CHAPTER

15 Commonsense Reasoning Using Answer Set Programming

Answer set programming is an approach to knowledge representation and reasoning. Knowledge is represented as *answer set programs*, and reasoning is performed by *answer set solvers*. Answer set programming enables default reasoning, which is required in commonsense reasoning. It supports event calculus reasoning and handles some types of event calculus formulas whose circumscription cannot be computed using predicate completion, including effect constraints, disjunctive event axioms, and definitions of compound events.

In this chapter, we discuss the use of answer set programming for commonsense reasoning. We first describe the syntax and semantics of answer set programs. We then present the theory and practice of answer set programming for event calculus reasoning. Finally, we discuss two useful tools for answer set programming: the F2LP program and the $\mathcal{E}$ language. Resources for this chapter are available at `decreasoner.sourceforge.net/csr/ecasp/`.

15.1 ANSWER SET PROGRAMMING

The syntax of answer set programs derives from the Prolog language. We use the syntax of the ASP-Core-2 standard. The semantics of answer set programs is defined by the *stable model semantics* introduced by Michael Gelfond and Vladimir Lifschitz.

An answer set program consists of a set of *rules* of the form

$$\alpha \texttt{ :- } \beta\texttt{.}$$

which represents that α, the *head* of the rule, is true if β, the *body* of the rule, is true. Here is an answer set program:

```
p.
r :- p, not q.
```

The first rule

```
p.
```

is called a *fact*. It has an empty body and is written without the `:-` (if) connective. The symbol `,` indicates conjunction ($\wedge$). The token `not` refers to *negation as failure* and is different from classical negation ($\neg$). The expression `not q` represents that `q` is not found to be true.

We can perform automated reasoning on this program by placing it in a file `example.lp` and running the answer set grounder and solver `clingo` on the file.

```
$ clingo -n 0 example.lp
clingo version 4.2.1
Reading from example.lp
Solving...
Answer: 1
p r
SATISFIABLE

Models       : 1
Calls        : 1
Time         : 0.001s
CPU Time     : 0.000s
$
```

The command-line option `-n 0` requests all stable models. `Models` refers to stable models, which are also called *answer sets*. The program has one `Answer` or stable model in which `p` and `r` are true.

If we add the following to the program:

```
s.
q :- s.
```

then `clingo` produces

```
Answer: 1
p s q
```

Notice that `r` is no longer found to be true, because `q` is found to be true.

We can also write rules with disjunction in the head:

```
r | q :- p.
```

(The `DLV` answer set grounder and solver uses `v` instead of `|`.) Given this rule and

```
p.
```

`clingo` produces two stable models:

```
Answer: 1
p r
Answer: 2
p q
```

A rule with an empty head is called a *constraint*. If we run `clingo` on the previous program and the constraint

```
:- r.
```

then we only get one stable model

```
Answer: 1
p q
```

The `clingo` program is a combination of the answer set grounder `gringo` and the answer set solver `clasp`. A grounder converts an answer set program containing variables into an equivalent program not containing any variables. It operates by replacing variables with ground terms. For example, if we ground

```
p(a,b).
q(X,Y) :- p(X,Y).
```

we get

```
p(a,b).
q(a,a) :- p(a,a).
q(a,b) :- p(a,b).
q(b,a) :- p(b,a).
q(b,b) :- p(b,b).
```

15.1.1 SYNTAX OF ANSWER SET PROGRAMS

The *syntax* of answer set programs is defined as follows.
Definition 15.1. A ***signature*** σ consists of the following disjoint sets:

- A set of ***constants***.
- For every $n \in \{1, 2, 3, \ldots\}$, a set of n-ary ***function symbols***.
- For every $n \in \{0, 1, 2, \ldots\}$, a set of n-ary ***predicate symbols***.

Given a signature σ and a set of ***variables*** disjoint from the signature, we define answer set programs as follows.
Definition 15.2. A ***term*** is defined inductively as follows:

- A constant is a term.
- A variable is a term.
- If τ_1 and τ_2 are terms, then `-` τ, τ_1 `+` τ_2, τ_1 `-` τ_2, τ_1 `*` τ_2, and τ_1 `/` τ_2 are terms. The symbols `+`, `-`, `*`, and `/` are the ***arithmetic symbols***.
- If ϕ is an n-ary function symbol and $\tau_1, \ldots, \tau_n$ are terms, then ϕ`(`τ_1`, ...,` τ_n`)` is a term.
- Nothing else is a term.

Definition 15.3. A ***ground term*** is a term containing no variables and no arithmetic symbols.
Definition 15.4. An ***atom*** is defined inductively as follows:

- If ρ is an n-ary predicate symbol and $\tau_1, \ldots, \tau_n$ are terms, then ρ`(`τ_1`, ...,` τ_n`)` is an atom.
- If ρ is a 0-ary predicate symbol, then ρ is an atom.
- If τ_1 and τ_2 are terms, then τ_1 `<` τ_2, τ_1 `<=` τ_2, τ_1 `=` τ_2, τ_1 `!=` τ_2, τ_1 `>` τ_2, and τ_1 `>=` τ_2 are atoms. The symbols `<`, `<=`, `=`, `!=`, `>`, and `>=` are the ***comparative predicates***.
- Nothing else is an atom.

Definition 15.5. A ***ground atom*** is an atom containing no variables, no arithmetic symbols, and no comparative predicates.
Definition 15.6. A ***rule*** is

$$\alpha_1 \mid \ldots \mid \alpha_k \texttt{ :- } \beta_1\texttt{ , }\ldots\texttt{ , }\beta_m\texttt{ , not }\gamma_1\texttt{ , }\ldots\texttt{ , not }\gamma_n\texttt{ .}$$

where $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n$ are atoms. $\alpha_1 \mid \ldots \mid \alpha_k$ is the ***head*** of the rule, and $\beta_1\texttt{ , }\ldots\texttt{ , }\beta_m\texttt{ , not }\gamma_1\texttt{ , }\ldots\texttt{ , not }\gamma_n$ is its ***body***.
Definition 15.7. A ***fact*** is a rule whose body is empty ($m = 0$ and $n = 0$).
Definition 15.8. A ***constraint*** is a rule whose head is empty ($k = 0$).
Definition 15.9. A ***ground rule*** is a rule containing no variables, no arithmetic symbols, and no comparative predicates.
Definition 15.10. A ***logic program***, ***answer set program***, or ***program*** is a set of rules.
Definition 15.11. A ***traditional rule*** is a rule whose head contains a single atom ($k = 1$).
Definition 15.12. A ***traditional program*** is a set of traditional rules.
Definition 15.13. A ***ground program*** is a program containing no variables, no arithmetic symbols, and no comparative predicates.

Answer set programming languages, such as those of `lparse`, `gringo`, and `DLV`, and the standard ASP-Core-2, further specify the following:

- Constants are integers or start with a lowercase letter.
- Variables start with an uppercase letter.
- Function symbols start with a lowercase letter.
- Predicate symbols start with a lowercase letter.

15.1.2 SEMANTICS OF ANSWER SET PROGRAMS

The stable models semantics of ground programs is defined as follows.
Definition 15.14. An ***interpretation*** I is a set of ground atoms. If α is a ground atom, then $\alpha \in I$ represents that α is true, whereas $\alpha \notin I$ represents that α is false.
Definition 15.15. An interpretation I ***satisfies*** a rule

$$\alpha_1 \mid \ldots \mid \alpha_k \texttt{ :- } \beta_1\texttt{ , }\ldots\texttt{ , }\beta_m\texttt{ , not }\gamma_1\texttt{ , }\ldots\texttt{ , not }\gamma_n\texttt{ .}$$

if and only if

$$\{\alpha_1, \ldots, \alpha_k\} \cap I \neq \varnothing \text{ or}$$
$$\{\beta_1, \ldots, \beta_m\} \not\subseteq I \text{ or}$$
$$\{\gamma_1, \ldots, \gamma_n\} \cap I \neq \varnothing$$

That is, a rule is satisfied by an interpretation if and only if the head of the rule is satisfied by the interpretation if the body of the rule is satisfied by the interpretation.
Definition 15.16. An interpretation I is a ***model*** of a program Π if and only if I satisfies r for every $r \in \Pi$.
Definition 15.17. The ***reduct*** of a program Π relative to an interpretation I, written Π^I is the set of all rules

$$\alpha_1 \mid \ldots \mid \alpha_k \texttt{ :- } \beta_1\texttt{ , }\ldots\texttt{ , }\beta_m$$

such that

$$\alpha_1 \mid \ldots \mid \alpha_k \texttt{ :- } \beta_1\texttt{ , }\ldots\texttt{ , }\beta_m\texttt{ , not }\gamma_1\texttt{ , }\ldots\texttt{ , not }\gamma_n\texttt{ .}$$

is an element of Π, and $\{\gamma_1, \ldots, \gamma_n\} \cap I = \varnothing$.

Definition 15.18. An interpretation I is a ***stable model*** of a program Π if and only if I is a minimal model of Π^I relative to set inclusion.

15.1.3 STABLE MODELS: EXAMPLE 1

Let Π be the following program:

```
holdsAt(awake,2) :- holdsAt(awake,1), not happens(fallAsleep,1).
holdsAt(awake,1).
```

This program has three models:

```
{happens(fallAsleep,1), holdsAt(awake,1)}
{happens(fallAsleep,1), holdsAt(awake,1), holdsAt(awake,2)}
{holdsAt(awake,1), holdsAt(awake,2)}
```

We can show that $I = \{$`holdsAt(awake,1), holdsAt(awake,2)`$\}$ is a stable model of Π. The reduct Π^I is

```
holdsAt(awake,2) :- holdsAt(awake,1).    (15.1)
holdsAt(awake,1).                        (15.2)
```

I satisfies (15.1) and (15.2). Therefore I is a model of the reduct Π^I. We can show that I is minimal by considering all its subsets.

1. $\{$`holdsAt(awake,2)`$\}$ satisfies (15.1), but does not satisfy (15.2).
2. $\{$`holdsAt(awake,1)`$\}$ satisfies (15.2), but does not satisfy (15.1).
3. $\varnothing$ satisfies (15.1), but does not satisfy (15.2).

Therefore I is a minimal model of Π^I, and I is a stable model of Π.

The output of `clingo` on this program is

```
Answer: 1
holdsAt(awake,1) holdsAt(awake,2)
```

The program Π has one stable model.

15.1.4 STABLE MODELS: EXAMPLE 2

Suppose we add `happens(fallAsleep,1)` to the previous program Π:

```
holdsAt(awake,2) :- holdsAt(awake,1), not happens(fallAsleep,1).
holdsAt(awake,1).
happens(fallAsleep,1).
```

The program now has two models:

```
{happens(fallAsleep,1), holdsAt(awake,1)}
{happens(fallAsleep,1), holdsAt(awake,1), holdsAt(awake,2)}
```

We can show that I = {`happens(fallAsleep,1)`, `holdsAt(awake,1)`}, which lacks `holdsAt(awake,2)`, is a stable model. The reduct Π^I is

```
holdsAt(awake,1).                    (15.3)
happens(fallAsleep,1).               (15.4)
```

I is a model of the reduct Π^I because I satisfies (15.3) and (15.4). We can also show that I is minimal.

1. {`holdsAt(awake,1)`} satisfies (15.3), but does not satisfy (15.4).
2. {`happens(fallAsleep,1)`} satisfies (15.4), but does not satisfy (15.3).
3. $\varnothing$ does not satisfy (15.3) or (15.4).

Therefore I is a stable model of Π.

The output of `clingo` on this program is

```
Answer: 1
holdsAt(awake,1) happens(fallAsleep,1)
```

15.1.5 STABLE MODELS: EXAMPLE 3

Here is an example of an interpretation that is a model but is not a stable model. Consider the following program Π:

```
a :- b.
b :- a.
```

This program has two models:

$\varnothing$

{a, b}

Although I = {a, b} is a model of Π, it is not a stable model of Π. The reduct Π^I is

```
a :- b.                              (15.5)
b :- a.                              (15.6)
```

I satisfies (15.5) and (15.6), but I is not minimal because $\varnothing$, which is a subset of I, also satisfies (15.5) and (15.6). $I = \varnothing$ is a stable model of Π.

15.1.6 CHOICE RULES

Ilkka Niemelä, Patrik Simons, and Timo Soininen introduce several extensions to answer set programming, including the *choice rule*.

Definition 15.19. A ***choice rule*** is

$$\{\alpha_1 ; \ldots ; \alpha_k\} \texttt{ :- } \beta_1 , \ldots , \beta_m , \texttt{ not } \gamma_1 , \ldots , \texttt{ not } \gamma_n .$$

where $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_n$ are atoms.

This specifies that, if the body is satisfied, then any subset of the atoms in the head can be true. For example, given the program

```
p.
{q;r} :- p.
```

`clingo` produces

```
Answer: 1
p
Answer: 2
p q
Answer: 3
p r
Answer: 4
p r q
```

A choice rule of the form

$$\{\alpha_1 ; \ldots ; \alpha_k\} \texttt{ :- } \beta_1 \texttt{ , } \ldots \texttt{ , } \beta_m \texttt{ , not } \gamma_1 \texttt{ , } \ldots \texttt{ , not } \gamma_n \texttt{ .}$$

can be translated into the following traditional rules:

$$\begin{aligned}
\beta' \texttt{ :- } & \beta_1 \texttt{ , } \ldots \texttt{ , } \beta_m \texttt{ , not } \gamma_1 \texttt{ , } \ldots \texttt{ , not } \gamma_n \texttt{ .} \\
\alpha_1 \texttt{ :- } & \beta' \texttt{ , not } \overline{\alpha_1} \texttt{ .} \\
& \vdots \\
\alpha_k \texttt{ :- } & \beta' \texttt{ , not } \overline{\alpha_k} \texttt{ .} \\
\overline{\alpha_1} \texttt{ :- } & \texttt{not } \alpha_1 \texttt{ .} \\
& \vdots \\
\overline{\alpha_k} \texttt{ :- } & \texttt{not } \alpha_k \texttt{ .}
\end{aligned}$$

where β' is a new atom that represents that the body of the choice rule is true, and $\overline{\alpha_1}, \ldots, \overline{\alpha_k}$ are new atoms that represent the negation of $\alpha_1, \ldots, \alpha_k$ respectively. This is one way of describing the semantics of choice rules. Answer set solvers implement choice rules in more efficient ways.

The choice rule $\{\alpha\}$ is useful for exempting an atom α from stability checking. This is used in event calculus reasoning to exempt *HoldsAt* and *Happens* from minimization:

```
{holdsAt(F,T)} :- fluent(F), time(T).
{releasedAt(F,T)} :- fluent(F), time(T).
```

15.2 EVENT CALCULUS IN ANSWER SET PROGRAMMING: THEORY

We present the theory behind event calculus reasoning in answer set programming. We discuss an alternative definition of stable models in terms of the operator *SM* and its relation to the traditional definition. We specify the conditions under which *SM* is equivalent to circumscription. We show that these conditions are satisfied by event calculus domain descriptions, enabling *SM* to be used instead of circumscription for event calculus reasoning. Finally, we discuss the *EC2ASP* translation of event calculus problems into logic programs and the correctness of the translation.

15.2.1 SM

Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz introduce an alternative definition of stable model in terms of the *SM* operator, whose definition is similar to that of circumscription. This definition of stable model is more general than the definition of Section 15.1.2 in several ways:

- *SM* applies to first-order formulas, which may be syntactically more complex than answer set programming rules.
- *SM* addresses non-Herbrand models. (Herbrand structures and models are defined in Appendix A.3.3.)
- *SM* allows us to specify which predicates are subject to stability checking. The predicates that are subject to stability checking are referred to as *intensional predicates*.

The intensional predicates specify the predicates we would like to characterize based on nonintensional predicates. In the event calculus, we would like to characterize intensional predicates such as *Initiates* and *Terminates* by a conjunction of formulas that contain nonintensional predicates such as *HoldsAt* and *ReleasedAt*.

The stable models of a first-order formula Γ relative to a list of intensional predicates $\rho_1, \ldots, \rho_n$ are defined as the models of $SM[\Gamma; \rho_1, \ldots, \rho_n]$, which is a second-order formula defined in Section A.8.

15.2.2 RELATION BETWEEN SM AND STABLE MODELS

Consider the following sentence:

$$\forall x\,(\neg P(x) \Rightarrow Q(x)) \tag{15.7}$$

$SM[(15.7); P, Q]$ is

$$\forall x\,(\neg P(x) \wedge Q(x))$$

Now consider the following logic program:

```
q(X) :- not p(X).
```
(15.8)

If we introduce at least one constant, then the Herbrand models of $SM[(15.7); P, Q]$ are identical to the stable models of (15.8). In general, we can characterize the relationship between *SM* and the stable models of Section 15.1.2 as follows.

Definition 15.20. If r is a rule of the form

$$\alpha_1 \mid \ldots \mid \alpha_k \texttt{ :- } \beta_1\texttt{ , }\ldots\texttt{ , }\beta_m\texttt{ , not }\gamma_1\texttt{ , }\ldots\texttt{ , not }\gamma_n\texttt{ .}$$

then the ***FOL-representation*** of r, written $FOL[r]$, is the first-order formula

$$\forall v_1, \ldots, v_p\,(\beta_1 \wedge \ldots \wedge \beta_m \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_n \Rightarrow \alpha_1 \vee \ldots \vee \alpha_k)$$

where $v_1, \ldots, v_p$ are all the variables in r.

Definition 15.21. If Π is a logic program, then the ***FOL-representation*** of Π, written $FOL[\Pi]$, is the conjunction of the FOL-representations of the rules of Π.

For example, if Π is the logic program

```
q(a).
p(X) :- q(X), not r(X).
```

then $FOL[\Pi]$ is the first-order formula

$$q(a) \wedge \forall X\,(q(X) \wedge \neg r(X) \Rightarrow p(X)) \tag{15.9}$$

To simplify the discussion in this chapter, we violate the case conventions for first-order logic described in Section 2.1.5. In (15.9), X is a variable, a is a constant, and p, q, and r are predicate symbols.

Definition 15.22. Let σ be a signature, and let $\mathcal{S}$ be a Herbrand structure of σ. Then $\overline{\mathcal{S}}$ is defined as the set of all $\rho(a_1, \ldots, a_n)$ such that $n \in \{0, 1, 2, \ldots\}$, ρ is an n-ary predicate symbol of σ, and $\langle a_1, \ldots, a_n\rangle \in \rho^{\mathcal{S}}$.

Theorem 15.1. *Let σ be a signature containing at least one constant and a finite number of predicate symbols $\rho_1, \ldots, \rho_n$. Let Π be a traditional logic program of σ, and let $\mathcal{S}$ be a Herbrand structure of σ. Then*

$$\overline{\mathcal{S}} \textit{ is a stable model of } \Pi$$

if and only if

$$\mathcal{S} \textit{ is a model of } SM[FOL[\Pi]; \rho_1, \ldots, \rho_n]$$

Proof. See the proof of Theorem 1 of Ferraris, Lee, and Lifschitz (2011). ■

15.2.3 RELATION BETWEEN SM AND CIRCUMSCRIPTION

In this section, we characterize the conditions under which *SM* and *CIRC* produce the same results. It is important to note that the definitions and results related to *SM* assume that the logical connectives are limited to $\bot$ (false), $\wedge$, $\vee$, and $\Rightarrow$. Other logical connectives are considered to be abbreviations:

- $\top \stackrel{\text{def}}{\equiv} \bot \Rightarrow \bot$
- $\neg\Gamma \stackrel{\text{def}}{\equiv} \Gamma \Rightarrow \bot$
- $\Gamma_1 \Leftrightarrow \Gamma_2 \stackrel{\text{def}}{\equiv} (\Gamma_1 \Rightarrow \Gamma_2) \wedge (\Gamma_2 \Rightarrow \Gamma_1)$

Definition 15.23. An occurrence of a symbol $\circ$ in a formula Γ is ***strictly positive*** if and only if the occurrence of $\circ$ is not in the antecedent of an implication.

For example, in the sentence

$$(\forall x\,(Q(x) \Rightarrow P(x))) \wedge (\forall x\,(P(x) \Rightarrow R(x)))$$

the first occurrence of the predicate symbol P is strictly positive and the second one is not. In the sentence

$$((P(A) \vee P(B)) \wedge (\forall x\,(P(x) \vee Q(x) \Rightarrow R(x)))$$

the first occurrence of the symbol $\vee$ is strictly positive and the second one is not.

Definition 15.24. A formula Γ is ***canonical*** relative to predicate symbols $\rho_1, \ldots, \rho_n$ if and only if, for every ρ_i,

1. no occurrence of ρ_i in Γ is in the antecedents of more than one implication, and
2. for every occurrence of ρ_i in Γ, if the occurrence is in the scope of a strictly positive occurrence of $\exists$ or $\vee$ in Γ, then the occurrence is strictly positive in Γ.

Here are several examples. The effect axiom

$$\forall a, t\,((P(a) \Rightarrow Q(a)) \Rightarrow Initiates(E(a), F(a), t))$$

is canonical relative to *Initiates* but not relative to P, because the occurrence of P is in the antecedents of two implications. Similarly,

$$\forall a, t\,(\neg P(a) \Rightarrow Initiates(E(a), F(a), t))$$

is not canonical relative to P, because $\neg P(a)$ is an abbreviation for $P(a) \Rightarrow \bot$, and P occurs in the antecedents of two implications:

$$\forall a, t\,((P(a) \Rightarrow \bot) \Rightarrow Initiates(E(a), F(a), t))$$

The sentence

$$\forall x(P(x) \vee (Q(x) \Rightarrow R(x)))$$

is not canonical relative to Q because the occurrence of Q is not strictly positive in the scope of the strictly positive occurrence of $\vee$.

The conjunction of two effect axioms

$$\forall a, e, t\,(HoldsAt(F(a), t) \wedge Initiates(e, G(a), t) \Rightarrow Initiates(e, H(a), t)) \wedge$$
$$\forall a, e, t\,(HoldsAt(J(a), t) \wedge Initiates(e, K(a), t) \Rightarrow Initiates(e, L(a), t))$$

is canonical relative to *Initiates* because every occurrence of *Initiates* satisfies the conditions of Definition 15.24.

We can now state a precise relationship between *SM* and *CIRC*. It turns out that, for canonical formulas, they are logically equivalent.

Theorem 15.2. *If formula* Γ *is canonical relative to predicate symbols* $\rho_1, \ldots, \rho_n$, *then*

$$SM[\Gamma; \rho_1, \ldots, \rho_n] \Leftrightarrow CIRC[\Gamma; \rho_1, \ldots, \rho_n]$$

Proof. See the proof of Theorem 1 of Lee and Palla (2012). ■

15.2.4 USE OF SM FOR EVENT CALCULUS REASONING

The circumscribed formulas in an event calculus domain description have the following properties:

- Σ is canonical relative to *Initiates*, *Terminates*, *Releases*.
- Δ is canonical relative to *Happens*.
- Θ is canonical relative to $Ab_1, \ldots, Ab_n$.

We can therefore use *SM* instead of *CIRC* for reasoning in the event calculus.

Theorem 15.3. *Let*

$$D_C = CIRC[\Sigma; Initiates, Terminates, Releases] \wedge$$
$$CIRC[\Delta; Happens] \wedge$$
$$CIRC[\Theta; Ab_1, \ldots, Ab_n] \wedge \Xi$$

be an event calculus domain description as specified in Definition 2.11, with $\Xi = \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge \mathrm{E} \wedge \mathrm{CC}$. *Let*

$$D_S = SM[\Sigma; Initiates, Terminates, Releases] \wedge$$
$$SM[\Delta; Happens] \wedge$$
$$SM[\Theta; Ab_1, \ldots, Ab_n] \wedge \Xi$$

Then $D_C \Leftrightarrow D_S$.

Proof. This follows from Theorem 15.2. ■

So far, we have shown that *SM* can be used for event calculus reasoning. But *SM* operates on formulas and not logic programs. We next discuss the translation of an event calculus domain description into an equivalent logic program.

15.2.5 EC2ASP

Joohyung Lee and Ravi Palla define a translation *EC2ASP* for turning an event calculus domain description into a logic program. Let

$$D = CIRC[\Sigma; Initiates, Terminates, Releases] \wedge$$
$$CIRC[\Delta; Happens] \wedge$$
$$CIRC[\Theta; Ab_1, \ldots, Ab_n] \wedge \Xi$$

be an event calculus domain description. The *EC2ASP* translation of *D* into a logic program, written

$$EC2ASP[\Sigma; \Delta; \Theta; \Xi; Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n]$$

is computed as follows (Lee & Palla, 2012).

EC2ASP: Step 1

We construct formula Ξ' from Ξ as follows. First, we rewrite all definitional axioms of the form

$$\forall v(\rho(v) \Leftrightarrow \Gamma)$$

as

$$\forall v(\Gamma^{\neg\neg} \Rightarrow \rho(v))$$

where $\Gamma^{\neg\neg}$ is obtained from Γ by prepending $\neg\neg$ (*double negation*) to all occurrences of the intensional predicates $Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n$. Second, we prepend $\neg\neg$ to all strictly positive occurrences of the intensional predicates in the remaining axioms of Ξ.

For example, CC1, which is

$$Started(f,t) \Leftrightarrow HoldsAt(f,t) \vee \exists e\,(Happens(e,t) \wedge Initiates(e,f,t))$$

is rewritten as

$$HoldsAt(f,t) \vee \exists e\,(\neg\neg Happens(e,t) \wedge \neg\neg Initiates(e,f,t)) \Rightarrow Started(f,t)$$

Prepending double negation is necessary to eliminate unwanted cycles in the predicate dependency graph. For example, CC1 has an edge in the predicate dependency graph from *Started* to *Happens*, and a causal constraint may have an edge from *Happens* to *Started*. Because of the minimality condition in the definition of a stable model (Definition 15.18), *Started* and *Happens* atoms would be eliminated, giving an incorrect result. See the example of Section 15.1.5.

EC2ASP: Step 2

We eliminate quantifiers from $\Sigma \wedge \Delta \wedge \Theta \wedge \Xi'$ using the method given in Section 5.2 of the paper by Lee and Palla (2012).

EC2ASP: Step 3

We convert the resulting quantifier-free formula into a logic program using a set of transformations defined by Cabalar, Pearce, and Valverde (2005).

EC2ASP: Step 4

We add choice rules to the resulting program for all nonintensional predicates.

15.2.6 CORRECTNESS OF EC2ASP

We have the following correctness result for the *EC2ASP* translation.

Definition 15.25. Let Γ_1 and Γ_2 be formulas of signature ρ, and let σ be a subsignature of ρ. We say that Γ_1 is σ***-equivalent*** to Γ_2, written $\Gamma_1 \Leftrightarrow_\sigma \Gamma_2$ if and only if the class of models of Γ_1 restricted to σ is identical to the class of models of Γ_2 restricted to σ.

Theorem 15.4. *Let*

$$\begin{aligned} D_C = {} & CIRC[\Sigma; Initiates, Terminates, Releases] \wedge \\ & CIRC[\Delta; Happens] \wedge \\ & CIRC[\Theta; Ab_1, \ldots, Ab_n] \wedge \Xi \end{aligned}$$

be an event calculus domain description of signature σ that contains a finite number of predicate symbols. Let

$$\begin{aligned} D_S = SM[FOL[EC2ASP[\Sigma; \Delta; \Theta; \Xi; \\ Initiates, Terminates, Releases, Happens, \\ Ab_1, \ldots, Ab_n]]] \end{aligned}$$

Then $D_C \Leftrightarrow_\sigma D_S$.

Proof. See the Proof of Theorem 8 of Lee and Palla (2012). ■

15.3 EVENT CALCULUS IN ANSWER SET PROGRAMMING: PRACTICE

We now discuss how to solve event calculus reasoning problems using answer set programming. We discuss how to write a domain description and how to run a solver on the domain description.

15.3.1 WRITING THE DOMAIN DESCRIPTION

Expressing event calculus domain descriptions as answer set programs is similar to expressing these domain descriptions in first-order logic. The main difference is that the syntax of answer set programs is more restrictive. What can be expressed as a single first-order formula may require several rules. For example, the formula

$$(P(x) \vee Q(x)) \wedge R(x) \Rightarrow S(x)$$

must be expressed as two rules:

```
s(X) :- p(X), r(X).
s(X) :- q(X), r(X).
```

The formula

$$(\neg \exists y\, P(x, y)) \wedge Q(x) \Rightarrow R(x)$$

must also be expressed as two rules:

```
p1(X) :- p(X,Y).
r(X) :- q(X), not p1(X).
```

where `p1` is a new predicate symbol.

Here are some examples of how event calculus formulas are expressed as answer set programming rules. (A complete list of the various types of event calculus formulas is given in Section 2.7.)

We use predicate symbols to represent that an entity is of a particular sort and declare entities using these symbols:

```
object(leaf).
```

We declare events and fluents in terms of these entities:

```
fluent(height(O,H)) :- object(O), height(H).
fluent(falling(O)) :- object(O).
event(startFalling(O)) :- object(O).
event(hitsGround(O)) :- object(O).
```

We specify the effects of events:

```
initiates(startFalling(O),falling(O),T) :- object(O), time(T).
releases(startFalling(O),height(O,H),T) :- object(O), height(H),
                                            time(T).
initiates(hitsGround(O),height(O,H),T) :- holdsAt(height(O,H),T),
                                          object(O),
```

```
  height(H), time(T).
terminates(hitsGround(O),falling(O),T) :- object(O), time(T).
```

We specify trajectories:

```
trajectory(falling(O),T,height(O,H2),X) :-
  holdsAt(height(O,H1),T), H2=H1-X*X, object(O), height(H1),
  offset(X), time(T).
```

We specify triggered events:

```
happens(hitsGround(O),T) :-
  holdsAt(falling(O),T), holdsAt(height(O,0),T), object(O),
  time(T).
```

We specify state constraints:

```
:- H1!=H2, holdsAt(height(O,H1),T), holdsAt(height(O,H2),T),
   object(O), height(H1), height(H2), time(T).
```

We specify initial conditions and event occurrences:

```
:- holdsAt(falling(leaf),0).
holdsAt(height(leaf,9),0).
happens(startFalling(leaf),0).
```

If the reasoning type is abduction or planning, then we exempt `happens` from stability checking:

```
{happens(E,T)} :- event(E), time(T), T<maxtime.
```

We may also wish to specify that fluents are not released at timepoint 0:

```
:- releasedAt(F,0), fluent(F).
```

15.3.2 RUNNING THE SOLVER

We select an answer set grounder and solver such as `clingo`. We place the domain description in a file such as `file.lp`. We also need the rules for the event calculus. The DEC rules (`dec.lp`) are provided in Figure 15.1, and the CC rules (`cc.lp`) are provided in Figure 15.2. The `clingo` program supports double negation as failure such as `not not p(X)`. In `lparse`, this can be represented as `{not p(X)}0`. We then run `clingo`:

```
clingo -c maxtime=3 -n 0 file.lp dec.lp | format-output 3
```

The `format-output` program developed by Kim (2009) converts the output of answer set solvers into the more readable format produced by the Discrete Event Calculus Reasoner. The fluents that are true at timepoint 0 are shown, followed by differences in the truth values of fluents from one timepoint to the next. Fluents that become true are preceded by + (plus sign), and fluents that become false are preceded by - (minus sign).

```
time(0..maxtime).
{holdsAt(F,T)} :- fluent(F), time(T).
{releasedAt(F,T)} :- fluent(F), time(T).
stoppedIn(T1,F,T2) :- happens(E,T), T1<T, T<T2,
  terminates(E,F,T), event(E), fluent(F), time(T),
  time(T1), time(T2).
startedIn(T1,F,T2) :- happens(E,T), T1<T, T<T2,
  initiates(E,F,T), event(E), fluent(F), time(T),
  time(T1), time(T2).
holdsAt(F2,T1+T2) :- happens(E,T1), initiates(E,F1,T1), 0<T2,
  trajectory(F1,T1,F2,T2), not stoppedIn(T1,F1,T1+T2),
  event(E), fluent(F1), fluent(F2), time(T1), time(T2),
  T1+T2<maxtime.
holdsAt(F2,T1+T2) :- happens(E,T1), terminates(E,F1,T1), 0<T2,
  antiTrajectory(F1,T1,F2,T2), not startedIn(T1,F1,T1+T2),
  event(E), fluent(F1), fluent(F2), time(T1), time(T2),
  T1+T2<maxtime.
initiated1(F,T) :- happens(E,T), initiates(E,F,T), event(E),
  fluent(F), time(T).
terminated1(F,T) :- happens(E,T), terminates(E,F,T), event(E),
  fluent(F), time(T).
released1(F,T) :- happens(E,T), releases(E,F,T), event(E),
  fluent(F), time(T).
holdsAt(F,T+1) :- holdsAt(F,T), not releasedAt(F,T+1),
  not terminated1(F,T), fluent(F), time(T), T<maxtime.
:- holdsAt(F,T+1), not holdsAt(F,T), not releasedAt(F,T+1),
  not initiated1(F,T), fluent(F), time(T), T<maxtime.
releasedAt(F,T+1) :- releasedAt(F,T), not initiated1(F,T),
  not terminated1(F,T), fluent(F), time(T), T<maxtime.
:- releasedAt(F,T+1), not releasedAt(F,T), not released1(F,T),
  fluent(F), time(T), T<maxtime.
holdsAt(F,T+1) :- happens(E,T), initiates(E,F,T), event(E),
  fluent(F), time(T), T<maxtime.
:- holdsAt(F,T+1), happens(E,T), terminates(E,F,T), event(E),
  fluent(F), time(T), T<maxtime.
releasedAt(F,T+1) :- happens(E,T), releases(E,F,T), event(E),
  fluent(F), time(T), T<maxtime.
:- releasedAt(F,T+1), happens(E,T), initiates(E,F,T),
  event(E), fluent(F), time(T), T<maxtime.
:- releasedAt(F,T+1), happens(E,T), terminates(E,F,T),
  event(E), fluent(F), time(T), T<maxtime.
```

FIGURE 15.1

DEC rules for event calculus reasoning (`dec.lp`)

```
started(F,T) :- holdsAt(F,T).
started(F,T) :- not not happens(E,T), not not initiates(E,F,T),
  event(E), fluent(F), time(T).
stopped(F,T) :- not holdsAt(F,T), fluent(F), time(T).
stopped(F,T) :- not not happens(E,T), not not terminates(E,F,T),
  event(E), fluent(F), time(T).
initiated(F,T) :- started(F,T), not terminated1(F,T).
terminated(F,T) :- stopped(F,T), not initiated1(F,T).
```

FIGURE 15.2

CC rules for event calculus reasoning (`cc.lp`)

15.3.3 EXAMPLE: RUNNING AND DRIVING

The circumscription of the running and driving example in Section 9.2.1 can be handled in answer set programming. We specify entities, fluents, and events:

```
agent(nathan).
location(bookstore).

fluent(tired(A)) :- agent(A).

event(go(A,L)) :- agent(A), location(L).
event(run(A,L)) :- agent(A), location(L).
event(drive(A,L)) :- agent(A), location(L).
```

We write a disjunctive event axiom that specifies that, if an agent goes to a location, then the agent runs or drives to that location:

```
happens(run(A,L),T) | happens(drive(A,L),T) :- happens(go(A,L),T).
```

We write an effect axiom that says that, if an agent runs to a location, then the agent will be tired:

```
initiates(run(A,L),tired(A),T) :- agent(A), location(L), time(T).
```

Now we specify that Nathan was initially not tired and went to the bookstore:

```
:- holdsAt(tired(nathan),0).
happens(go(nathan,bookstore),0).
```

Finally, we specify that Nathan was tired afterward:

```
holdsAt(tired(nathan),1).
```

We specify that fluents are not released at timepoint 0:

```
:- releasedAt(F,0), fluent(F).
```

We then run `clingo` as follows:

```
clingo -c maxtime=1 -n 0 dec.lp rundrive.lp | format-output 1
```

It is correctly inferred that Nathan ran to the bookstore:

```
Answer: 1
0
happens(run(nathan,bookstore),0)
happens(go(nathan,bookstore),0)
1
+tired(nathan)
```

15.3.4 EXAMPLE: CARRYING A BOOK

The circumscription of the example of carrying a book in Section 6.4.1 can be handled in answer set programming. We specify entities, fluents, and events:

```
agent(nathan).
object(book).
object(A) :- agent(A).
room(livingRoom).
room(kitchen).

event(letGoOf(A,O)) :- agent(A), object(O).
event(pickUp(A,O)) :- agent(A), object(O).
event(walk(A,R1,R2)) :- agent(A), room(R1), room(R2).
fluent(inRoom(O,R)) :- object(O), room(R).
fluent(holding(A,O)) :- agent(A), object(O).
```

We use effect axioms to represent that, if an agent walks from room r_1 to room r_2, then the agent will be in r_2 and will no longer be in r_1:

```
initiates(walk(A,R1,R2),inRoom(A,R2),T) :- agent(A), room(R1),
    room(R2),  time(T).

terminates(walk(A,R1,R2),inRoom(A,R1),T) :- R1!=R2, agent(A),
  room(R1), room(R2), time(T).
```

We specify that an object is in one room at a time:

```
:- R1!=R2, holdsAt(inRoom(O,R1),T), holdsAt(inRoom(O,R2),T),
   object(O),  room(R1), room(R2), time(T).
```

We specify that, if an agent is in the same room as an object and the agent picks up the object, then the agent will be holding the object:

```
initiates(pickUp(A,O),holding(A,O),T) :- holdsAt(inRoom(A,R),T),
  holdsAt(inRoom(O,R),T), agent(A), object(O), room(R), time(T).
```

Further, if an agent is holding an object and the agent lets go of the object, then the agent will no longer be holding the object:

```
terminates(letGoOf(A,O),holding(A,O),T) :- holdsAt(holding(A,O),T),
  agent(A), object(O), time(T).
```

We use effect constraints to represent the indirect effects of walking from one room to another:

```
initiates(E,inRoom(O,R),T) :- initiates(E,inRoom(A,R),T),
  holdsAt(holding(A,O), T).

terminates(E,inRoom(O,R),T) :- terminates(E,inRoom(A,R),T),
  holdsAt(holding(A,O), T).
```

Now we specify that Nathan and the book start out in the living room. Nathan picks up the book and walks into the kitchen. He lets go of the book and walks into the living room.

```
holdsAt(inRoom(nathan,livingRoom),0).
holdsAt(inRoom(book,livingRoom),0).
:- holdsAt(holding(nathan,book),0).
:- holdsAt(holding(A,A),T), agent(A).

happens(pickUp(nathan,book),0).
happens(walk(nathan,livingRoom,kitchen),1).
happens(letGoOf(nathan,book),2).
happens(walk(nathan,kitchen,livingRoom),3).
```

We specify that fluents are not released at timepoint 0:

```
:- releasedAt(F,0), fluent(F).
```

Finally, we run `clingo` as follows:

```
clingo -c maxtime=4 -n 0 dec.lp book.lp | format-output 4
```

It is correctly inferred that, at the end, the book is in the kitchen and Nathan is in the living room:

```
Answer: 1
0
happens(pickUp(nathan,book),0)
inRoom(book,livingRoom)
inRoom(nathan,livingRoom)
1
+holding(nathan,book)
inRoom(book,livingRoom)
inRoom(nathan,livingRoom)
happens(walk(nathan,livingRoom,kitchen),1)
2
-inRoom(book,livingRoom)
-inRoom(nathan,livingRoom)
+inRoom(book,kitchen)
+inRoom(nathan,kitchen)
holding(nathan,book)
happens(letGoOf(nathan,book),2)
3
```

```
-holding(nathan,book)
inRoom(nathan,kitchen)
inRoom(book,kitchen)
happens(walk(nathan,kitchen,livingRoom),3)
4
-inRoom(nathan,kitchen)
+inRoom(nathan,livingRoom)
inRoom(book,kitchen)
```

15.4 F2LP

The F2LP program, which was developed by Joohyung Lee and Ravi Palla, is a useful tool for answer set programming. It automates the translation of first-order formulas under the stable models semantics into answer set programs. The logical symbols supported by F2LP are shown in Table 15.1. Answer set program rules may also be provided in the input to F2LP; these are written to output unchanged.

15.5 $\mathcal{E}$

The $\mathcal{E}$ action language, which was developed by Antonis Kakas and Rob Miller, can be used as a high-level language for answer set programming. We define $\mathcal{E}$ and describe how an $\mathcal{E}$ domain description can be translated into an answer set program.

Definition 15.26. An $\mathcal{E}$ ***signature*** σ consists of disjoint sets of fluents, events, and timepoints.

For the purposes of our translation of $\mathcal{E}$ into answer set programs, we assume that fluents and events are answer set program atoms, and we assume that timepoints are answer set program terms. We define an $\mathcal{E}$ *statement* and the translation of a statement into an answer set program rule as follows.

Table 15.1 Symbols Supported by F2LP

Name	F2LP	First-order logic
Conjunction	`&`	$\wedge$
Disjunction	`|`	$\vee$
Implication	`->`	$\Rightarrow$
Implication	`<-`	
Default negation	`not`	
Classical negation	`-`	$\neg$
True	`true`	$\top$
False	`false`	$\bot$
Universal quantification	`![X,Y,Z]:`	$\forall x,y,z$
Existential quantification	`?[X,Y,Z]:`	$\exists x,y,z$

Definition 15.27. If β is a fluent and τ is a timepoint, then

$$\beta \texttt{ holds-at } \tau$$

is a ***statement***. It is translated into the rule

$$\texttt{holdsAt(}\beta\texttt{, }\tau\texttt{).}$$

Definition 15.28. If β is a fluent and τ is a timepoint, then

$$\texttt{-}\beta \texttt{ holds-at } \tau$$

is a ***statement***. It is translated into the rule

$$\texttt{:- holdsAt(}\beta\texttt{, }\tau\texttt{).}$$

Definition 15.29. If α is an event and τ is a timepoint, then

$$\alpha \texttt{ happens-at } \tau$$

is a ***statement***. It is translated into the rule

$$\texttt{happens(}\alpha\texttt{, }\tau\texttt{).}$$

Definition 15.30. If α is an event, β is a fluent, $\gamma_1, \ldots, \gamma_n$ are fluents, and π is `initiates` or `terminates`, then

$$\alpha\ \pi\ \beta \texttt{ when } (\texttt{-})\gamma_1\texttt{, ..., }(\texttt{-})\gamma_n$$

is a ***statement***. It is translated into the rule

$$\pi\texttt{(}\alpha\texttt{, }\beta\texttt{, }\tau\texttt{) :- (not) holdsAt(}\gamma_1\texttt{, }\tau\texttt{), ..., (not) holdsAt(}\gamma_n\texttt{, }\tau\texttt{).}$$

where τ is a new variable.

BIBLIOGRAPHIC NOTES

The stable models semantics for logic programming was introduced by Gelfond and Lifschitz (1988). Book-length treatments of answer set programming are by Baral (2003), Gebser, Kaminski, Kaufmann, and Schaub (2013), and Gelfond and Kahl (2014). Answer set programming is also discussed by Gelfond (2008).

Prolog is discussed by Clocksin and Mellish (2003). ASP-Core-2 is specified by Calimeri et al. (2012). The `lparse` program is described by Syrjänen (2000). The `gringo`, `clasp`, and `clingo` programs are described by Gebser, Kaminski, Ostrowski, Schaub, and Thiele (2009), Gebser, Ostrowski, Kaminski, Kaufmann, and Schaub (2010), and Gebser, Kaminski, Kaufmann, and Schaub (2013). The `DLV` program is described by Bihlmeyer, Faber, Ielpa, Lio, and Pfeifer (2014).

Gelfond and Lifschitz (1991) extend answer set programming with classical negation. Niemelä, Simons, and Soininen (1999) introduce the choice rule. The translation of choice rules into traditional rules is from Janhunen and Niemelä (2011).

SM was defined by Ferraris, Lee, and Lifschitz (2007, 2011). Lee and Palla (2012), Palla (2012), and Kim, Lee, and Palla (2009) developed the theory behind solving event calculus reasoning problems using answer set programming. They define the

translation *EC2ASP* and prove its correctness. The FOL-representation of a logic program is defined by Lee, Lifschitz, and Palla (2008). Predicate dependency graphs are described by Lee and Palla (2012) and Palla (2012). Double negation as failure in logic programs is discussed by Lifschitz, Tang, and Turner (1999).

The F2LP program is described by Lee and Palla (2009). The $\mathcal{E}$ action language is described by Kakas and Miller (1997a, 1997b). Dimopoulos, Kakas, and Michael (2004) provide a translation of $\mathcal{E}$ domain descriptions into answer set programs without the use of the `holdsAt`, `happens`, `initiates`, and `terminates` predicate symbols. For example, `f holds-at t` is translated into `f(t)` rather than `holdsAt(f,t)`. The $\mathcal{E}$ language has been further developed into $\mathcal{M}$odular-$\mathcal{E}$ (Kakas, Michael, & Miller, 2005, 2011), which addresses the ramification and qualification problems.

EXERCISES

15.1 Let Π be the following program:

```
p.
r :- p, not q.
s :- t, not r.
```

Show that $I = \{$`p, r`$\}$ is a stable model of Π.

15.2 Let Π be the following program:

```
p | q :- r.
r :- not t.
t | u :- v.
v.
```

Show that $I = \{$`p, r, u, v`$\}$ is a stable model of Π.

15.3 Write a program to convert $\mathcal{E}$ domain descriptions into answer set programs using the translation described in Section 15.5.