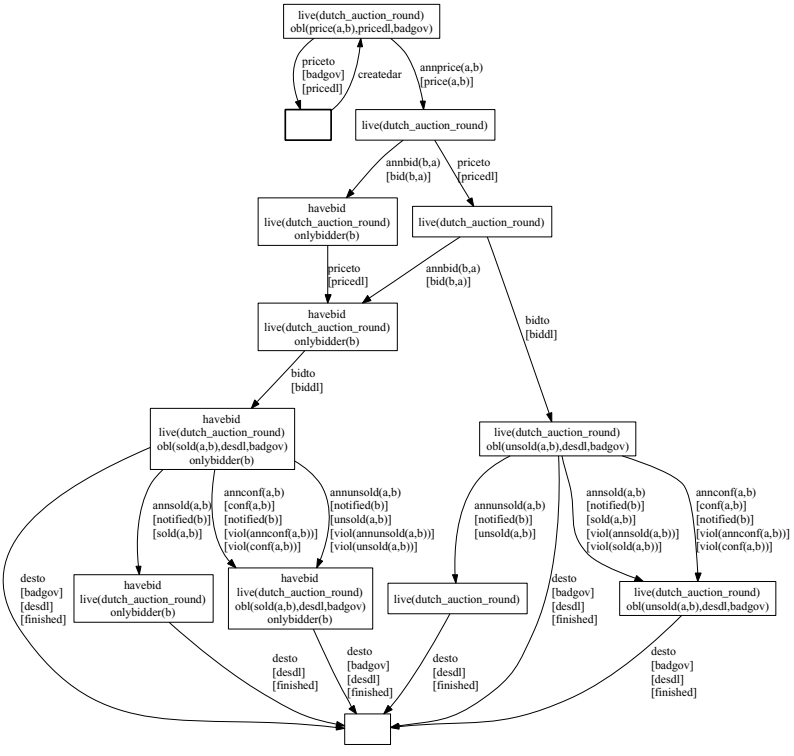

Fig. 3. States of the auction round for a single bidder

```
restarted ← occurred(desdl, I), holdsat(conflict, I),
            instant(I).                                    (Q3)
        ⊥ ← not restarted.
```

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

```
startstate(F) ← holdsat(F, I1), occurred(createdar, I0),
                next(I0, I1), ifluent(F).                  (Q4)
```

The fifth query can be used to verify the protocol. 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).) 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 this has been done, but if we still obtain a trace with this query we know something has gone wrong.

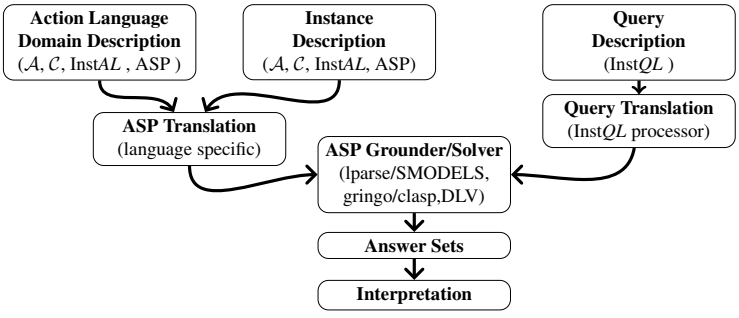

Fig. 4. The Data Flow of for Designing Institutions

```
  startstate(F) ← holdsat(F, I1), occurred(createdar, I0),
                  next(I0, I1), ifluent(F).
restartstate(F) ← holdsat(F, I1), occurred(desdl, I0),
                  holdsat(conflict, I0),
                  next(I0, I1), ifluent(F).
    missing(F) ← startstate(F), not restartstate(F), ifluent(F).
      added(F) ← restartstate(F), not startstate(F), ifluent(F).
       invalid ← missing(F), ifluent(F).
       invalid ← added(F), ifluent(F).
             ⊥ ← not invalid.
```
(Q5)

From the above, it is clear that it is possible to express these queries in $AnsProlog$, but it requires a solid knowledge of the formalism and implementation detail to get the order of events and fluents correct. InstQL was designed to remove these difficulties and allow designers to write queries in a language more closely related to natural language.

5 InstQL

In this section we introduce the query language, InstQL, that can either be used in conjunction with InstAL or directly with an $AnsProlog$ program representing the institution, whether the program is derived from the formal description or InstAL.

Figure 4 shows the flow chart of the relationships between the various components. A designer will first have to specify the institution. This can either be written directly in $AnsProlog$ or using InstAL by providing the domain description and the institutional description which are then translated into $AnsProlog$. For verification, queries on the traces are specified in InstQL and then translated into $AnsProlog$. Both programs are then merged and passed to the grounder and solver. The returned answer sets are then interpreted. A possible course of action might be that the description of the institution needs to be changed or that a new query is required.

The InstQL queries act as filters on the valid traces of the institutions. Instead of returning all traces we use the queries to return only the queries that satisfy the query, in a similar way as, for example, SQL queries.

InstQL has two basic concepts: (i) *constraint:* an assertion of a property that must be satisfied by a valid trace (for example, a restriction on which traces are considered), and (ii) *condition:* a specification of properties that may hold for a given 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.

5.1 Syntax

Inst*QL* provides two *predicates* that form the basis of all Inst*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).

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 into sub-queries, or in Inst*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*.

The building block of query conditions is the *term*:

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

The after expression also allows for the 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 to 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 specify properties of the trace that must be true:

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

For example, consider the following Inst*QL* query:

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

This indicates that only traces in which event `e` occurs should be considered.

Table 1. Inst*QL* Syntax

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_literal>	::=	<literal>|<condition_literal>
<while_expr>	::=	<while_literal> while <while_expr> | <while_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>	::=	<term> or <disjunction> | <term>
<condition_decl>	::=	condition <identifier> : <disjunction>; | condition <identifier> : <conjunction>;
<constraint>	::=	constraint <disjunction> ; | constraint <conjunction>;

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 follows:

```
condition disj: happens(e2) or happens(e3);
condition conj: happens(e1) and disj;
```

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, Inst*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`) or condition literals involving condition names. A while expression is defined as follows:

```
<while_literal} ::= <literal> | <condition_literal>
<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 left on:

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

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;
```

5.2 Semantics

The semantics of an InstQL query is defined by the translation function T which translates InstQL into $AnsProlog$. This function takes a fragment of InstQL 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:

$$T(\texttt{happens(e)}) = \texttt{occurred(e, I)}, \texttt{event(e)}$$
$$T(\texttt{holds(f)}) = \texttt{holdsat(f, I)}, \texttt{ifluent(f)}$$

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

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

while for a condition literal they are:

$$T(\texttt{conditionName}) = \texttt{conditionName(I)}$$
$$T(\texttt{not conditionName}) = \texttt{not conditionName(I)}$$

and a conjunction of terms is:

$$T(c_1 \texttt{ and } c_2 \texttt{ and} \cdots \texttt{ and } c_n) = T(c_1), T(c_2), \ldots, T(c_n)$$

A disjunction translates to more than one rule. However, this is defined slightly differently depending on whether it is part of a condition declaration or a constraint.

$$\begin{array}{ll} T(\texttt{condition conditionName} : & c_1 \texttt{ or } c_2 \texttt{ or } \cdots \texttt{or } c_n;) = \\ & \{\texttt{conditionName} \leftarrow T(c_i).\ \mid\ 1 \leq i \leq n\} \\ T(\texttt{constraint} & c_1 \texttt{ or } c_2 \texttt{ or } \cdots \texttt{or } c_n;) = \\ & \{\texttt{newName} \leftarrow T(c_i).\ \mid\ 1 \leq i \leq n\} \cup \\ & \{\bot \leftarrow \texttt{not newName}.\} \end{array}$$

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. This atom becomes true if one of the sub-queries in the disjunction becomes true. In order to satisfy the entire query at least one the sub-queries not to be true, as expressed by the constraint. 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 Inst*QL* query. Recall that a condition name may be parameterised: since an Inst*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).`".

Notice that so far only the translation of constraint and condition provide a specification for time (`instance`). Because of the grammar of our language, the translation of other terms results in a set of literals which will appear in a rule that already include this atom.

The semantics for `while` is:

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

We give the semantics for the binary operator `after(n)`. This readily generalises for sequences of `after(n)` operators mixed with `after` operators.

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

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 $\texttt{n} > 0$.

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

$$\begin{array}{l} T\ (\texttt{happens(e) while holds(f) after happens(d) after(3) holds(g)}) = \\ \texttt{occurred(e, } \texttt{t}_\texttt{i}\texttt{), event(e), holdsat(f, } \texttt{t}_\texttt{i}\texttt{), ifluent(f),} \\ \texttt{instant(}\texttt{t}_\texttt{i}\texttt{), occurred(d, } \texttt{t}_\texttt{j}\texttt{), event(d), instant(}\texttt{t}_\texttt{j}\texttt{),} \\ \texttt{holdsat(g, } \texttt{t}_\texttt{k}\texttt{), ifluent(g), instant(}\texttt{t}_\texttt{k}\texttt{),} \\ \texttt{after(}\texttt{t}_\texttt{i}\texttt{, } \texttt{t}_\texttt{j}\texttt{), after(}\texttt{t}_\texttt{j}\texttt{, } \texttt{t}_\texttt{k}\texttt{, 3).} \end{array}$$

5.3 The Dutch Auction Queries

Having defined the query language Inst*QL*, we return to the example queries for the Dutch auction from Section 4.

For (Q1) the following InstQL query is equivalent:

```
condition bad: happens(badgov);
constraint not 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;
```

An InstQL query that is equivalent to (Q2) is:

```
constraint holds(conflict);
```

The following query is equivalent to (Q3):

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

For (Q4), the following InstQL query is equivalent:

```
condition startstate(F): holds(F) after(1) happens(createdar);
```

For (Q5) the following InstQL 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);
condition added(F): restartstate(F) and not startstate(F);
constraint missing(F) or added(F);
```

While queries 1-5 demonstrate the capabilities of our query language they might not be the only queries a designer of the Dutch Auction would pose.

The following query verifies it is never the case that an agent has permission to perform an action while not having the power. A correct protocol will return no traces.

```
condition permission(F): holds(perm(F)) while not holds(pow(F));
constraint permission(F);
```

The following query returns traces containing violations that have not been detected.

```
condition violation(F): not happens(badgov) after happens(viol(F));
constraint violation(F);
```

As a designer you also want to verify the order of the important events that need to take place. The following query verifies that the two deadlines occur in the correct order and that the corresponding obligations are fulfilled.

```
condition order: pricedl after desdl;
condition obl: not holds(obligation(E,D,V)) while
               not happens(V) after holds(obligation(E,D,V));
constraint: order and not happens(badgov) and obl;
```

6 Reasoning

6.1 Common Reasoning Tasks

Following the description of InstQL in the preceding section, we now illustrate how it can be used to perform three common tasks [24] in computational reasoning: prediction, postdiction and planning.

Prediction is the problem of ascertaining the resulting state for a given (partial) sequence of events/actions and initial state. That is, suppose some transition system is in state $s \in \mathcal{S}$ with $\mathcal{S}$ the set of all possible states of the system and a sequence $A = a_1, \ldots, a_n$ of actions/events occurs. Then the prediction problem (s, A) is to decide the set of states $\{S' \subseteq \mathcal{S}\}$ 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 $\{S \subseteq \mathcal{S}\}$ of states that could have held before

A. The planning problem (s, s') is to decide which sequence(s) of actions, $\{A' \subseteq \mathcal{A}\}$, with $\mathcal{A}$ all possible sequences of actions/events, 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 Inst*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 $f_{1\ldots k}$ are fluents which must hold in the matched state and $g_{1\ldots k}$ 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 Inst*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 $\{S' \subseteq \mathcal{S}\}$.

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 Inst*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;
```

Reasoning with institutions: There are two distinct types of reasoning about institutions. The first is the verification and exploration of normative properties. After specifying an institutions, queries can be used to determine that desired properties of the model are present or to elicit emergent properties that were perhaps not intended. The second kind is for the participants/agents within that institution to use the available information in their decision processes. The participants could, using the current state and the specification apply prediction to determine previous actions of other participants, postdiction to evaluate possible effects of their actions or planning to determine the actions necessary to achieve certain goals. Using $AnsProlog$ as the underlying formalism, designers and institutional participants can use partial information to reason about the institution itself of other participants.

6.2 Modelling Linear Temporal Logic

LTL [22] is a commonly used temporal logic used for model checking transitions systems. In this section we show that LTL style reasoning can also modelled using our Inst*QL*. We opted for LTL since it shares the same linear time structure as our model and also allows complex expressions of temporal properties between states. Traditional LTL syntax is often considered difficult to write and we believe that Inst*QL* would be a valuable alternative, especially if one wants to reason about events and fluents at the same time.

Linear Temporal Logic: (LTL) [22] 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 [10] is defined as follows: (i) $p \in AP$ is a formula of LTL (ii) $\neg f$ is a formula if f is a formula (iii) $f \vee g$ is a formula if f and g are formulae (iv) $f \wedge g$ is a formula if f and g are formulae (v) $\Diamond f$ is a formula if f is a formula ("sometimes f") (vi) fUg is a formula if f and g are formulae ("f until g"). We abbreviate $\neg\Diamond\neg f$ by $\Box f$ ("always f").

The semantics of LTL is given with respect to a structure $M = (\mathbf{S}, \mathbf{X}, \mathbf{L})$ and a path of state transitions. M contains 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. In addition, we use $first(x)$ to denote the first state in path x.

The semantics of LTL is defined inductively in terms of interpretations (paths) over a linear structure (time) by the relation $\models$ [10, 9, 25, 16, 4]. Without loss of generality we use the natural numbers $\mathcal{N}$ as our structure. An interpretation is a function $\pi : \mathcal{N} \rightarrow \mathbb{P}(AP)$, which assigns a truth value to each element of AP at every instant $i \in \mathcal{N}$.

Let M be a structure and $x \in \mathbf{X}$, then:

$$
\begin{array}{ll}
\pi, i \models p \in AP & \iff p \in \pi(i) \\
\pi, i \models \neg f & \iff p, i \not\models f \\
\pi, i \models f \vee g & \iff \pi, i \models f \text{ or } \pi, i \models g \\
\pi, i \models f \wedge g & \iff \pi, i \models f \text{ and } \pi, i \models g \\
\pi, i \models \Diamond f & \iff \exists j \geq i \cdot \pi, j \models f \\
\pi, i \models fUg & \iff \exists j \geq i \cdot \pi, j \models g \wedge (\forall i \leq k < j \cdot \pi, k \models f)
\end{array}
$$

Where the structure is understood, we will omit it from the relation and write $x \models f$.

In principle LTL (originally) only refers to states, and as a general observation, the merging of actions and fluents inside LTL is non-trivial as you are merging state-relative and transition-relative concepts. With institutions we want to reason about both fluents and events, so $AP = \mathcal{E} \cup \mathcal{F}$.

Expressing LTL in Inst*QL*: There is an important difference between LTL and Inst*QL* in the sense that Inst*QL* is not designed for model checking but for model generation. Given a query, it will generate those paths that satisfy the criteria. If π is the path given to LTL for verification, Inst*QL* will return all traces that satisfy the query which may or may not include the path given for verification. To solve this problem one can provide the path itself as a constraint to the Inst*QL* query. This can be easily done using a combination of `while` and `after` in the same way as be defined event ordering above. This will restrict the search space to those traces in which the path is satisfied. If the path itself is invalid (e.g. two observed events during the same time, fluents that are in a state while they should not be), then the query will automatically not be satisfied.

The LTL query itself can then be expressed in Inst*QL*. We will briefly describe how the various formulae may be expressed as conditions in Inst*QL*. Each sub-formula S of the formula F that is to be checked is translated as a condition with a unique name `cond-S`. To make a formula F effective (i.e. only compute traces for which F is true) we add a constraint to the query that specifies the condition for F must hold: "`constraint cond-F;`". Atomic elements a of AP and their negation simply become conditions with `happens(a)` or `holds(a)` or their negation depending on the type of a. Consequently, LTL disjunction can be handled be handled as a disjunction in Inst*QL*. Conjunction is LTL is like our Inst*QL* while as all sub-formulas need to be evaluated over the same time instant.

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

```
condition diamond-F: cond-F;
```

Although it might seem similar to the encoding of atomic elements, this encoding guarantees a possible different time instance.

Defining until (FUG) is more subtle. Naïvely, we could define "F until G" as:

```
condition false_before(cond-F,cond-G): cond-F after not cond-G;
condition cond-FUG: & not false_before(cond-F, cond-G);
```

However, translating this into $AnsProlog$ we see that the condition is too strong. To make the example easier assume that F is a fluent and G an event and that we skip the encoding for the sub-formula:

$$\texttt{false\_before(F,E)} \leftarrow \texttt{occurred(E, I), event(E), instant(I),}$$
$$\texttt{not holdsat(F, J), ifluent(F), instant(J), after(I, J).}$$
$$\texttt{until(F, E)} \leftarrow \texttt{not false\_before(F, E).}$$

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 is that f cannot be false before the *first* occurrence of e [16].

In order to achieve the correct semantics, we need to introduce new fluents `happened(e)` to the domain for each event $e \in \mathcal{E}$ to indicate that e occurred for

the first time. This is done automatically when we translate Inst*QL* to $AnsProlog$ to indicate when an event has happened at any time in the past during the current trace.

```
holdsat(happened(E), I) ← occurred(E, I), event(E), instant(I).
holdsat(happened(E), I) ← occurred(E, J), after(I, J),
                          event(E), instant(I), instant(J).
```

To allow for this we need for each event E that is part of the query and the until statement the condition `condition con-E: holds(happened(E));`.

This allows us to then specify FUG as follows:

```
condition fb(cond-F, cond-G): not cond-F while not cond_G;
    condition cond-FUG: not fb(cond-F, cond-G) and cond-E
                                        and cond-F;
```

6.3 Institutional Designer and Reasoning Tools: Inst*Suite*

Both Inst*QL* and Inst*AL* were designed and implemented to make representing and reasoning about institutions more intuitive and effective. While they were designed to work together they can be used independently from each other. Inst*AL* and Inst*QL* specifications can be written in any text processor and then translated into an answer set program and passed on to an answer set solver that computes the requested traces and models. To provide normative designer more support, we have developed an integrated development environment Inst*Edit* with syntax highlighting. Together they are referred to as Inst*Suite* , which source code, a combination of Java and perl, can be obtained from `http://agents.cs.bath.ac.uk/InstSuite/`

7 Discussion

Previous work in [2, 1] (using the action language $\mathcal{C}^+$ [11]), has shown that action languages are particularly suited to modelling normative domains, where actions in the language are equated with institutional events. In [7] we extend this approach with the language Inst*AL* which incorporates normative properties directly into the syntax of the language and operates by translating institutional specifications into $AnsProlog$. In this case we are able to directly leverage the reasoning capabilities inherent in the underlying logic programming platform to query properties of models. By building Inst*QL* upon this model we are able to offer an equivalent level of abstraction to Inst*AL* while at the same time remaining independent of the action language itself Inst*AL*.

Inst*QL* was designed for institutions, but it can be used a general query language for action domains, provided their descriptions can be mapped to $AnsProlog$. Compared to existing query languages for action domains, Inst*QL* allows for simultaneous actions and the definition of conditions which can then be used to create more complex queries.

In [15], the authors present four query languages: $\mathcal{P}, \mathcal{Q}, \mathcal{Q}_n, \mathcal{R}$. Queries expressed in those languages can also be expressed using Inst*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 Inst*QL* using the techniques discussed in Section 6. `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 $\mathcal{Q}$, $\mathcal{Q}_n$ and $\mathcal{R}$. Given the action ordering technique used, we can assign specific times to each of the fluents. Inst*QL* can express all the same kinds of queries as the query languages above, but in addition Inst*QL* is capable of modelling simultaneous actions and fluents, which permits the expression of complex queries using disjunctions and conjunctions of conditions and, above all, allows reasoning with incomplete information, thus fully exploiting the reasoning power of answer set programming.

The Causal Calculator (CCALC) [12] is a versatile tool for modelling action domains. While queries are possible in CCALC, Inst*QL* has been designed specifically as a query language, providing constructs to make specifying queries more natural. Relative ordering of actions or states is much more difficult in CCALC than it is Inst*QL* , nor does CCALC allow for the formulation of composite queries (condition literals).

As it stands Inst*QL* is an intuitive and versatile query and abduction language for action domains. The language is succinct and without redundancy (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)` predicates and the possibility to construct general logical expression without the need for condition statements.

At the moment Inst*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, alternating time is more suited to open domains.

In [7] we introduced the concept of multi-institutions; groups of institutions that can influence each others' state. In the near future we want to extend Inst*QL* to multi-institution specifications.

References

[1] Artikis, A., Sergot, M., Pitt, J.: Specifying electronic societies with the Causal Calculator. In: Giunchiglia, F., Odell, J.J., Weiss, G. (eds.) AOSE 2002. LNCS, vol. 2585, pp. 1–15. Springer, Heidelberg (2003)

[2] Artikis, A., Sergot, M., Pitt, J.: Specifying norm-governed computational societies. ACM Trans. Comput. Logic 10(1), 1–42 (2009)

[3] Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge Press, Cambridge (2003)

[4] Calvanese, D., Vardi, M.Y.: Reasoning about actions and planning in LTL action theories. In: Proc. KR 2002 (2002)

[5] Cliffe, O.: Specifying and Analysing Institutions in Multi-Agent Systems using Answer Set Programming. PhD thesis, University of Bath (2007)

[6] Cliffe, O., De Vos, M., Padget, J.: Answer set programming for representing and reasoning about virtual institutions. In: Inoue, K., Satoh, K., Toni, F. (eds.) CLIMA 2006. LNCS (LNAI), vol. 4371, pp. 60–79. Springer, Heidelberg (2007)
[7] Cliffe, O., De Vos, M., Padget, J.: Specifying and reasoning about multiple institutions. In: Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., Matson, E. (eds.) COIN 2006. LNCS (LNAI), vol. 4386, pp. 63–81. Springer, Heidelberg (2007)
[8] Eiter, T., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The KR system dlv: Progress report, comparisons and benchmarks. In: Cohn, A.G., Schubert, L., Shapiro, S.C. (eds.) KR 1998: Principles of Knowledge Representation and Reasoning, pp. 406–417. Morgan Kaufmann, San Francisco (1998)
[9] Allen Emerson, E.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 995–1072. Elsevier, Amsterdam (1990)
[10] Emerson, E.A., Halpern, J.Y.: "sometimes" and "not never" revisited: on branching versus linear time temporal logic. Journal of the ACM 33(1), 151–178 (1986)
[11] Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence 153, 49–104 (2004)
[12] Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence 153, 49–104 (2004)
[13] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-Driven Answer Set Solving. In: Proceeding of IJCAI 2007, pp. 386–392 (2007)
[14] Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3-4), 365–386 (1991)
[15] Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. Artif. Intell. 2, 193–210 (1998)
[16] Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 200–212. Springer, Heidelberg (2001)
[17] Hopton, L., Cliffe, O., De Vos, M., Padget, J. A.: Aql: A query language for action domains modelled using answer set programming. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 437–443. Springer, Heidelberg (2009)
[18] Searle, J.R.: The Construction of Social Reality. The Penguin Press, Allen Lane (1995)
[19] Kowalski, R.A., Sadri, F.: Reconciling the event calculus with the situation calculus. Journal of Logic Programming 31(1-3), 39–58 (1997)
[20] Niemelä, I., Simons, P.: Smodels: An implementation of the stable model and well-founded semantics for normal LP. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS (LNAI), vol. 1265, pp. 420–429. Springer, Heidelberg (1997)
[21] Noriega, P.: Agent mediated auctions: The Fishmarket Metaphor. PhD thesis, Universitat Autonoma de Barcelona (1997)
[22] Pnueli, A.: The Temporal Logic of Programs. In: 19th Annual Symp. on Foundations of Computer Science (1977)
[23] Rodríguez, J.-A., Noriega, P., Sierra, C., Padget, J.: FM 96.5 A Java-based Electronic Auction House. In: Proceedings of 2nd Conference on Practical Applications of Intelligent Agents and MultiAgent Technology (PAAM 1997), pp. 207–224 (1997) ISBN 0-9525554-6-8
[24] Sergot, M.: $\mathcal{C}+^{++}$: An action language for modelling norms and institutions. Technical Report 8, Department of Computing, Imperial College, London (2004)
[25] Sistla, A.P., Clarke, E.M.: The complexity of propostional linear temporal logics. Journal of the ACM 32(3), 733–749 (1985)