Chapter 2

Background

In this chapter, we review some basic notions of the event calculus, which is a logical language for representing and reasoning about actions and their effects. It was first presented by Robert Kowalski and Marek Sergot in 1986. After that, we introduce to the reader the answer set semantics, created by M. Gelfond and V. Lifschitz in 1988. We compare it with other formalisms and mention important definitions and properties of answer set programs.

2.1 Event calculus

Event calculus is a narrative-based formalism for reasoning about action. The event calculus was introduced by Kowalski and Sergot in 1986. Here, we define the original version.

In their definition, Kowalski and Sergot introduced the term $After(e, f)$ to represent a period of time of a fluent. That is, the period of time initiated by the event e during which the fluent f is true. In addition, the fluent $Before(e, f)$ stands for the time terminated by event e during which the fluent f is true. The event e is the actual occurrence of the event. The formula $Holds(p)$ means that the fluent in the period p

holds from the beginning to the end of p [Shanahan, 2006].

The effects of events are described using predicates $Initiates$ and $Terminates$.

$$Holds(After(e, f)) \leftarrow Initiates(e, f)$$

$$Holds(Before(e, f)) \leftarrow Terminates(e, f)$$

Additionally, the formulas $Start(p, e)$ and $End(p, e)$ mean that the period p begins with the event e and that the period p ends with the event e, respectively.

$$Start(After(e, f), e)$$

$$End(Before(e, f), e)$$

$$Start(Before(e2, f), e1) \leftarrow Eq(After(e1, f),\ Before(e2, f))$$

$$End(After(e1, f), e2) \leftarrow Eq(After(e1, f),\ Before(e2, f))$$

The formula $Eq(p1, p2)$ means that periods $p1$ and $p2$ are the same. In the last two rules, $After(e1, f)$ corresponds to the period $p1$ and $Before(e2, f)$ is the period $p2$. Other formulas are the following:

$$\begin{aligned} Eq(After(e1, f), Before(e2, f)) \leftarrow\ & Holds(After(e1, f)) \wedge \\ & Holds(Before(e2, f)) \wedge Time(e1, t1) \wedge Time(e2, t2) \wedge \\ & t1 < t2 \wedge not\ Broken(t1, f, t2) \end{aligned}$$

$$\begin{aligned} Broken(t1, f1, t3) \leftarrow\ & Holds(After(e, f2)) \wedge Incompatible(f1, f2) \wedge \\ & Time(e, t2) \wedge t1 < t2 \wedge t2 < t3 \end{aligned}$$

$$Broken(t1, f1, t3) \leftarrow Holds(Before(e, f2)) \wedge Incompatible(f1, f2) \wedge Time(e, t2) \wedge t1 < t2 \wedge t2 < t3$$

The formula $Broken(t1, f, t2)$ represents that the fluent f ceases to hold between times $t1$ and $t2$. The formula $Incompatible(f1, f2)$ means that the fluents $f1$ and $f2$ cannot hold at the same time. An example of that situation is when $f1$ represents the negation of $f2$, therefore if $f1$ and $f2$ are both present will cause inconsistency. The clauses for $Incompatible$ are part of the domain description. The intended meaning of the formula $t1 < t2$, where $t1$ and $t2$ are time points, is obvious. The formula $Time(e, t)$ represents that event e occurs at time t. Finally, the formula $HoldsAt(f, t)$ says that fluent f holds at time t, and is constrained by the following axioms, where $In(t, p)$ means that time point falls within period p.

$$HoldsAt(f, t) \leftarrow Holds(After(e, f)) \wedge In(t, After(e, f)) \tag{2.1}$$

$$HoldsAt(f, t) \leftarrow Holds(Before(e, f)) \wedge In(t, Before(e, f)) \tag{2.2}$$

$$In(t, p) \leftarrow Start(p, e1) \wedge End(p, e2) \wedge Time(e1, t1) \wedge \tag{2.3}$$

$$Time(e2, t2) \wedge t1 < t \wedge t < t2 \tag{2.4}$$

$$In(t, p) \leftarrow Start(p, e1) \wedge not\ End(p, e2) \wedge Time(e1, t1) \wedge t1 < t \tag{2.5}$$

$$In(t, p) \leftarrow End(p, e1) \wedge not\ Start(p, e2) \wedge Time(e1, t1) \wedge t < t1. \tag{2.6}$$

Axiom 2.5 is used in the case when a period has no known end. In that case, the period is assumed to go on forever. On the other hand, axiom 2.6 caters for the case when a period has no known beginning.

The axioms described previously can be used as follows: given a description of the effects of events (actions) and narrative description, the axioms supply conclusions

of the form $HoldsAt(f, t)$. The effects of the events are described using the clauses $Initiates$, $Terminates$ and $Incompatible$. Kowalski and Sergot used a technique, in which event descriptions are decomposed into binary relationships [Shanahan, 2006]. In order to illustrate that technique, consider the Blocks World example, which inspired different works in Artificial Intelligence.

The Blocks World comprises a number of stackable, cube-shaped blocks on a table. The task is to describe the effects of moving these blocks from one place to another. For instance, using situation calculus, the fluent $On(x, y)$ denotes that block x is on y, where y is either another block or the table. In addition, $Clear(x)$ denotes that there is room for a block on the top of x, where x is either a block or the table. We assume that the table is always clear. There is only one action, which is $Move(x, y)$ and it means the moving of x to y [Shanahan, 2006].

Going back to the technique of Kowalski and Sergot, consider the following clause:

$$\begin{aligned} Initiates(e, On(x, y)) \leftarrow Act(e, Move) \wedge Object(e, x) \wedge Destination(e, y) \wedge \\ Time(e, t) \wedge HoldsAt(Clear(x), t) \wedge HoldsAt(Clear(y), t) \wedge \\ Diff(x, y) \wedge Diff(x, Table) \end{aligned}$$

In this clause, preconditions appear as $HoldsAt$ atoms in the body of a clause (the right hand side). Event occurrences are described in terms of the Act, $Object$, and $Destination$ predicates. Additionally, each event occurrence is given a unique name. For example, a simple narrative in which A is moved onto D at time 5, and then B is moved onto A at time 10 would be represented as follows:

$$Act(E1, Move)$$

$$Object(E1, A)$$

$$Destination(E1, D)$$

$$Time(E1, 5)$$

$$Act(E2, Move)$$
$$Object(E2, B)$$
$$Destination(E2, A)$$
$$Time(E2, 10)$$

However, the use of *Object* and *Destination* can be replaced by a parameterized *Move* action instead. In continuation is the full set of clauses for the Blocks World:

$$Initiates(e, On(x, y)) \leftarrow Act(e, Move(x, y)) \wedge Time(e, t) \wedge HoldsAt(Clear(x), t) \wedge HoldsAt(Clear(y), t) \wedge Diff(x, y) \wedge Diff(x, Table)$$

$$Terminates(e, On(x, z)) \leftarrow Act(e, Move(x, y)) \wedge Time(e, t) \wedge HoldsAt(Clear(x), t) \wedge HoldsAt(Clear(y), t) \wedge Diff(x, y) \wedge HoldsAt(On(x, z), t) \wedge Diff(y, z)$$

$$Incompatible(On(x, y), On(x, z)) \leftarrow Diff(y, z)$$

$$Initiates(e, Clear(z)) \leftarrow Act(e, Move(x, y)) \wedge Time(e, t) \wedge HoldsAt(Clear(x), t) \wedge HoldsAt(Clear(y), t) \wedge Diff(x, y) \wedge HoldsAt(On(x, z), t) \wedge Diff(y, z)$$

$$Terminates(e, Clear(y)) \leftarrow Act(e, Move(x, y)) \wedge Time(e, t) \wedge$$
$$HoldsAt(Clear(x), t) \wedge HoldsAt(Clear(y), t) \wedge$$
$$Diff(x, y) \wedge Diff(x, Table)$$

$$Incompatible(Clear(y), On(x, y))$$

The narrative description is correspondingly simpler without the Object and Destination predicates:

$$Act(E1, Move(A, D))$$
$$Time(E1, 5)$$
$$Act(E2, Move(B, A))$$
$$Time(E2, 10)$$

And finally, the initial situation can be described as follows:

$$Initiates(E0, On(C, Table))$$
$$Initiates(E0, On(B, C))$$
$$Initiates(E0, On(A, B))$$
$$Initiates(E0, On(D, Table))$$
$$Initiates(E0, Clear(A))$$
$$Initiates(E0, Clear(D))$$
$$Initiates(E0, Clear(Table))$$
$$Time(E0, 0)$$

where $E0$ is the initial event. The whole narrative is illustrated in figure 2.1.

In conclusion, event calculus is one way to represent commonsense reasoning. Let review again the basic notions of the event calculus: *event*, $fluent$ and *timepoint*. An *event* is an action that may occur in the world. For example, a person *driving* a car. A $fluent$ is a property of the world, which is time-varying. For instance, a person *in the garden*. Finally, a *timepoint* is just an instance of time, such as 8:30 a.m.

An event may happen at a given timepoint. A fluent has a truth value (true or false) at a timepoint or over an interval. When an action occurs, it may change the

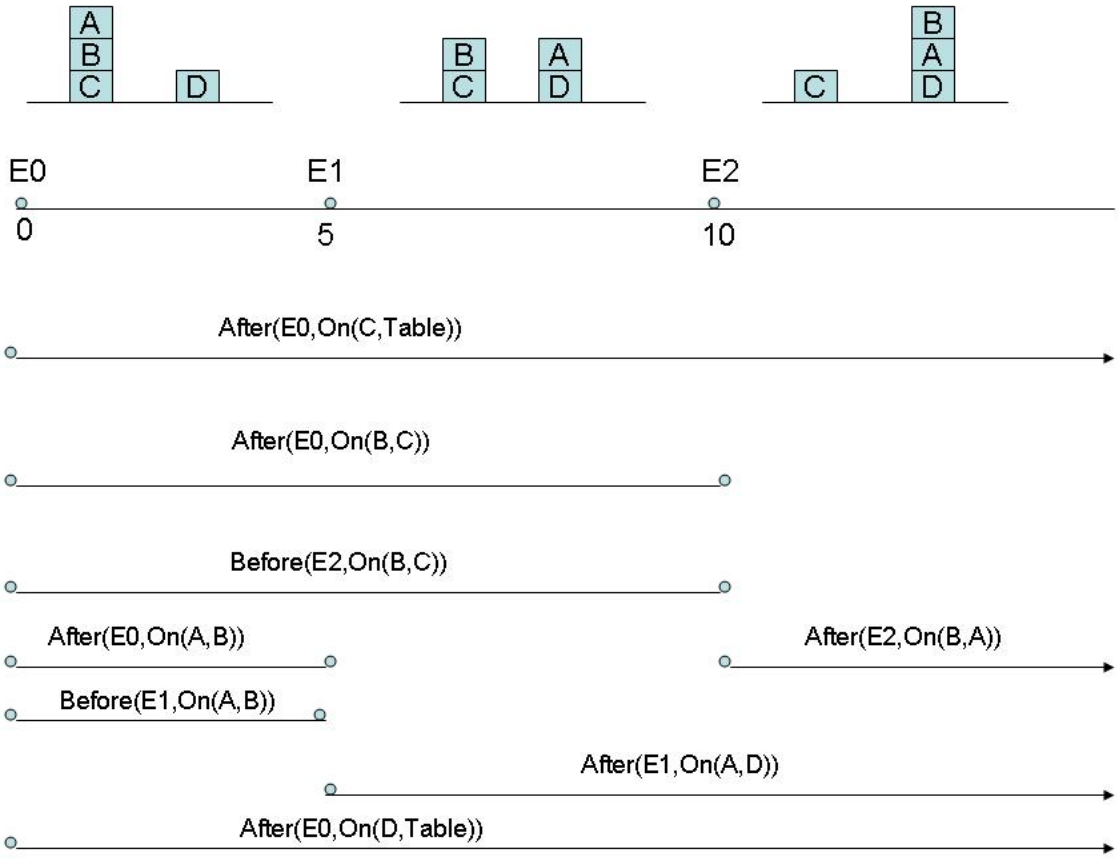

Figure 2.1: The blocks world narrative

value of the fluents. We have commonsense knowledge about the effects that actions have on fluents. We can express the previous notions as:

$HoldsAt(f, t)$, fluent f is true at time t.

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

$Initiates(e, f, t)$, if action e occurs at time t, then the fluent f becomes true.

$Terminates(e, f, t)$, if event e occurs at time t, then the fluent f becomes false [Mueller, 2006]. Given these notions of the event calculus, let us introduce the discrete event calculus.

2.1.1 Discrete event calculus

In [Mueller, 2004], Eric Mueller describes the discrete event calculus, which is a classical logic axiomatization of the event calculus with the time point restricted to the integers. The following are most of the axioms of the discrete event calculus, used in the DECReasoner system of Mueller [Mueller, 2005]. In the following description, e stands for an event, f for a fluent, and t for time.

1. Axiom DEC1:
   $Clipped(t_1, f, t_2) \stackrel{def}{\equiv} \exists e, t(Happens(e, t) \wedge t_1 \leq t < t_2 \wedge Terminates(e, f, t))$.
2. Axiom DEC2:
   $Declipped(t_1, f, t_2) \stackrel{def}{\equiv} \exists e, t(Happens(e, t) \wedge t_1 \leq t < t_2 \wedge Initiates(e, f, t))$.
3. Axiom DEC3:
   $Happens(e, t1) \wedge Initiates(e, f_1, t_1) \wedge 0 < t_2 \wedge$
   $Trajectory(f_1, t_1, f_2, t_2) \wedge \neg Clipped(t_1, f_1, t_1 + t_2)) \Longrightarrow$
   $HoldsAt(f_2, t_1 + t_2)$.
4. Axiom DEC4:
   $(Happens(e, t_1) \wedge Terminates(e, f_1, t_1) \wedge 0 < t_2 \wedge$

$AntiTrajectory(f_1, t_1, f_2, t_2) \wedge \neg Declipped(t_1, f_1, t_1 + t_2)) \Longrightarrow$

$HoldsAt(f_2, t_1 + t_2)$.

5. Axiom DEC5:

   $(HoldsAt(f, t) \wedge \neg(ReleasedAt(f, t + 1) \wedge$

   $\neg\exists e(Happens(e, t) \wedge Terminates(e, f, t))) \Longrightarrow$

   $HoldsAt(f, t + 1)$.

6. Axiom DEC6:

   $(\neg HoldsAt(f, t) \wedge \neg ReleasedAt(f, t + 1) \wedge$

   $\neg\exists e(Happens(e, t) \wedge Initiates(e, f, t))) \Longrightarrow$

   $\neg HoldsAt(f, t + 1)$.

7. Axiom DEC7:

   $(ReleasedAt(f, t) \wedge$

   $\neg\exists e(Happens(e, t) \wedge (Initiates(e, f, t) \vee Terminates(e, f, t)))) \Longrightarrow$

   $ReleasedAt(f, t + 1)$.

8. Axiom DEC8:

   $(\neg ReleasedAt(f, t) \wedge \neg\exists e(Happens(e, t) \wedge Releases(e, f, t))) \Longrightarrow$

   $\neg ReleasedAt(f, t + 1)$.

9. Axiom DEC9:

   $(Happens(e, t) \wedge Initiates(e, f, t)) \Longrightarrow HoldsAt(f, t + 1)$

10. Axiom DEC10:

    $(Happens(e, t) \wedge Terminates(e, f, t)) \Longrightarrow \neg HoldsAt(f, t + 1)$.

11. Axiom DEC11:

    $(Happens(e, t) \wedge Releases(e, f, t)) \Longrightarrow ReleasedAt(f, t + 1)$.

12. Axiom DEC12:

$$(Happens(e,t) \wedge (Initiates(e,f,t) \vee Terminates(e,f,t))) \Longrightarrow$$
$$\neg ReleasedAt(f,t+1).$$

We decided to show the basic concepts of event calculus and discrete event calculus, because Erik Mueller uses them in his examples of commonsense reasoning. However, In Section 2.2, we describe briefly Answer Set Programming, which is our preferred method. The reasons will be explained throughout this work.

2.2 Answer Set Programming

Answer Set Programming (ASP) is a formal, logical language designed for declarative problem solving, knowledge representation and real world reasoning. It represents a modern approach to logic programming. It has a background on first logic and epistemic logic, but it does not work as a classical theorem prover [de Vos and Watson, 2005].

A rule is an expression of the form:

$$L_0 \; or \ldots or \; L_k \leftarrow \; L_{k+1}, \ldots, L_m, \; not \; L_{m+1}, \ldots, not \; L_n,$$

where each of the L_is is a literal in the sense of classical logic. Intuitively, the above rule means that if $L_{k+1}, \ldots, L_m$ are true and if $L_{m+1}, \ldots, L_n$ can be assumed to be false, then at least one of $L_0, \ldots, L_k$ must be true [Gelfond and Lifschitz, 1988]. The logical connective *not* is called *negation as failure* [Baral and Gelfond, 1994].

The ASP language is very suitable for knowledge representation, reasoning, and declarative problem solving. The non-classical symbols $\leftarrow$ and *not* give a specific structure. As a result, syntactic sub-classes are defined and their properties are studied. These sub-classes have different complexity and expressiveness. Therefore, different sub-classes are used for different applications. Moreover, ASP has efficient implementations, which help to program large applications. In conclusion, the reason to use ASP

is that there exists a great support structure around it (including implementations and theoretical results), which allow systematic creation of ASP programs and assimilation of new information [Baral, 2003]. In Chapter 3, we compare ASP with other well-known logical formalisms.

2.3 ASP as a knowledge representation language

ASP is a suitable knowledge representation language. Some of its properties are:

1. *Restricted monotonicity* (An ASP program behaves monotonically with respect to addition of literals about certain predicates. This is useful when we do not want future information to change the meaning of a definition.). Let consider the following example from [Baral, 2003], which illustrate a simple restricted monotonicity property. These rules are based on the well known example of the Yale Shooting Scenario: there is a turkey and a gun, the gun can be loaded or not, and the turkey will be dead when shooting with the gun loaded. In the following code, *ab* stands for abnormal, *res* stands for result, and *loaded* means that the gun is loaded. In addition, the symbol ¬ is a denotation for the classical negation as opposed to the negation as failure (denoted by *not*). Negation as failure has nonmonotonic character, which is an advantage over the classical negation. A rule with a negation as failure is similar to a default.

   $r_1 : holds(alive, s_0) \leftarrow .$

   $r_2 : holds(F, res(A, S)) \leftarrow \neg holds(F, S), not\ ab(F, A, S).$

   $r_3 : \neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), not\ ab(F, A, S).$

   $r_4 : \neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S).$

   $r_5 : ab(alive, shoot, S) \leftarrow not\ \neg holds(loaded, S).$

If we add the additional rule:

$r_6 : holds(alive, res(shoot, s_0)) \leftarrow .$

we can intuitively conclude that the gun is not loaded in s_0. But the program $\Pi = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ does not entail $\neg holds(loaded, s_0)$. If we add the following explicit rules:

$r_7 : holds(loaded, S) \leftarrow holds(alive, S), \neg holds(alive, res(shoot, S)).$

$r_8 : \neg holds(loaded, S) \leftarrow holds(alive, res(shoot, S)).$

$r_9 : \neg holds(F, S) \leftarrow \neg holds(F, res(A, S)), not\ ab(F, A, S).$

$r_{10} : holds(F, S) \leftarrow holds(F, res(A, S)), not\ ab(F, A, S).$

the program $\Pi' = \Pi \cup \{r_7, r_8, r_9, r_{10}\}$ entails $\neg holds(loaded, s_0)$, which accomplishes our goal.

2. *Language independence* and *language tolerance*

   Definition 2.1 (Language independence)**.** An answer set program Π is *language independent* if, for two languages L_1 and L_2 that are permissible [1] for Π, the ground programs $ground(\Pi, L_1)$ and $ground(\Pi, L_2)$ have the same consistent answer sets [Baral, 2003].

   Definition 2.2. An answer set program Π is *language tolerant* if, for any two languages L_1 and L_2 that are permissible for Π, the following holds: If A_1 is a consistent answer set for the ground program $ground(\Pi, L_1)$, then there is a consistent answer set A_2 for the ground program $ground(\Pi, L_2)$ such that $A_1 \cup lit(L_2) = A_2 \cup Lit(L_1)$ [Baral, 2003].

3. *Sort-ignorable* The sorts can be ignored through language tolerance.

4. Addition of new knowledge through $filtering$.

[1] Let L be an arbitrary language, and let Π be an answer set program. If every rule in Π is a rule in L, we say that L is permissible for Π.

In addition, ASP provides *compact representation* in certain knowledge representation problems. In Section 2.4, some of the applications of ASP are listed.

2.4 Applications of ASP

ASP has many applications to database query languages, knowledge representation, reasoning, and planning. Some examples, as described in [Baral, 2003], are:

- ASP has a greater ability than Datalog in expressing database query features.
- ASP is used in planning, as well as in approximate planning, in the presence of incompleteness. In addition, ASP can be used for conformant planning [2] .
- ASP is useful in product configuration, in representation of constraint satisfaction problems, and of dynamic constraint satisfaction problems.
- ASP has been used for scheduling, supply chain planning, and in solving combinatorial auctions.
- ASP has been used in formalizing deadlock and reachability in Petri nets. In addition, it is used in characterizing monitors, and in cryptography.
- ASP has been used in verification of contingency plans for shuttles and also in verifying correctness of circuits in the presence of delays.
- ASP has been used in knowledge representation problems such as reasoning about actions, plan verification, the frame problem, reasoning with inheritance hierarchies, and with prioritized defaults.
- ASP is appropriate for reasoning with incomplete information.

[2] Conformant planning is the problem of finding, in a nondeterministic domain, a sequence of actions which will achieve the goal for all possible contingencies.

Different applications have been created using ASP. For instance, recently, a group of researchers in Arizona State University combined ASP with Link Grammar and WordNet. In [L.Tari and C.Baral, 2005], the authors use Link Grammar and WordNet to extract facts and disambiguate verbs and nouns. They developed a system for deep reasoning about the travel domain. The same domain has been studied and implemented by Gelfond [Gelfond, 2006]. More examples of created software are the Monkey and Bananas domain, the World Block domain, the Space shuttle and the Zoo example.

2.5 Syntax of ASP programs

An *answer set framework* consists of two alphabets (an axiom alphabet and a query alphabet), two languages (an axiom language and a query language) defined over the two alphabets, a set of axioms, and an entailment relation between sets of axioms and queries [Gelfond and Lifschitz, 1988]. Let us first review the axiom alphabet.

Definition 2.3. The axiom alphabet is composed by the following:

1. variables,
2. constants,
3. function symbols,
4. predicate symbols,
5. connectives,
6. punctuation symbols, and
7. the symbol $\perp$,

where the connectives are $\neg$, *or*, $\leftarrow$, *not*, and ",". The punctuation symbols are "(", ")" and "."[Baral, 2003].

Usually, variables are combinations of letters and numbers that start with an upper-case letter. On the other hand, constants, predicate symbols, and function symbols are strings that start with a lower-case letter.

Definition 2.4. A *term* is defined as follows:

- A variable is a term.
- A constant is a term.
- If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

Definition 2.5. A term is said to be *ground*, if no variable occurs in it [Baral, 2003].

An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and each t_i is a term. If each of the terms is ground, then the atom is said to be ground.

A *literal* is either an atom or an atom preceded by the symbol $\neg$. The former is referred to as a positive literal and the latter as a negative literal. A literal is said to be ground, if the atom in it is ground.

Definition 2.6 (Herbrand Universe and Herbrand Base)**.** The *Herbrand Universe* of a language L, denoted by HU_L, is the set of all ground terms, which can be formed with the functions and constants in L.

The *Herbrand Base* of a language L, denoted by HB_L, is the set of all ground atoms that can be formed with predicates from L and terms from HU_L.

Definition 2.7. A *rule* is of the form:

L_0 *or* $\ldots$ *or* $L_k \leftarrow L_{k+1}, \ldots, L_m,$ *not* $L_{m+1}, \ldots,$ *not* $L_n,$

where L_is are literals (or when $k = 0$, L_0 may be the symbol $\perp$), $k \geq 0$, $m \geq k$, and $n \geq m$.

A rule is referred to as ground if all the literals of the rule are ground.

The part on the left of the $\leftarrow$ is the *head* (or *conclusion*) and the part on the right is the *body* (or *premise*) of the rule.

A rule with an empty body and a single disjunct in the head (in other words, $k = 0$) is called a $fact$. In that case, if L_0 is a ground literal we refer to it as a ground fact. A fact can be written without the use of $\leftarrow$ as:

$$L_0.$$

When $k = 0$, and $L_0 = \perp$, the rule is called a *constraint*. The symbol $\perp$ in the heads of constraints is usually eliminated and the rule is written as:

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

Definition 2.8. Let r be a rule in a language L. The grounding of r in L, denoted by $ground(r, L)$, is the set of all rules obtained from r by all possible substitutions of elements of HU_L for the variables in r.

Definition 2.9. The *answer set language* given by an alphabet consists of the set of all ground rules constructed from the symbols of the alphabet.

The *signature* of the answer set framework is denoted by $\sigma = (O, F, P)$, where O are the constants, F are the function symbols, and P are the predicate symbols of the language.

2.6 ASP programs

An *answer set program* is a finite set of rules of the form:

$$L_0 \; or \ldots \; or \; L_k \; \leftarrow \; L_{k+1}, \ldots, L_m, \; not \; L_{m+1}, \ldots, not \; L_n,$$

and is used to express a set of axioms of an answer set framework. Intuitively, a logic program can be viewed as a specification for building possible theories of the world. In addition, the rules can be viewed as constraints that those theories should satisfy. As a result, semantics of logic programs differ in the way they define satisfiability of the rules [Baral and Gelfond, 1994]. We are interested particularly in answer set semantics. There are different types of answer set programs, some of them are listed below. More details can be found in [Baral, 2003].

2.6.1 General logic programs

This is the most popular sub-class and it is also referred to as *normal logic programs*. A general logic program is a set of rules where L_is are atoms and $k = 0$.

Example 2.1. The following is an example of a general logic program:

$$fly(X) \leftarrow bird(X), not \; ab(X)$$
$$ab(X) \leftarrow penguin(X).$$
$$bird(X) \leftarrow penguin(X).$$
$$bird(tweety) \leftarrow .$$
$$penguin(skippy) \leftarrow .$$

2.6.2 Definite logic programs

Also known as *Horn* logic programs, the definite logic programs are sets of rules where L_is are atoms, $k = 0$, and $m = n$. Therefore, *definite programs* are general logic programs that do not have negation as failure (*not*) [Baral and Gelfond, 1994].

Example 2.2. An example of a definite logic program is:

$$anc(X,Y) \leftarrow par(X,Y).$$
$$anc(X,Y) \leftarrow par(X,Z), anc(Z,Y).$$
$$par(a,b).$$
$$par(b,c).$$
$$par(d,e).$$

2.6.3 Extended logic programs

An extended logic program is a set of rules with $k = 0$.

Example 2.3. For instance,

$$fly(X) \leftarrow bird(X),\ not\ \neg fly(X).$$
$$\neg fly(X) \leftarrow penguin(X).$$
$$bird(tweety).$$
$$bird(rocky).$$
$$penguin(rocky).$$

2.6.4 Normal disjunctive logic programs

A normal disjunctive program is a set of rules where L_is are atoms. If $m = n$, then we have a *disjunctive logic program*.

Example 2.4. For example,

$$bird(X)\ or\ reptile(X) \leftarrow lays\_egg(X).$$
$$lays\_egg(slinky).$$

In Section 2.7, we define the semantics of ASP programs. As opposed to the syntax, which describes the construction of complex signs from simpler signs, the semantics refers to the aspects of meaning.

2.7 Semantics of ASP programs

Definite programs are the simplest class of declarative logic programs. The semantics of such programs can be defined basically in two ways: using a model theoretical characterization (explained in Section 2.7.1) or a iterated fixpoint characterization (explained in Section 2.7.2).

2.7.1 Model theoretical characterization

In order to define what is an answer set for a definite program, we need to introduce few more definitions. A *Herbrand interpretation* of a definite program Π is any subset $I \subseteq HB_\Pi$ of its Herbrand base. Answer sets are defined as particular Herbrand interpretations that satisfy certain properties with respect to the program and are *minimal*. An interpretation I is *minimal* among $\{I_1, \ldots, I_n\}$ if there does not exist a j, with $1 \leq j \leq n$ such that I_j is a strict subset of I. An interpretation I is *least* among the set $\{I_1, \ldots, I_n\}$, if for all j, where $1 \leq j \leq n$, $I \subseteq I_j$.

A *Herbrand interpretation* S of Π is said to satisfy the rule

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

if the following conditions hold:

1. $L_0 \neq \perp$: $\{L_1, \ldots, L_m\} \subseteq S$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$ implies that $L_0 \in S$.
2. $L_0 = \perp$: $\{L_1, \ldots, L_m\} \not\subseteq S$ or $\{L_{m+1}, \ldots, L_n\} \cap S \neq \emptyset$.

A *Herbrand model* M of a program Π is a Herbrand interpretation of Π such that it satisfies all rules in Π. We also say that M is closed under Π.

Definition 2.10. An *answer set* of a definite program Π is a Herbrand model of Π which is minimal among the Herbrand models of Π [Gelfond and Lifschitz, 1988].

Definition 2.11. An *answer set* of a definite program Π is a minimal subset S of HB that is closed under $ground(\Pi)$ [Baral, 2003].

Proposition 2.1. *Definite programs have unique answer sets [Baral, 2003].*

Proposition 2.2. *The intersection of the Herbrand models of a definite program is its unique minimal Herbrand model [Baral, 2003].*

2.7.2 Iterated fixpoint characterization

In order to explain the fixpoint characterization, let us assume the definite program Π to be a possibly infinite set of ground rules. In addition, let 2^{HB_Π} be the set of all Herbrand interpretations of Π. We define an operator $T \overset{0}{\Pi}: 2^{HB_\Pi} \rightarrow 2^{HB_\Pi}$ as:

$$T \overset{0}{\Pi}(I) = \{L_0 \in HB_\Pi \mid \Pi \text{ contains a rule } L_0 \leftarrow L_1, \ldots, L_m, \text{ such that } \{L_1, \ldots, L_m\} \subseteq I \text{ holds } \}.$$

The previously declared operator is known as *immediate consequence operator*. The intuitive meaning of $T \overset{0}{\Pi}(I)$ is the set of atoms that can be derived from a single application of Π given the atoms in I [Baral, 2003].

The operator $T \overset{0}{\Pi}(I)$ is monotone. That is, $I \subseteq I' \rightarrow T \overset{0}{\Pi}(I) \subseteq T \overset{0}{\Pi}(I')$.

Let assign the empty set to $T \overset{0}{\Pi} \uparrow 0$. Additionally, let define $T \overset{0}{\Pi} \uparrow (i+1)$ to be $T \overset{0}{\Pi}(T \overset{0}{\Pi} \uparrow i)$. It is easy to see that, $T \overset{0}{\Pi} \uparrow 0 \subseteq T \overset{0}{\Pi} \uparrow 1$. By the monotonicity and transitivity property of $\subseteq$, we have $T \overset{0}{\Pi} \uparrow i \subseteq T \overset{0}{\Pi} \uparrow (i+1)$. In the case of a finite

Herbrand base, the repeated application of $T\overset{0}{\Pi}$ starting from the empty set, will take us to a fixpoint of $T\overset{0}{\Pi}$. Moreover, this fixpoint is the least fixpoint of $T\overset{0}{\Pi}$. The case of an infinite Herbrand base is similar. For more details the reader should refer to [Baral, 2003]. In conclusion, a definite program Π can be characterized by its least fixpoint.

Proposition 2.3. *For any definite program, $lfp(T\overset{0}{\Pi})$ (least fixpoint) is the unique minimal Herbrand model of Π and the answer set of Π.*

Next we give the definition of answer sets for general logic program, which allow the operator *not* in the body of the rules. The previous approach of minimal models and iterated fixpoints cannot be directly applied in this case, because general logic programs have multiple answer sets [Baral, 2003].

The approach to define answer sets of general programs is to use a fixpoint definition. The steps are the following: given a candidate answer set S for a general program Π, we first transform Π with respect to S and obtain a definite program denoted by Π^S. Then, S is defined as an answer set of Π, if S is the answer set of the transformed definite program Π^S. This operation is called the Gelfond-Lifschitz transformation (or reduct) and it was introduced by Gelfond and Lifschitz in [Gelfond and Lifschitz, 1988].

Definition 2.12. Let Π be a ground general program. For any set S of atoms, let Π^S be a program obtained from Π by deleting:

- each rule that has a literal *not* L in its body, where $L \in S$, and
- literals of the form *not* L in the bodies of the remaining rules.

The transformed program Π^S does not contain *not*, so it is a definite program and its answer set is already defined. If this answer set exists and moreover, it coincides with S, then we say that S is an *answer set* of Π. In other words, an answer set of Π is characterized by the equation:

$$S = M_0(\Pi^S),$$

where M_0 is the minimal model.

2.8 Closing remarks

In this chapter, we first reviewed the basics of the event calculus. After, we introduced the basics of Answer Set Programming, as well as the syntax and semantics of Answer Set Programs. In brief, ASP is non-monotonic, it has the ability to represent defaults (sentences that start with *Normally*), and it has both negation as failure as well as classical negation, which allow us to express all kind of rules. The reader may refer to [M.Osorio and Giannella, 2001] and [M. Osorio and Arrazola, 2001] for more results related to ASP.