Representing the Language of the Causal Calculator in Answer Set Programming

Michael Casolary and Joohyung Lee

School of Computing, Informatics and Decision Systems Engineering
Arizona State University, Tempe, AZ, USA
`Michael.Casolary@asu.edu, joolee@asu.edu`

Abstract

Action language $\mathcal{C}+$, a formalism based on nonmonotonic causal logic, was designed for describing properties of actions. The definite fragment of $\mathcal{C}+$ was implemented in system the Causal Calculator (CCALC), based on a reduction of nonmonotonic causal logic to propositional logic. On the other hand, in this paper, we represent the language of CCALC in answer set programming (ASP), by translating nonmonotonic causal logic into formulas under the stable model semantics. We design 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, our prototype implementation of this approach, which we call CPLUS2ASP, achieves functionality close to CCALC while taking advantage of answer set solvers to yield efficient computation that is orders of magnitude faster than CCALC on several benchmark examples.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases answer set programming, nonmonotonic causal logic, action languages

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.51

1 Introduction

Action languages are formal models of parts of natural language that are used for describing properties of actions. Among them, language $\mathcal{C}+$ [9] and its predecessor $\mathcal{C}$ [10] are based on nonmonotonic causal logic. The definite fragment of nonmonotonic causal logic can be turned into propositional logic by the literal completion method, which resulted in an efficient way to compute $\mathcal{C}+$ using satisfiability (SAT) solvers. The Causal Calculator (CCALC) is an implementation of this idea. Version 1 of CCALC was created by McCain [16], accepting $\mathcal{C}$ as its input language; Version 2 is an enhancement described in [11], which accepts $\mathcal{C}+$ as its input language. Language $\mathcal{C}+$ is significantly more enhanced than $\mathcal{C}$ in several ways, such as being able to represent multi-valued formulas, defined fluents, additive fluents, rigid constants and defeasible causal laws. Although CCALC was not aimed at large scale applications, it has been applied to several challenging commonsense reasoning problems, including problems of nontrivial size [1], to provide a group of robots with high-level reasoning [4], to give executable specifications of norm-governed computational societies [3], and to automate the analysis of business processes under authorization constraints [2].

An alternative way to compute $\mathcal{C}+$ is to turn it into answer set programs and to use existing answer set solvers. This can be achieved by first turning multi-valued causal logic into Boolean-valued causal logic as described in [11] and then turning the latter into answer set programs as described in [16, 5, 15]. In fact, a system called COALA (*Co*mpiler for *a*ction *la*nguages) was implemented based on this idea [8]. The system turns a fragment of language

© Michael Casolary and Joohyung Lee;
licensed under Creative Commons License NC-ND

$\mathcal{C}+$ into the input language of GRINGO[1], but that fragment lacks several important features of $\mathcal{C}+$ mentioned above, which are available in CCALC.

In this paper, we provide a way to encode CCALC input language in answer set programming, and present a prototype implementation called CPLUS2ASP based on this idea. Our approach differs from that of COALA in a few ways. First, CPLUS2ASP can handle multi-valued constants, which COALA does not allow. Second, we turn the language of CCALC into formulas under the stable model semantics [7], and use system F2LP[2] ("*f*ormulas *to* *l*ogic *p*rograms") [14] to turn them into the input language of ASP solvers. This allows users to write the same complex formulas as in the input language of CCALC. Third, we design a standard library that defines the constructs of the CCALC input language in terms of the language of GRINGO. Using the standard library and reusing the existing software F2LP and CLINGO[3] allowed a simple design of CPLUS2ASP that achieves functionality close to CCALC. Also thanks to the efficiency of answer set solvers, our experiments show that CPLUS2ASP is orders of magnitude faster than CCALC in several benchmark examples.

The paper is organized as follows. Section 2 provides preliminaries, and Section 3 shows how to encode the language of CCALC in ASP, and presents the prototype implementation CPLUS2ASP. We compare the efficiency of CPLUS2ASP against that of CCALC in Section 4.

2 Preliminaries

2.1 Nonmonotonic Causal Theories and $\mathcal{C}+$

Due to lack of space, the reviews in this section are rather dense. We refer the reader to [9] for the details. In $\mathcal{C}+$, formulas are multi-valued. A *(multi-valued propositional) signature* is a set σ of symbols called *constants*, along with a nonempty finite set $Dom(c)$ of symbols called the *domain* of c. An *atom* of a signature σ is an expression of the form $c\!=\!v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *(multi-valued) formula* of σ is a propositional combination of atoms. A *causal rule* is an expression of the form

$$F \Leftarrow G$$

where F and G are multi-valued propositional formulas. A *causal theory* is a set of causal rules.

Language $\mathcal{C}+$ is a high level notation for causal theories that was designed for describing transition systems—directed graphs whose vertices represent states and edges are labeled by actions that affect the states. In $\mathcal{C}+$, constants are partitioned into *fluent* constants and *action* constants. Fluent constants are further partitioned into *simple* and *statically determined* fluents. A *fluent formula* is a formula where all constants occurring in it are fluent constants. An *action formula* is a formula that contains at least one action constant and no fluent constants. A *static law* is an expression of the form

$$\textbf{caused } F \textbf{ if } G \tag{1}$$

where F and G are fluent formulas. An *action dynamic law* is an expression of the form (1) in which F is an action formula and G is a formula. A *fluent dynamic law* is an expression

[1] `http://potassco.sourceforge.net`
[2] `http://reasoning.eas.asu.edu/f2lp`
[3] CLINGO is a system that combines GRINGO and CLASP in a monolithic way, available from the same link as the one in Footnote 1.

of the form

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \qquad (2)$$

where F and G are fluent formulas and H is a formula, provided that F does not contain statically determined constants. A *causal law* is a static law, or an action dynamic law, or a fluent dynamic law. An *action description* is a set of causal laws.

The semantics of $\mathcal{C}+$ in [9] is described via a translation into causal logic. For any action description D and any nonnegative integer m, the causal theory D_m is defined as follows. The signature of D_m consists of the pairs $i\!:\!c$ such that

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

The domain of $i\!:\!c$ is the same as the domain of c. By $i\!:\!F$ we 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. The rules of D_m are

$$i\!:\!F \Leftarrow i\!:\!G \qquad (3)$$

for every static law (1) in D and every $i \in \{0, \dots, m\}$, and for every action dynamic law (1) in D and every $i \in \{0, \dots, m-1\}$;

$$i{+}1\!:\!F \Leftarrow (i{+}1\!:\!G) \wedge (i\!:\!H) \qquad (4)$$

for every fluent dynamic law (2) in D and every $i \in \{0, \dots, m-1\}$;

$$0\!:\!c{=}v \Leftarrow 0\!:\!c{=}v \qquad (5)$$

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

The causal models of D_m correspond to the paths of length m in the transition system described by D.

2.2 Language of the Causal Calculator

The language of CCalc provides a convenient way of expressing $\mathcal{C}+$ descriptions. It allows us to declare `sorts`, `objects`, `variables` and `constants`, as well as to describe causal laws. A causal law may contain variables, which are understood in terms of grounding. Such causal laws are *schemas* for ground instances, as in answer set programming.

The left column of Figure 1 is a simple $\mathcal{C}+$ description in the language of CCalc. The symbol `>>` in the sort declaration between the names of two sorts expresses that the second is a subsort of the first, so that every object that belongs to the second sort belongs also to the first. Lines 1–2 declare that `s_num` is a subsort of `num`. Lines 11–13 introduce objects of the two sorts. The integers from 0 to `n` − 1 belong to sort `s_num`; the integers from 0 to `n` belong to sort `num`. Line 19 declares that `has` is an inertial fluent whose domain is `num`. CCalc understands this line the same as the declaration

```
has   :: simpleFluent(num)
```

followed by the fluent dynamic law

```
caused has=X if has=X after has=X
```

where `X` ranges over all objects of sort `num`. Similarly, Line 22 declares that `buy` is an exogenous action with Boolean values. Lines 32–35 represent a simple query for finding a path of length 3 from the initial state where `has=2` to the goal state where `has=4`.

2.3 Stable Model Semantics of First-Order Formulas and System F2LP

We refer the reader to [7, 14] for the details. The stable model semantics from [7] is defined for first-order formulas, which allow for arbitrary nesting of connectives and quantifiers as in first-order logic. Strong negation ($\sim$) occurs only in front of an atom. For instance,

$$\neg \sim p(x) \rightarrow p(x) \tag{6}$$

expresses that x belongs to p by default.

System F2LP can be used to turn any "almost universal sentence" into an answer set program so that answer set solvers can be used to compute the Herbrand stable models of the almost universal sentence. As far as this paper is concerned, it is sufficient to know that any sentence where every quantifier is in the scope of negation is almost universal. (6) can be encoded in the language of F2LP as

```
p(X) <- not -p(X).
```

(In the language of F2LP, the default negation ($\neg$) is expressed as `not`; the strong negation ($\sim$) is encoded as `-`, following the convention in the input language of ASP solvers. In addition, `?` and `!` denote the existential and universal quantifiers, respectively; `|` denotes disjunction and `&` denotes conjunction.) The input language of F2LP also allows aggregates and choice rules as in the language of GRINGO.

```
:- sorts
  num >> s_num.                              sort(num).
                                             #domain num(V_num).
                                             sort_object(num,V_num).

                                             sort(s_num).
                                             #domain s_num(V_s_num).
                                             sort_object(num,V_s_num).

                                             num(V_s_num).
:- objects
  0..n-1   :: s_num;                         s_num(0..n-1).
  n        :: num.                           num(n).

:- variables
  K        :: s_num.                         #domain s_num(K).

:- constants
  has :: inertialFluent(num);                inertialFluent(has).
                                             constant_sort(has,num).

  buy :: exogenousAction(boolean).           exogenousAction(buy).
                                             constant_sort(buy,boolean).

buy causes has=K+1 if has=K.                 h(eql(has,K+1),V_astep+1) <-
                                                h(eql(buy,true),V_astep) &
                                                h(eql(has,K),V_astep).

nonexecutable buy if has=n.                  false <-
                                                h(eql(buy,true),V_astep) &
                                                h(eql(has,n),V_astep).
:- query
maxstep :: 3;                                false <- query_label(0) &
0: has=2;                                             not (h(eql(has,2),0) &
maxstep: has=4.                                            h(eql(has,4),maxstep)).
```

Figure 1 Simple Transition System in the Language of CCALC and in the Language of F2LP

3 Representing the Language of the Causal Calculator in ASP

3.1 Translating $\mathcal{C}+$ into Answer Set Programs

We consider a finite definite $\mathcal{C}+$ description D of signature σ, where the heads of the rules are 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 following these steps: (i) turn D into the corresponding multi-valued causal theory D_m (as explained in Section 2.1); (ii) turn D_m into a Boolean-valued causal theory D'_m; (iii) turn D'_m into formulas under the stable model semantics; (iv) turn the result further into a logic program (using F2LP as explained in Section 2.3). In this section we explain Steps (ii) and (iii).

Definite Elimination of Multi-Valued Constants Consider the causal theory D_m with signature σ_m consisting of rules of the form (3), (4) and (5). Consider all constants $i:c$ $(0 \le i \le m)$ in σ_m, where c is a fluent constant of D. By σ^c_m we denote the signature obtained from σ_m by replacing every constant $i:c$ with Boolean constants $i:eql(c,v)$ for all $v \in Dom(c)$.

The causal theory D^c_m with signature σ^c_m is obtained from D_m by replacing each occurrence of an atom $i : c = v$ in D_m with $i : eql(c,v) = \mathbf{t}$, and adding the causal rules

$$i:eql(c,v')=\mathbf{f} \Leftarrow i:eql(c,v)=\mathbf{t} \qquad (0 \le i \le m) \tag{7}$$

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

The following proposition is a simplification of Proposition 9 from [11].[4]

▶ Proposition 1. There is a 1-1 correspondence between the models of D_m and the models of D^c_m.

The elimination of multi-valued action constants is similar.

Turning Boolean-valued Action Descriptions into SM In [6], McCain's translation is modified and extended as follows. Take any set T of causal rules of the forms

$$A \Leftarrow G, \tag{8}$$

$$\neg A \Leftarrow G, \tag{9}$$

$$\bot \Leftarrow G, \tag{10}$$

where A is an atom and G is an arbitrary propositional formula. For each rule (8), take the formula $\neg\neg G \rightarrow A$; for each rule (9), the formula $\neg\neg G \rightarrow \sim A$; for each rule (10), the formula $\neg G$. Also add the following completeness constraints for all atoms A:

$$\neg A \wedge \neg{\sim}A \rightarrow \bot\ . \tag{11}$$

Note that, for T, which is definite, the modified McCain translation yields a first-order theory that is tight [7].

Consider D'_m 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

[4] Proposition 9 from [11] involves adding two kinds of rules. Vladimir Lifschitz pointed out that one kind of rules can be dropped if the given theory is definite.

occurrence of $i : eql(c,v)=\mathbf{t}$ in it with $h(eql(c,v),i)$ and every occurrence of $i : eql(c,v)=\mathbf{f}$ with $\sim h(eql(c,v),i)$. According to the modified McCain translation, the causal rules (3) that represent a static law (1) are represented by formulas under the stable model semantics as

$$h(i:F) \leftarrow \neg\neg h(i:G) \tag{12}$$

($i \in \{0,\dots,m\}$). The translation of causal rules for an action dynamic law is similar except that i ranges over $\{0,\dots,m-1\}$. In the special case when $h(i:F)$ and $h(i:G)$ are the same literal, (12) can be represented using choice rules in ASP:

$$\{h(i{:}F)\}. \tag{13}$$

This is because (12) is strongly equivalent to $h(i{:}F) \vee \neg h(i{:}F)$, which can be abbreviated as (13) [12]. In fact, we observe in many cases (13) can be used in place of (12).

Similarly, the modified McCain translation turns the causal rules (4) that correspond to a fluent dynamic law (2) into

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

In fact, we can also turn (4) into

$$h(i{+}1{:}F) \;\leftarrow\; \neg\neg h(i{+}1{:}G) \;\wedge\; h(i{:}H) \tag{14}$$

because the change does not affect the stable models of the resulting theory, which is tight [7]. Similarly, certain occurrences of $\neg\neg$ in (12) and (14) can be further dropped if removing them does not cause the resulting theory to become non-tight.

Again in the special case when $h(i{+}1{:}F)$ and $h(i{+}1{:}G)$ are the same literal, (14) can be represented using choice rules as follows:

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

3.2 Representing Domain Descriptions in the Language of F2LP

Figure 1 shows a side-by-side comparison of an example CCALC input program and its representation in the language of F2LP. As shown, the translation is modular. For each sort name S that is declared in the CCALC input program, the translation introduces a fact `sort`(S) and a variable V_S that ranges over all objects of sort S by the line #`domain` $S(V_S)$, and relates the sort name and the objects by the fact `sort_object`(S, V_S). (This "meta predicate," together with another meta predicate `constant_sort` that is shown later, is used in the standard library to associate constants with their domains.) As an example, Lines 2–8 in the right column of Figure 1 is a representation of sort declarations in the language of GRINGO. In addition, 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 example.

A constant declaration in the language of CCALC of the form

$$C :: \mathit{CompositeSort}(V)$$

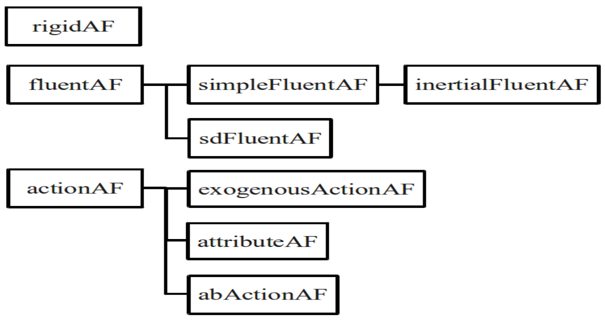

Figure 2 The Hierarchy of Atomic Formulas

is turned into a fact *CompositeSort*(C), followed by another fact `constant_sort`(C, V), which is used in the standard library. See Lines 19–23 for example.

Encoding causal laws in the language of F2LP follows the method in Section 3.1. Like the input language of CCALC, the variables in the F2LP rules are understood as schemas for ground terms. Lines 25–31 are example encodings of causal laws in the language of F2LP. Since every variable is sorted, these F2LP rules are safe according to the definition of safety in [13], and its translation into an ASP program also results in a safe logic program.

Note that Figure 1 does not contain the other causal laws (the inertial assumption for `has` and the exogeneity assumption for `buy`). Since such causal laws and rules are frequently used, they are described in the standard library, which we explain in the next section.

3.3 Standard Library

The standard library[5] declares built-in sorts and objects, such as sort `boolean` and its objects `true` and `false`; sorts `step` and `astep` and the integer objects that belong to the sorts ($0, \dots,$ `maxstep` for `step` and $0, \dots,$ `maxstep` $- 1$ for `astep`). More importantly, it contains postulates specific to each kind of fluents and actions.

3.3.1 Postulates for Specific Fluents and Actions

First, we assume the presence of certain meta-variables that are used in the postulates. For instance, `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 the domain description in Figure 1, `V_inertialFluentAF` ranges over `eql(has, 0)`, `eql(has, 1)`, $\dots$, `eql(has, n)`. Similarly, we have other meta-variables `V_fluentAF`, `V_simpleFluentAF`, `V_sdFluentAF`, `V_rigidAF`, `V_actionAF`, `V_exogenousActionAF`, `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.

- The inertial assumption for `inertialFluents` is represented by

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

[5] See `http://reasoning.eas.asu.edu/cplus2asp` for the complete file.

- The exogeneity assumption (5) for simple fluents at time 0 is represented by

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

- The exogeneity assumption for `exogenousAction` is stated as

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

- The completeness assumption (11) 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 (7) 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).
  ```

  (Recall that `sort_object` is introduced in translating `sort` declarations from the domain description and `constant_sort` is introduced in translating `constant` declarations from the domain description.)
  The definite elimination rules and the completeness assumptions for action constants are similar to those for fluent constants.

3.4 Meta-Sorts and Meta-Variables

In order to make grounding replace all meta-variables with the corresponding ground terms as intended in the previous section, we first introduce meta-level sorts for representing the constant hierarchy in $\mathcal{C}+$. This is done in the same way as the object-level (user-defined) sorts are introduced. For instance, the following are sort declarations for `simpleFluent` and `inertialFluent`, and 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 1, the constant declarations included the fact `inertialFluent(has)`; the variable `V_simpleFluent` ranges over all simple fluent constants, including the inertial fluent `has`.

Similarly, we introduce meta-level sorts for different kinds of atomic formulas depending on the different kinds of constants. For instance, the following is a part of the declaration for `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 *domain predicates* $Constant$`AF`s which contain terms 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 `inertialFluentAF` is a domain predicate that contains all ground terms of the form `eql`(c, v) where c is an `inertialFluent`, and v is a value in the domain of c, using the meta-predicate `constant_object`.

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

(Recall the definition of `constant_object` in the previous section.) The grounding process replaces the meta-variable `V_inertialFluentAF` by every ground term of the form `eql`(c, v) where c is a constant of meta-level sort `inertialFluent` and v is an element in the domain of c, as specified by the `constant_object` relation. So once the user declares that c is an `inertialFluent` (or one of its subsorts) in the domain description, the postulates for the inertial assumption for c are generated by ASP grounders.

4 Implementation and Experiments

We implemented the techniques described in Section 3 in a prototype implementation, which we call CPLUS2ASP. The system achieves functionality close to CCALC by using the standard library and the combination of the existing software F2LP and CLINGO. As documented in Figure 3, the system turns the CCALC domain description into the language of F2LP, calls F2LP to turn it into the language of CLINGO, calls CLINGO to find answer sets, and displays them in CCALC-style solutions (`as2transition` is a post-processor that takes answer sets and transforms them into a format of CCALC output.) CPLUS2ASP is designed to be compatible with the input language of CCALC. It supports most of the basic features of CCALC, but does not yet handle features like user-defined macros, `where` clauses, and shifting. The system was written in C++, utilizing the tools `flex` and `bison` to aid in the creation of a CCALC language grammar and syntax parser.

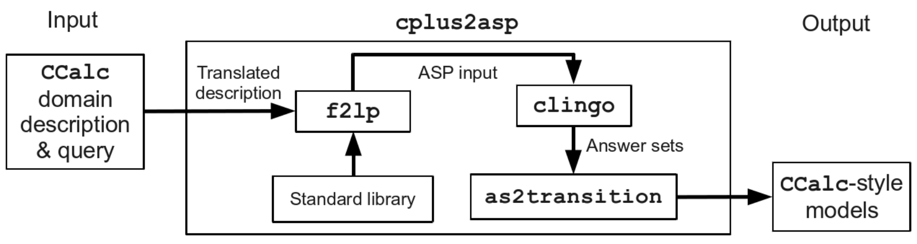

Figure 3 The Architecture of CPLUS2ASP

Problem	CCALC with ZCHAFF			CPLUS2ASP with CLINGO		
	Total	Preparation[a]	Solving	Total	Preparation[b]	Solving[c]
Traffic World: Scenario 1 (maxstep=5)	1.55s	1.52s (1.06s + 0.29s + 0.17s)	0.03s	0.22s	0.20s (0.11s + 0.09s)	0.02s (0.02s + 0.00s)
Traffic World: Scenario 2 (maxstep=3)	22.74s	22.38s (17.85s + 3.45s + 1.08s)	0.36s	1.42s	1.14s (0.11s + 1.03s)	0.28s (0.28s + 0.00s)
Traffic World: Scenario 3 (maxstep=5)	6.29s	6.05s (4.48s + 0.99s + 0.58s)	0.24s	0.52s	0.40s (0.08s + 0.32s)	0.12s (0.12s + 0.00s)
Traffic World: Scenario 3-1 (maxstep=11)	608.76s	558.46s (415.06s + 85.45s + 57.95s)	50.30s	59.84s	34.42s (0.08s + 33.34s)	25.42s (17.95s + 7.47s)

[a] (grounding time + completion time + shifting & writing input clause time)
[b] (CPLUS2ASP processing time + CLINGO grounding time)
[c] (CLINGO pre-processing time + solving time)

Figure 4 Experiments with CCALC and CPLUS2ASP

We ran both CCALC and CPLUS2ASP on a series of benchmark problems designed to utilize a variety of CCALC syntactic elements and tested the speed of each program with respect to grounding and solving. These tests included all examples from [9] along with specific versions of the larger domains described in [1]. Figure 4 shows the performance comparison of CCALC and CPLUS2ASP on the Traffic World domain from [1] using the same scenarios described there, plus one more that is a scaled-up version of Scenario 3. All tests were run in a native install of Ubuntu on a machine with a 3.2 GHz Pentium 4 processor and 2 GB of RAM. Overall CPLUS2ASP consistently performs significantly faster than CCALC, producing identical solutions to those of CCALC. The preparation times that are spent for CPLUS2ASP in producing the input to CLINGO are negligible, as they do not involve grounding.

5 Conclusion

Based on the theoretical result that turns nonmonotonic causal logic into the stable model semantics, we presented a method that represents the language of the Causal Calculator in answer set programming, and implemented it in a prototype called CPLUS2ASP. In comparison with COALA, CPLUS2ASP allows the full expressivity of action language $\mathcal{C}+$ using input language syntax that is almost identical to the language of CCALC. Our ongoing work involves making CPLUS2ASP fully compatible with CCALC by implementing the remaining features of CCALC missing in CPLUS2ASP.

Acknowledgements: We are grateful to Michael Bartholomew, Vladimir Lifschitz, Yunsong Meng, Ravi Palla, and anonymous referees for their useful comments on this paper. The authors were partially supported by the National Science Foundation under Grant IIS-0916116 and by the IARPA SCIL program.

References

1 Varol Akman, Selim Erdoğan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the Zoo World and the Traffic World in the language of the Causal Calculator. *Artificial Intelligence*, 153(1–2):105–140, 2004.

2 Alessandro Armando, Enrico Giunchiglia, and Serena Elisa Ponta. 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)*, 2009.

3 Alexander Artikis, Marek Sergot, and Jeremy Pitt. Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, 9(1), 2009.

4 Ozan Caldiran, Kadir Haspalamutgil, Abdullah Ok, Can Palaz, Esra Erdem, and Volkan Patoglu. Bridging the gap between high-level reasoning and low-level control. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2009.

5 Paolo Ferraris. A logic program characterization of causal theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 366–371, 2007.

6 Paolo Ferraris, Joohyung Lee, Yuliya Lierler, Paolo Lifschitz, and Fangkai Yang. Representing first-order causal theories by logic programs. *TPLP*, 2010. To appear.

7 Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.

8 Martin Gebser, Torsten Grote, and Torsten Schaub. Coala: A compiler from action languages to asp. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, pages 360–364, 2010.

9 Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

10 Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 623–630. AAAI Press, 1998.

11 Joohyung Lee. *Automated Reasoning about Actions*[6]. PhD thesis, University of Texas at Austin, 2005.

12 Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. A reductive semantics for counting and choice in answer set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 472–479, 2008.

13 Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Safe formulas in the general theory of stable models (preliminary report). In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 672–676, 2008.

14 Joohyung Lee and Ravi Palla. System F2LP – computing answer sets of first-order formulas. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 515–521, 2009.

15 Vladimir Lifschitz and Fangkai Yang. Translating first-order causal theories into answer set programming. In *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)*, 2010.

16 Norman McCain. *Causality in Commonsense Reasoning about Actions*[7]. PhD thesis, University of Texas at Austin, 1997.

[6] `http://peace.eas.asu.edu/joolee/papers/dissertation.pdf`
[7] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.gz`