Representing the Language of the Causal Calculator in Answer Set Programming

by

Michael Casolary

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2011 by the
Graduate Supervisory Committee:

Joohyung Lee, Chair
Gail-Joon Ahn
Chitta Baral

ARIZONA STATE UNIVERSITY

August 2011

ABSTRACT

Action language C+ is a formalism for describing properties of actions, which is based on nonmonotonic causal logic. The definite fragment of C+ is implemented in the Causal Calculator (CCalc), which is based on the reduction of nonmonotonic causal logic to propositional logic. This thesis describes the language of CCalc in terms of answer set programming (ASP), based on the translation of nonmonotonic causal logic to formulas under the stable model semantics. I designed a standard library which describes the constructs of the input language of CCalc in terms of ASP, allowing a simple modular method to represent CCalc input programs in the language of ASP. Using the combination of system F2LP and answer set solvers, this method achieves functionality close to that of CCalc while taking advantage of answer set solvers to yield efficient computation that is orders of magnitude faster than CCalc for many benchmark examples. In support of this, I created an automated translation system Cplus2ASP that implements the translation and encoding method and automatically invokes the necessary software to solve the translated input programs.

To my family and friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The field of knowledge representation is a specialization of artificial intelligence that focuses on designing, implementing, and utilizing logical formalisms that encode facts and relationships in a way that allows computers to store them and reason about them in a flexible and efficient manner. In pursuit of this goal, many formalisms have been created, each aiming to capture certain domains or solve specific problems related to creating intuitive logics. Action-oriented formalisms focus on the concept of reasoning about actions and their effects on the state of a given world. Common issues action-oriented formalisms address are the frame problem (McCarthy & Hayes, 1969), which is how to encode the idea that objects have persistent states, and the ramification problem (Finger, 1986), which addresses capturing indirect effects of actions.

In particular, nonmonotonic causal logic (Giunchiglia, Lee, Lifschitz, McCain, & Turner, 2004), as the name implies, is a formalism designed around representing causal relationships. $\mathcal{C}+$ (Giunchiglia et al., 2004) is a high-level notation of nonmonotonic causal logic that is designed to describe actions and their effects. The Causal Calculator (CCALC[1]), an implementation of $\mathcal{C}+$, reduces a fragment of nonmonotonic causal logic to propositional logic and uses satisfiability (SAT) solvers to produce solutions for given action descriptions. CCALC was originally created as a prototype, but it has been successfully applied to several challenging commonsense reasoning problems; it has been used on problems of nontrivial size (Akman, Erdoğan, Lee, Lifschitz, & Turner, 2004),

[1]`http://www.cs.utexas.edu/users/tag/cc/`

to provide a group of robots with high-level reasoning (Caldiran et al., 2009), to give executable specifications of norm-governed computational societies (Artikis, Sergot, & Pitt, 2009; Chopra & Singh, 2003), and to automate the analysis of business processes under authorization constraints (Armando, Giunchiglia, & Ponta, 2009).

It has been shown by McCain (1997), Ferraris (2007), and Ferraris et al. (2010) that nonmonotonic causal logic can be turned into logic programs under the answer set semantics (Gelfond & Lifschitz, 1988). This provides a way to compute nonmonotonic causal logic using answer set solvers and is an alternative to the computing method that is implemented in CCALC. As there are currently many efficient answer set solvers available, this is a promising approach.

In a related work, Kim, Lee, and Palla (2009) showed that the event calculus (Shanahan, 1995) can be reformulated in answer set programming. That work enabled Kim to create an implementation of the event calculus (Kim, 2009) that utilized this embedding to translate event calculus into the language of answer set programming (ASP), taking advantage of ASP solvers to significantly improve the efficiency of solving event calculus descriptions.

Taking inspiration from the work on event calculus, in this thesis we examine the effectiveness of using a similar approach with causal logic. Our aim is to create a system that is a step toward combining the expressivity of $\mathcal{C}+$ with the speed of modern ASP solvers. We create an enhanced translation and encoding method designed to translate action descriptions in the input language of CCALC into the input language of ASP solvers. We then implement CPLUS2ASP, a software tool that performs this translation in an automated fashion. Figure 1.1 shows the design of CPLUS2ASP. Our experiments show

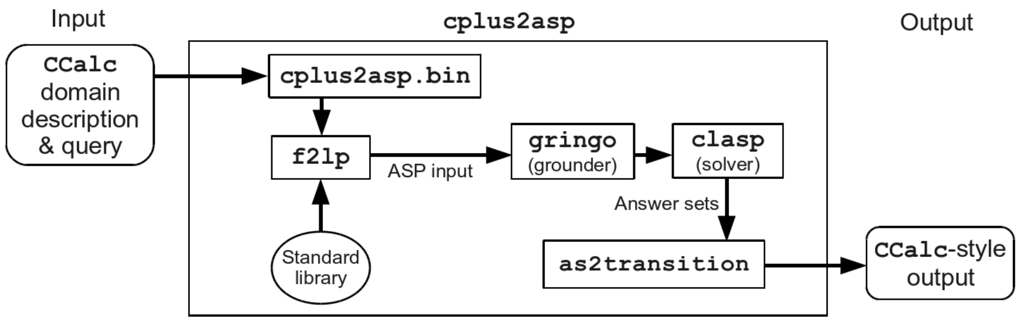


Figure 1.1. Design of the Cplus2ASP System

that CPLUS2ASP achieves functionality close to CCALC while taking advantage of ASP solvers to yield efficient computation that is orders of magnitude faster than CCALC on several benchmark examples. This work paves the way to implement a new version of CCALC that will incorporate recent advances in answer set programming, such as incremental grounding, online reasoning, modular programming, constraint solving, and aggregates.

In Chapter 2, we introduce background material. Chapter 3 covers preliminary content used as a basis for this work. The actual method for translating and encoding CCALC input into ASP is documented in Chapter 4. Information about the software system CPLUS2ASP and its components is presented in Chapter 5, and the results of testing the software against various benchmark domains can be found in Chapter 6. Chapter 7 addresses and compares related works, and in Chapter 8, we comment on our contribution as a whole and discuss possible future enhancements to CPLUS2ASP.

CHAPTER 2

BACKGROUND

2.1 Nonmonotonic Causal Theories and C+

Action language $\mathcal{C}+$ (Giunchiglia et al., 2004) is a high-level notation of nonmonotonic causal logic designed to describe transition systems. A *transition system* is a directed graph whose vertices represent possible states of a world and whose edges, labeled with actions, represent transitions from one state to another. To aid in these representations, $\mathcal{C}+$ takes common syntactic and semantic concepts, such as inertia and conditional causation, and represents them as English-like "laws" that are generally easier to parse and understand than their logic formula equivalents. $\mathcal{C}+$ is an extension of $\mathcal{C}$ (McCain, 1997) that overcomes several essential limitations of its predecessor by including features such as multi-valued constants, defined fluents, additive fluents, rigid constants, attributes, and defeasible causal laws. $\mathcal{C}+$ was further extended to express distant causation (Craven & Sergot, 2005), preferred states and actions (Sergot & Craven, 2006), and to allow for probabilistic reasoning (Eiter & Lukasiewicz, 2003).

2.2 The Causal Calculator

The Causal Calculator was originally created by McCain (1997) as an implementation of action language $\mathcal{C}$. To support $\mathcal{C}+$, Lee extended CCALC (2005) by adding several features, including those mentioned above. While Lee's extended CCALC was originally intended to be a proof-of-concept software tool demonstrating that an implementation of $\mathcal{C}+$ was feasible, CCALC benefitted from the expressivity of $\mathcal{C}+$ such that it became desirable for use in a variety of problem-solving applications. Several representative examples of such applications were mentioned in the introduction.

2.3 Answer Set Programming

Answer set programming is a declarative programming paradigm that is oriented towards difficult NP-hard search problems. Its goal is to reduce the given search problem to computing stable models, using an ASP solver to perform the search. There are many answer set solvers available, such as GRINGO and CLASP[1], SMODELS[2], CMODELS[3], and DLV[4]. In addition, a biannual ASP competition is held to encourage the development and further improvement of ASP solvers. Thanks in part to the versatility of answer set solvers, ASP has been applied to a wide range of problems, from code optimization (Brain, Crick, Vos, & Fitch, 2006) and model checking (Liu, Ramakrishnan, & Smolka, 1998) to music composition (Boenn, Brain, Vos, & Fitch, 2008) and multi-agent planning (Son, Pontelli, & Sakama, 2009).

2.4 F2LP

The input to ASP solvers is limited to rule forms, which are analogous to clausal normal form in classical logic. System F2LP ("Formulas To Logic Programs") (Lee & Palla, 2009) is a front end that allows ASP solvers to compute stable models of first-order formulas, as defined by Ferraris (2007) and Ferraris et al.

[1]`http://potassco.sourceforge.net/`
[2]`http://www.tcs.hut.fi/Software/smodels/`
[3]`http://www.cs.utexas.edu/users/tag/cmodels.html`
[4]`http://www.dlvsystem.com/dlvsystem/index.php/Home`

(2011). Allowing first-order formulas as input makes it easier to express concepts like nested connectives and quantifiers. For example, the formula

```
terminal(X) <- vertex(X) & not ?[Y]:edge(X,Y)
```

succinctly describes the conditions under which `x` is considered to be a terminal vertex of a directed graph by using existential quantification over `Y` (denoted by `?[Y]`). F2LP can also be used to compute event calculus (Shanahan, 1995) and situation calculus (McCarthy & Hayes, 1969; Reiter, 2001) by using ASP solvers (Kim et al., 2009; Lee & Palla, 2010).

CHAPTER 3

PRELIMINARIES

The following sections document prior research and experiments that we use as a basis for our enhancements and contribution.

3.1 Nonmonotonic Causal Logic

Our work focuses on the version of causal logic supported by CCALC (multi-valued definite propositional causal logic), so what follows is the definition of nonmonotonic causal logic given in the paper "Nonmonotonic Causal Theories" (Giunchiglia et al., 2004).

In causal logic, a *multi-valued propositional signature* consists of:

- A set σ of symbols called *constants*, and
- for each constant c in σ, a nonempty, finite set $\mathit{Dom}(c)$ consisting of at least two elements that comprise the *domain* of c.

An *atom* of σ is an expression $c\!=\!v$, which signifies that the value of the constant c is equal to v.

The expression $c = d$ is a commonly seen shortcut representing that the value of c equals the value of d for some value from the intersection of their domains. Stated formally:

$$c\!=\!d \equiv \bigvee_{v \in \mathit{Dom}(c) \cap \mathit{Dom}(d)} c\!=\!v \wedge d\!=\!v.$$

Another syntactic shortcut that works for constants with Boolean domains is to use c in a causal rule to stand for $c=$**true** and $\neg c$ to stand for $c=$**false**.

An *interpretation* I of σ is a function that maps each constant in σ to a value in its domain.

Causal logic is defined in terms of *causal rules* of the form

$$F \Leftarrow G \tag{3.1}$$

where this is understood as, "There is a cause for F if G is true." (This is in contrast to the classical version

$$G \rightarrow F$$

of (3.1), which would be read, "If G is true then so is F.") In the general case, both F and G may be arbitrary *formulas*, which are propositional combinations of elements from σ, $\top$ (universal truth), and $\bot$ (universal falsehood). If F is $\bot$ in a causal rule, we call that rule a *constraint*.

A *causal theory* T of σ is a finite set of causal rules. If all of the heads of the rules in a causal theory are either $\bot$ or a single literal, then we call that causal theory *definite*.

An interpretation I satisfies an atom $c=v$ (represented as $I \models c=v$) if $I(c)=v$. As expected, every interpretation I satisfies $\top$ ($I \models \top$ for every I), and no I satisfies $\bot$ ($I \not\models \bot$ for every I). Satisfaction is extended to arbitrary formulas using classical truth tables for the standard propositional connectives.

The semantics of causal logic is defined in terms of a fixpoint definition. For a causal theory T and an interpretation I, the *reduct* T^I of T under I is the set of heads of the causal rules of T whose bodies are satisfied by I. I is a *model* of T if I is the unique interpretation of σ that satisfies T^I. A causal theory T is called *satisfiable* if it has at least one model and *unsatisfiable* (or *inconsistent*) if it has no models.

For example, take causal theory T to be

$$\begin{aligned} p &\Leftarrow q, \\ q &\Leftarrow q, \\ \neg q &\Leftarrow \neg q. \end{aligned} \tag{3.2}$$

If we let interpretation I_1 be the set $\{p, \neg q\}$ (i.e., $I_1(p) = \textbf{true}$ and $I_1(q) = \textbf{false}$), this would not be a model of T. I_1 only satisfies the body of the third rule in the theory, causing the reduct T^{I_1} to be $\{\neg q\}$. Interpretation $\{p, \neg q\}$ does not uniquely satisfy the reduct (the set $\{\neg p, \neg q\}$ would as well); therefore, I_1 is not a model of T. However, if we let interpretation I_2 be the set $\{p, q\}$ (i.e., $I_2(p) = \textbf{true}$ and $I_2(q) = \textbf{true}$), then the first two bodies of T would be satisfied by I_2, making the reduct $T^{I_2} = \{p, q\}$. Therefore, I_2 uniquely satisfies T^{I_2}, and thus is a model of T. It turns out that this is the only model of the theory.

3.2 Action Language C+

Action language $\mathcal{C}+$ is a high-level notation for causal logic that is designed to describe transition systems in a succinct way. In $\mathcal{C}+$, constants are divided into two groups: *fluent* constants and *action* constants. Fluent constants are further partitioned into *simple* and *statically determined* fluents. Using these new categories, $\mathcal{C}+$ also distinguishes between various types of formulas. A *fluent formula* is a formula where any constants occurring in it are fluent constants. An *action formula* is a formula that contains at least one action constant and no fluent constants.

Where causal logic utilized causal rules as its fundamental logic sentences, $\mathcal{C}+$ uses higher level *causal laws*. There are three basic kinds of causal laws:

- *Static laws* are expressions of the form

  $$\textbf{caused } F \textbf{ if } G \tag{3.3}$$

  where F and G are fluent formulas.

- *Action dynamic laws* are expressions with the same form as (3.3), except F is an action formula and G is any kind of formula.

- *Fluent dynamic laws* are expressions of the form

  $$\textbf{caused } F \textbf{ if } G \textbf{ after } H \tag{3.4}$$

  where F and G are fluent formulas and H is any kind of formula. In this case, F cannot contain statically determined constants.

Static laws are typically used to express relationships and dependencies between fluents in the same state. Action dynamic laws perform a similar function, but they are specific to dependencies between actions. The bulk of $\mathcal{C}+$ laws found in the average action description are fluent dynamic laws, as they are the ones that define how a given state changes over time as a result of the effects of actions or other conditions.

In addition to the basic causal laws shown in (3.3) and (3.4), several "shortcut" causal laws were defined in terms of these basic laws in Giunchiglia et al. (2004, Appendix B) to make it easier to express common concepts of transition systems.

The $\mathcal{C}+$ analogue of a causal theory from causal logic is an *action description*, which is a set of causal laws.

The semantics of $\mathcal{C}+$ in Giunchiglia et al. (2004) is defined in terms of a translation into causal logic. In order to perform the translation, the earlier definition of causal logic must be extended to include the concept of time via the inclusion of ordered states. The definition of a constant is extended to include a nonnegative integer time stamp: $i\!:\!c$. This represents the constant c at time i. The domain of $i\!:\!c$ remains the same as the domain of c. Using this new definition, an atom of the form $i\!:\!c\!=\!v$ signifies, "The constant c has the value v at time i."

For any action description D and any nonnegative integer m, the causal theory D_m is defined in the following manner. The signature of D_m consists of extended constants $i\!:\!c$ such that

- $i \in \{0, \ldots, m\}$ if c is a fluent constant of D, or
- $i \in \{0, \ldots, m-1\}$ if c is an action constant of D.

We use the expression $i : F$ to denote the result of inserting $i :$ in front of every occurrence of every constant in a formula F, and similarly for a set of formulas. For every static causal law (3.3) in D, add the causal rule

$$i\!:\!F \Leftarrow i\!:\!G \tag{3.5}$$

to D_m, where $i \in \{0, \ldots, m\}$. Do the same thing for every action dynamic law in D, using $i \in \{0, \ldots, m-1\}$.

Transform each fluent dynamic law (3.4) in D into

$$i{+}1 : F \Leftarrow (i{+}1 : G) \land (i : H) \tag{3.6}$$

in D_m, using $i \in \{0, \ldots, m-1\}$. Simple fluent constants have a property that their values are initially *exogenous* (i.e., by default they can take on any value from their domain unless constrained otherwise). This is represented in causal logic by adding the rule

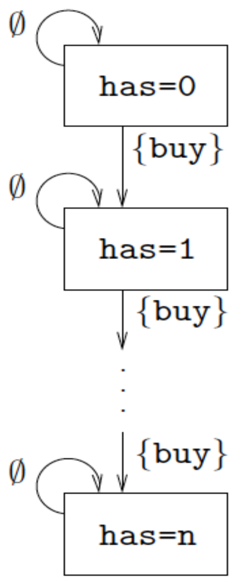
$$0 : c{=}v \Leftarrow 0 : c{=}v \tag{3.7}$$

for every simple fluent constant c and every $v \in \mathit{Dom}(c)$.

The causal models of D_m correspond to paths through a transition system representing the action description D. Viewed as a directed graph, the nodes (or "states") consist of the possible values for each constant in σ, and the edges from one state to another correspond to the actions that change the state of the system to a new state.

3.3 The Language of CCalc

The language of CCALC provides a convenient way of representing $\mathcal{C}+$. In addition to supporting all of the causal laws mentioned above, CCALC also provides mechanisms for declaring constants, declaring named domains ("sorts"), and populating those domains with values ("objects"). This is illustrated in Figure 3.1 using an action description that models a simple transition system. This domain models the concept of a person who has a certain quantity of items and can choose to buy another item to increase how many they possess by one, up to a given limit. Line 2 declares two domains, `num` and `s_num`, that can be populated and used later in the description.



```
1   :- sorts
2     num >> s_num.
3
4   :- objects
5     0..4   :: s_num;
6     5      :: num.
7
8   :- variables
9     K      :: s_num.
10
11  :- constants
12    has :: inertialFluent(num);
13    buy :: exogenousAction.
14
15  buy causes has=K+1 if has=K.
16  nonexecutable buy if has=5.
17
18  :- query
19    maxstep :: 3;
20    0: has=2;
21    maxstep: has=4.
```

Figure 3.1. Simple Transition System and Its Action Description in the Language of CCalc

The use of >> signifies that `s_num` is a *subsort* of `num`, which means that any objects added to `s_num` are automatically added to `num` as well. Lines 5 and 6 populate the `s_num` and `num` sorts (respectively) with objects. `s_num` has the numbers from `0` to 4 added to it, as does `num` (by virtue of being a *supersort* of `s_num`), which also has the number 5 added to its domain.

Line 9 declares that `K`, when used in a causal law, may stand for any value from the sort `s_num`. Variable declarations like this allow for the creation of causal laws that are automatically grounded by CCALC into sets of laws representing all possible values from the variables referenced in each law.

Line 12 declares `has` as a fluent constant with the domain of `num`. In addition to `simpleFluent`, `sdFluent` (statically determined fluent), and `action`, CCALC allows declarations of constants using special keywords that combine a constant declaration with an implicit inclusion of certain causal laws to predefine the behavior of that constant. In this case, `inertialFluent` is a keyword that declares `has` as a simple fluent constant, then implicitly adds the law

```
inertial has,
```

which is short for a set of laws

```
caused has=v if has=v after has=v
```

for every value `v` in the domain of `has`. This grants the property of inertia to `has`.

Similarly, line 13 declares `buy` as an action constant. The lack of a named domain following the constant type signifies that `buy` is to have the default Boolean domain. Like `inertialFluent`, `exogenousAction` is another CCALC keyword that automatically adds the law

```
exogenous buy
```

to the description, which stands for the set of laws

```
caused buy=v if buy=v
```

for every value `v` in the domain of `buy`. Since `buy` happens to have the Boolean domain, the set can be explicitly unfolded as

```
caused buy if buy,
caused -buy if -buy.
```

As mentioned earlier, if a constant c has the Boolean domain, using it as a bare keyword (i.e., `buy`) is understood as $c = \mathbf{true}$ (in this case, `buy=true`). Negating a bare Boolean constant $\neg c$ is shorthand for $c = \mathbf{false}$ (i.e., `buy=false`).

The causal laws in lines 15 and 16 define the behavior of the `buy` action. The first law states that an effect of executing the `buy` action is to increment the value of `has` by one. The law is in the form

$$F \textbf{ causes } G \textbf{ if } H, \tag{3.8}$$

which, because G (`has`) is a fluent formula, is equivalent in this case to the basic causal law

$$\textbf{caused } G \textbf{ if } \top \textbf{ after } F \wedge H.$$

An **if** or **after** clause can generally be omitted if it is trivially $\top$, resulting in the more compact form

$$\textbf{caused } G \textbf{ after } F \wedge H,$$

or, in the specific case of line 15,

```
caused has=K+1 after buy & has=K.
```

The law on line 16 conditionally restricts the `buy` action from executing if the person already has the maximum number of items. It too is a shortcut law, this time of the form

$$\textbf{nonexecutable } F \textbf{ if } G \tag{3.9}$$

which is shorthand for

$$\textbf{caused } \bot \textbf{ after } F \wedge G.$$

Thus, the law in the action description is equivalent to

```
caused false after buy & has=5.
```

In CCALC, `false` used as a bare keyword stands for $\bot$. Similarly, `true` stands for $\top$.

```
% Shifting atoms and clauses... done. (0.00 seconds)
% After shifting: 47 atoms (including new atoms), 135 clauses
% Writing input clauses... done.  (0.00 seconds)
% Calling ZChaff... done.
% Reading output file(s) from SAT solver... done.
% Solution time: 0 seconds.

0:  has=2

1:  has=2

ACTIONS:  buy

2:  has=3

ACTIONS:  buy

3:  has=4
```

Figure 3.2. Output of Running the Simple Transition System in CCalc

Finally, lines 18–21 define a query that can be run on this action description. In this case, the query specifies that time steps shall range from `0` to 3, and its conditions are that at time step `0`, `has` is equal to 2, and at the final time step, `has` is equal to 4. CCALC transforms these conditions into constraints, merges them with the base action description, and tries to find models of the resulting causal theory.

Running CCALC on this action description, configuring it to return the first model it finds, and invoking the included query produces the output in Figure 3.2.

As can be seen from the output, CCALC has found a model where the `buy` action is executed in time steps 1 and 2, resulting in the value of `has` increasing from 2 to 4, just as the query stipulated. Note that this is not the only model of this query; there are two other models that change when the `buy` actions are executed, but both models are similar to the one above.

Though they do not appear in the action description of Figure 3.1, two more keywords deserve mention: `rigid` and `attribute`. Declaring a constant c as rigid makes it a fluent constant, adding the law

$$\textbf{rigid}\ c \tag{3.10}$$

implicitly, which is short for the set of laws

$$\textbf{caused}\ \bot\ \textbf{if}\ c{=}v_1\ \textbf{after}\ c{=}v \wedge v \neq v_1$$

for all values v and v_1 in the domain of c. In actuality, CCALC does not do this. CCALC understands that an explicitly rigid fluent constant cannot change its value from the one initially assigned to it. As a result, instead of adding the laws above for each rigid constant c, CCALC simply strips the time stamp from them, reverting them to $c = v$ style atoms instead of the time-stamped $i : c = v$ style. This improves efficiency when using these constants by avoiding the creation of unnecessary copies of causal laws.

The other important keyword to note is `attribute`. An *attribute* is a special kind of non-Boolean exogenous action constant designed to attach to a Boolean action constant and act as a property of that action. For example, in the description in Figure 3.1, we could add an attribute `howMany` to the action `buy` that indicated how many items the person bought each time `buy` was executed. This could be declared in CCALC with the statement

```
howMany :: attribute(num) of buy,
```

which would bind the new constant `howMany` to `buy`, letting `howMany` range over the domain `num`. Upon encountering this declaration, CCALC automatically creates a supersort `num*` of `num`, adding the object `none` to `num*` and assigning `num*` as the actual domain of `howMany`. CCALC would then add the causal law

```
always howMany=none <-> -buy
```

to the action description, which is shorthand for

```
caused false after -(howMany=none <-> -buy).
```

This ties `howMany` to `buy` such that `howMany` only has a value that is not `none` if `buy` is executed. If `buy` is not executed, `howMany` must take on the value `none`. Once we define the attribute `howMany`, we can then modify the action description to support purchasing multiple items.

3.4 Stable Model Semantics

For the purposes of this work, it is sufficient to restrict our attention to propositional stable model semantics. What follows is a definition of stable model semantics similar to Ferraris's definition (2005), with extensions to include strong (i.e., classical) negation in addition to default negation. Notationally, we will distinguish between the two by using "$\neg$" to indicate default negation (negation as failure), and "$\sim$" to indicate strong negation.

A *propositional signature* σ is a set of *atoms*, which inherit their definition from classical logic. A *literal* is an atom that is optionally preceeded by $\sim$. All *propositional connectives* $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$, $\bot$, and $\top$ are allowed. $\top$ is shorthand for $\bot \rightarrow \bot$, $\neg F$ is shorthand for $F \rightarrow \bot$, and $F \leftrightarrow G$ is an abbreviation of $(F \rightarrow G) \wedge (G \rightarrow F)$. *Formulas* are combinations of literals and connectives, as defined in propositional logic.

With the understanding that $F \leftarrow G$ is an alternate representation of $G \rightarrow F$, a *rule* is a formula of the form

$$F \leftarrow G, \tag{3.11}$$

where F and G may be arbitrary (propositional) formulas. In this work, we restrict our attention to rules where F and G are finite. A *program* Π is a finite set of rules.

An *interpretation* of Π is a mapping of each atom in σ to one of the truth values **true** or **false**. We identify an interpretation I with the set of atoms X that are true in the interpretation. The definition of satisfaction for atoms is the same as in classical logic. A literal containing strong negation is satisfied if its atom is not, and vice-versa. Satisfaction for a formula is defined in a manner similar to that of classical logic.

Like the semantics for causal theories, the semantics of a program is defined in terms of a reduct. The *reduct* Π^X of a program Π relative to X is the result of replacing each maximal subformula in each rule of Π that is not satisfied by X with $\bot$. X is an *answer set* of Π if X is the minimal set satisfying Π^X. An answer set X of Π is considered *coherent* if it does not contain both an atom p and its strong negation $\sim p$. For the purposes of this work, we restrict our attention to coherent answer sets.

As an example, consider the program

$$\begin{aligned} p &\leftarrow \neg q, \\ q &\leftarrow \sim r, \\ \sim r &\leftarrow \neg\neg \sim r. \end{aligned} \tag{3.12}$$

The set $X = \{p\}$ is an answer set of the program, as can be seen by forming the reduct

$$p \leftarrow \neg\bot,$$

$$\bot \leftarrow \bot,$$

$$\bot \leftarrow \bot,$$

and observing that the latter two rules are equivalent to $\top$ and the remaining rule reduces to $p \leftarrow \top$, and therefore $\{p\}$ is the minimal set satisfying that rule. In addition, $\{q, \sim r\}$ is also an answer set of the program. The reduct relative to $\{q, \sim r\}$

$$\bot \leftarrow \bot,$$

$$q \leftarrow \sim r,$$

$$\sim r \leftarrow \neg\bot,$$

is equivalent to the conjunction of rules $q \leftarrow \sim r$ and $\sim r \leftarrow \top$, which has $\{q, \sim r\}$ as its minimally satisfying set. Further experimentation shows that these are the only two answer sets of the given program.

Observe that in the program (3.12), strong negation ($\sim$) behaves differently in the reduct from negation as failure ($\neg$). This reflects the intuitive reading of $\neg p$ as "p is not known to be true" versus that of $\sim p$, which is read as "p is known to be false." We utilize this difference in behavior between the two negations to embed features from causal logic in stable model semantics.

3.5 Translating Causal Logic Into Logic Programs

In (Ferraris et al., 2010), McCain's translation (McCain, 1997) is extended as follows. Assuming a Boolean signature σ, take any set T of definite causal rules, each of which has the form

$$a \Leftarrow G, \tag{3.13}$$

$$\neg a \Leftarrow G, \tag{3.14}$$

or

$$\bot \Leftarrow G, \tag{3.15}$$

where a is an atom and G is an arbitrary propositional formula. Define the set of rules under the stable model semantics T' as follows. For each rule (3.13), replace it with the formula $a \leftarrow \neg\neg G$; replace each rule (3.14) with the formula $\sim\!a \leftarrow \neg\neg G$; and for each rule (3.15), replace it with the formula $\neg G$. Then, add the following completeness constraints (3.16) for all atoms a:

$$\bot \leftarrow \neg a \land \neg \sim\!a. \tag{3.16}$$

Note that for T, which is definite, the modified McCain's translation yields a program that is tight (Ferraris et al., 2011). Given that definite propositional causal theories are automatically in clausal form as defined in (Ferraris, 2007), we can justify the extended McCain's translation using a simplified form of Theorem 2 from that paper:

Theorem 1 *A set of literals X from atoms in σ is a causal model of T iff X is also an answer set of T'.*

CHAPTER 4

ENCODING CAUSAL LOGIC IN ASP

4.1 Translating C+ Into Logic Programs

Consider a finite definite $\mathcal{C}+$ action description D with signature σ, where the head of each of its rules is either an atom or $\bot$. Without losing generality, we assume that, for any constant c in σ, $Dom(c)$ has at least two elements. Description D can be turned into a logic program by following these steps:

1. Turn D into a corresponding multi-valued causal theory D_m, as described in Section 3.2;
2. Turn D_m into a Boolean-valued causal theory D_m^c;
3. Turn the causal rules of D_m^c into rules under the stable model semantics;
4. Turn the result further into a logic program using F2LP, as explained in Section 4.2.

4.1.1 Definite Elimination of Multi-Valued Constants.

Consider the causal theory D_m with signature σ_m consisting of rules of the form (3.5), (3.6), and (3.7). Consider all constants $i\!:\!c$ $(0 \leq i \leq m)$ in σ_m, where c is a fluent constant of D. Create a new signature σ_m^c from σ_m by replacing each constant $i\!:\!c$ with Boolean constants $i\!:\!eql(c,v)$ for all $v \in Dom(c)$.

The causal theory D_m^c with signature σ_m^c is obtained from D_m by replacing each occurrence of an atom $i : c = v$ in D_m with $i : eql(c,v) = \mathbf{true}$

and adding the causal rules

$$i\!:\!eql(c,v')\!=\!\textbf{false} \Leftarrow i : eql(c,v)\!=\!\textbf{true} \qquad (0 \leq i \leq m) \tag{4.1}$$

for all $v, v' \in Dom(c)$ such that $v \neq v'$.

The following proposition is a simplification of Proposition 9 from Lee's paper (2005).[1]

Proposition 1 *There is a 1-1 correspondence between the models of D_m and the models of D_m^c.*

The elimination of multi-valued action constants is similar.

4.1.2 Turning Boolean-Valued Action Descriptions Into Logic Programs.

Consider D_m^c, which is obtained from D_m by eliminating all multi-valued constants in favor of Boolean constants. $h(i\!:\!F)$ is a formula obtained from $i\!:\!F$ by replacing every occurrence of $i\!:\!eql(c,v)\!=\!\textbf{true}$ in it with $h(eql(c,v),i)$[2] and every occurrence of $i\!:\!eql(c,v)\!=\!\textbf{false}$ with $\sim\! h(eql(c,v),i)$[3]. According to the modified McCain's translation, the causal rules (3.5) that represent static laws (3.3) are represented by formulas under the stable model semantics as

$$h(i\!:\!F) \leftarrow \neg\neg h(i\!:\!G) \tag{4.2}$$

($i \in \{0,\ldots,m\}$).The translation of causal rules for action dynamic laws is similar, except that i ranges over $\{0,\ldots,m-1\}$.

[1]Proposition 9 involves adding two kinds of rules. Vladimir Lifschitz pointed out that one of the kinds of rules can be dropped if the given theory is definite.

[2]In the case of atoms with rigid constants that do not require time stamps, i.e. $eql(c,v)$, we replace them with $h(eql(c,v))$.

[3]If the domain of c was Boolean originally, as an optimization we replace $h(eql(c,\textbf{false}),i)$ with $\sim\! h(eql(c,\textbf{true}),i)$ and $\sim\! h(eql(c,\textbf{false}),i)$ with $h(eql(c,\textbf{true}),i)$.

In the special case when $h(i\!:\!F)$ and $h(i\!:\!G)$ are the same literal, (4.2) can be represented using choice rules in ASP:

$$\{h(i\!:\!F)\}. \tag{4.3}$$

This is because when $h(i : F)$ and $h(i : G)$ are the same literal, (4.2) is strongly equivalent to $h(i\!:\!F) \vee \neg h(i\!:\!F)$, which can be abbreviated as (4.3) (Lee, Lifschitz, & Palla, 2008a). In fact, we observe that in many cases (4.3) can be used place of (4.2).

Similarly, the modified McCain's translation turns the causal rules (3.6) that correspond to fluent dynamic laws (3.4) into

$$h(i{+}1\!:\!F) \;\leftarrow\; \neg\neg\Big(h(i{+}1\!:\!G) \;\wedge\; h(i\!:\!H)\Big). \tag{4.4}$$

We can also turn (3.6) into

$$h(i{+}1\!:\!F) \;\leftarrow\; \neg\neg h(i{+}1\!:\!G) \;\wedge\; h(i\!:\!H) \tag{4.5}$$

because the change does not affect the stable models of the resulting theory, which is tight (Ferraris et al., 2011). Similarly, certain occurrences of $\neg\neg$ in (4.2) and (4.5) can be further dropped if removing them does not cause the resulting theory to become non-tight, which may change the stable models.

Again in the special case when $h(i{+}1 : F)$ and $h(i{+}1 : G)$ are the same literal, (4.5) can be represented using choice rules as follows:

$$\{h(i{+}1\!:\!F)\} \;\leftarrow\; h(i\!:\!H).$$

4.2 Representing Domain Descriptions in the Language of F2LP

Figure 4.1 shows a side-by-side comparison of an example CCALC input program (on the left) and its representation in the language of F2LP (on the right). As the

```
1   :- sorts
2     num >> s_num.
3
4
5
6
7
8
9
10
11  :- objects
12    0..4   :: s_num;
13    5      :: num.
14
15  :- variables
16    K      :: s_num.
17
18  :- constants
19    has :: inertialFluent(num);
20
21
22    buy :: exogenousAction.
23
24
25  buy causes has=K+1 if has=K.
26
27
28
29  nonexecutable buy if has=5.
30
31
32  :- query
33    maxstep :: 3;
34    0: has=2;
35    maxstep: has=4.
```

```
1
2   sort(num).
3   #domain num(V_num).
4   sort_object(num,V_num).
5
6   sort(s_num).
7   #domain s_num(V_s_num).
8   sort_object(num,V_s_num).
9
10  num(V_s_num).
11
12  s_num(0..4).
13  num(5).
14
15
16  #domain s_num(K).
17
18
19  inertialFluent(has).
20  constant_sort(has,num).
21
22  exogenousAction(buy).
23  constant_sort(buy,boolean).
24
25  h(eql(has,K+1),V_astep+1) <-
26     h(eql(buy,true),V_astep) &
27     h(eql(has,K),V_astep).
28
29  false <-
30     h(eql(buy,true),V_astep) &
31     h(eql(has,5),V_astep).
32
33  false <- query_label(0) &
34           not (h(eql(has,2),0) &
35                h(eql(has,4),maxstep)).
```

Figure 4.1. Simple Transition System in the Language of CCalc and in the Language of F2LP

example shows, the translation is modular. For each sort name S that is declared in the CCALC input program, the translation introduces a fact `sort`(S) along with a variable V_S that ranges over all objects of the sort S (expressed by the line `#domain` $S(V_S)$). The translation relates the sort name to the objects of the sort by the fact `sort_object`(S, V_S). The declaration that S_1 is a supersort of S_2 is represented by $S_1(V_{S_2})$, as illustrated in line 10.

The ASP representation of the object and variable declarations are straightforward. The declaration that O is an object of sort S is encoded as a fact $S(O)$. In order to declare a user-defined variable V of sort S, we write `#domain` $S(V)$. See lines 12–13 and line 16 for examples.

A constant declaration in the language of CCALC of the form

$$C :: CompositeSort(V)$$

is turned into a fact $CompositeSort(C)$, followed by the declaration of a meta-predicate `constant_sort`(C, V), which is used in the standard library. Lines 19–23 are an example.

Encoding causal laws in the language of F2LP follows the method in Section 3.5. Like in the input language of CCALC, variables in the F2LP rules are understood as schemas for ground terms. Lines 25–31 show an encoding of causal laws in the language of F2LP. Since every variable is sorted, these F2LP rules are safe according to the definition of safety by Lee et al. (2008b), and the translation of these rules into an ASP program also results in a safe logic program.

Note that the translation in Figure 4.1 does not include certain causal laws from the complete action description (in particular, the inertial assumption for `has` and the exogeneity assumption for `buy` are not present). Since such causal laws and rules are frequently used, they are expressed in a general form in the standard library, as explained in the next section.

4.3 Standard Library File

The standard library[4] contains declarations of predicates, variables, and postulates that are common to all translated action descriptions. Certain declarations, like that of the `boolean` sort and its objects `true` and `false`, mirror internal CCALC declarations. Other declarations, like that of the `constant_object` predicate, are specific to the translation and encoding method, and are used to mimic features of CCALC that are not handled by the software components of the CPLUS2ASP system.

4.3.1 Postulates for Different Fluents and Actions. First, we assume the presence of certain meta-variables that are used in the postulates. `V_step` is a variable of the sort `step`, whose objects range over the values $0, 1, \dots,$ `maxstep`. `V_astep` is a variable of the sort `astep`, whose objects range over the values $0, 1, \dots,$ `maxstep` $- 1$. `V_inertialFluentAF` is a meta-variable that ranges over all ground terms of the form `eql`(c, v), where c is an `inertialFluent` and v is an object in the domain of c as introduced in the domain description. For example, for the domain description in Figure 4.1, `V_inertialFluentAF` ranges over the values `eql(has,0)`, `eql(has,1)`$, \dots,$ `eql(has,5)`. Similarly, we have other meta-variables `V_fluentAF`, `V_simpleFluentAF`, `V_sdFluentAF`, `V_rigidAF`, `V_actionAF`, `V_exogenousActionAF`, and `V_attributeAF` that range over ground terms of the form `eql`(c, v), where c and v range over corresponding constants and values.

We show later how to prepare a program so that meta-variables range over the atoms as intended.

[4]Presented in Appendix A and available at `http://reasoning.eas.asu.edu/cplus2asp`

The inertial assumption for `inertialFluent` constants is represented by

```
h(V_inertialFluentAF,V_astep+1) <-
    not not h(V_inertialFluentAF,V_astep+1) &
    h(V_inertialFluentAF,V_astep),
```

or equivalently as

```
 {h(V_inertialFluentAF,V_astep+1)} <-
  h(V_inertialFluentAF,V_astep).
```

The exogeneity assumption (3.7) for simple fluents in the initial time step is represented by

```
  h(V_simpleFluentAF,0) <- not not h(V_simpleFluentAF,0),
```

or equivalently as

```
  {h(V_simpleFluentAF,0)}.
```

The exogeneity assumptions for `exogenousAction` and `attribute` constants are stated as

```
  {h(V_exogenousActionAF,V_astep)}
```

and

```
  {h(V_attributeAF,V_astep)},
```

here shortened like other simple exogeneity rules.

In addition, we say that attributes take the special value `none` iff the corresponding action is not executed, per the definition of attributes in Section 3.3.

```
  false <- not
      ( h(eql(V_attribute,none),V_astep) <->
        -h(eql(V_action,true),V_astep) )
      & action_attribute(V_action,V_attribute).
```

action_attribute is obtained from the declaration of attributes. It records the relation between an action and its attributes.

The completeness assumption (3.16) for fluents is represented as follows.

```
  false <- not h(V_fluentAF,V_step) & not -h(V_fluentAF,V_step),
```

or equivalently as

```
  false <- {h(V_fluentAF,V_step), -h(V_fluentAF,V_step)}0.
```

The definite elimination rules for multi-valued fluent constants corresponding to (4.1) can be represented as

```
-h(eql(V_fluent,Object1),V_step) <-
   h(eql(V_fluent,Object),V_step) &
   constant_object(V_fluent,Object) &
   constant_object(V_fluent,Object1) & Object != Object1.
```

Here, V_fluent is a meta-variable that ranges over all fluent constants. The predicate constant_object is defined in terms of sort_object and constant_sort:

```
  constant_object(V_constant,Object) <-
      constant_sort(V_constant,V_sort) &
      sort_object(V_sort,Object).
```

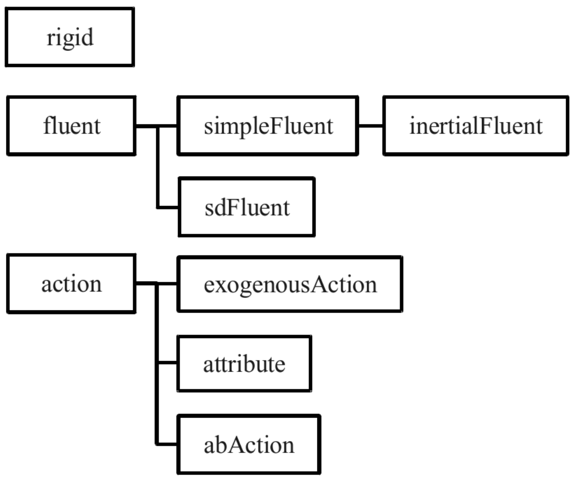


Figure 4.2. The Hierarchy of Constant Meta-Sorts

As stated earlier, `sort_object` is introduced as part of translating sort declarations from the domain description, and `constant_sort` is introduced while translating constant declarations in the domain description.

The definite elimination rules and the completeness assumptions for action constants are similar to those for fluent constants. The rules for rigid fluent constants are the same as the ones for fluent constants that are not rigid, except rigid fluent constants do not have a time stamp (i.e., `V_step` is omitted).

4.3.2 Meta-Sorts and Meta-Variables. In order to have grounding replace all meta-variables with the corresponding ground atoms as intended in the previous section, we introduce meta-level sorts for representing the constant hierarchy, shown in Figure 4.2. This is done in the same way as introducing user-defined sorts. For instance, the following are declarations for the `simpleFluent` and `inertialFluent` meta-sorts, along with the declaration of their subsort relation.

```
sort(simpleFluent).
#domain simpleFluent(V_simpleFluent).
sort_object(simpleFluent,V_simpleFluent).

sort(inertialFluent).
#domain inertialFluent(V_inertialFluent).
sort_object(inertialFluent,V_inertialFluent).

simpleFluent(V_inertialFluent).
```

Recall that in Figure 4.1, line 19 of the F2LP program declared the fact `inertialFluent(has)` as part of a constant declaration. As a result of the declarations above, the variable `V_simpleFluent` ranges over all simple fluent constants, including the inertial fluent `has` in the example.

Similarly, we introduce meta-level sorts for different categories of atomic formulas that are related to each kind of constant. For example, the following is a part of the declaration for the meta-sorts `simpleFluentAF` and `inertialFluentAF`.

```
sort(simpleFluentAF).
#domain simpleFluentAF(V_simpleFluentAF).
sort_object(simpleFluentAF,V_simpleFluentAF).

sort(inertialFluentAF).
#domain inertialFluentAF(V_inertialFluentAF).
sort_object(inertialFluentAF,V_inertialFluentAF).

simpleFluentAF(V_inertialFluentAF).
```

These declarations are used to define *ConstantAF domain predicates* which contain atomic formulas of the form eql(c, v), where c is a constant of meta-level sort *Constant* and v is a value in the domain of c. For instance, the following represents that simpleFluentAF and inertialFluentAF are domain predicates that contain all atomic formulas of the form eql(c, v), where c is a simpleFluent or inertialFluent (respectively), and v is a value in the domain of c, using the meta-predicate constant_object.

```
simpleFluentAF(eql(V_simpleFluent,Object)) <-
   constant_object(V_simpleFluent,Object).

inertialFluentAF(eql(V_inertialFluent,Object)) <-
   constant_object(V_inertialFluent,Object).
```

Recall that constant_object is derived from a combination of constant_sort and sort_object declarations.

The grounding process replaces the meta-variable V_simpleFluentAF by every ground term of the form eql(c, v) where c is a constant of the meta-level sort simpleFluent (and its subsorts as well) and v is an element in the domain of c, as specified by the constant_object relation. Once the user declares that c is a simpleFluent (or one of its subsorts) in the domain description, the postulates for initial exogeneity of the value of c are automatically generated by the ASP grounders. If c is declared an inertialFluent, the inertial assumption for c is automatically generated as well.

As a result of this process, the ground ASP program produced from the encoding of a translated action description will contain identical rules to that of the modified McCain's translation of D^c_m, as described in Section 4.1.2. The use of the standard library's meta-predicates will introduce additional rules and predicates, but these will just be intermediate declarations in support of the postulates in the standard library; removing these extra rules and predicates from the ground program will produce a program identical to the one created in Section 4.1.2.

CHAPTER 5

SOFTWARE SUPPORT

While we have shown that this translation and encoding method is sound and convenient due to its modular form, it remains a nontrivial process to perform the translation of any relatively large domain by hand; one must have a firm grasp of CCALC syntax and $\mathcal{C}+$ semantics, along with a solid understanding of F2LP syntax and ASP semantics, to be able to faithfully follow the steps outlined above. To make the translation process more practical, we utilize a combination of existing software and newly created programs (collectively referred to as "the tool chain") to handle the complete process of translation, solution generation, and interpretation of results, making it much easier to translate action descriptions and significantly reducing the possibility of human error.

5.1 Cplus2ASP

CPLUS2ASP was written to act as an overarching system designed to automate the process of calling the tool chain in the correct order and with the appropriate options. By default, CPLUS2ASP takes CCALC input, processes it with the translator module (`cplus2asp.bin`) and F2LP, invokes GRINGO and CLASP to ground and solve the resulting ASP program, and finally passes the answer sets returned by CLASP to `as2transition` so they can be transformed back into a more human-readable form, presenting CCALC-style models of the original input (i.e., similar to Figure 3.2) as its final output.

In addition to managing the tool chain, CPLUS2ASP can also interactively run queries and automatically determine various parameters and settings based on the input to and output from various stages of the tool chain.

5.2 Cplus2ASP.bin

The program `cplus2asp.bin` was created as a module designed to accept CCALC input and automatically use the encoding method described earlier to produce equivalent F2LP input. It not only handles $\mathcal{C}+$ causal laws but also supports many extra features of CCALC, such as rigid constants, automatic insertion of necessary postulates for CCALC constant keywords (like `inertialFluent`), support for all "shortcut" causal law forms, elimination of multi-valued constants in favor of Boolean constants, and translation of queries in CCALC syntax into constraints in the language of F2LP. These features allow `cplus2asp.bin` to process many action descriptions without any changes required to the original CCALC input. However, `cplus2asp.bin` is currently a prototype, and as such, certain advanced features of CCALC are not yet fully supported, including macro expansion, handling of nested constants, full evaluation of "`where`" clauses in causal laws, and support for certain dynamic declarations and syntactic shortcuts. That said, `cplus2asp.bin` can still properly parse and translate many action descriptions, including all examples from "Nonmonotonic Causal Theories" (Giunchiglia et al., 2004), without any modifications required to the original input files.

`cplus2asp.bin` forms the core of CPLUS2ASP, as it makes the system practical for use by anyone with a good working knowledge of CCALC. It was created in C++ using `flex` and `bison` to streamline the process of creating an

```
clasp version 1.3.7
Reading from stdin
Solving...
Answer: 1
constant_sort(has,num) constant_sort(buy,boolean) inertialFluent(has)
simpleFluent(has) fluent(has) exogenousAction(buy) action(buy) s_num(0)
s_num(1) s_num(2) s_num(3) s_num(4) num(4) num(3) num(2) num(1) num(0)
num(5) h(eql(buy,true),2) h(eql(buy,true),1) h(eql(has,2),0) h(eql(has,2),1)
h(eql(has,3),2) h(eql(has,4),3) -h(eql(has,5),3) -h(eql(has,5),0)
-h(eql(has,5),1) -h(eql(has,5),2) -h(eql(has,0),3) -h(eql(has,0),0)
-h(eql(has,0),1) -h(eql(has,0),2) -h(eql(has,1),3) -h(eql(has,1),0)
-h(eql(has,1),1) -h(eql(has,1),2) -h(eql(has,2),3) -h(eql(has,2),2)
-h(eql(has,3),3) -h(eql(has,3),0) -h(eql(has,3),1) -h(eql(has,4),0)
-h(eql(has,4),1) -h(eql(has,4),2) -h(eql(buy,true),0) sort(s_num)
sort(num) query_label(0)
```

Figure 5.1. Answer Set of the Translated Simple Transition System

acceptable grammar that recognizes CCALC input and parses it correctly. The entire program consists of approximately eleven thousand lines of code, including the definitions of the lexical analyzer and parser.

5.3 AS2Transition

The output of CPLUS2ASP at this point is a series of answer set outputs from the ASP solver. While it is possible to parse this output in its native format, the transformations performed during the translation and encoding of an action description make it difficult to understand and analyze the answer sets in terms of the original action description. As a demonstration, Figure 5.1 shows the raw output of an answer set from CLASP after running CPLUS2ASP on the simple transition system in Figure 3.1.

To improve readability of the output of CPLUS2ASP, `as2transition` was written to take the answer sets output by ASP solvers and transform them into output similar to that of CCALC. In addition to turning $h(eql(c, v), i)$ predicates back into their original $i : c = v$ forms, `as2transition` also separates

```
Solution 1:
0:  has=2
1:  has=2
  ACTIONS:  buy
2:  has=3
  ACTIONS:  buy
3:  has=4
```

Figure 5.2. Transformed Answer Set of the Translated Simple Transition System

action constants from fluent constants and arranges them in order of time stamp, making it much easier to parse and analyze the output from CPLUS2ASP. Figure 5.2 shows the result of passing the output in Figure 5.1 to `as2transition`. The similarity between the output of `as2transition` and CCALC allows users familiar with CCALC to use CPLUS2ASP instead with little to no change in how they write their action descriptions or how they collect and analyze their data.

5.4 Using Cplus2ASP

CPLUS2ASP was designed to gracefully handle the complexity of the underlying tool chain, the goal being to make it easy to use from an end-user perspective. For example, if the action description in Figure 3.1 were saved in a file called `has.cp`, passing the file to CPLUS2ASP using the default options (ask which query to run, return the first solution found) would only require the following command line:

```
cplus2asp has.cp
```

This causes CPLUS2ASP to translate the input file, automatically find and remember any queries defined, and interactively prompt the user for which query to run before calling the answer set solver and `as2transition`. The number `0` can be added to the end of the command line to have CPLUS2ASP return all solutions found by the answer set solver; positive numbers used in this way tell CPLUS2ASP to look for a specific number of solutions.

Normally, CPLUS2ASP does not show any intermediate information or data, just the output from `as2transition` along with any errors that may have been reported by the programs in the tool chain. However, it is possible to stop CPLUS2ASP partway through its translating and solving process and have it show the ASP input being sent to the answer set solver. Having a copy of the input to the answer set solver can come in handy as a debugging aid if the grounder reports an error in the translated description.

CPLUS2ASP is designed to use GRINGO and CLASP as the respective grounder and solver for the ASP part of the tool chain. However, it can also be adapted to use a combined grounder and solver, like CLINGO, if desired.

CHAPTER 6

EXPERIMENTS

6.1 Benchmark Problems

During the creation and subsequent use of CCALC, several "benchmark" domains were created or adapted from prior systems in order to test both the efficiency and the expressive capabilities of CCALC. These domains were selected to be translated (both by hand and using the automated translator) and evaluated with respect to the correctness of the translations, along with the speed and efficiency of the ASP-based solving system. Note that for the purposes of benchmarking, CLINGO was used instead of GRINGO and CLASP, as the integrated grounding and solving capabilities of CLINGO generally make it a faster and more efficient solver compared to invoking its components separately.

6.1.1 NMCT Benchmarks. Each of the sample domains presented in "Nonmonotonic Causal Theories" (Giunchiglia et al., 2004) either demonstrate capabilities of $\mathcal{C}+$ or formalize classic knowledge representation problems. Producing correct solutions for these domains shows that a given system can overcome key challenges of knowledge representation, such as the frame problem. As these domains conveniently cover a wide set of capabilities, we also chose them as tests of CPLUS2ASP. Table 6.1 shows the comparative results of running each of the NMCT example domains through CCALC versus the performance of CPLUS2ASP on the same input.

Table 6.1

Comparative Performance of CCalc and Cplus2ASP: NMCT Domains

Problem	CCALC with ZCHAFF				CPLUS2ASP with CLINGO			
	Total	Preparation[a]	Solving	Size[b]	Total	Preparation[c]	Solving[d]	Size[e]
Going to Work (`maxstep`=1)	0.03s	0.03s (0.03 + 0.00 + 0.00)	0.00s	A:14 C:36	0.03s	0.03s (0.02 + 0.01)	0.00s (0.00 + 0.00)	A:170 R:200
Lifting the Table (`maxstep`=1)	0.03s	0.03s (0.02 + 0.01 + 0.00)	0.00s	A:13 C:26	0.03s	0.03s (0.03 + 0.00)	0.00s (0.00 + 0.00)	A:193 R:226
Monkey and Bananas (`maxstep`=4)	0.06s	0.06s (0.05 + 0.01 + 0.00)	0.00s	A:91 C:438	0.05s	0.04s (0.02 + 0.02)	0.01s (0.01 + 0.00)	A:819 R:1277
Pendulum (`maxstep`=2)	0.01s	0.01s (0.01 + 0.00 + 0.00)	0.00s	A:5 C:8	0.02s	0.02s (0.02 + 0.00)	0.00s (0.00 + 0.00)	A:92 R:106
Publishing Papers (`maxstep`=3)	0.16s	0.15s (0.07 + 0.06 + 0.02)	0.01s	A:834 C:2096	0.30s	0.28s (0.02 + 0.26)	0.02s (0.02 + 0.00)	A:2710 R:33613
Shooting Turkeys (`maxstep`=6)	0.03s	0.03s (0.02 + 0.01 + 0.00)	0.00s	A:66 C:178	0.04s	0.03s (0.02 + 0.01)	0.01s (0.00 + 0.01)	A:668 R:946

[a]grounding time + completion time + shifting and writing input clause time
[b]atoms and clauses of ground SAT solver input
[c]`cplus2asp.bin` and F2LP processing time + CLINGO grounding time
[d]CLINGO preprocessing time + solving time
[e]atoms and rules of ground ASP input

What follows are brief descriptions of each domain, including the feature or problem it represents.

- Going to Work: This domain formalizes the concept of a person who starts at home and can either walk or drive to work. It utilizes the concept of nondeterministic actions and demonstrates the ability of $\mathcal{C}+$ to support this.
- Lifting the Table: An object is on a large table that requires two people to lift. They must lift each end at the same time or else the object will fall off. Correct models require that actions be allowed to execute simultaneously.
- Monkey and Bananas: This is a classic domain used to demonstrate various aspects of planning problems via a monkey that wants to get bananas that are hung from the ceiling of a room.
- Pendulum: A pendulum swings back and forth unless someone holds on to it to stop its motion temporarily. This is a demonstration of defeasible actions and redefining inertia.
- Publishing Papers: A professor publishes papers of varying length to different venues and must keep track of what kinds of papers (conference, journal, etc.) have been published. The domain utilizes attributes to elaborate on a basic publishing action.
- Shooting Turkeys: A slightly less homicidal version of the Yale Shooting Problem with two turkeys pursued by a hunter. This domain is a classic formalization of the frame problem; while the hunter is reloading his gun

and aiming at the second turkey, the one shot first should remain dead and not return to life as a zombie turkey.

6.1.2 Zoo World and Traffic World. Zoo World and Traffic World (Akman et al., 2004) are both nontrivial formalizations of medium-sized action domains suitable for testing the ability of $\mathcal{C}+$ to express and reason about complex relationships, including indirect effects of actions, chain reactions of cause and effect, and conditional or nondeterministic actions. Indirectly, they also serve as a stress test for CCalc and Cplus2ASP, as even the smallest Zoo World or Traffic World example is far larger (in terms of size of the grounded program) than any of the NMCT examples, and it is relatively simple to scale up the Zoo World and Traffic World domains in a nontrivial fashion. Tables 6.2 and 6.3 show the results of running various scenarios in both systems.

Table 6.2

Comparative Performance of CCalc and Cplus2ASP: Zoo World

Problem	CCALC with ZCHAFF				CPLUS2ASP with CLINGO			
	Total	Preparation[a]	Solving	Size[b]	Total	Preparation[c]	Solving[d]	Size[e]
Zoo World: Test 1 (`maxstep=4`)	50.45s	49.83s (42.70 + 4.36 + 2.77)	0.62s	A:8833 C:129521	1.86s	1.49s (0.05 + 1.44)	0.37s (0.37 + 0.00)	A:36409 R:72597
Zoo World: Test 2 (`maxstep=2`)	14.61s	14.41s (11.03 + 2.99 + 0.39)	0.20s	A:3952 C:41295	0.66s	0.52s (0.04 + 0.48)	0.14s (0.14 + 0.00)	A:22892 R:33166
Zoo World: Test 3 (`maxstep=2`)	44.67s	44.40s (39.76 + 4.26 + 0.38)	0.27s	A:5527 C:70965	1.05s	0.84s (0.04 + 0.80)	0.21s (0.20 + 0.01)	A:26894 R:46777
Zoo World: Test 4 (`maxstep=4`)	47.49s	46.96s (40.64 + 4.35 + 1.97)	0.53s	A:7180 C:100243	1.75s	1.34s (0.04 + 1.30)	0.41s (0.36 + 0.05)	A:36386 R:72615
Zoo World: Test 5 (`maxstep=1`)	132.70s	132.43s (124.57 + 7.05 + 0.81)	0.27s	A:4824 C:67264	1.00s	0.82s (0.05 + 0.77)	0.18s (0.17 + 0.01)	A:24898 R:43285
Zoo World: Test 6 (`maxstep=1`)	45.66s	45.53s (40.86 + 4.30 + 0.37)	0.13s	A:29864 C:41687	0.63s	0.51s (0.05 + 0.46)	0.12s (0.12 + 0.00)	A:22148 R:33811
Zoo World: Big Cage Shuffle (`maxstep=19`)	-.--s	3184.92s (2463.45 + 333.40 + 388.07)	-.--s[f]	A:282956 C:12448077	187.04s	77.79s (0.05 + 77.74)	109.25s (19.24 + 90.01)	A:801367 R:2779945

[a]grounding time + completion time + shifting and writing input clause time
[b]atoms and clauses of ground SAT solver input
[c]`cplus2asp.bin` and F2LP processing time + CLINGO grounding time
[d]CLINGO preprocessing time + solving time
[e]atoms and rules of ground ASP input
[f]ZCHAFF was unable to process this domain

Table 6.3

Comparative Performance of CCalc and Cplus2ASP: Traffic World

Problem	CCALC with ZCHAFF				CPLUS2ASP with CLINGO			
	Total	Preparation[a]	Solving	Size[b]	Total	Preparation[c]	Solving[d]	Size[e]
Traffic World: Scenario 1 (`maxstep=5`)	1.55s	1.52s (1.06 + 0.29 + 0.17)	0.03s	A:2324 C:13482	0.19s	0.17s (0.07 + 0.10)	0.02s (0.02 + 0.00)	A:5436 R:7590
Traffic World: Scenario 2 (`maxstep=3`)	22.74s	22.38s (17.85 + 3.45 + 1.08)	0.36s	A:17994 C:110128	1.30s	0.89s (0.04 + 0.85)	0.41s (0.41 + 0.00)	A:71407 R:92057
Traffic World: Scenario 3 (`maxstep=5`)	6.29s	6.05s (4.48 + 0.99 + 0.58)	0.24s	A:8792 C:56176	0.48s	0.36s (0.04 + 0.32)	0.12s (0.12 + 0.00)	A:29038 R:37315
Traffic World: Scenario 3-1 (`maxstep=11`)	608.76s	558.46s (415.06 + 85.45 + 57.95)	50.30s	A:531552 C:3671940	53.44s	28.82s (0.07 + 28.75)	24.62s (17.59 + 7.03)	A:2722511 R:3341332

[a]grounding time + completion time + shifting and writing input clause time
[b]atoms and clauses of ground SAT solver input
[c]`cplus2asp.bin` and F2LP processing time + CLINGO grounding time
[d]CLINGO preprocessing time + solving time
[e]atoms and rules of ground ASP input

Most of the Zoo World scenarios are structured as planning problems, such as how to get a certain number of people and animals into a cage within a certain period of time. The "Big Cage Shuffle" Zoo World scenario is a larger-scale version of the world described in the other scenarios, and while it is also structured as a planning problem ("How quickly can all people and animals in the world visit every cage in the zoo?"), its more open-ended nature, larger size, and much longer solution all present challenges for both systems. The use of query conditions based on a variable time stamps caused CCALC to handle the query in an unexpected manner, resulting in ZCHAFF returning incorrect solutions.

The senarios for Traffic World include a mix of planning problems and prediction. The challenges of computing solutions to the Traffic World scenarios are similar to those of the Zoo World scenarios in the sense that the domain (with multiple cars and road segments) can quickly become large. However, the Traffic World scenarios utilize numeric computation more frequently. Scenario 3-1 is an example of what scaling can do to the computation time required to produce solutions to Traffic World scenarios, as it is effectively a scaled-up version of Scenario 3 with more cars and longer roads.

6.1.3 Other Domains. Two other domains demonstrate interesting properties of and test the limitations of both systems. The Tower of Hanoi is a classic puzzle involving moving discs along a set of pegs, guided by a set of simple rules regarding which discs can move and where they can move to. It is an interesting domain due to the fact that increasing the number of discs in the puzzle geometrically increases the number of moves required to solve it, also geometrically increasing the size of the grounded action description.

The Knapsacks of Marbles domain is a puzzle variant of the knapsack problem: how to fit a certain number (or configuration) of items of varying dimensions into a knapsack of a given size. As presented here, the puzzle version of the problem describes piles of marbles of varying size and a set of bags, which are also of varying size. The challenge is to put the marbles in the bags such that all of the marbles fit and none are left outside of a bag. This puzzle is formalized as a reasoning about state problem, meaning that placing a marble in a bag does not involve performing an action; instead, the bags are filled with marbles all at once. As a result, solutions can be found via manipulation and analysis of the initial state, meaning that the workload for reasoning about this domain is left almost entirely to the grounders of the respective systems. In addition, the Knapsacks of Marbles puzzle heavily utilizes numeric computation.

Table 6.4

Comparative Performance of CCalc and Cplus2ASP: Miscellaneous Domains

Problem	CCALC with ZCHAFF				CPLUS2ASP with CLINGO			
	Total	Preparation[a]	Solving	Size[b]	Total	Preparation[c]	Solving[d]	Size[e]
Knapsacks of Marbles (`maxstep`=0)	43.60s	43.42s (37.37 + 5.58 + 0.47)	0.18s	A:409 C:55806	3.07s	1.52s (0.07 + 1.45)	1.55s (1.54 + 0.01)	A:56756 R:115444
Tower of Hanoi: 4 Discs (`maxstep`=15)	1.75s	1.39s (0.44 + 0.23 + 0.72)	0.36s	A:6043 C:58059	0.37s	0.23s (0.03 + 0.20)	0.14s (0.07 + 0.07)	A:11422 R:21852
Tower of Hanoi: 5 Discs (`maxstep`=31)	22.36s	3.94s (0.86 + 0.49 + 2.59)	18.42s	A:17487 C:192947	3.05s	0.64s (0.03 + 0.61)	2.41s (0.28 + 2.13)	A:29811 R:61141
Tower of Hanoi: 6 Discs (`maxstep`=63)	193.01s	9.40s (1.18 + 0.68 + 7.54)	183.61s	A:47505 C:590295	41.34s	1.96s (0.02 + 1.94)	39.38s (0.93 + 38.45)	A:77270 R:166390

[a] grounding time + completion time + shifting and writing input clause time
[b] atoms and clauses of ground SAT solver input
[c] `cplus2asp.bin` and F2LP processing time + CLINGO grounding time
[d] CLINGO preprocessing time + solving time
[e] atoms and rules of ground ASP input

As can be seen from the results in Table 6.4, the domains stress test different parts of the systems, allowing us to observe comparative efficiencies between individual components. It is of interest to note that CCALC was able to keep the size of the ground Knapsacks of Marbles domain relatively small compared to ASP. This is likely due to the fact that the Knapsacks of Marbles action description makes frequent use of rigid constants; CCALC is able to perform several optimizations when rigid constants are used in action descriptions, whereas CPLUS2ASP performs fewer optimizations and still has to include all meta-predicate declarations for all rigid constants.

6.2 Size of Domains

A side effect of the translation of CCALC action descriptions into ASP and the use of our encoding method and standard library is that the translated domain, while relatively compact thanks to the use of meta-postulates in the standard library, will still increase in size as a result of the translation. Table 6.5 demonstrates this by comparing the word counts of each of the NMCT examples, the Zoo World domain, and the Traffic World domain before translation (as CPLUS2ASP input), after translation (as F2LP input), and finally as raw ASP code suitable for use with CLINGO.

6.3 Analysis

Based on the results outlined in the tables above, CPLUS2ASP demonstrates itself to be a competitive system compared to CCALC; in many cases CPLUS2ASP can also outperform CCALC in terms of solving speed by over an order of magnitude.

Table 6.5

Sizes of Domains as Cplus2ASP Input, F2LP Input, and Clingo Input

Problem	Word Count		
	CPLUS2ASP Input	F2LP Input[a]	CLINGO Input[b]
Going To Work	77	160	923
Lifting the Table	47	108	867
Monkey and Bananas	216	521	1336
Pendulum	26	83	826
Publishing Papers	64	193	978
Shooting Turkeys	60	157	925
Zoo World	1023	1994	3751
Traffic World	847	1709	3280

[a]after processing with `cplus2asp.bin`
[b]after processing with F2LP and including the standard library

In addition, further testing for correctness of the solutions for each domain showed that CPLUS2ASP produced identical solutions to CCALC in all examples for all queries. CCALC consistently requires far fewer atoms than CPLUS2ASP to ground domains, but CPLUS2ASP almost always outperforms CCALC when it comes to the number of rules created in CPLUS2ASP versus the number of clauses used by CCALC.

As the data shows, when CPLUS2ASP is able to solve problems faster than CCALC, it is often due to much lower grounding times. The grounding techniques used by GRINGO make it very efficient, especially with domains that have numeric computations, allowing it to outperform the grounding methods of CCALC for those domains. This development presents interesting possibilities, as prior to this, numerically dense domains were often off-limits to CCALC due to problems it could have grounding these domains because of their size. With further optimization, it appears possible to utilize CPLUS2ASP to tackle a wide variety of reasoning problems involving numeric domains.

It should be noted that all of the benchmark tests were only run on one query from each domain. CCALC has a slight advantage with respect to grounding when multiple queries are run on the same domain, as its internal atom and clause shifting mechanisms can efficiently copy the base domain over multiple queries and time steps. In contrast, CPLUS2ASP must process and fully ground a domain each time a different query is run, even if the only change is to alter the range of time steps being considered. This is a current limitation of the answer set solvers CPLUS2ASP utilizes, and it is expected that as incremental solvers like ICLINGO (Gebser et al., 2008) improve, they can be coupled with CPLUS2ASP to potentially allow CPLUS2ASP to use dynamic grounding methods to emulate the shifting ability of CCALC.

The domain size comparison in Table 6.5 points to another advantage of using CPLUS2ASP, in particular the automated translator `cplus2asp.bin`. The increased word counts of the domains reflect an increase in the difficulty of translating them, as can be observed from the translation of the simple transition system in Figure 4.1. The translation and encoding method described in Chapter 4 can be performed without automated aids, but doing so becomes prone to human error and requires advanced knowledge of CCALC and F2LP syntax when translating nontrivial domains.

CHAPTER 7

RELATED WORK

CPLUS2ASP is a new addition to a set of systems designed to transform action formalisms into ASP. There are three other predecessors to CPLUS2ASP that indirectly inspired its creation. The first was created by Doğandağ et al. (2001) and turned action language $\mathcal{C}$ into the language of SMODELS, an early implementation of stable model semantics. The work described a complete encoding method, but the implementation is not publicly available.

Another was the $\mathcal{ALM}$ to ASP system[1] (Gelfond & Inclezan, 2010), which takes action language $\mathcal{ALM}$ and translates it into ASP. $\mathcal{ALM}$ (Gelfond & Inclezan, 2009) can be thought of as a cousin to the Modular Action Description language (MAD) (Lifschitz & Ren, 2006), with syntax similar to a modified version of $\mathcal{C}+$ that supports the concept of description-independent modules.

The last inspiration for CPLUS2ASP was COALA[2] (Gebser, Grote, & Schaub, 2010), a system that translates a limited verion of $\mathcal{C}+$, in addition to other similar formalisms like action languages $\mathcal{B}$ and $\mathcal{AL}$, into ASP so that they can be computed by answer set solvers. COALA has many of the same basic features and abilities of CPLUS2ASP but lacks support for features like multi-valued constants or non-exogenous actions, which prevents COALA from supporting constructs like attributes. It also requires the use of a specialized syntax, meaning native CCALC action descriptions cannot be used with COALA without first translating them into the input language of COALA. As an example, Figure 7.1 shows the syntax for a simple domain involving opening a closed door. As the

[1] `http://www.webpages.ttu.edu/dincleza/ALM/`
[2] `http://www.cs.uni-potsdam.de/wv/coala/`

```
<action> openDoor.
<fluent> closed.

<caused> closed <if> closed.

<caused> -closed <if> <true> <after> openDoor.
```

Figure 7.1. Sample Coala Action Description

example shows, COALA action descriptions take inspiration from CCALC action descriptions, but utilize a significantly different syntax for constant declarations in addition to a more restrictive format for causal laws.

CHAPTER 8

CONCLUSION

We have created an efficient and modular encoding method for $\mathcal{C}+$ that enables it to be translated into the language of ASP. Our experiments demonstrate that this new translation faithfully captures the features of $\mathcal{C}+$. In addition, we have shown that using answer set solvers to compute models of translated action descriptions is generally far faster and more efficient than using CCALC.

The software system CPLUS2ASP automates the process of performing this translation and calling the necessary programs to produce CCALC-style output, making this encoding method practical for use by anyone familiar with CCALC and causal logic.

By transforming CCALC input into the language of ASP, we capture the best of both worlds: we retain the expressivity and ease of use of $\mathcal{C}+$ while also taking advantage of the rapid improvement of modern answer set solvers. Due to the general and modular nature of our translation and encoding method, future advances in the efficiency of answer set solvers can be easily integrated into our software system, in many cases automatically improving the capabilities of CPLUS2ASP.

Work has already begun on enhancing the translator (`cplus2asp.bin`) so that it supports more features of the original CCALC system. As answer set solvers improve, investigations could be made into the possibility of extending the input language of CPLUS2ASP beyond the capabilities of CCALC, incorporating concepts such as aggregate expressions and nondefinite causal theories into the system. Coupled with recent developments in answer set programming such as incremental grounding with the ICLINGO system (Gebser et al., 2008) and hybrid constraint solving with the CLINGCON system[1], CPLUS2ASP's efficiency could improve even further, especially with respect to solving speed and the sizes and types of domains that can be formalized using the system.

As the overall CPLUS2ASP system improves, investigations into representing new domains and scenarios can be conducted, including topics such as online reasoning and semantic processing. We hope this work will serve as a foundation to increase the visibility and popularity of $\mathcal{C}+$, particularly with those whose specialties lie outside the field of knowledge representation.

[1]`http://www.cs.uni-potsdam.de/clingcon/`

REFERENCES

Akman, V., Erdoğan, S., Lee, J., Lifschitz, V., & Turner, H. (2004). Representing the Zoo World and the Traffic World in the language of the Causal Calculator. *Artificial Intelligence*, *153(1–2)*, 105–140.

Armando, A., Giunchiglia, E., & Ponta, S. E. (2009). Formal specification and automatic analysis of business processes under authorization constraints: an action-based approach. In *Proceedings of the 6th international conference on trust, privacy and security in digital business (trustbus'09).*

Artikis, A., Sergot, M., & Pitt, J. (2009). Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, *9*(1).

Boenn, G., Brain, M., Vos, M. de, & Fitch, J. (2008). Automatic composition of melodic and harmonic music by answer set programming. In M. G. de la Banda & E. Pontelli (Eds.), *Proceedings of the twenty-fourth international conference on logic programming* (Vol. 5366, p. 160-174). Springer-Verlag.

Brain, M., Crick, T., Vos, M. de, & Fitch, J. (2006). Toast: Applying answer set programming to superoptimisation. In *Proceedings of 22nd international conference on logic programming (ICLP)* (p. 270-284). Springer.

Caldiran, O., Haspalamutgil, K., Ok, A., Palaz, C., Erdem, E., & Patoglu, V. (2009). Bridging the gap between high-level reasoning and low-level control. In *Proceedings of international conference on logic programming and nonmonotonic reasoning (LPNMR).*

Chopra, A., & Singh, M. (2003). Nonmonotonic commitment machines. In *Agent communication languages and conversation policies AAMAS 2003 workshop.*

Craven, R., & Sergot, M. (2005). Distant causation in C+. *Studia Logica*, *79*(1), 73–96.

Doğandağ, S., Alpaslan, F. N., & Akman, V. (2001). Using stable model semantics (SMODELS) in the Causal Calculator (CCALC). In *Proceedings 10th Turkish Symposium on Artificial Intelligence and Neural Networks* (pp. 312–321).

Eiter, T., & Lukasiewicz, T. (2003). Probabilistic reasoning about actions in nonmonotonic causal theories. In *Proceedings nineteenth conference on uncertainty in artificial intelligence (UAI-2003)* (pp. 192–199). Morgan Kaufmann Publishers.

Ferraris, P. (2005). Answer sets for propositional theories. In *Proceedings of international conference on logic programming and nonmonotonic reasoning (LPNMR)* (pp. 119–131).

Ferraris, P. (2007). A logic program characterization of causal theories. In *Proceedings of international joint conference on artificial intelligence (IJCAI)* (pp. 366–371).

Ferraris, P., Lee, J., Lierler, Y., Lifschitz, P., & Yang, F. (2010). Representing first-order causal theories by logic programs. *TPLP*. (To appear)

Ferraris, P., Lee, J., & Lifschitz, V. (2011). Stable models and circumscription. *Artificial Intelligence*, *175*, 236–263.

Finger, J. (1986). *Exploiting constraints in design synthesis*. Unpublished doctoral dissertation, Stanford University. (PhD thesis)

Gebser, M., Grote, T., & Schaub, T. (2010). Coala: a compiler from action languages to ASP. In *Proceedings of european conference on logics in artificial intelligence (JELIA)*.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Thiele, S. (2008). Engineering an incremental ASP solver. In M. Garcia de la Banda & E. Pontelli (Eds.), *Proceedings of the twenty-fourth international conference on logic programming (ICLP'08)* (Vol. 5366, p. 190-205). Springer-Verlag.

Gelfond, M., & Inclezan, D. (2009). Yet another modular action language. In *Proceedings of the second international workshop on software engineering for answer set programming*[2] (pp. 64–78).

Gelfond, M., & Inclezan, D. (2010). *Reasoning about dynamic domains in modular action language ALM*[3].

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In R. Kowalski & K. Bowen (Eds.), *Proceedings of international logic programming conference and symposium* (pp. 1070–1080). MIT Press.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., & Turner, H. (2004). Nonmonotonic causal theories. *Artificial Intelligence*, *153(1–2)*, 49–104.

[2]http://www.sea09.cs.bath.ac.uk/downloads/sea09proceedings.pdf

[3]http://www.webpages.ttu.edu/dincleza/alm/alm-technical-report-2010.pdf

Kim, T.-W. (2009). *Implementing and experimenting with answer set programming based event calculus reasoner*. Unpublished master's thesis, Arizona State University.

Kim, T.-W., Lee, J., & Palla, R. (2009). Circumscriptive event calculus as answer set programming. In *Proceedings of international joint conference on artificial intelligence (IJCAI)* (p. 823-829).

Lee, J. (2005). *Automated reasoning about actions*[4]. Unpublished doctoral dissertation, University of Texas at Austin.

Lee, J., Lifschitz, V., & Palla, R. (2008a). A reductive semantics for counting and choice in answer set programming. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)* (pp. 472–479).

Lee, J., Lifschitz, V., & Palla, R. (2008b). Safe formulas in the general theory of stable models (preliminary report). In *Proceedings of international conference on logic programming (ICLP)* (pp. 672–676).

Lee, J., & Palla, R. (2009). System F2LP – computing answer sets of first-order formulas. In *Procedings of international conference on logic programming and nonmonotonic reasoning (LPNMR)* (p. 515-521).

Lee, J., & Palla, R. (2010). Situation calculus as answer set programming. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)* (pp. 309–314).

Lifschitz, V., & Ren, W. (2006). A modular action description language. In *Proceedings of national conference on artificial intelligence (AAAI)* (pp. 853–859).

Liu, X., Ramakrishnan, C. R., & Smolka, S. A. (1998). Fully local and efficient evaluation of alternating fixed points. In *Tools and algorithms for construction and analysis of systems.*

McCain, N. (1997). *Causality in commonsense reasoning about actions*[5]. Unpublished doctoral dissertation, University of Texas at Austin.

McCarthy, J., & Hayes, P. (1969). Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer & D. Michie

[4]http://peace.eas.asu.edu/joolee/papers/dissertation.pdf

[5]ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz

(Eds.), *Machine intelligence* (Vol. 4, pp. 463–502). Edinburgh: Edinburgh University Press.

Reiter, R. (2001). *Knowledge in action: Logical foundations for specifying and implementing dynamical systems*. MIT Press.

Sergot, M., & Craven, R. (2006, July). The deontic component of action language nC+. In *DEON 2006* (Vol. 4048, pp. 222–237).

Shanahan, M. (1995). A circumscriptive calculus of events. *Artif. Intell.*, *77*(2), 249-284.

Son, T., Pontelli, E., & Sakama, C. (2009). Logic programming for multiagent planning with negotiation. In *Proceedings of 25th international conference on logic programming (ICLP)* (p. 99-114). Springer.

APPENDIX A

F2LP STANDARD FILE

```
% Standard description-independent declarations and rules
% that embed CCalc constructs in ASP.

% A derived binary relation between a constant and its domain objects,
% connected via constant_sort and sort_object.
constant_object(V_constant,X_Object) <-
   constant_sort(V_constant,X_Sort) &
   sort_object(X_Sort,X_Object).
%-----------------------------------------------------------
% Description-independent declarations of sorts and objects.

sort(boolean).
#domain boolean(V_boolean).
sort_object(boolean,V_boolean).

boolean(true).
%-----------------------------------------------------------
% Time steps

sort(step).
#domain step(V_step).
sort_object(step,V_step).

sort(astep).
#domain astep(V_astep).
sort_object(astep,V_astep).

% astep is a subsort of step
step(V_astep).

step(0..maxstep).
astep(0..maxstep-1).
%-----------------------------------------------------------
% Constants hierarchy

% Meta-constants to group categories of constants.
sort(constant).
#domain constant(V_constant).
sort_object(constant,V_constant).
%----
% Rigid constants

sort(rigid).
#domain rigid(V_rigid).
sort_object(rigid,V_rigid).
%---
% Fluent-based constants.

sort(fluent).
#domain fluent(V_fluent).
sort_object(fluent,V_fluent).

sort(simpleFluent).
#domain simpleFluent(V_simpleFluent).
sort_object(simpleFluent,V_simpleFluent).

sort(inertialFluent).
#domain inertialFluent(V_inertialFluent).
sort_object(inertialFluent,V_inertialFluent).

sort(sdFluent).
#domain sdFluent(V_sdFluent).
sort_object(sdFluent,V_sdFluent).
```

```
%---
% Action-based constants.

sort(action).
#domain action(V_action).
sort_object(action,V_action).

sort(exogenousAction).
#domain exogenousAction(V_exogenousAction).
sort_object(exogenousAction,V_exogenousAction).

sort(abAction).
#domain abAction(V_abAction).
sort_object(abAction,V_abAction).

sort(attribute).
#domain attribute(V_attribute).
sort_object(attribute,V_attribute).
%---
% Subsort relations.

constant(V_fluent).
constant(V_action).
constant(V_rigid).
fluent(V_simpleFluent).
simpleFluent(V_inertialFluent).
fluent(V_sdFluent).
action(V_exogenousAction).
action(V_abAction).
action(V_attribute).
%------------------------------------------------------------
% Sort declaration: atomic formulas

%---
sort(fluentAtomicFormula).
#domain fluentAtomicFormula(V_fluentAtomicFormula).
sort_object(fluentAtomicFormula,V_fluentAtomicFormula).

sort(simpleFluentAtomicFormula).
#domain simpleFluentAtomicFormula(V_simpleFluentAtomicFormula).
sort_object(simpleFluentAtomicFormula,V_simpleFluentAtomicFormula).

sort(inertialFluentAtomicFormula).
#domain inertialFluentAtomicFormula(V_inertialFluentAtomicFormula).
sort_object(inertialFluentAtomicFormula,V_inertialFluentAtomicFormula).

sort(sdFluentAtomicFormula).
#domain sdFluentAtomicFormula(V_sdFluentAtomicFormula).
sort_object(sdFluentAtomicFormula,V_sdFluentAtomicFormula).

sort(rigidAtomicFormula).
#domain rigidAtomicFormula(V_rigidAtomicFormula).
sort_object(rigidAtomicFormula,V_rigidAtomicFormula).

%---
sort(actionAtomicFormula).
#domain actionAtomicFormula(V_actionAtomicFormula).
sort_object(actionAtomicFormula,V_actionAtomicFormula).

sort(exogenousActionAtomicFormula).
#domain exogenousActionAtomicFormula(V_exogenousActionAtomicFormula).
sort_object(exogenousActionAtomicFormula,V_exogenousActionAtomicFormula).
```

```
sort(abActionAtomicFormula).
#domain abActionAtomicFormula(V_abActionAtomicFormula).
sort_object(abActionAtomicFormula,V_abActionAtomicFormula).

sort(attributeAtomicFormula).
#domain attributeAtomicFormula(V_attributeAtomicFormula).
sort_object(attributeAtomicFormula,V_attributeAtomicFormula).

%---
% Subsort relations.

fluentAtomicFormula(V_simpleFluentAtomicFormula).
simpleFluentAtomicFormula(V_inertialFluentAtomicFormula).
fluentAtomicFormula(V_sdFluentAtomicFormula).
actionAtomicFormula(V_exogenousActionAtomicFormula).
actionAtomicFormula(V_abActionAtomicFormula).
actionAtomicFormula(V_attributeAtomicFormula).
%------------------------------------------------------------
% Object declaration: atomic formulas

rigidAtomicFormula(eql(V_rigid,X_Object)) <-
   constant_object(V_rigid,X_Object).

simpleFluentAtomicFormula(eql(V_simpleFluent,X_Object)) <-
   constant_object(V_simpleFluent,X_Object).

inertialFluentAtomicFormula(eql(V_inertialFluent,X_Object)) <-
   constant_object(V_inertialFluent,X_Object).

sdFluentAtomicFormula(eql(V_sdFluent,X_Object)) <-
   constant_object(V_sdFluent,X_Object).

actionAtomicFormula(eql(V_action,X_Object)) <-
   constant_object(V_action,X_Object).

exogenousActionAtomicFormula(eql(V_exogenousAction,X_Object)) <-
   constant_object(V_exogenousAction,X_Object).

abActionAtomicFormula(eql(V_abAction,X_Object)) <-
   constant_object(V_abAction,X_Object).

attributeAtomicFormula(eql(V_attribute,X_Object)) <-
   constant_object(V_attribute,X_Object).

%------------------------------------------------------------
% Description-independent rules to encode common CCalc constructs.

% Exogeneity for exogenous actions.
{h(V_exogenousActionAtomicFormula,V_astep)}.
% Negative version for Booleans.
{-h(eql(V_exogenousAction,true),V_astep)} <-
   constant_sort(V_exogenousAction,boolean).

% abActions default to false.
{-h(eql(V_abAction,true),V_astep)}.

% Exogeneity for attributes.
{h(V_attributeAtomicFormula,V_astep)}.
```

```
% Restriction that attributes will take on the value "none"
% if and only if their linked action does not execute.
false <-
   not (( h(eql(V_attribute,none),V_astep) -> -h(eql(V_action,true),V_astep) ) &
        ( -h(eql(V_action,true),V_astep) -> h(eql(V_attribute,none),V_astep) )) &
   action_attribute(V_action,V_attribute).

% Inertia for inertial fluents
{h(V_inertialFluentAtomicFormula,V_astep+1)} <-
   h(V_inertialFluentAtomicFormula,V_astep).
% Negative version for Booleans.
{-h(eql(V_inertialFluent,true),V_astep+1)} <-
   -h(eql(V_inertialFluent,true),V_astep) &
   constant_sort(V_inertialFluent,boolean).

%------------------------------------------------------------
% Exogeneity for simple fluents at time 0.

{h(V_simpleFluentAtomicFormula,0)}.
% Negative version for Booleans.
{-h(eql(V_simpleFluent,true),0)} <-
   constant_sort(V_simpleFluent,boolean).

%------------------------------------------------------------
% exogenous: Grants exogeneity to a constant.

% Rigids

{h(eql(V_rigid,X_Object))} <-
   exogenous(V_rigid) &
   constant_object(V_rigid,X_Object).
% Negative version if it's Boolean.
{-h(eql(V_rigid,true))} <-
   exogenous(V_rigid) &
   constant_sort(V_rigid,boolean).

% Fluents

{h(eql(V_fluent,X_Object),V_step)} <-
   exogenous(V_fluent) &
   constant_object(V_fluent,X_Object).
% Negative version if it's Boolean.
{-h(eql(V_fluent,true),V_step)} <-
   exogenous(V_fluent) &
   constant_sort(V_fluent,boolean).

% Actions

{h(eql(V_action,X_Object),V_astep)} <-
   exogenous(V_action) &
   constant_object(V_action,X_Object).
% Negative version if it's Boolean.
{-h(eql(V_action,true),V_astep)} <-
   exogenous(V_action) &
   constant_sort(V_action,boolean).

%------------------------------------------------------------
% inertial: Grants inertia to a (non-rigid) fluent.

{h(eql(V_fluent,X_Object),V_astep+1)} <-
   inertial(V_fluent) &
   h(eql(V_fluent,X_Object),V_astep) &
   constant_object(V_fluent,X_Object).
```

```
% Negative version if it's Boolean.
{-h(eql(V_fluent,true),V_astep+1)} <-
   inertial(V_fluent) &
   -h(eql(V_fluent,true),V_astep) &
   constant_sort(V_fluent,boolean).

%------------------------------------------------------------
% noconcurrency: If stated as a fact, prevents
% concurrent execution of Boolean actions.

false <-
   noconcurrency &
   action(V_action_1) &
   not (h(eql(V_action,true),V_astep) &
        h(eql(V_action_1,true),V_astep)
        -> V_action=V_action_1).

%------------------------------------------------------------
% Existence and uniqueness for every constant relative to its domain.

% Rigids

-h(eql(V_rigid,X_Object_1)) <-
   h(eql(V_rigid,X_Object)) &
   constant_object(V_rigid,X_Object) &
   constant_object(V_rigid,X_Object_1) &
   X_Object != X_Object_1 &
   not constant_sort(V_rigid,boolean).

% Fluents

-h(eql(V_fluent,X_Object_1),V_step) <-
   h(eql(V_fluent,X_Object),V_step) &
   constant_object(V_fluent,X_Object) &
   constant_object(V_fluent,X_Object_1) &
   X_Object != X_Object_1 &
   not constant_sort(V_fluent,boolean).

% Actions

-h(eql(V_action,X_Object_1),V_astep) <-
   h(eql(V_action,X_Object),V_astep) &
   constant_object(V_action,X_Object) &
   constant_object(V_action,X_Object_1) &
   X_Object != X_Object_1 &
   not constant_sort(V_action,boolean).

% Only complete interpretations allowed.

false <-
   {h(V_rigidAtomicFormula),
    -h(V_rigidAtomicFormula)}0.

false <-
   {h(V_fluentAtomicFormula,V_step),
    -h(V_fluentAtomicFormula,V_step)}0.

false <-
   {h(V_actionAtomicFormula,V_astep),
    -h(V_actionAtomicFormula,V_astep)}0.
```

```
%-----------------------------------------------------------
% Hide most of the internal predicates to avoid cluttering the answer sets.

#hide sort(step).
#hide sort(astep).
#hide sort(boolean).
#hide step/1.
#hide astep/1.
#hide boolean/1.

#hide sort(constant).
#hide sort(fluent).
#hide sort(action).
#hide sort(abAction).
#hide sort(attribute).
#hide sort(exogenousAction).
#hide sort(inertialFluent).
#hide sort(rigid).
#hide sort(sdFluent).
#hide sort(simpleFluent).
#hide constant/1.
#hide sort_object/2.
#hide constant_object/2.

#hide sort(actionAtomicFormula).
#hide sort(fluentAtomicFormula).
#hide sort(abActionAtomicFormula).
#hide sort(attributeAtomicFormula).
#hide sort(exogenousActionAtomicFormula).
#hide sort(inertialFluentAtomicFormula).
#hide sort(rigidAtomicFormula).
#hide sort(simpleFluentAtomicFormula).
#hide sort(sdFluentAtomicFormula).
#hide actionAtomicFormula/1.
#hide fluentAtomicFormula/1.
#hide abActionAtomicFormula/1.
#hide attributeAtomicFormula/1.
#hide exogenousActionAtomicFormula/1.
#hide inertialFluentAtomicFormula/1.
#hide rigidAtomicFormula/1.
#hide sdFluentAtomicFormula/1.
#hide simpleFluentAtomicFormula/1.
```