IMPLEMENTING AND EXPERIMENTING WITH ANSWER SET PROGRAMMING BASED EVENT CALCULUS REASONER

by

Tae-Won Kim

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

ARIZONA STATE UNIVERSITY

October 2009

IMPLEMENTING AND EXPERIMENTING WITH ANSWER SET PROGRAMMING BASED EVENT CALCULUS REASONER

by

Tae-Won Kim

has been approved

October 2009

Graduate Supervisory Committee:

Joohyung Lee, Chair
Chitta Baral
Subbarao Kambhampati

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

Recently, Lee and Palla showed that circumscription can be embedded into the stable model semantics in the first order case, and based on this, showed how to reformulate circumscriptive event calculus in terms of answer set programming (ASP). This opens up a new opportunity for answer set solvers to be used for event calculus reasoning. We have implemented an ASP-based event calculus reasoner ECASP and compared it with a satisfiability (SAT) based event calculus system called the discrete event calculus (DEC) reasoner. Assuming that the domain is closed and finite, unlike the DEC reasoner, ECASP can handle all axioms of the event calculus. Moreover, it shows significant speed-up over the DEC reasoner for all benchmark problems that we tested.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Knowledge representation and reasoning is the field concerned with representing knowledge in a language with a precise formal semantics and deriving useful conclusions from the encoded knowledge. Reasoning about actions is a subfield of knowledge representation and reasoning that concentrates on representing effects of actions and reasoning about them. Researchers in this area have encountered the fundamental problems such as the frame [19](how to describe things that do not change by default), the ramification [19](how to describe indirect effects of an action), and the qualification problems [18](how to ensure successful execution of an action). First-order logic, although widely used in artificial intelligence and computer science, has serious limitations to solving the problems. It mainly lacks of modeling default reasoning. Thus several formalisms have been proposed. Over the years, each formalism has evolved significantly, but it is not clear how they are related to each other, especially due to the fact that nonmonotonic logics and classical logic have been viewed distant from each other. The formalisms, for example, the situation calculus [19], the event calculus [9], and temporal action logic [2], are based on some extensions of first-order logic such as circumscription [18], while some others, for example, action languages [6], are based on nonmonotonic logics such as the stable model semantics [5] or causal logic [7].

However, it is shown recently that some representative nonmonotonic logics are closely related to each other. Lin and Zhao [17] showed that logic programs under the stable model semantics can be turned into propositional formulas, which allowed to compute the stable models using efficient implementation of SAT solvers that are widely available. Furthermore, Ferraris, Lee, and Lifschitz [4] proposed a new definition of a stable model which includes neither grounding nor the fixpoints unlike in the traditional stable model semantics. The new definition is given in terms of second-order (classical) logic which looks similar to

circumscription. Indeed, Ferraris, Lee, and Lifschitz [4] showed that the new stable model semantics can be characterized by circumscription. Also, Lee and Lin [11] showed that circumscription can be embedded into the stable model semantics in the propositional case. This result was further extended to the first-order case in [13]. Based on the relationship between circumscription and the stable model semantics, Lee and Palla [13] showed how to turn circumscriptive event calculus [24] into answer set programming (ASP), a new declarative programming paradigm based on the stable model semantics.

In this thesis, we test the hypothesis that an ASP-based event calculus called ECASP [1] handles more expressive reasoning than the existing SAT-based approach with exploiting the computational benefits of ASP solvers. The thesis is organized as follows. In the next chapter, we review circumscription and the event calculus based on it. We show how to compute event calculus reasoning in the theoretical and implementation view; the Yale Shooting problem [8] is used as the main example. In addition, we present a slight extension of the answer set semantics from [16], and review ASP and the general language of stable models [4]. Chapter 3 is a review of [13] that shows how to turn circumscription into the stable model semantics and how to reformulate circumscriptive event calculus as ASP. In Chapter 4, we introduce ECASP and show how to solve various event calculus problems in that system. In Chapter 5, we compare the performance of different systems on reasoning problems including the 14 benchmark problems [26], [27]. Finally, we conclude in Chapter 6. Appendix A is the domain description of Robby's apartment [3] in the input language of ECASP.

[1] http://reasoning.eas.asu.edu/ecasp/.

2. BACKGROUND

2.1. Circumscription

Circumscription is a nonmonotonic formalism introduced by John McCarthy. It allows default reasoning by minimizing the extents of predicates. Mathematically, circumscription turns a first-order sentence into a stronger second-order sentence [14]. Let $\mathbf{p}$ be a list of distinct predicate constants $p_1, \ldots, p_n$ and let $\mathbf{u}$ be a list of distinct predicate variables $u_1, \ldots, u_n$ whose length is the same as the length of $\mathbf{p}$. Expression $\mathbf{u} \leq \mathbf{p}$ stands for the conjunction of formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \rightarrow p_i(\mathbf{x}))$ where $i = 1, \ldots n$ and $\mathbf{x}$ is a list of distinct object variables of the same length as the arity of p_i. Expression $\mathbf{u} = \mathbf{p}$ stands for the conjunction of formulas $\forall \mathbf{x}(u_i(\mathbf{x}) \leftrightarrow p_i(\mathbf{x}))$. Moreover, expression $\mathbf{u} < \mathbf{p}$ stands for $(\mathbf{u} \leq \mathbf{p}) \wedge \neg(\mathbf{u} = \mathbf{p})$ and the expression intuitively means that the set denoted by $\mathbf{u}$ is a strict subset of the set denoted by $\mathbf{p}$. Given a first-order sentence F, CIRC$[F; \mathbf{p}]$ is defined as the second-order sentence

$$F \wedge \neg \exists \mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F(\mathbf{u}))$$

where $\mathbf{p}$ is the list $p_1, \ldots, p_n$ of predicate constants occurring in F, $\mathbf{u}$ is a list $u_1, \ldots, u_n$ of distinct predicate variables corresponding to $\mathbf{p}$, and $F(\mathbf{u})$ is the formula obtained from F by substituting $\mathbf{u}$ for $\mathbf{p}$. For example, let F be $P(a)$ where P is the predicate constant occurring in F and a is an object constant. Then, CIRC$[F; P]$ is

$$P(a) \wedge \neg \exists u((u < P) \wedge u(a)).$$

This formula means that a is in the set P and there is no proper subset u of P such that a belongs to u. That is, P is a singleton set whose member is only a.

Consider the following formulas in the event calculus.

Nathan wakes up at timepoint 1:

$$Happens(WakeUp(Nathan), 1). \tag{2.1}$$

If an agent wakes up, then the agent will be awake:

$$Initiates(WakeUp(a), Awake(a), t). \tag{2.2}$$

If an agent falls asleep, then the agent will no longer be awake:

$$Terminates(FallAsleep(a), Awake(a), t). \tag{2.3}$$

However, these formulas do not tell us about events and effects that are not mentioned above: there may be other (unexpected) events that do occur or (unexpected) effects of $WakeUp$ that are not described above. To avoid such unexpected cases, we minimize $Happens$, $Initiates$, and $Terminates$ using circumscription. CIRC[(2.1); $Happens$] is

$$(e = WakeUp(Nathan) \wedge t = 1) \leftrightarrow Happens(e, t). \tag{2.4}$$

$WakeUp(Nathan)$ is the only event that happened. CIRC[(2.2); $Initiates$] is

$$\exists a(e = WakeUp(a) \wedge f = Awake(a)) \leftrightarrow Initiates(e, f, t). \tag{2.5}$$

The positive effect of $WakeUp(a)$ on $Awake(a)$ is the only positive effect known. CIRC[(2.3); $Terminates$] is

$$\exists a(e = FallAsleep(a) \wedge f = Awake(a)) \leftrightarrow Terminates(e, f, t). \tag{2.6}$$

The negative effect of $FallAsleep(a)$ on $Awake(a)$ is the only negative effect known.

2.2. Computing Circumscription

Theorem 1 *[14, Proposition 2] Let P be a predicate constant, let $\mathbf{x}$ be a list of variables of the same length as the arity of P, and let $F(\mathbf{x})$ be a formula. If $F(\mathbf{x})$ does not contain P, then the circumscription*

$$\mathrm{CIRC}[\forall \mathbf{x}(F(\mathbf{x}) \rightarrow P(\mathbf{x});\ P]$$

is equivalent to

$$\forall \mathbf{x}(F(\mathbf{x}) \leftrightarrow P(\mathbf{x})).$$

The transformation that turns the implication into the equivalence is called *predicate completion* [1]. In general, circumscription is not reducible to first-order formula but as we saw in the previous section, it can be turned into predicate completion under certain assumption. For example, (2.1) can be rewritten as

$$(e = WakeUp(Nathan) \wedge t = 1) \rightarrow Happens(e, t). \tag{2.7}$$

Predicate completion turns (2.7) into

$$(e = WakeUp(Nathan) \wedge t = 1) \leftrightarrow Happens(e, t). \tag{2.8}$$

In view of Theorem 1, (2.8) is equivalent to CIRC[(2.1); $Happens$].

An occurrence of a predicate constant in a formula is *positive* if the number of implications whose antecedent contains the occurrence is even. For example, consider $(p \rightarrow q) \rightarrow r$. p and r have a positive occurrence in the formula.

Theorem 2 *[14, Proposition 14] Let $\mathbf{p}$ be a list of predicate constants $p_1, \ldots, p_n$ occurring in a formula F. If every predicate constant in $\mathbf{p}$ has a positive occurrence in F, then the*

parallel circumscription

$$\text{CIRC}[F; p_1, \ldots, p_i]$$

is equivalent to

$$\bigwedge_{i=1}^{n} \text{CIRC}[F; p_i].$$

For example, we compute CIRC[(2.2) $\wedge$ (2.3); $Initiates$, $Terminates$] by Theorem 1 and Theorem 2 :

$$(Initiates(e, f, t) \leftrightarrow \exists a(e = WakeUp(a) \wedge f = Awake(a))) \ \wedge$$
$$(Terminates(e, f, t) \leftrightarrow \exists a(e = FallAsleep(a) \wedge f = Awake(a))).$$

2.3. The Event Calculus

The event calculus is a formalism for commonsense reasoning. The original version of the event calculus [9] was formulated as a logic program that can be executed in PROLOG. Later, Shanahan reformulated the event calculus in terms of classical logic using circumscription [24]. Circumscriptive event calculus has evolved significantly in [26], [27], [28]. Mueller introduced the discrete event calculus [20] that limits the timepoint sort to integers, which simplifies *EC* axioms. Based on the reduction of circumscription to predicate completion under certain condition, event calculus reasoning can be reduced to propositional satisfiability (SAT). This led to implementations of the event calculus using SAT solvers to compute event calculus problems. An event calculus planner [29] and the discrete event calculus reasoner [20] were implemented based in this idea.

2.3.1. Circumscriptive Event Calculus

We follow the syntax of the event calculus described in Chapter 2 from [21]. (Circumscriptive) event calculus is based on many sorted first-order logic. The sorts such as events,

fluents, timepoints, and domain objects are declared. Here, an *event* is an action that can happen. A *fluent* is a property that can change by events that occur. A *timepoint* is a time point. A *condition* used in the event calculus is defined recursively as follows:

- A comparison is a condition. A comparison is $\tau_1 < \tau_2$, $\tau_1 \leq \tau_2$, $\tau_1 \geq \tau_2$, $\tau_1 > \tau_2$, $\tau_1 = \tau_2$, or $\tau_1 \neq \tau_2$ where τ_1 and τ_2 are terms;
- If f is a fluent term and t is a timepoint term, then $HoldsAt(f, t)$ and $\neg HoldsAt(f, t)$ are conditions;
- If γ_1 and γ_2 are conditions, then $\gamma_1 \wedge \gamma_2$ and $\gamma_1 \vee \gamma_2$ are conditions;
- If v is a variable and γ is a condition, then $\exists v\gamma$ is a condition.

We will use the symbols e and e_i $(1 \leq i \leq n)$ as event terms, f and f_i $(1 \leq i \leq n)$ as fluent terms, t and t_i $(1 \leq i \leq n)$ as timepoint terms, γ and γ_i $(1 \leq i \leq n)$ as conditions, and Ab_i $(1 \leq i \leq n)$ as abnormal predicates. The meanings of some basic predicates used in the event calculus are shown in Table I on the next page.

An event calculus domain description is defined as

$$\text{CIRC}[\Sigma;\ \mathit{Initiates}, \mathit{Terminates}, \mathit{Releases}] \wedge \text{CIRC}[\Delta_1 \wedge \Delta_2;\ \mathit{Happens}] \wedge$$
$$\text{CIRC}[\Theta;\ \mathit{Ab}_1, \ldots, \mathit{Ab}_n] \wedge \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge E$$

where

- Σ is a conjunction of the axioms of the form;
  - Positive effect axiom: $\gamma \rightarrow \mathit{Initiates}(e, f, t)$
  - Negative effect axiom: $\gamma \rightarrow \mathit{Terminates}(e, f, t)$
  - Release axiom: $\gamma \rightarrow \mathit{Releases}(e, f, t)$
  - Effect constraint: $\gamma \wedge \pi_1(e, f_1, t) \rightarrow \pi_2(e, f_2, t)$ where π_1 and π_2 are

Table I. Event Calculus Predicates

Predicate	**Meaning**
$HoldsAt(f,t)$	f is true at t.
$Happens(e,t)$	e occurs at t.
$ReleasedAt(f,t)$	f is released from the commonsense law of inertia at t.
$Initiates(e,f,t)$	If e occurs at t, then f will be true and not released from the commonsense law of inertia after t.
$Terminates(e,f,t)$	If e occurs at t, then f will be false and not released from the commonsense law of inertia after t.
$Releases(e,f,t)$	If e occurs at t, then f will be released from the commonsense law of inertia after t.
$Trajectory(f_1,t_1,f_2,t_2)$	If f_1 is initiated by an event that occurs at t_1 and $t_2 > 0$, then f_2 will be true at t_1+t_2.
$AntiTrajectory(f_1,t_1,f_2,t_2)$	If f_1 is terminated by an event that occurs at t_1 and $t_2 > 0$, then f_2 will be true at t_1+t_2.

$Initiates$ or $Terminates$

- Positive cumulative effect axiom:

$\gamma \wedge [\neg]Happens(e_1,t) \wedge \cdots \wedge [\neg]Happens(e_n,t) \rightarrow Initiates(e,f,t)$

- Negative cumulative effect axiom:

$\gamma \wedge [\neg]Happens(e_1,t) \wedge \cdots \wedge [\neg]Happens(e_n,t) \rightarrow Terminates(e,f,t)$

- Δ_1 is a conjuction of the axioms of the form;

  - Event occurrence formula: $Happens(e,t)$

  - Temporal ordering formula: $t_1 < t_2$, $t_1 \leq t_2$, $t_1 \geq t_2$, $t_1 > t_2$, $t_1 = t_2$, or $t_1 \neq t_2$

- Δ_2 is a conjuction of the axioms of the form;

  - Trigger axioms: $\gamma \rightarrow Happens(e,t)$

  - Causal constraints: $\sigma(f,t) \wedge \pi_1(f_1,t) \wedge \cdots \wedge \pi_n(f_n,t) \rightarrow Happens(e,t)$ where σ is *Stopped* or *Started*, and π_i $(1 \leq i \leq n)$ is $Initiated$ or $Terminated$. Each predicate is defined in the following axioms:

[$CC1$] $Started(f,t) \leftrightarrow (HoldsAt(f,t) \vee \exists e(Happens(e,t) \wedge Initiates(e,f,t)))$

[$CC2$] $Stopped(f,t) \leftrightarrow (\neg HoldsAt(f,t) \vee \exists e(Happens(e,t) \wedge Terminates(e,f,t)))$

[$CC3$] $Initiated(f,t) \leftrightarrow (Started(f,t) \wedge \neg\exists e(Happens(e,t) \wedge Terminates(e,f,t)))$

[$CC4$] $Terminated(f,t) \leftrightarrow (Stopped(f,t) \wedge \neg\exists e(Happens(e,t) \wedge Initiates(e,f,t)))$

- Disjunctive event axiom: $Happens(e,t) \rightarrow Happens(e_1,t) \vee \cdots \vee Happens(e_n,t)$

- Θ is a conjunction of cancellation axioms containing the abnormal predicates;

  - Cancellation axiom: $\gamma \rightarrow Ab_i(\tau_1,\ldots,\tau_n,t)$ where τ_i $(1 \leq i \leq n)$ is a term

- Ω is a conjunction of unique names axioms;

  - Unique names axiom: $U[\phi_1,\ldots,\phi_n]$ where ϕ_i $(1 \leq i \leq n)$ is a function under the unique name assumption.

- Ψ is a conjunction of the axioms of the form;

  - State constraint: γ_1, $\gamma_1 \rightarrow \gamma_2$, or $\gamma_1 \leftrightarrow \gamma_2$

  - Action precondition axiom: $\gamma \rightarrow Happens(e,t)$

  - Event occurrence constraint [1]: $Happens(e_1,t) \wedge \gamma \wedge [\neg]Happpens(e_2,t) \rightarrow \perp$

- Π is a conjunction of axioms of the form;

  - Trajectory axiom: $\gamma \rightarrow Trajectory(f_1,t_1,f_2,t_2)$

  - Antitrajectory axiom: $\gamma \rightarrow AntiTrajectory(f_1,t_1,f_2,t_2)$

- Γ is a conjunction of observations such as $HoldsAt(f,t)$ and $ReleasedAt(f,t)$;

- E is a conjunction of event calculus axioms such as EC or DEC [20]. DEC includes 2 definitions and 10 axioms as follows:

[1] We rewrite the formula $Happens(e_1,t) \wedge \gamma \rightarrow [\neg]Happens(e_2,t)$ from [21] so that the $Happens$ predicate does not occur in the consequent. It will result in a succinct representation [13].

$[DEC1]$ $StoppedIn(t_1, f, t_2) \leftrightarrow \exists e, t(Happens(e, t) \wedge t_1 < t < t_2 \wedge Terminates(e, f, t))$

$[DEC2]$ $StartedIn(t_1, f, t_2) \leftrightarrow \exists e, t(Happens(e, t) \wedge t_1 < t < t_2 \wedge Initiates(e, f, t))$

$[DEC3]$ $(Happens(e, t_1) \wedge Initiates(e, f_1, t_1) \wedge 0 < t_2 \wedge \neg StoppedIn(t_1, f_1, t_1 + t_2)$

$\wedge Trajectory(f_1, t_1, f_2, t_2)) \rightarrow HoldsAt(f + 2, t_1 + t_2)$

$[DEC4]$ $(Happens(e, t_1) \wedge Terminates(e, f_1, t_1) \wedge 0 < t_2 \wedge \neg StartedIn(t_1, f_1, t_1 + t_2)$

$\wedge AntiTrajectory(f_1, t_1, f_2, t_2)) \rightarrow HoldsAt(f + 2, t_1 + t_2)$

$[DEC5]$ $(HoldsAt(f, t) \wedge \neg ReleasedAt(f, t + 1)$

$\wedge \neg\exists e(Happens(e, t) \wedge Terminates(e, f, t))) \rightarrow HoldsAt(f, t + 1)$

$[DEC6]$ $(\neg HoldsAt(f, t) \wedge \neg ReleasedAt(f, t + 1)$

$\wedge \neg\exists e(Happens(e, t) \wedge Initiates(e, f, t))) \rightarrow \neg HoldsAt(f, t + 1)$

$[DEC7]$ $(ReleasedAt(f, t) \wedge \neg\exists e(Happens(e, t)$

$\wedge (Initiates(e, f, t) \vee Terminates(e, f, t)))) \rightarrow ReleasedAt(f, t + 1)$

$[DEC8]$ $(\neg ReleasedAt(f, t) \wedge \neg\exists e(Happens(e, t) \wedge Releases(e, f, t)))$

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

$[DEC9]$ $(Happens(e, t) \wedge Initiates(e, f, t)) \rightarrow HoldsAt(f, t + 1)$

$[DEC10]$ $(Happens(e, t) \wedge Terminates(e, f, t)) \rightarrow \neg HoldsAt(f, t + 1)$

$[DEC11]$ $(Happens(e, t) \wedge Releases(e, f, t)) \rightarrow ReleasedAt(f, t + 1)$

$[DEC12]$ $(Happens(e, t) \wedge (Initiates(e, f, t) \vee Terminates(e, f, t)))$

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

$DEC1$ and $DEC2$ are definitional axioms for the predicates $StoppedIn$ and $StartedIn$. $DEC3$ and $DEC4$ describe the behavior of trajectories to represent gradual change like falling objects. $DEC5$ to $DEC8$ describe inertia of $HoldsAt$ and $ReleasedAt$ to handle truth values of fluents. $DEC9$ to $DEC12$ describe effects of events on fluents.

The event calculus supports several kinds of reasonings:

- **Temporal projection** is a reasoning task to decide a resulting state given the initial state and a sequence of events that occur;
- **Planning** is a reasoning task to find a sequence of events given the initial state and a final state correspond to the goal;
- **Postdiction** is a reasoning task to decide the initial state given a final state and a sequence of events that occur.

Moreover, the event calculus can represent the following:

- *The commonsense law of inertia.* A fluent remains unchanged if there is no action that affects it. For example, an object stays in the current position if there is no action that affects the location of the object;
- *Conditional effects of events.* The effects of events depends on the state in which the events happen. For example, if a person turns on a light but the electricity was turned off, then the light does not turn on. The result of turning on the light depends on the status of the electricity;
- *Indirect effects of events.* For example, if a monkey has bananas, and walks to another location, then not only the monkey's location changes (as the direct effect of walking), but also the location of bananas change (as an indirect effect of walking);
- *Events with nondeterministic effects.* For example, tossing a coin has two nondeterministic effects such as head or tail;
- *Continuous (gradual) change.* For example, the height of a falling ball changes continuously until the ball hits the ground;

- *Triggered events.* When a specific condition is satisfied, an event must occur. For example, an alarm clock beeps when the current time is the alarm time.

2.3.2. Example: Yale Shooting

The Yale Shooting problem was introduced in [8]. Initially a turkey is alive and a gun is not loaded. Then the gun is loaded and after sneezing (waiting), the gun is shot. From this scenario, our commonsense tells us the turkey is dead. How do we automate this commonsense reasoning? We answer the question by formalizing the domain in the event calculus. There are two fluents *Alive* and *Loaded*, and three events *Load*, *Sneeze*, and *Shoot*.

If the person loads the gun, then the gun will be loaded:

$$Initiates(Load, Loaded, t). \tag{2.9}$$

If the gun is loaded and the person shoots the gun, then the turkey will not be alive any more:

$$HoldsAt(Loaded, t) \rightarrow Terminates(Shoot, Alive, t). \tag{2.10}$$

If the gun is shot, then the gun will not be loaded any more:

$$Terminates(Shoot, Loaded, t). \tag{2.11}$$

Initially, the turkey is alive and the gun is not loaded:

$$HoldsAt(Alive, 0) \tag{2.12}$$

$$\neg HoldsAt(Loaded, 0). \tag{2.13}$$

Consider the following sequence of events. The person loads the gun at timepoint 0, sneezes at timepoint 1, and shoots the gun at timepoint 2:

$$Happens(Load, 0) \tag{2.14}$$

$$Happens(Sneeze, 1) \tag{2.15}$$

$$Happens(Shoot, 2). \tag{2.16}$$

Given the conjunction of the DEC axioms, $\Sigma = (2.9) \wedge (2.10) \wedge (2.11)$, $\Delta = (2.14) \wedge (2.15) \wedge (2.16)$, $\Omega = U[Load, Sneeze, Shoot] \wedge U[Loaded, Alive]$, and $\Gamma = (2.12) \wedge (2.13)$, we can check that the turkey is not alive at timepoint 3. We represent this as follows:

Theorem 3

$$\text{CIRC}[\Sigma;\ Initiates,\ Terminates] \wedge \text{CIRC}[\Delta;\ Happens] \wedge \Omega \wedge \Gamma \wedge DEC$$
$$\models \neg HoldsAt(Alive, 3). \tag{2.17}$$

Proof By Theorem 1 and Theorem 2, we compute circumscription using predicate completion. Then we have

$$(Initiates(e, f, t) \leftrightarrow (e = Load \wedge f = Loaded)) \quad \wedge \tag{2.18}$$
$$(Terminates(e, f, t) \leftrightarrow (e = Shoot \wedge f = Alive \wedge HoldsAt(Loaded, t)) \vee$$
$$(e = Shoot \wedge f = loaded)) \quad \wedge$$
$$\neg Releases(e, f, t)$$

$$Happens(e, t) \leftrightarrow (e = Load \wedge t = 0) \quad \vee \quad (e = Sneeze \wedge t = 1) \quad \vee$$
$$(e = Shoot \wedge t = 2). \tag{2.19}$$

From $DEC9$, (2.18), and (2.19), we have

$$HoldsAt(Loaded, 1). \tag{2.20}$$

From $DEC12$, (2.18), and (2.19), we have

$$\neg ReleasedAt(Loaded, 1). \tag{2.21}$$

From $DEC8$, (2.18), and (2.21), we have

$$\neg ReleasedAt(Loaded, 2). \tag{2.22}$$

From $DEC5$, (2.18), (2.19), (2.20), and (2.22), we have

$$HoldsAt(Loaded, 2). \tag{2.23}$$

From $DEC10$, (2.18), (2.19), and (2.23), we finally have $\neg HoldsAt(Alive, 3)$.

We notice that (2.17) is a temporal projection problem. We can also solve a planning and postdiction problem by the event calculus [21:41-43].

2.3.3. The Discrete Event Calculus Reasoner

The Discrete Event Calculus (DEC) Reasoner [2] was developed by Erik Mueller. The system turns a domain description in the event calculus into a satisfiability problem and finds models using SAT solvers. The block diagram of the system is shown in Figure 1.

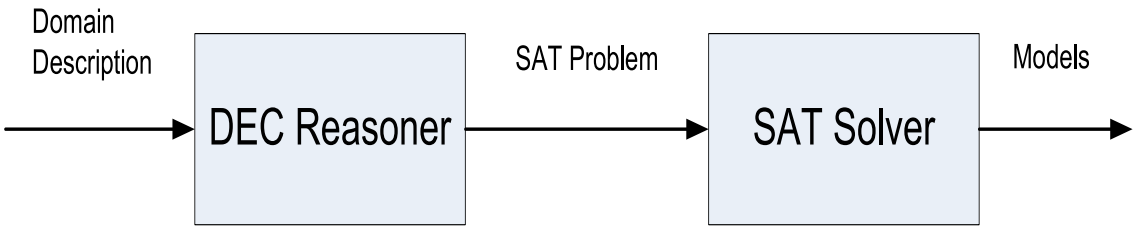

Fig. 1. Discrete Event Calculus Reasoner System

The following domain description (`Yale3.e`) shows the Yale Shooting problem in the input language of the DEC reasoner:

[2] http://decreasoner.sourceforge.net/.

```
; Yale3.e

load foundations/Root.e

load foundations/EC.e

event Load()

event Shoot()

event Sneeze()

fluent Loaded()

fluent Alive()


; Effect Axioms

[time] Initiates(Load(),Loaded(),time).

[time] HoldsAt(Loaded(),time) -> Terminates(Shoot(),Alive(),time).

[time] Terminates(Shoot(),Loaded(),time).

; Initial Condition

HoldsAt(Alive(),0).

!HoldsAt(Loaded(),0).

Happens(Load(),0).


Happens(Sneeze(),1).

Happens(Shoot(),2).

completion Happens

range time 0 3

range offset 1 1
```

The first line is for loading the **Root.e** file, which contains the declaration of the basic sorts

such as the boolean, integer, predicate, and function sorts. The second line is for loading the **EC.e** file, which includes the declaration of the time, offset [3], fluent, and event sorts and predicates of the event calculus [22]. Next events and fluents are defined. The next eight lines encode the formulas (2.9),. . . ,(2.16). The *completion* statement instructs the DEC reasoner to do predicate completion on *Happens* predicate. Last, the ranges of the time sort and the offset sort are defined.

Upon reading the input file above, the DEC reasoner produces the following output:

```
Discrete Event Calculus Reasoner 1.0

loading examples/Yale3.e

loading foundations/Root.e

loading foundations/EC.e

28 variables and 64 clauses

relsat solver

1 model

---

model 1:

0

Alive().

Happens(Load(), 0).

1

+Loaded().

Happens(Sneeze(), 1).

2
```

[3]In DEC, the values of offset are integer timepoints to represent gradual changes of fluents [23].

```
Happens(Shoot(), 2).
3
-Alive().
-Loaded().
...
encoding 0.1s
solution 0.0s
total 0.2s
```

The DEC reasoner calls the RELSAT [4] solver to compute models. This output shows the fluents and the events that are true at each timepoint. A plus sign indicates that a fluent becomes true at the timepoint. A minus sign indicates that a fluent becomes false at the timepoint. For example, the turkey is alive, the gun is not loaded, and the event *Load* occurs at timepoint 0. The turkey is no longer alive and the gun is no longer loaded at timepoint 3.

2.4. Answer Set Semantics and Answer Set Programming

2.4.1. Answer Set Semantics

The answer set semantics was proposed to explain the meaning of negation as failure in logic programming [5]. Here we present a slight extension of the answer set semantics from [16] that allows variables. An *atom* is defined as in first-order logic: an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate constant of arity n ($n \geq 0$) and $t_1, \ldots, t_n$ are terms. A *nested expression* is defined recursively as follows:

- Every atom is a nested expression;

[4] http://www.bayardo.org/resources.html.

- 0-place connectives ($\top$ and $\bot$) are nested expressions;
- If F is a nested expression, then $\neg F$ is a nested expression;
- If F and G are nested expressions, then $F \vee G$ and $F \wedge G$ are nested expressions.

A *rule* is of the form

$$H \leftarrow B \tag{2.26}$$

where H and B are nested expressions. We call H the *head* of the rule and B the *body* of the rule. A *(logic) program* is a set of rules.

Under the answer set semantics, variables occurring in a program can be eliminated by means of grounding — the process that replaces every variable with every ground term, which can be built using function and object constants occurring in the program, in all possible ways. Let Π be a program that contains no variables and let $\sigma(\Pi)$ be the signature consisting of object, function and predicate constants that occur in Π. A *ground atom* is an atom with no variables. Let M be a set of ground atoms of $\sigma(\Pi)$. The reduct Π^M of Π relative to M is the negation-free program obtained from Π by replacing each maximal occurrence of the form $\neg F$ (where F is a nested expression) by $\bot$ if $M \models F$ and by $\top$ otherwise. M is an *answer set* of the program Π if M is a minimal set of ground atoms satisfying Π^M. The minimality here is in terms of set inclusion. For example, consider the program:

$$\begin{aligned} &happy(John) \leftarrow \neg hungry(John) \wedge \neg cold(John) \\ &happy(John) \leftarrow married(John). \end{aligned} \tag{2.27}$$

Let $M = \{happy(John)\}$. The reduct of the program relative to M is

$$happy(John) \leftarrow \top \wedge \top \tag{2.28}$$
$$happy(John) \leftarrow married(John).$$

The minimal set of ground atoms satisfying (2.28) is M and thus it is the answer set of the program (2.27).

2.4.2. Answer Set Programming

Answer set programming is a new declarative programming paradigm based on the stable model (answer set) semantics [15]. The main idea of ASP is to solve a combinatorial search problem by writing a logic program whose answer sets correspond to the solutions of the problem and using ASP solvers to compute them. Several efficient ASP solvers have been developed, such as ASSAT [5], CMODELS [6], DLV [7], SMODELS [8], SUP [9], CLASP and CLINGO [10].

The following are useful ASP language constructs:

- *Constraint.* By a constraint we denote a rule (2.26) whose head is $\perp$. We often omit the head $\perp$. For example, consider the rule:

$$\leftarrow happy(p) \wedge cold(p). \tag{2.29}$$

The set of ground atoms that contains both $happy(p)$ and $cold(p)$ violates the con-

[5] http://assat.cs.ust.hk/.
[6] http://www.cs.utexas.edu/users/tag/cmodels.html/.
[7] http://www.dbai.tuwien.ac.at/proj/dlv/.
[8] http://www.tcs.hut.fi/Software/smodels/.
[9] http://www.cs.utexas.edu/users/tag/sup/.
[10] http://potassco.sourceforge.net/.

straint (2.29), and it cannot be an answer set of the program that includes the constraint. In the input language of LPARSE [11], the rule is represented as

```
:- happy(P), cold(P).
```

- *Choice formula.* By a choice formula for a finite set X of ground atoms, we denote the formula

$$\bigwedge_{A \in X}(A \vee \neg A). \tag{2.30}$$

The answer sets of (2.30) are arbitrary subsets of X. For example, consider the rule:

$$(happy(John) \vee \neg happy(John)) \wedge (happy(Peter) \vee \neg happy(Peter)) \leftarrow \neg hot.$$

It means that if the weather is not hot, there are four possible cases:

  - Nobody is happy;
  - John is happy;
  - Peter is happy;
  - John and Peter are happy.

In the input language of LPARSE, the rule is represented as

```
{happy(john), happy(peter)} :- not hot.
```

- *Strong negation.* By strong negation ($\sim$) we represent explicit false. It is useful for representing commonsense law of inertia. For example, consider the rules:

$$alive(t+1) \leftarrow alive(t) \wedge \neg \sim alive(t+1) \tag{2.31}$$

$$\sim alive(t+1) \leftarrow \sim alive(t) \wedge \neg alive(t+1).$$

[11] LPARSE is a front-end for several ASP solvers, such as SMODELS, CMODELS, CLASP, ASSAT. The main role of LPARSE is grounding. http://www.tcs.hut.fi/Software/smodels/.

The first rule means that, for time t, an object is alive at $t+1$ if the object is alive at t and there is no evidence that the object is not alive at $t+1$. Here, $\sim alive(t)$ represents that the object is known to be not alive, while $\neg alive(t)$ represents that the object is not known to be alive. Semantically, strong negation can be eliminated by introducing a new atom. For example, the answer sets of (2.31) have one-to-one correspondence with the answer sets of the following program:

$$\begin{aligned} &alive(t+1) \leftarrow alive(t) \wedge \neg alive'(t+1) \\ &alive'(t+1) \leftarrow alive'(t) \wedge \neg alive(t+1) \\ &\leftarrow alive(t) \wedge alive'(t). \end{aligned} \tag{2.32}$$

$\sim alive(t)$ in (2.31) can be identified with $alive'(t)$ in (2.32). In the input language of LPARSE, (2.31) is represented as

```
alive(T+1) :- alive(T),  not -alive(T+1), time(T).
-alive(T+1) :- -alive(T), not alive(T+1),  time(T).
```

The symbol '–' indicates strong negation.

2.5. New Language of Stable Models

Ferrais, Lee and Lifschitz [4] viewed logic programs as a special class of first-order sentences. For instance, the first-order logic (FOL) representation of the program

$$holdsAt(happy(John), 0)$$

$$holdsAt(happy(p), t) \leftarrow \neg holdsAt(hungry(p), t) \ \wedge \neg holdsAt(cold(p), t)$$

is

$$holdsAt(happy(John), 0) \wedge$$

$$\forall\ p, t((\neg holdsAt(hungry(p), t) \wedge \neg holdsAt(cold(p), t)) \rightarrow holdsAt(happy(p), t)).$$

Given a first-order sentence F, SM$[F;\mathbf{p}]$ is defined as the second-order sentence

$$F \wedge \neg\exists\mathbf{u}((\mathbf{u} < \mathbf{p}) \wedge F^*(\mathbf{u}))$$

where $\mathbf{p}$ is the list of predicate constants $p_1, \ldots, p_n$ called *intensional*, $\mathbf{u}$ is a list of predicate variables $u_1, \ldots, u_n$ that correspond to predicate constants $\mathbf{p}$, and $F^*(\mathbf{u})$ is defined recursively as follows:

- $p_i(t_1, \ldots, t_m)^* = u_i(t_1, \ldots, t_m)$ if p_i belongs to $\mathbf{p}$.

  $p_i(t_1, \ldots, t_m)^* = p_i(t_1, \ldots, t_m)$ otherwise;

- $(t_1 = t_2)^* = (t_1 = t_2)$;

- $\bot^* = \bot$;

- $(F \wedge G)^* = F^* \wedge G^*$;

- $(F \vee G)^* = F^* \vee G^*$;

- $(F \to G)^* = (F^* \to G^*) \wedge (F \to G)$;

- $(\forall x F)^* = \forall x F^*$;

- $(\exists x F)^* = \exists x F^*$.

If predicate constants $\mathbf{p}$ are all predicates occurring in F, we can abbreviate SM$[F;\mathbf{p}]$ as SM$[F]$.

The authors defined that a *stable* model of a first-order sentence F is a model (in the sense of first-order logic) that satisfies SM$[F]$. The new definition of a stable model refers to neither grounding nor fixpoints unlike the traditional semantics [5]. It is similar to the definition of circumscription.

According to [4, Proposition 1], given a logic program Π whose signature contains at least one object constant, an *Herbrand stable model* of F is a model that satisfies $\mathrm{SM}[F; \mathbf{p}]$, where F is the FOL-representation of Π and $\mathbf{p}$ is the list of all predicate constants occurring in F. An Herbrand stable model of F is called an answer set of F.

3. TURNING CIRCUMSCRIPTIVE EVENT CALCULUS INTO ASP

Lee and Palla [13] showed how to turn circumscription into the stable model semantics, and based on this, how to reformulate circumscriptive event calculus as answer set programming. This chapter is a review of the main results of that paper.

3.1. Turning Circumscription into SM

An occurrence of a predicate constant in a formula is *strictly positive* if there is no implication whose antecedent contains the occurrence. For example, consider the formula $(p \to q) \to r$. Since p and q are in the antecedent of the whole implication, only r has a strictly positive occurrence in the formula.

For any set $\mathbf{p}$ of predicate constants and any formulas G and H, the form $G \to H$ is called *canonical implication relative to* $\mathbf{p}$ [13] if the following conditions are satisfied:

- Every predicate constant from $\mathbf{p}$ in G has a strictly positive occurrence in G;
- Every predicate constant from $\mathbf{p}$ in H has a strictly positive occurrence in H.

For instance, $\neg HoldsAt(f,t) \to Happens(f,t)$ is not a canonical implication relative to $\{HoldsAt,\ Happens\}$ because $HoldsAt$ does not have a strictly positive occurrence in the antecedent.

Proposition 1 *[13, Proposition 2] Let F be the universal closure of the conjunction of canonical implications relative to $\mathbf{p}$.*

$$\mathrm{CIRC}[F;\mathbf{p}] \leftrightarrow \mathrm{SM}[F;\mathbf{p}]$$

is logically valid.

3.2. Turning Event Calculus Descriptions into SM

Interestingly, for Σ, Δ $(= \Delta_1 \wedge \Delta_2)$, and Θ defined in Chapter 2, all axioms in Σ can be viewed as canonical implications relative to *Initiates*, *Terminates*, and *Releases*. All axioms in Δ can be viewed as canonical implications relative to *Happens*. All axioms in Θ can be viewed as canonical implications relative to Ab_i.

Proposition 2 *[13, Theorem 2] Given an event calculus domain description defined in Chapter 2, let* $\mathbf{p}$ *be the set of all predicate constants occurring in it, and let* $Choice(\mathbf{p})$ *be the conjunction of choice formulas* $\forall \mathbf{x}(p(\mathbf{x}) \vee \neg p(\mathbf{x}))$ *for every constant* p *in* $\mathbf{p}$ *where* $\mathbf{x}$ *is a list of distinct object variables of the same length as the arity of* p*. The following are equivalent to each other:*

(a) $\text{CIRC}[\Sigma;\ Initiates,\ Terminates,\ Releases] \wedge \text{CIRC}[\Delta;\ Happens] \wedge$
$\text{CIRC}[\Theta;\ Ab_1, \ldots, Ab_n] \wedge \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge E;$

(b) $\text{SM}[\Sigma;\ Initiates,\ Terminates,\ Releases] \wedge \text{SM}[\Delta;\ Happens] \wedge$
$\text{SM}[\Theta;\ Ab_1, \ldots, Ab_n] \wedge \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge E$

(c) $\text{SM}[\Sigma \wedge \Delta \wedge \Theta \wedge \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge E;$
$Initiates,\ Terminates,\ Releases,\ Happens,\ Ab_1, \ldots,\ Ab_n]$

(d) $\text{SM}[\Sigma \wedge \Delta \wedge \Theta \wedge \Omega \wedge \Psi \wedge \Pi \wedge \Gamma \wedge E \wedge$
$Choice(\mathbf{p} \setminus \{Initiates, Terminates, Releases, Happens, Ab_1, \ldots, Ab_n\});\ \mathbf{p}].$

3.3. Turning Event Calculus Descriptions into Answer Set Programs

3.3.1. RASPL-1^M Programs

Many-sorted first-order logic is an extension of first-order logic. Instead of a homogeneous sort in a signature, we specify the sort of each constant, variable, and argument of each

function and predicate constant. Based on the extension, Lee and Palla [13] defined a RASPL-1^M program as a many-sorted extension of a RASPL-1 program from [10]. The underlying signature of a RASPL-1^M program contains an integer sort and integer constants. It also includes arithmetic functions such as $+$, $-$, and comparison operators such as $<$, $\leq$, $>$, $\geq$. An *atomic formula* is an expression of the form $p(t_1, \ldots, t_n)$, equality $(t_1 = t_2)$ or comparison (*e.g.* $t_1 < t_2$) where p is a predicate constant of arity n $(n \geq 0)$ and $t_1, \ldots, t_n$ are terms. A *rule* is of the form

$$A_1 \ ; \ldots ; \ A_k \leftarrow \ A_{k+1}, \ldots, A_m, not \ A_{m+1}, \ldots, not \ A_n, \\ not \ not \ A_{n+1}, \ldots, not \ not \ A_p \tag{3.1}$$

where $0 \leq k \leq m \leq n \leq p$ and A_i is an atomic formula. Using double negations ("*not not*") in (3.1), one can represent choice rules in RASPL-1^M programs [10]. In other words, a choice rule of the form

$$\{A_0\} \leftarrow A_1, \ldots, A_m, not \ A_{m+1}, \ldots, not \ A_n$$

can be understood as shorthand for

$$A_0 \leftarrow A_1, \ldots, A_m, not \ A_{m+1}, \ldots, not \ A_n, not \ not \ A_0.$$

A RASPL-1^M program is a finite set of rules. Let Π be a RASPL-1^M program and let $\sigma(\Pi)$ be the signature that contains object, function and predicate constants occurring in Π. The answer sets of Π are defined as the Herbrand interpretations of $\sigma(\Pi)$ that satisfies SM$[F; \mathbf{p}]$, where F is the FOL-representation of Π and $\mathbf{p}$ is the list of all predicate constants occurring in F.

3.3.2. Turning Event Calculus Descriptions into RASPL-1^M Programs

Definition 1 *For any formulas F, G, H and K, transformations C_1 and C_2 are defined as follows:*

- *C_1 is the transformation that turns an expression of the form $(F \vee G) \wedge H \rightarrow K$ into the conjunction of $F \wedge H \rightarrow K$ and $G \wedge H \rightarrow K$ (eliminating disjunction in the antecedent).*

- *C_2 is the transformation that turns an expression of the form $K \rightarrow (F \wedge G) \vee H$ into the conjunction of $K \rightarrow F \vee H$ and $K \rightarrow G \vee H$ (eliminating conjunction in the consequent).*

The following procedure turns an event calculus description into a RASPL-1^M program. It is slightly modified from the procedure described in [13]. We assume that outermost universal quantifiers in each axiom are already dropped.

Definition 2 (Translation EC2ASP)

1. *Rewrite all definitional axioms of the form*

$$p(\mathbf{x}) \overset{def}{\leftrightarrow} G$$

   as $G \rightarrow p(\mathbf{x})$.

2. *For each axiom, apply C_1 and C_2 repeatedly until there is no further change.*

3. *For each axiom, rewrite it equivalently so that all maximal occurrences of the form $\exists \mathbf{y} G$ are only in the antecedent.*

4. *For each axiom, repeat the following until there is no existential quantifier:*

   (a) *Replace maximal negative occurrences of $\exists \mathbf{y} G(\mathbf{y})$ in the axiom by $G(\mathbf{z})$ where $\mathbf{y}$ is a list of variables and $\mathbf{z}$ is the list of new variables corresponding to $\mathbf{y}$.*

(b) *Replace maximal positive occurrences of* $\exists \mathbf{y} G(\mathbf{x}, \mathbf{y})$ *in the axiom where* $\mathbf{y}$ *is a list of variables and* $\mathbf{x}$ *is the list of all free variables of* $\exists \mathbf{y} G(\mathbf{x}, \mathbf{y})$*, by the formula* $p_G(\mathbf{x})$ *where* p_G *is a new predicate constant, and add the axiom*

$$G(\mathbf{x}, \mathbf{y}) \to p_G(\mathbf{x}) \tag{3.2}$$

5. *Apply* C_1 *to all resulting implications from the previous step until there is no change.*

6. *Turn all resulting formulas into rules of the form:*

   (a) *Replace every* $\wedge$ *by a comma, every* $\vee$ *by a semicolon, and every* $\neg$ *by not;*

   (b) *Rewrite every formula* $Body \to Head$ *to a rule* $Head \leftarrow Body$*.*

7. *Add choice rules* $\{p(\mathbf{x})\}$ *for all predicate constants* $\mathbf{p}$ *except for the following predicate constants:*

   - *Initiates, Terminates, Releases, Happens, and Abnormal;*
   - *All predicate constants* $\mathbf{p}$ *in Step 1;*
   - *All new predicate constants* $\mathbf{p_G}$ *in Step 4(b).*

Step 1 is due to the fact that completion coincides with the stable model semantics for *tight* formulas [13, Proposition 3]. Step 2 removes outermost disjunction in the antecedent and outermost conjunction in the consequent. It turns each axiom into the form

$$F_1 \wedge \ldots \wedge F_n \to F_{n+1} \vee \ldots \vee F_m \tag{3.3}$$

where $0 \leq n \leq m$ and each F_i is an atomic formula or expression of the form $\exists \mathbf{y} G$, possibly followed by $\neg$. Step 3 is to prepare for the existential quantifier elimination in Step 4. Step 4(a) is one of the steps in standard prenex form conversion, which preserves *strong*

equivalence [12, Theorem 2]. Step 4(b) eliminates each occurrence of $\neg\exists\mathbf{y}G(\mathbf{x},\mathbf{y})$ in the antecedent of every implication by introducing new atoms. After this step, some axioms may contain disjunction in the antecedent. Step 5 eliminates them by applying C_1, and the resulting formulas are the form (3.3) where F_i is an atomic formula, possibly followed by $\neg$. In the view of the equivalence between conditions (c) and (d) from Proposition 2, Step 7 extends the list of all intensional predicates to the set of all predicates occurring in the given event calculus description.

Consider the axiom:

$$\begin{aligned} Happens(GraspBananas,t) \rightarrow \neg HoldsAt(HasBananas,t) \wedge HoldsAt(OnBox,t) \\ \wedge \exists location(HoldsAt(At(Bananas,location),t) \\ \wedge HoldsAt(At(Monkey,location),t)). \end{aligned} \tag{3.4}$$

This action precondition axiom is in the Monkey and Bananas domain from [7] and it represents that a monkey can grasp bananas only if the monkey does not have bananas, is on the box, and is in the same location as bananas.

Step 1 does not change (3.4) because the axiom is not a definitional axiom. Step 2 eliminates conjunction in the consequent of the axiom:

$$\begin{aligned} &Happens(GraspBananas,t) \rightarrow \neg HoldsAt(HasBananas,t) \\ &Happens(GraspBananas,t) \rightarrow HoldsAt(OnBox,t) \\ &Happens(GraspBananas,t) \rightarrow \exists location(HoldsAt(At(Bananas,location),t) \wedge \\ &\qquad HoldsAt(At(Monkey,location),t)). \end{aligned} \tag{3.5}$$

Then, we apply Step 3 as follows:

$$Happens(GraspBananas, t) \rightarrow \neg HoldsAt(HasBananas, t) \tag{3.6}$$

$$Happens(GraspBananas, t) \rightarrow HoldsAt(OnBox, t)$$

$$Happens(GraspBananas, t) \wedge \neg\exists location(HoldsAt(At(Bananas, location), t) \wedge HoldsAt(At(Monkey, location), t)) \rightarrow \bot.$$

Next we apply step 4(b): introducing a new predicate constant $newPre$, adding the formula

$$HoldsAt(At(Bananas, location), t) \wedge HoldsAt(At(Monkey, location), t)) \rightarrow newPre(t). \tag{3.7}$$

and replacing the last formula in (3.6) with

$$Happens(GraspBananas, t) \wedge \neg newPre(t) \rightarrow \bot. \tag{3.8}$$

Step 5 does not change the resulting formulas (the first two formulas in (3.6), (3.7), and (3.8)) and step 6 turns them into rules

$$\neg HoldsAt(HasBananas, t) \leftarrow Happens(GraspBananas, t) \tag{3.9}$$

$$HoldsAt(OnBox, t) \leftarrow Happens(GraspBananas, t)$$

$$newPre(t) \leftarrow HoldsAt(At(Bananas, location), t),\ HoldsAt(At(Monkey, location), t)$$

$$\leftarrow Happens(GraspBananas, t),\ not\ newPre(t).$$

In the input language of LPARSE, (3.9) is represented as

```
:- happens(graspBananas,T), not holdsAt(hasBananas,T), time(T).
holdsAt(onBox,T) :- happens(graspBananas,T), time(T).
newPre(T) :- holdsAt(at(bananas,Location),T),
             holdsAt(at(monkey,Location),T), loc(Location), time(T).
:- happens(graspBananas,T), not newPre(T), time(T).
```

As shown above, we need to rewrite the resulting RASPL-1^M program to conform to the input language of LPARSE:

- We add domain predicates for all variables occurring in the rule to the body (e.g. `loc(Location), time(T)`);
- We move equality, comparison, or negated atomic formulas from the head to the body (e.g. $t_1 = t_2 \leftarrow \ldots$ into $\leftarrow \ldots, \ t_1 \neq t_2, \quad t_1 \geq t_2 \leftarrow \ldots$ into $\leftarrow \ldots, \ t_1 < t_2,$ and *not* $p(t) \leftarrow \ldots$ into $\leftarrow \ldots, \ p(t)$).

4. IMPLEMENTATION

This chapter presents *ECASP* - a prototype implementation of the translation *EC2ASP* given in Chapter 3. We compare ECASP with Mueller's work.

4.1. ECASP

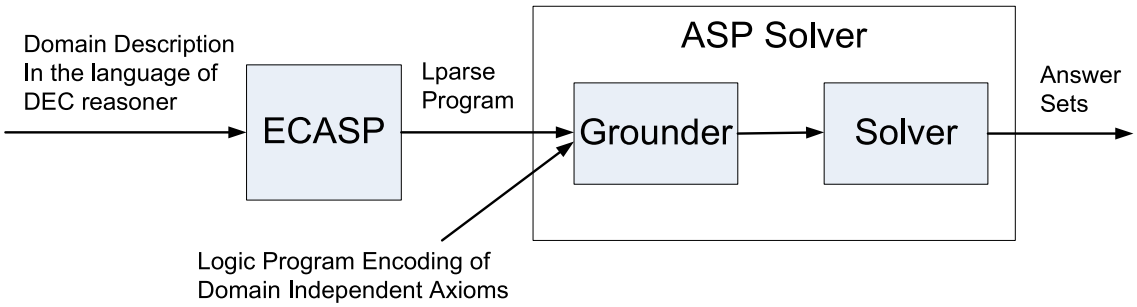

Fig. 2. Architecture of ECASP

ECASP turns an event calculus description in the input language of the DEC reasoner into the input language of LPARSE that is accepted by several ASP solvers. The architecture of ECASP is shown in Figure 2. The system is available from `http://reasoning.eas.asu.edu/ecasp/`.

4.1.1. Procedures

The procedures implemented in ECASP are as follows:

- **eliminateUniQuan**(F) eliminates all universal quantifiers in a formula F. For example, the procedure turns

```
sort agent
.....
[agent2,time]HoldsAt(Happy(agent2),time).
```

in the input language of the DEC reasoner into

```
#domain agent(Agent2), time(Time).

holdsAt(happy(Agent2),Time).
```

According to the convention of the input language of the DEC reasoner [22], variable names are of the form "sortname[number]". The sort name will be used to declare variables in the input language of LPARSE.

- **applyC1(F)** applies the transformation C_1 from Definition 1 in Chapter 3 to formula F. This procedure is used for Step 2 and Step 5 of Definition 2.

- **applyC2(F)** applies the transformation C_2 in Definition 1 to formula F. This procedure is used for Step 2.

- **moveExstQuan(F)** turns the formula of the form (3.3) into a formula such that all maximal occurrences of subformulas of the form $\exists \mathbf{y} G$ occur in the antecedent only. This procedure is used for Step 3.

- **eliminateExstQuan(F)** eliminates all existential quantifiers in F, where F is a form of (3.3) and all existential quantifiers occur only in the antecedent of F. This procedure calls the following subroutines:

  (a) **eliminateNegQuan(G)** removes maximal negative occurrences of $\exists \mathbf{y} G$ in F. This procedure is used for Step 4(a);

  (b) **eliminatePosQuan(G)** removes maximal positive occurrences of $\exists \mathbf{y} G$ in F. This procedure is used for Step 4(b).

- **convertToLP(F)** turns a formula F into a RASPL-1^M rule in the input language of LPARSE. This procedure is used for Step 6.

4.2. Domain Independent Axioms in ECASP

To compute event calculus reasoning in ASP, in addition to the domain description, we need logic program encoding of domain independent axioms, such as DEC, EC, CC ($CC1$ to $CC4$), or EC for events with duration [21, Appendix C] as shown in Figure 2. Since domain independent axioms are used every time, we store logic program encoding in separate files. There are a few versions depending on the choice of domains.

4.2.1. DEC.lp

`DEC.lp` is the logic program encoding of DEC axioms by Mueller (Section 2.3.1):

```
% DEC.lp
#domain fluent(F;F1;F2),event(E),time(T;T1;T2).
time(0..maxstep).
% DEC 1
stoppedIn(T1,F,T2) :- happens(E,T), T1<T, T<T2, terminates(E,F,T).
% DEC 2
startedIn(T1,F,T2) :- happens(E,T), T1<T, T<T2, initiates(E,F,T).
% DEC 3
holdsAt(F2,T1+T2) :- happens(E,T1), initiates(E,F1,T1), 0<T2,
                     trajectory(F1,T1,F2,T2), not stoppedIn(T1,F1,T1+T2),
                     T1+T2<=maxstep.
% DEC 4
holdsAt(F2,T1+T2) :- happens(E,T1), terminates(E,F1,T1), 0<T2,
                     antiTrajectory(F1,T1,F2,T2), not startedIn(T1,F1,T1+T2),
                     T1+T2<=maxstep.
```

```
initiated2(F,T) :- happens(E,T), initiates(E,F,T).

terminated2(F,T) :- happens(E,T), terminates(E,F,T).

released2(F,T) :- happens(E,T), releases(E,F,T).


% DEC 5

holdsAt(F,T+1) :- holdsAt(F,T), not releasedAt(F,T+1), not terminated2(F,T),

                  T<maxstep.

% DEC 6

:- not holdsAt(F,T), not releasedAt(F,T+1), not initiated2(F,T),

   holdsAt(F,T+1), T<maxstep.

% DEC 7

releasedAt(F,T+1) :- releasedAt(F,T),

                     not initiated2(F,T), not terminated2(F,T), T<maxstep.

% DEC 8

:- not releasedAt(F,T), not released2(F,T), releasedAt(F,T+1), T<maxstep.

% DEC 9

holdsAt(F,T+1) :- happens(E,T), initiates(E,F,T), T<maxstep.

% DEC 10

:- happens(E,T), terminates(E,F,T), holdsAt(F,T+1), T<maxstep.

% DEC 11

releasedAt(F,T+1) :- happens(E,T), releases(E,F,T), T<maxstep.

% DEC 12

:- happens(E,T), initiates(E,F,T), releasedAt(F,T+1), T<maxstep.
```

```
:- happens(E,T), terminates(E,F,T), releasedAt(F,T+1), T<maxstep.


{holdsAt(F,T)}.
{releasedAt(F,T)}.
```

The first rule of this file is the declaration of domain predicates such as fluent, event, and time. The next rule specifies that the range of time is from 0 to maxstep. Next, all *DEC* axioms are encoded — three new atoms such as *initiated*2, *terminated*2, and *released*2 are introduced by Step 4(b) in Definition 2. The last two rules come from Step 7.

Using `DEC.lp`, we show how to compute event calculus reasoning in ECASP. The following file (`Yale3-ea.e`) shows the Yale Shooting problem in the input language of ECASP:

```
; Yale3-ea.e
event Load()
event Shoot()
event Sneeze()
fluent Loaded()
fluent Alive()
range time 0 3
range offset 1 1


; Effect Axioms
[time](Initiates(Load(),Loaded(),time)).
[time](HoldsAt(Loaded(),time) -> Terminates(Shoot(),Alive(),time)).
```

```
[time]Terminates(Shoot(),Loaded(),time).

; Inertial Fluents

!ReleasedAt(Loaded(),0).

!ReleasedAt(Alive(),0).

; Initial Condition

HoldsAt(Alive(),0).

!HoldsAt(Loaded(),0).

Happens(Load(),0).


Happens(Sneeze(),1).

Happens(Shoot(),2).
```

The input to ECASP that formalizes the Yale Shooting problem is almost same as the file `Yale3.e` in Chapter 2. The above file can also be an input to the DEC reasoner.

The next file (`Yale3-ea.lp`) is the logic program obtained from the `Yale3-ea.e` file by running ECASP:

```
% Yale3-ea.lp

#domain time(Time).

time(0..3).

offset(1..1).

fluent(loaded).

fluent(alive).

event(load).

event(shoot).

event(sneeze).
```

```
initiates(load,loaded,Time).

terminates(shoot,alive,Time) :- holdsAt(loaded,Time).

terminates(shoot,loaded,Time).

:- releasedAt(loaded,0).

:- releasedAt(alive,0).

holdsAt(alive,0).

:- holdsAt(loaded,0).

happens(load,0).

happens(sneeze,1).

happens(shoot,2).

hide.

show holdsAt(F,T), happens(E,T), happens3(E,T,T2).
```

The first two rules declare `Time` as a variable ranging over the time domain {0,...,3}. Next events and fluents are defined. The next eight rules are generated by ECASP from formulas (2.9),...,(2.16). The next two rules represent that the fluents *loaded* and *alive* should not be subject to the commonsense law of inertia at timepoint 0. `hide` directs ASP solvers not to print out any atoms. The last line (`show`) tells ASP solvers to output atoms of the form `holdsAt(F,T)`, `happens(E,T)`, or `happens3(E,T,T2)` in the answer set.

To run the Yale Shooting domain description, we use

```
lparse -c maxstep=3 DEC.lp Yale3-ea.lp | cmodels.
```

Any grounder that accepts LPARSE input language (e.g. LPARSE or GRINGO) can be substituted for LPARSE. Any solver that accepts LPARSE output (e.g. SMODELS, SUP, CLASP, or CLASPD) can be substituted for CMODELS.

Upon reading the program that consists of `DEC.lp` and `Yale3-ea.lp` as input of LPARSE (see Figure 2), CMODELS produces the following output:

```
cmodels version 3.79 Reading...done
Answer: 1
 Answer set: holdsAt(alive,1) holdsAt(alive,2) holdsAt(loaded,1)
 holdsAt(loaded,2) happens(shoot,2) happens(sneeze,1) happens(load,0)
 holdsAt(alive,0)
Number of loops 0
```

As shown in the answer set, the turkey is alive, the gun is not loaded, and the event *load* happens at timepoint 0. After the sequence of events, the turkey is no longer alive and the gun is no longer loaded at timepoint 3. This answer set is the same as the model returned by the DEC reasoner as shown in Chapter 2. The last line means that the number of loops [17] in the program is zero.

4.2.2. EC.lp

`EC.lp` is the logic program encoding of *EC* axioms whose timepoint sort is a subsort of the real number sort [21]. Mueller [20] proved the equivalence between *EC* and *DEC* where timepoints are the integers. Since ECASP restricts the timepoint sort to the integers, we can use either `EC.lp` or `DEC.lp` to solve event calculus problems using ECASP (for the DEC reasoner, the timepoint sort defined in `EC.e` is also the integer sort). `EC.lp` is represented as

```
% EC.lp
#domain fluent(F;F1;F2), event(E), timepoint(T;T1;T2).
```

```
timepoint(0..maxstep).

% EC 1

clipped(T1,F,T2) :- happens(E,T), T1<=T, T<T2, terminates(E,F,T).

% EC 2

declipped(T1,F,T2) :- happens(E,T), T1<=T, T<T2, initiates(E,F,T).

% EC 3

stoppedIn(T1,F,T2) :- happens(E,T), T1<T, T<T2, terminates(E,F,T).

% EC 4

startedIn(T1,F,T2) :- happens(E,T), T1<T, T<T2, initiates(E,F,T).

% EC 5

holdsAt(F2,T1+T2) :- happens(E,T1), initiates(E,F1,T1), 0<T2,
                     trajectory(F1,T1,F2,T2), not stoppedIn(T1,F1,T1+T2),
                     T1+T2<=maxstep.

% EC 6

holdsAt(F2,T1+T2) :- happens(E,T1), terminates(E,F1,T1), 0<T2,
                     antiTrajectory(F1,T1,F2,T2), not startedIn(T1,F1,T1+T2),
                     T1+T2<=maxstep.

% EC 7

releasedAtBetween(T1,F,T2) :- releasedAt(F,T), T1<T, T<=T2.

persistsBetween(T1,F,T2) :- not releasedAtBetween(T1,F,T2).

% EC 8

releasedBetween(T1,F,T2) :- happens(E,T), T1<=T, T<T2, releases(E,F,T).

% EC 9

holdsAt(F,T2) :- holdsAt(F,T1), T1<T2, persistsBetween(T1,F,T2),
```

```
                not clipped(T1,F,T2).
% EC 10
:- not holdsAt(F,T1), T1<T2, persistsBetween(T1,F,T2),
   not declipped(T1,F,T2), holdsAt(F,T2).
% EC 11
releasedAt(F,T2) :- releasedAt(F,T1), T1<T2, not clipped(T1,F,T2),
                    not declipped(T1,F,T2).
% EC 12
:- not releasedAt(F,T1), T1<T2,
   not releasedBetween(T1,F,T2), releasedAt(F,T2).
% EC 13
releasedIn(T1,F,T2) :- happens(E,T), T1<T, T<T2, releases(E,F,T).
% EC 14
holdsAt(F,T2) :- happens(E,T1), initiates(E,F,T1), T1<T2,
                 not stoppedIn(T1,F,T2), not releasedIn(T1,F,T2).
% EC 15
:- happens(E,T1), terminates(E,F,T1), T1<T2,
   not startedIn(T1,F,T2), not releasedIn(T1,F,T2), holdsAt(F,T2).
% EC 16
releasedAt(F,T2) :- happens(E,T1), releases(E,F,T1),T1<T2,
                    not stoppedIn(T1,F,T2), not startedIn(T1,F,T2).
% EC 17
:- happens(E,T1), initiates(E,F,T1), T1<T2,
   not releasedIn(T1,F,T2), releasedAt(F,T2).
```

```
:- happens(E,T1), terminates(E,F,T1), T1<T2,
   not releasedIn(T1,F,T2), releasedAt(F,T2).


{holdsAt(F,T)}.
{releasedAt(F,T)}.
```

$EC1$ to $EC4$ are definitional axioms for the predicates *Clipped*, *Declipped*, *StoppedIn*, and *StartedIn*. $EC5$ and $EC6$ describe the behavior of trajectories to represent continuous change like falling objects. $EC7$ and $EC8$ are definitional axioms for the predicates *PersistsBetween* and *ReleasedBetween*. Using these two predicates, $EC9$ to $EC12$ describe inertia of *HoldsAt* and *ReleasedAt* to handle truth values of fluents. $EC13$ to $EC17$ describe effects of events on fluents.

To run the Yale Shooting domain description, we use

```
ecasp Yale3-ea.e
gringo -c maxstep=3 EC.lp Yale3-ea.lp | clasp.
```

CLASP produces the same answer set as CMODELS generates using `DEC.lp` in the previous section. Hence, now we can replace `EC.lp` by `DEC.lp`.

4.2.3. ECCausal.lp

`ECCausal.lp` is the logic program encoding of CC, which is used for handling ramification problems. Consider the example of Thielscher's circuit in Chapter 6 from [21] as shown in Figure 3 on the next page. When both switch 1 and switch 2 are closed, the light is on. When both switch 1 and switch 3 are closed, the relay is activated. The relay is connected to switch 2, and switch 2 is opened when the relay is activated. Assume that initially

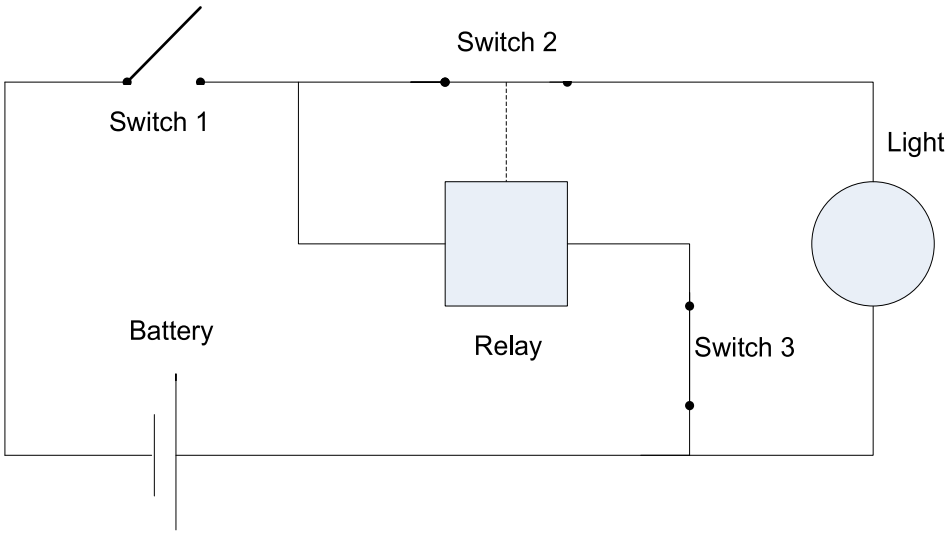

Fig. 3. Thielscher's Circuit

switch 1 is open, switch 2 and switch 3 are closed, the relay is not activated, and the light is off. What happens if we close switch 1? The direct effect of the event is that switch 1 is closed and the relay is activated as an indirect effect. Activating the relay opens switch 2, and finally the light is off. The key to solving this problem is how to describe a number of dependencies among the five fluents; indirect effects of events are instantaneously interactive to each other. This brings the development of causal constraint axioms:

```
% ECCausal.lp
#domain fluent(Fluent), event(Event), time(Time).
% CC 1
started(Fluent,Time) :- holdsAt(Fluent,Time).
started(Fluent,Time) :- {not happens(Event,Time)}0,
                        {not initiates(Event,Fluent,Time)}0.
% CC 2
stopped(Fluent,Time) :- not holdsAt(Fluent,Time).
```

```
stopped(Fluent,Time) :- {not happens(Event,Time)}0,
                        {not terminates(Event,Fluent,Time)}0.
% CC 3
terminated2(Fluent,Time) :- happens(Event,Time), terminates(Event,Fluent,Time).
initiated(Fluent,Time) :- started(Fluent,Time), not terminated2(Fluent,Time).
% CC 4
initiated2(Fluent,Time) :- happens(Event,Time), initiates(Event,Fluent,Time).
terminated(Fluent,Time) :- stopped(Fluent,Time), not initiated2(Fluent,Time).
```

$Started(f,t)$ means that a fluent f is true at timepoint t or is initiated by an event that occurs at t. $Stopped(f,t)$ means that a fluent f is false at timepoint t or is terminated by an event that occurs at t. $Initiated(f,t)$ means that a fluent f is started at timepoint t and the fluent is not terminated by an event that occurs at t. $Terminated(f,t)$ means that a fluent f is stopped at timepoint t and the fluent is not initiated by an event that occurs at t [21].

Thielscher's circuit domain description (`ThielscherCircuit1-ea.e`) is encoded in ECASP as follows:

```
; ThielscherCircuit1-ea.e
; Sort Declaration
sort switch
sort relay
sort light

; Constant Declaration
```

```
switch S1, S2, S3

relay R

light L


event Light(light)

event Close(switch)

event Open(switch)

event Activate(relay)

fluent Lit(light)

fluent Closed(switch)

fluent Activated(relay)


range time 0 1

range offset 1 1


; Causal Constraints

[time](Stopped(Lit(L),time) & Initiated(Closed(S1),time) &

       Initiated(Closed(S2),time) -> Happens(Light(L),time)).

[time](Started(Closed(S2),time) & Initiated(Activated(R),time)

       -> Happens(Open(S2),time)).

[time](Stopped(Activated(R),time) & Initiated(Closed(S1),time) &

       Initiated(Closed(S3),time) -> Happens(Activate(R),time)).

; Effect Axioms

[switch,time]Initiates(Close(switch),Closed(switch),time).
```

```
[switch,time]Terminates(Open(switch),Closed(switch),time).

[relay,time]Initiates(Activate(relay),Activated(relay),time).

[light,time]Initiates(Light(light),Lit(light),time).

; Inertial Fluents

[light]!ReleasedAt(Lit(light),0).

[switch]!ReleasedAt(Closed(switch),0).

[relay]!ReleasedAt(Activated(relay),0).

; Initial Condition

!HoldsAt(Closed(S1),0).

HoldsAt(Closed(S2),0).

HoldsAt(Closed(S3),0).

!HoldsAt(Activated(R),0).

!HoldsAt(Lit(L),0).

Happens(Close(S1),0).


completion Happens
```

When we close switch 1, the fluent $Closed(S1)$ is initiated and the event $Activate(R)$ occurs immediately by the third causal constraint. Consequently, the fluent $Activated(Relay)$ is initiated. The event $Open(S2)$ also occurs instantaneously by the second causal constraint. As the effect of the event $Open(S2)$, the fluent $Closed(S2)$ is terminated. Finally, the event $Light(L)$ does not occur by the first causal constraint and the minimized $Happens$. In conclusion, the event $Close(S1)$ indirectly influences both $Activated(R)$ and $Closed(S2)$, and the fluent $Lit(L)$ depends on all other fluents.

To run Thielscher's circuit domain description, we use

```
ecasp ThielscherCircuit1-ea.e
gringo -c maxstep=1 DEC.lp ECCausal.lp ThielscherCircuit1-ea.lp | clasp.
```

Upon reading the program that consists of `DEC.lp`, `ECCausal.lp` and `ThielscherCircuit1-ea.lp` (obtained from `ThielscherCircuit1-ea.e` by running ECASP) as input of GRINGO, CLASP produces the following output:

```
clasp version 1.2.1
.............
Solving...
Answer: 1
holdsAt(closed(s2),0) holdsAt(closed(s3),0) happens(close(s1),0)
happens(activate(r),0) happens(open(s2),0) holdsAt(closed(s1),1)
holdsAt(closed(s3),1) holdsAt(activated(r),1)
```

As shown in the answer set, switch 1 and switch 3 are closed, and the relay is activated at timepoint 1. However, switch 2 is terminated, and the light is off at the same timepoint.

4.2.4. EC_dur.lp

`EC_dur.lp` is the logic program encoding of *EC* axioms for events with duration. To represent events with duration, we substitute the predicate $Happens3(e, t_1, t_2)$ [27] for $Happens(e, t)$; this predicate represents that an event e occurs between t_1 and t_2 ($t_1 \leq t_2$). Especially, `EC_dur.lp` is used to solve a problem involving compound events that consist of a sequence of sub-events. `EC_dur.lp` is represented as

```
% EC_dur.lp
#domain fluent(F;F1;F2), event(E), timepoint(T;T1;T2;T3;T4).
```

```
timepoint(0..maxstep).
% EC' 1
clipped(T1,F,T4) :- happens3(E,T2,T3), T1<=T3, T2<T4, terminates(E,F,T2).
% EC' 2
declipped(T1,F,T4) :- happens3(E,T2,T3), T1<=T3, T2<T4, initiates(E,F,T2).
% EC' 3
stoppedIn(T1,F,T4) :- happens3(E,T2,T3), T1<T3, T2<T4, terminates(E,F,T2).
% EC' 4
startedIn(T1,F,T4) :- happens3(E,T2,T3), T1<T3, T2<T4, initiates(E,F,T2).
% EC' 5
holdsAt(F2,T2+T3) :- happens3(E,T1,T2), initiates(E,F1,T1), 0<T3,
                     trajectory(F1,T1,F2,T3), not stoppedIn(T1,F1,T2+T3),
                     T2+T3<=maxstep.
% EC' 6
holdsAt(F2,T2+T3) :- happens3(E,T1,T2), terminates(E,F1,T1), 0<T3,
                     antiTrajectory(F1,T1,F2,T3),
                     not startedIn(T1,F1,T2+T3), T2+T3<=maxstep.
% EC' 7 (as same as EC 7)
releasedAtBetween(T1,F,T2) :- releasedAt(F,T), T1<T, T<=T2.
persistsBetween(T1,F,T2) :- not releasedAtBetween(T1,F,T2).
% EC' 8
releasedBetween(T1,F,T4) :- happens3(E,T2,T3), T1<=T3, T2<T4,
                            releases(E,F,T2).
% EC' 9 (as same as EC 9)
```

```
holdsAt(F,T2) :- holdsAt(F,T1), T1<T2, persistsBetween(T1,F,T2),
                 not clipped(T1,F,T2).
% EC' 10 (as same as EC 10)
:- not holdsAt(F,T1), T1<T2, persistsBetween(T1,F,T2),
   not declipped(T1,F,T2), holdsAt(F,T2).
% EC' 11 (as same as EC 11)
releasedAt(F,T2) :- releasedAt(F,T1), T1<T2, not clipped(T1,F,T2),
                    not declipped(T1,F,T2).
% EC' 12 (as same as EC 12)
:- not releasedAt(F,T1), T1<T2,
   not releasedBetween(T1,F,T2), releasedAt(F,T2).
% EC' 13
releasedIn(T1,F,T4) :- happens3(E,T2,T3), T1<T3, T2<T4, releases(E,F,T2).
% EC' 14
holdsAt(F,T3) :- happens3(E,T1,T2), initiates(E,F,T1), T2<T3,
                 not stoppedIn(T1,F,T3), not releasedIn(T1,F,T3).
% EC' 15
:- happens3(E,T1,T2), terminates(E,F,T1), T2<T3,
   not startedIn(T1,F,T3), not releasedIn(T1,F,T3), holdsAt(F,T3).
% EC' 16
releasedAt(F,T3) :- happens3(E,T1,T2), releases(E,F,T1),T2<T3,
                    not stoppedIn(T1,F,T3), not startedIn(T1,F,T3).
% EC' 17
:- happens3(E,T1,T2), initiates(E,F,T1), T2<T3,
```

```
   not releasedIn(T1,F,T3), releasedAt(F,T3).
:- happens3(E,T1,T2), terminates(E,F,T1), T2<T3,
   not releasedIn(T1,F,T3), releasedAt(F,T3).
% EC' 18
:- happens3(E,T1,T2), T1 > T2.
% EC' 19
happens(E,T) :- happens3(E,T,T).


{holdsAt(F,T)}.
{releasedAt(F,T)}.
```

Compared to `EC.lp`, $EC7$, $EC9$, $EC10$, $EC11$, and $EC12$ are unchanged. To define $Happens3$, $EC'18$ and $EC'19$ are added. $Happens(e,t)$ is replaced with $Happens3(e,t_i,t_j)$ in all other EC axioms.

Let's consider the Commuter example [27], which describes a commuter goes to work from home using a compound event. The following file (`Commuter15-ea.e`) represents the Commuter domain description in ECASP:

```
; Commuter15-ea.e
sort place
place Work, Home
sort station: place
station HerneHill, Victoria, SouthKen


fluent At(place)
fluent Train(station, station)
```

```
event WalkTo(place)

event TrainTo(station)

event GoToWork()

range time 0 15

range offset 1 1


; Effect Axioms

[place, time](Initiates(WalkTo(place), At(place), time)).

[place1, place2, time](place1!=place2 & HoldsAt(At(place1),time)

                      -> Terminates(WalkTo(place2), At(place1), time) ).

[station1, station2, time]

(HoldsAt(Train(station1, station2), time) & HoldsAt(At(station1), time)

 -> Initiates(TrainTo(station2), At(station2), time) ).

[station1, station2, time]

(HoldsAt(Train(station1, station2), time) & HoldsAt(At(station1), time)

 -> Terminates(TrainTo(station2), At(station1), time) ).

[time](Initiates(GoToWork(), At(Work), time)).

[place1, time](HoldsAt(At(place1),time) & place1!=Work

               -> Terminates(GoToWork(), At(place1), time)).


; Compound Event

[time1, time2, time3, time4]

(Happens3(WalkTo(HerneHill), time1, time1) &

 Happens3(TrainTo(Victoria), time2, time2) &
```

```
 Happens3(TrainTo(SouthKen), time3, time3) &
 Happens3(WalkTo(Work), time4, time4) &
 time1<time2 & time2<time3 & time3<time4 &
 !Clipped(time1, At(HerneHill), time2) &
 !Clipped(time2, At(Victoria), time3) &
 !Clipped(time3, At(SouthKen), time4)
 -> Happens3(GoToWork(), time1, time4)).

; Train Routes
[time](HoldsAt(Train(HerneHill, Victoria),time)).
[time](HoldsAt(Train(Victoria, SouthKen),time)).

; Inertial Fluents
[place](!ReleasedAt(At(place), 0)).
[station1, station2](!ReleasedAt(Train(station1, station2), 0)).

; State Constraints
[place1, place2, time]
(HoldsAt(At(place1), time) & HoldsAt(At(place2), time) -> place1=place2).
[station1, station2, time]
(HoldsAt(Train(station1, station2), time) -> station1!=station2).
[station1, station2, time](station1!=HerneHill & station1!=Victoria
                           -> !HoldsAt(Train(station1,station2),time)).
[station1, station2, time](station2!=Victoria & station2!=SouthKen
```

```
                          -> !HoldsAt(Train(station1,station2),time)).
[station1, station2, time]
(HoldsAt(Train(station1, station2),time) & station1=HerneHill
 -> station2!=SouthKen).


HoldsAt(At(Home),0).
Happens3(WalkTo(HerneHill), 1, 1).
Happens3(TrainTo(Victoria), 6, 6).
Happens3(TrainTo(SouthKen), 10, 10).
Happens3(WalkTo(Work), 12, 12).
```

The first four lines define sorts and their constants. After effect axioms, the compound event $GoToWork$ is defined in terms of the events $WalkTo$ and $TrainTo$. It ensures that a commuter will be at work after $time4$ if the commuter walks to Hernehill at $time1$, takes a train to Victoria at $time2$, takes a train to Southken at $time3$, and walks to work at $time4$ under the condition that the commuter keeps staying at the same place after every sub-event occurs. The predicate *Clipped* is used to guarantee this condition. That is, using the compound event we can expect the effects of the event that should be consistent with the effects of sub-events [25]. The next two formulas represent train routes from HerneHill to Victoria and from Victoria to SouthKen.

To run the Commuter domain description, we use

```
ecasp Commuter15-ea.e
gringo -c maxstep=15 EC_dur.lp Commuter15-ea.lp | clasp.
```

Upon reading the program that consists of `EC_dur.lp` and `Commuter15-ea.lp` (obtained from `Commuter15-ea.e` by running ECASP) as input of GRINGO, CLASP produces the following output:

```
clasp version 1.2.1

.............

Solving...

Answer: 1

happens3(walkTo(herneHill),1,1) happens3(trainTo(victoria),6,6)

happens3(trainTo(southKen),10,10) happens3(walkTo(work),12,12)

holdsAt(at(home),0) holdsAt(train(herneHill,victoria),0)

holdsAt(train(herneHill,victoria),1) holdsAt(train(herneHill,victoria),2)

holdsAt(train(herneHill,victoria),3) holdsAt(train(herneHill,victoria),4)

................................... ...................................

holdsAt(train(herneHill,victoria),15) holdsAt(train(victoria,southKen),0)

holdsAt(train(victoria,southKen),1) holdsAt(train(victoria,southKen),2)

................................... ...................................

holdsAt(train(victoria,southKen),13) holdsAt(train(victoria,southKen),14)

holdsAt(train(victoria,southKen),15) happens3(goToWork,1,12)

holdsAt(at(home),1) holdsAt(at(herneHill),2)

holdsAt(at(herneHill),3) holdsAt(at(herneHill),4)

holdsAt(at(herneHill),5) holdsAt(at(herneHill),6)

holdsAt(at(victoria),7) holdsAt(at(victoria),8)

holdsAt(at(victoria),9) holdsAt(at(victoria),10)

holdsAt(at(southKen),11) holdsAt(at(southKen),12)
```

```
holdsAt(at(work),13) holdsAt(at(work),14)
holdsAt(at(work),15) happens(walkTo(work),12)
happens(trainTo(victoria),6) happens(trainTo(southKen),10)
happens(walkTo(herneHill),1)
```

As shown in the answer set, the commuter gets to work at timepoint 13 after walking and taking trains. The commuter's place does not change unless the commuter takes the next action.

4.3. Different Encoding from the DEC Reasoner

The input language of ECASP differs from the input language of the DEC reasoner [22] in the following ways:

- The DEC reasoner supports `option` statements. For instance, `option solver minisat` instructs the DEC reasoner to use MINISAT [1] as the SAT solver. However, ECASP ignores this statement;
- The DEC reasoner supports `load` statements that load other supported files such as `DEC.e`, `EC.e` or `ECCausal.e`. However, ECASP ignores this statement (a user loads logic program encodings of domain independent axioms);
- The DEC reasoner supports `noninertial` statements that the specified fluents should not be subject to the commonsense law of inertia at all timepoints. However, ECASP ignores this statement, and we should add `ReleasedAt(f,t)` for non-inertial fluents;
- The DEC reasoner automatically inserts `!ReleasedAt(f,0)` for all fluents that are not defined as non-inertial. However, ECASP does not add the formulas, and we

[1] http://www.minisat.se/.

should add for all initially inertial fluents. For the Yale Shooting problem, we add `!ReleasedAt(Loaded(),0)` and `!ReleasedAt(Alive(),0)`;

- The convention of using parentheses in the input language of ECASP follows the standard convention in first-order logic. For example,

$$\forall x(P(x) \rightarrow Q(x))$$

is encoded instead of

$$\forall x\ P(x) \rightarrow Q(x)$$

in the input language of the DEC reasoner.

- ECASP defines `range` statements of time and offset before axioms in order to check out whether values of time and offset in every axiom are valid.

4.4. ECASP Features

4.4.1. Handling the Full Version of the Event Calculus

The DEC reasoner is not able to handle effect constraints, disjunctive event axioms, and compound events because circumscription is reducible to predicate completion only under certain condition as mentioned in Chapter 2. Consider the following benchmark problems:

- `WalkingTurkey` [27]. This example is an extension of the Yale Shooting problem that introduces a new fluent $Walking$. Assume that initially the gun is loaded, and the turkey is alive and walking. How do we describe the relationship between the fluent $Alive$ and the additional fluent $Walking$ after the event $Shoot$ happens? A possible solution is to use the state constraint

$$HoldsAt(Walking, t) \rightarrow HoldsAt(Alive, t).$$

However, this makes a contradiction in the domain for the following reason: $Walking$ not only holds by $DEC5$ since there is no event that terminates the fluent, but also does not hold by the state constraint. Hence, we need to formalize that $Walking$ is indirectly terminated by $Shoot$. We consider the relationship between these two fluents as a ramification problem, and add the effect constraint

$$Terminates(Shoot, Alive, t) \rightarrow Terminates(Shoot, Walking, t).$$

It describes that if the gun is shot, then the turkey is no longer alive as the direct effect of the event and the turkey is no longer walking as an indirect effect. The DEC reasoner cannot handle this axiom since $Terminates$ occurs in the antecedent, and thus Theorem 1 in Chapter 2 does not apply.

- `BusRide` [26]. This example formalizes an event with nondeterministic effects of a person taking a bus to work. The bus can be either red or yellow:

$$Happens(Board, t) \rightarrow$$
$$Happens(BoardRed, t) \vee Happens(BoardYellow, t).$$

This disjunctive event axiom is not able to be handled by the DEC reasoner. The reason is that the consequent is a disjunctive formula, and Theorem 1 in Chapter 2 does not apply.

- `Commuter`. This example involves the following compound event as described in Section 4.2.4:

$$Happens3(WalkTo(HerneHill), t1, t1) \wedge$$
$$Happens3(TrainTo(Victoria), t2, t2) \wedge$$
$$Happens3(TrainTo(SouthKen), t3, t3) \wedge$$
$$Happens3(WalkTo(Work), t4, t4) \wedge$$
$$t1 < t2 \wedge t2 < t3 \wedge t3 < t4 \ \wedge \neg Clipped(t1, At(HerneHill), t2) \wedge$$
$$\neg Clipped(t2, At(Victoria), t3) \ \wedge \neg Clipped(t3, At(SouthKen), t4)$$
$$\rightarrow Happens3(GoToWork, t1, t4)$$

  Because the circumscribed predicate $Happens3$ is in the antecedent, Theorem 1 in Chapter 2 does not apply. Therefore, the DEC reasoner cannot handle the compound event.

On the other hand, under the stable model semantics, the rules translated from effect constraints, disjunctive event axioms, and compound events by ECASP can be directly handled by answer set solvers. These three examples can be found from the ECASP homepage.

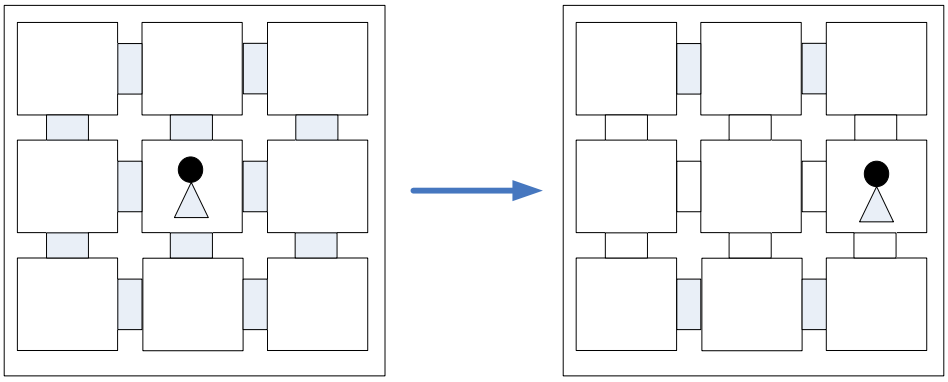

Fig. 4. Robby's Apartment

4.4.2. Allowing to write ASP rules

ECASP allows a user to write ASP rules directly. Consider the example of *Robby's apartment*. The robot Robby is in the apartment that consists of 9 rooms, and 12 doors between adjacent rooms. Initially Robby is in the middle of the apartment and all doors are locked (all colored doors are locked in Figure 4 on the previous page). Robby's goal is to make every room accessible from every other room in the least steps. To achieve the goal, we formalize the transitive closure of the accessibility relation in the input language of ECASP:

```
_asp { accessible(Room1,Room2,Time) :- not holdsAt(locked(Door),Time),
                                       sides(Room1,Room2,Door).
       accessible(Room,Room2,Time) :- accessible(Room,Room1,Time),
                                       accessible(Room1,Room2,Time). }.
```

The rules inside `_asp {...}` are simply copied to the output. The first rule represents that `Room1` is accessible from `Room2` if there exists a door between two rooms and the door is not locked. The second rule represents that `Room` is accessible from `Room2` if `Room` is accessible from `Room1` and `Room1` is accessible from `Room2`. On the other hand, the DEC reasoner is not able to handle transitive closure because it relies on completion.

We also add the rule:

```
_ASP { :- not accessible(Room1,Room2,11). }.
```

This ASP rule represents that any room is accessible from any other room at timepoint 11. The constraint guarantees the achievement of the goal by removing any answer set that does not include every accessibility relation between every pair of rooms in the apartment. The timepoint 11 is the smallest timepoint for completing Robby's assignment.

Next, consider the formula:

```
[room1,room2,door]
 (Sides(room1,room2,door) <->
   (room1=1 & room2=2 & door=D12) | (room1=2 & room2=1 & door=D12) |
   (room1=2 & room2=3 & door=D23) | (room1=3 & room2=2 & door=D23) |
   ..........................  | .......................... |
   (room1=8 & room2=9 & door=D89) | (room1=9 & room2=8 & door=D89)
 ).
```

This formula in the input language of the DEC reasoner is for building the apartment. `Sides(room1,room2,door)` represents that there is the `door` between `room1` and `room2`. Using this formula in ECASP is not efficient regarding to the number of grounding atoms and the time spent. Thus, we use the ASP construction as follows:

```
_asp {
       sides(1,2,d12). sides(2,1,d12). sides(2,3,d23). sides(3,2,d23).
       ............. ............. ............. .............
       sides(7,8,d78). sides(8,7,d78). sides(8,9,d89). sides(9,8,d89).
}.
```

The full domain description of Robby's apartment in the input language of ECASP is in Appendix A.

4.5. Enhancing Output Formats

To enhance the readability of the output by ECASP, we have implemented the system `format-output`. The more readable format is as follows:

```
Answer: 1
0
alive
happens(load,0)
1
+loaded
alive
happens(sneeze,1)
2
loaded
alive
happens(shoot,2)
3
-loaded
-alive
```

For each timepoint, a fluent with a minus sign represents that the fluent is made false, a fluent with a plus sign represents that the fluent is made true, and a fluent without any sign represents that its truth value has not changed with respect to the previous timepoint. To run the system, we use

```
lparse -c maxstep=3 DEC.lp Yale3-ea.lp | cmodels | format-output 3.
```

The last argument `3` represents the maximum timepoint.

5. EXPERIMENTS

We have compared the performance of the following systems on the 14 benchmark problems from [Shanahan, 1997; 1999a] and other problems from [21]:

- The DEC reasoner (v 1.0) running RELSAT (v 2.0),
- ECASP (v 0.9) with LPARSE (v 1.1.1) + CMODELS 3.79 running RELSAT (v 2.0),
- ECASP (v 0.9) with LPARSE (v 1.1.1) + SUP (v 0.4) running MINISAT (v 1.12b),
- ECASP (v 0.9) with GRINGO (v 2.0.3) + CLASP (v 1.2.1) [CLASPD (v 1.1)], and
- ECASP (v 0.9) with CLINGO (v 2.0.3).

ECASP turns an event calculus description in the input language of the DEC reasoner into the input language of LPARSE. The resulting program can be accepted by LPARSE and GRINGO, which eliminates variables by the process of grounding. CMODELS forms the completion of this ground program (for a non-tight ground program, loop formulas are added to its completion), turns the completion into a set of clauses, and finds answer sets by calling a SAT solver. On the other hand, SUP does not form the completion, but applies "non-clausal constraint" mechanism of MINISAT to compute supported models. CLASP computes answer sets with some advanced SAT and ASP solving techniques such as "conflict-driven nogood learning". CLASPD is an extension of CLASP to handle disjunctive logic programs. CLINGO is an integrated system of grounder GRINGO and solver CLASP. All experiments were run on a machine with a 3.00 GHz Intel Pentium D CPU and 2 gigabytes RAM running 64-bit Linux.

- `BusRide` [26] describes the nondeterministic effects of an event $Board$, being on the red bus or being on the yellow bus, using the disjunctive effect axiom. See Section 4.4.1 for details.

- `CoinToss` [27] describes the nondeterministic effects of an event $Toss$, head holding or tails holding, using the *determining fluent* [1] $ItsHeads$ as follows:

$$ReleasedAt(ItsHeads, t)$$
$$HoldsAt(ItsHeads, t) \rightarrow Initiates(Toss, Heads, t)$$
$$\neg HoldsAt(ItsHeads, t) \rightarrow Terminates(Toss, Heads, t).$$

- `ChessBoard` [27] describes the nondeterministic effects of an event $Throw$, a coin being on a white square, a black square, or both on a chess board, using the determining fluents $ItsBlack$ and $ItsWhite$, and the state constraint for their relationship:

$$ReleasedAt(ItsBlack, t)$$
$$ReleasedAt(ItsWhite, t)$$
$$HoldsAt(ItsWhite, t) \rightarrow Initiates(Throw, OnWhite, t)$$
$$HoldsAt(ItsBlack, t) \rightarrow Initiates(Throw, OnBlack, t)$$
$$HoldsAt(ItsWhite, t) \vee HoldsAt(ItsBlack, t).$$

- `Commuter` [27] formalizes a compound event $GoToWork$ that comprises a sequence of sub-events. The concept of events with duration is introduced. See Section 4.2.4 for details.

[1] A *determining fluent* should not be subject to the commonsense law of inertia and is used to decide whether other fluents hold or not by events.

- `DeadOrAlive` [26] extends the Yale Shooting problem with a new fluent *Dead*, which represents that the turkey is not alive using the state constraint:

$$ReleasedAt(Dead, t)$$

$$HoldsAt(Dead, t) \leftrightarrow \neg HoldsAt(Alive, t).$$

- `Happy` [27] describes the indirect effect *Happy* of an event *Feed* using the state constraint:

$$Terminates(Feed(p), Hungry(p), t)$$

$$ReleasedAt(Happy(p), t)$$

$$HoldsAt(Happy(p), t) \leftrightarrow \neg HoldsAt(Hungry(p), t) \wedge \neg HoldsAt(Cold(p), t).$$

- `KitchenSink` [26] describes continuous change using the *Releases* axiom and the *Trajectory* axiom:

$$Releases(TapOn, Height(h), t)$$

$$HoldsAt(Height(h1), t) \wedge h2 = h1 + o \rightarrow Trajectory(Filling, t, Height(h2), o).$$

The first axiom represents that the height is released from the commonsense law of inertia after the tap is turned on. The second axiom represents incremental (continuous) change of the height while the kitchen sink is filled with water.

- `RussianTurkey` [27] extends the Yale Shooting problem with the determining fluent, and describes the nondeterministic effects of an event *Shoot*:

$$Releases(Spin, Loaded, t)$$

$$HoldsAt(Loaded, t) \rightarrow Terminates(Shoot, Alive, t).$$

The event *Sneeze* in the Yale Shooting problem is replaced with the event *Spin*, which makes the gun loaded or unloaded at every timepoint.

- `StolenCar` [26] is a planning problem given the initial state (the owner parks the car) and a final state (the car is not parked). To solve the planning problem, the predicate $Happpens$ is not minimized.

- `StuffyRoom` [26] describes the indirect effect ($Stuffy$) of events ($Start$ and $Close2$) using the state constraint:

$$Initiates(Start, Blocked1, t)$$
$$Initiates(Close2, Blocked2, t)$$
$$ReleasedAt(Stuffy, t)$$
$$HoldsAt(Stuffy, t) \leftrightarrow HoldsAt(Blocked1, t) \wedge HoldsAt(Blocked2, t).$$

- `SupermarketTrolley` [26] represents the new effect $Spinning$ of the concurrent events $Push$ and $Pull$, and represents that one event can cancel the effect of another event using cumulative effect axioms:

$$Happens(Push, t) \rightarrow Initiates(Pull, Spinning, t)$$
$$\neg Happens(Pull, t) \rightarrow Terminates(Push, Backwards, t)$$
$$\neg Happens(Push, t) \rightarrow Terminates(Pull, Forwards, t).$$

- `ThielscherCircuit` [27] represents that indirect effects of events interact to each other without any delay using causal constraints. See Section 4.2.3 for details.

- `WalkingTurkey` [27] extends the Yale Shooting problem with a new fluent $Walking$, and describes the indirect effect $Walking$ of an event $Shoot$ using the effect constraint. See Section 4.4.1 for more details.

- `Yale` [26] is the Yale Shooting problem and represents the commonsense law of inertia using the effect axioms. See Section 2.3.2 for details.

5.2. Other Problems

- `FallingObjectwithAntiTrajectory` describes continuous change using the *Trajectory* axiom and the *AntiTrajectory* axiom:

$$ReleasedAt(Height(obj, h), t) \tag{5.1}$$

$$HoldsAt(Height(obj, h1), t) \wedge h2 = h1 - o \rightarrow Trajectory(Falling(obj), t, Height(obj, h2), o)$$

$$HoldsAt(Height(obj, h), t) \rightarrow AntiTrajectory(Falling(obj), t, Height(obj, h), o). \tag{5.2}$$

(5.2) represents that the height of the object does not change after the object stops falling down (actually, the object hits the ground).

- `FallingObjectwithEvents` describes continuous change using the *Release* axiom and the *Trajectory* axiom:

$$Releases(Drop(a, obj), Height(obj, h), t) \tag{5.3}$$

$$HoldsAt(Height(obj, h1), t) \wedge h2 = h1 - o \rightarrow Trajectory(Falling(obj), t, Height(obj, h2), o)$$

$$HoldsAt(Height(obj, h), t \rightarrow Initiates(HitGround(obj), Height(obj, h), t). \tag{5.4}$$

Instead of (5.1), (5.3) represents that the height of the object should not be subject to the commonsense law of inertia. (5.2) is replaced with (5.4) to represent that the height does not change after the object hits the ground.

- `HotAirBalloon` describes continuous change using the $Trajectory$ axiom and the $AntiTrajectory$ axiom:

$$ReleasedAt(Height(b,h),t)$$
$$HoldsAt(Height(b,h1),t) \wedge h2 = h1 + o$$
$$\rightarrow Trajectory(HearterOn(b),t,Height(b,h2),o)$$
$$HoldsAt(Height(b,h1),t) \wedge h2 = h1 - o$$
$$\rightarrow AntiTrajectory(HeaterOn(b),t,Height(b,h2),o).$$

  While both (5.2) and (5.4) makes the height unchanged (zero), the above $AntiTrajectory$ axiom represents decremental change of the height. This results in multiple models.

- `Telephone` describes the effects of events occurred from using a telephone. The positive and negative effect axioms are used:

$$HoldsAt(Idle(p),t) \rightarrow Initiates(PickUp(a,p),DialTone(p),t)$$
$$HoldsAt(Connected(p1,p2),t)$$
$$\rightarrow Terminates(SetDown(a,p2),Connected(p1,p2),t).$$

5.3. Analysis

As described in Section 4.4.1 and the next tables, the DEC reasoner solves 11 of the 14 benchmark problems, while ECASP solves all the problems: the DEC reasoner is not able to solve `BusRide`, which includes the disjunctive effect axiom, `Commuter` which includes the compound event, and `WalkingTurkey` which includes the effect constraint. ECASP turns disjunctive effect axioms into disjunctive rules, which can be handled by neither SUP nor CLINGO, but can be done by CMODELS and CLASPD. See `BusRide` and `ChessBoard`.

Table II. Results on Benchmark Problems (a)

Problem (max. step)	DEC reasoner	ECASP with LPARSE + CMODELS	ECASP with LPARSE + SUP	ECASP with GRINGO + CLASP(D)	ECASP with CLINGO
BusRide (2)	Can't handle Disjunctive Effect Axioms	0.05 (0.04+0.01) A: 29 / C: 31 R: 180	Can't handle Disjunctive Rules	0.02 (0.02+0.00) A: 83 R: 115	Can't handle Disjunctive Rules
BusRide (40)	Can't handle Disjunctive Effect Axioms	7.98 (7.24+0.74) A: 709 / C: 714 R: 130481	Can't handle Disjunctive Rules	0.47 (0.38+0.09) A: 3408 R: 51705	Can't handle Disjunctive Rules
ChessBoard (2)	0.00 (0.00+0.00) A:27 / C:52	0.03 (0.02+0.01) A: 38 / C: 55 R: 103	Can't handle Disjunctive Rules	0.02 (0.02+0.00) A: 65 R: 84	Can't handle Disjunctive Rules
ChessBoard (40)	0.10 (0.10+0.00) A:369 / C:812	0.14 (0.10+0.04) A: 736 / C: 1391 R: 1939	Can't handle Disjunctive Rules	0.04 (0.03+0.01) A: 1483 R: 1730	Can't handle Disjunctive Rules
CoinToss (4)	0.00 (0.00+0.00) A:25 / C:66	0.03 (0.02+0.01) A: 58 / C: 109 R: 155	0.03 (0.02+0.01) A: 88 / C: 0 R: 155	0.02 (0.02+0.00) A: 82 R: 119	0.02
CoinToss (40)	0.00 (0.00+0.00) A:205 / C:606	0.37 (0.21+0.16) A: 1462 / C: 3646 R:4691	0.25 (0.21+0.04) A: 2860 / C: 0 R: 4691	0.06 (0.05+0.01) A: 2782 R: 4476	0.05
Commuter (15)	Can't handle Compound Events	572.78 (519.43+53.35) A: 5159 / C: 5198 R: 8733799	570.01 (519.43+50.58) A: 29282 / C: 0 R: 8733799	52.61 (44.45+8.16) A: 26652 R: 6520825	31.71
DeadOrAlive (3)	0.00 (0.00+0.00) A:36 / C:76	0.03 (0.02+0.01) A: 77 / C: 0 R: 140	0.03 (0.02+0.01) A: 77 / C: 0 R: 140	0.02 (0.02+0.00) A: 70 R: 98	0.02
DeadOrAlive (40)	0.10 (0.10+0.00) A:369 / C:890	0.20 (0.18+0.02) A: 2574 / C: 0 R: 3978	0.19 (0.18+0.01) A: 2574 / C: 0 R: 3978	0.05 (0.04+0.01) A: 2533 R: 3652	0.04
Happy (2)	0.00 (0.00+0.00) A:24 / C:46	0.03 (0.03+0.00) A: 38 / C: 0 R: 77	0.03 (0.03+0.00) A: 38 / C: 0 R: 77	0.03 (0.02+0.01) A: 38 R: 63	0.02
Happy (40)	0.10 (0.10+0.00) A:328 / C:730	0.09 (0.08+0.01) A: 754 / C: 0 R: 1401	0.09 (0.08+0.01) A: 754 / C: 0 R: 1401	0.04 (0.04+0.00) A: 754 R: 1197	0.04
KitchenSink (5)	0.20 (0.20+0.00) A:102 / C:593	0.34 (0.31+0.03) A: 1344 / C: 0 R: 3648	0.34 (0.31+0.03) A: 1344 / C: 0 R: 3648	0.05 (0.05+0.00) A: 1158 R:1828	0.05
KitchenSink (25)	71.90 (71.50+0.40) A:1014 / C:12109	43.24 (37.79+5.45) A: 121580 / C: 0 R: 480146	43.39 (37.79+5.60) A: 121580 / C: 0 R: 480146	2.43 (1.70+0.73) A: 114968 R: 179195	1.96
KitchenSink (40)	12257.2 (204.29m) (12251.8+5.4) A: 2419 / C: 40199	270.74 (228.40+42.34) A: 701155 / C: 0 R: 2908221	271.52 (228.40+43.12) A: 701155 / C: 0 R: 2908221	14.88 (9.91+4.97) A: 676343 R: 1042040	12.26
Russian Turkey (4)	0.00 (0.00+0.00) A:35 / C:87	0.03 (0.02+0.01) A: 55 / C: 64 R: 132	0.02 (0.02+0.00) A: 85 / C: 0 R: 132	0.02 (0.02+0.00) A: 82 R: 110	0.02
Russian Turkey (40)	0.10 (0.10+0.00) A:287 / C:807	0.18 (0.13+0.05) A: 1171 / C: 1513 R: 3021	0.15 (0.13+0.02) A: 1966 / C: 0 R: 3021	0.04 (0.04+0.00) A: 1963 R: 2800	0.03
StolenCar (2)	0.00 (0.00+0.00) A:10 / C:20	0.02 (0.01+0.01) A: 21 / C: 26 R: 51	0.02 (0.01+0.01) A: 37 / C: 0 R: 51	0.01 (0.01+0.00) A: 30 R: 43	0.01
StolenCar (40)	0.00 (0.00+0.00) A:162 / C:364	2.38 (0.86+1.52) A: 1917 / C: 15192 R: 22056	1.00 (0.86+0.14) A: 1975 / C: 0 R: 22056	0.14 (0.11+0.03) A: 1930 R: 17947	0.10
StuffyRoom (2)	0.00 (0.00+0.00) A:27 / C:57	0.03 (0.02+0.01) A: 47 / C: 0 R: 86	0.04 (0.02+0.02) A: 47 / C: 0 R: 86	0.02 (0.02+0.00) A: 47 R: 68	0.02
StuffyRoom (40)	0.10 (0.10+0.00) A:369 / C:931	0.10 (0.09+0.01) A: 1464 / C: 0 R: 2111	0.10 (0.09+0.01) A: 1464 / C: 0 R: 2111	0.05 (0.03+0.02) A: 1464 R: 1902	0.04

A: number of atoms, C: number of clauses, R: number of ground rules

Table III. Results on Benchmark Problems (b)

Problem (max. step)	DEC reasoner	ECASP with LPARSE + CMODELS	ECASP with LPARSE + SUP	ECASP with GRINGO + CLASP(D)	ECASP with CLINGO
Supermarket (12)	0.00 (0.00+0.00) A:104 / C:752	0.05 (0.05+0.00) A: 315 / C: 0 R: 464	0.05 (0.05+0.00) A: 315 / C: 0 R: 464	0.03 (0.03+0.00) A: 315 R: 458	0.03
Supermarket (40)	0.10 (0.10+0.00) A:328 / C:2488	0.12 (0.11+0.01) A: 1969 / C: 0 R: 2454	0.11 (0.11+0.00) A: 1969 / C: 0 R: 2454	0.05 (0.05+0.00) A: 1969 R: 2448	0.05
Thielscher Circuit (1)	0.10 (0.10+0.00) A:312 / C:922	0.06 (0.05+0.01) A: 306 / C: 0 R: 529	0.06 (0.05+0.01) A: 306 / C: 0 R: 529	0.03 (0.03+0.00) A: 108 R: 140	0.03
Thielscher Circuit (20)	14.10 (13.80+0.30) A:5138 / C:16122	0.54 (0.50+0.04) A: 3916 / C: 0 R: 10979	0.53 (0.50+0.03) A: 3916 / C: 0 R: 10979	0.08 (0.07+0.01) A: 1743 R: 5669	0.08
Thielscher Circuit (40)	54.90 (54.40+0.50) A:10218 / C:32122	2.56 (2.31+0.25) A: 9276 / C: 0 R: 53699	3.35 (2.31+1.04) A: 9276 / C: 0 R: 53699	0.32 (0.23+0.09) A: 4812 R: 35458	0.21
Walking Turkey (3)	Can't handle Effect Constraints	0.02 (0.02+0.00) A: 64 / C: 0 R: 102	0.03 (0.02+0.01) A: 64 / C: 0 R: 102	0.02 (0.02+0.00) A: 64 R: 96	0.02
Walking Turkey (40)	Can't handle Effect Constraints	0.11 (0.10+0.01) A: 1664 / C: 0 R: 2035	0.11 (0.10+0.01) A: 1664 / C: 0 R: 2035	0.04 (0.04+0.00) A: 1664 R: 2029	0.04
Yale (3)	0.00 (0.00+0.00) A:28 / C:64	0.02 (0.02+0.00) A: 59 / C: 0 R: 95	0.02 (0.02+0.00) A: 59 / C: 0 R: 95	0.02 (0.01+0.01) A: 56 R: 78	0.01
Yale (40)	0.10 (0.10+0.00) A:287 / C:767	0.13 (0.12+0.01) A: 1873 / C: 0 R: 2729	0.13 (0.12+0.01) A: 1873 / C: 0 R: 2729	0.04 (0.03+0.01) A: 1872 R: 2613	0.03

A: number of atoms, C: number of clauses, R: number of ground rules

Table IV. Results on Other Problems (a)

Problem (max. step)	DEC reasoner	ECASP w/ LPA + CMO	ECASP w/ LPA + SUP	ECASP w/ GRI + CLA	ECASP w/ CLI
Falling w/ AntiTraj(5)	1.60 (1.60+0.00) A: 72 / C: 199	0.08 (0.08+0.00) A: 398 / C: 0 R: 630	0.08 (0.08+0.00) A: 398 / C: 0 R: 630	0.04 (0.04+0.00) A: 290 R: 437	0.04
Falling w/ AntiTraj(15)	272.20 (271.40+0.80) A: 416 / C: 3056	0.76 (0.68+0.08) A: 4989 / C:0 R: 9877	0.75 (0.68+0.07) A: 4989 / C: 0 R: 9877	0.10 (0.10+0.00) A: 3791 R: 7172	0.10
Falling w/ AntiTraj(25)	Failed (24.3 minutes)	3.68 (3.00+0.68) A: 20773 / C: 0 R: 44285	3.70 (3.00+0.70) A: 20773 / C: 0 R: 44285	0.45 (0.33+0.12) A: 16405 R: 33212	0.36
Falling w/ Events(5)	0.10 (0.10+0.00) A: 72 / C: 311	0.17 (0.16+0.01) A: 780 / C: 0 R: 1554	0.17 (0.16+0.01) A: 780 / C: 0 R: 1554	0.05 (0.05+0.00) A: 660 R: 1064	0.05
Falling w/ Events(15)	8.00 (7.90+0.10) A: 416 / C: 3221	4.94 (4.41+0.53) A: 21810 / C: 0 R: 54263	4.95 (4.41+0.54) A: 21810 / C: 0 R: 54263	0.45 (0.33+0.12) A: 20103 R: 31490	0.36
Falling w/ Events(25)	109.20 (108.80+0.40) A: 1092 / C: 12351	36.44 (30.98+5.46) A: 147920 / C: 0 R: 388239	36.43 (30.98+5.45) A: 147920 / C: 0 R: 388239	3.00 (2.06+0.94) A: 140700 R: 213684	2.38
Falling w/ Events(40)	Failed (25.2 minutes)	318.23 (256.98+61.25) A: 1201885 / C: 0 R: 3146363	320.35 (256.98+63.37) A: 1201885 / C: 0 R: 3146363	25.13 (16.40+8.73) A: 1169328 R: 1676615	21.10

A: number of atoms, C: number of clauses, R: number of ground rules

Table V. Results on Other Problems (b)

Problem (max. step)	DEC reasoner	ECASP w/ LPA + CMO	ECASP w/ LPA + SUP	ECASP w/ GRI + CLA	ECASP w/ CLI
HotAir w/ Balloon(3)	0.40 (0.40+0.00) A: 40 / C: 75	0.06 (0.05+0.01) A: 95 / C: 0 R: 186	0.06 (0.05+0.01) A: 95 / C: 0 R: 186	0.04 (0.04+0.00) A: 94 R: 133	0.04
HotAir w/ Balloon(15)	61.50 (61.50+0.00) A: 288 / C: 1163	0.21 (0.19+0.02) A: 489 / C: 678 R: 2446	0.20 (0.19+0.01) A: 1139 / C: 0 R: 2446	0.07 (0.06+0.01) A: 1137 R: 1909	0.06
HotAir w/ Balloon(20)	242.70 (242.70+0.00) A: 462 / C: 2422	0.67 (0.33+0.34) A: 870 / C: 1302 R: 4830	0.69 (0.33+0.36) A: 2256 / C: 0 R: 4830	0.09 (0.08+0.01) A: 2254 R: 3916	0.08
HotAir w/ Balloon(25)	10731.7 (178.9 m) (10730.5+1.2) A: 780 / C:5949	0.81 (0.73+0.08) A: 1127 / C: 1361 R: 11212	0.80 (0.73+0.07) A: 5261 / C: 0 R: 11212	0.15 (0.13+0.02) A: 5259 R: 9569	0.13
HotAir w/ Balloon(40)	Failed (24.8 minutes)	3.04 (2.80+0.24) A: 20079 C: 0 R: 42102	3.10 (2.80+0.30) A: 20079 / C: 0 R: 42102	0.48 (0.39+0.09) A: 20078 R: 37876	0.38
Telephone (3)	0.50 (0.50+0.00) A: 461 / C:3184	0.28 (0.28+0.00) A: 861 / C: 0 R: 1452	0.28 (0.28+0.00) A: 861 / C: 0 R: 1452	0.19 (0.19+0.00) A: 840 R: 1272	0.19
Telephone (40)	18.20 (17.70+0.50) A: 5419 / C:41590	1.80 (1.63+0.17) A: 21354 / C: 0 R: 27217	1.79 (1.63+0.16) A: 21354 / C: 0 R: 27217	0.48 (0.42+0.06) A: 21333 R: 27037	0.42

A: number of atoms, C: number of clauses, R: number of ground rules

5.3.1. Time

The reported times for the DEC reasoner are the sum of encoding time and SAT solving time, which are provided by the system. On the other hand, we use the Linux `time` command to measure the sum of the time spent by ECASP for translating an event calculus description into a logic program, the time spent by a grounder, and the time spent by an ASP solver. All reported times are the times for finding one model.

As we solve most of problems within one second, we increase the timepoints to observe more significant differences in performance. Indeed, for `KitchenSink` and `ThielscherCircuit` of the benchmark problems and all of other problems, the time differences between the DEC reasoner and ECASP with ASP solvers are more notable as the timepoints become larger. Especially, the grounding times of LPARSE and GRINGO are faster than those of the DEC reasoner. As we see the results in all tables, ECASP with CLINGO computes the fastest. However, when the timepoints increase, the DEC reasoner

fails to solve some problems such as `FallingObjectWithAntiTrajectory` at timepoint 25, `FallingObjectWithEvents` at timepoint 40, and `HotAirBalloon` at timepoint 40.

5.3.2. Atom

The atoms such as $Initiates(e, f, t)$, $Terminates(e, f, t)$, $Releases(e, f, t)$, and $Trajectory(f_1, t_1, f_2, t_2)$ produce many ground instances. To avoid this problem, the DEC reasoner adopts the "intelligent" encoding method given in [20]. Compared to the other systems, the DEC reasoner generates much less number of atoms in most of problems.

5.3.3. Clause

For some problems, CMODELS does not generate a clause. The reason is that the preprocessing of CMODELS finds out that the well-founded model is the unique answer set, and thus CMODELS does neither clausify the given program nor invoke a SAT solver. For all problems, SUP does not generate a clause.

Table VI. Results on Thielsher's Circuit with `option renaming on` and `off`

Example	option renaming on	option renaming off
Thielscher Circuit (1)	0.10 (0.10+0.00) A: 312 / C: 922	0.20 (0.20+0.0) A: 68 / C: 1114
Thielscher Circuit (20)	14.10 (13.80+0.30) A: 5138 / C: 16122	5.30 (5.00+0.30) A: 714 / C: 21254
Thielscher Circuit (40)	54.90 (54.40+0.50) A: 10218 / C: 32122	11.20 (10.60+0.60) A: 1394 / C: 42454

A: number of atoms, C: number of clauses

We notice that adding the `option` statement to Thielscher's circuit example affects the experimental results when executed using the DEC reasoner. By default, the value of the `option` statement is `on` and this statement (`option renaming on`) renames sub-formulas of the original formula by introducing new atoms so that clausifying the original formula does not blow up exponentially. If we add `option renaming off` explicitly to that example, then the number of atoms is dramatically reduced, but the number of clauses is increased as shown in Table VI. On the other hand, the experimental results of all the remaining examples do not change even though we use this `option` statement.

6. CONCLUSION

We have implemented an ASP-based event calculus reasoner ECASP. The following are the main achievements:

- The ASP approach is able to handle all axioms of the event calculus with the certain condition that the domain is closed and finite, while the SAT-based approach does not;
- The ASP approach is able to compute certain benchmark problems faster than the SAT-based approach does.

REFERENCES

[1] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[2] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. TAL: Temporal action logics language specification and tutorial.[1] *Linköping Electronic Articles in Computer and Information Science ISSN 1401-9841*, 3(015), 1998.

[3] Semra Doğandağ, Paolo Ferraris, and Vladimir Lifschitz. Almost definite causal theories. In *Proc. 7th Int'l Conference on Logic Programming and Nonmonotonic Reasoning*, pages 74–86, 2004. Extended version with proofs[2].

[4] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 372–379, 2007.

[5] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[6] Michael Gelfond and Vladimir Lifschitz. Action languages.[3] *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998.

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

[8] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[1] `http://www.ep.liu.se/ea/cis/1998/015/ .`
[2] `http://www.cs.utexas.edu/users/otto/papers/adct.ps`
[3] `http://www.ep.liu.se/ea/cis/1998/016/ .`

[9] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[10] 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.

[11] Joohyung Lee and Fangzhen Lin. Loop formulas for circumscription. *Artificial Intelligence*, 170(2):160–185, 2006.

[12] Joohyung Lee and Ravi Palla. Yet another proof of the strong equivalence between propositional theories and logic programs. In *Working Notes of the Workshop on Correspondence and Equivalence for Nonmonotonic Theories*, 2007.

[13] Joohyung Lee and Ravi Palla. Classical logic event calculus as answer set programming. In *Working Notes of the Workshop on Answer Set Programming and Other Computing Paradigms*, 2008.

[14] Vladimir Lifschitz. Circumscription. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in AI and Logic Programming*, volume 3, pages 298–352. Oxford University Press, 1994.

[15] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597, 2008.

[16] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[17] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, 2004.

[18] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980.

[19] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[20] Erik T. Mueller. Event calculus reasoning through satisfiability. *J. Log. Comput.*, 14(5):703–730, 2004.

[21] Erik T. Mueller. *Commonsense reasoning*. Elsevier, 2006.

[22] Erik T. Mueller. Discrete event calculus reasoner documentation, 2007.

[23] Erik T. Mueller. Event calculus. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 671–708. Elsevier, 2008.

[24] Murray Shanahan. A circumscriptive calculus of events. *Artif. Intell.*, 77(2):249–284, 1995.

[25] Murray Shanahan. Event calculus planning revisited. In *Proceedings 4th European Conference on Planning (ECP 97), Springer Lecture Notes in Artificial Intelligence no. 1348*, pages 390–402. Springer, 1997.

[26] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[27] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today*, LNCS 1600, pages 409–430. Springer, 1999.

[28] Murray Shanahan. The ramification problem in the event calculus. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 140–146, 1999.

[29] Murray Shanahan and Mark Witkowski. Event calculus planning through satisfiability. *J. Log. Comput.*, 14(5):731–745, 2004.

APPENDIX A

THE DOMAIN DESCRIPTION OF ROBBY'S APARTMENT IN THE INPUT LANGUAGE OF ECASP

```
; Apartment-ea.e
sort room: integer
sort door
door D12, D23, D14, D25, D36, D45, D56, D47, D58, D69, D78, D89

fluent Locked(door)
fluent InRoom(room)
event Lock(door)
event Unlock(door)
event Go(room)

; ECASP will pass:
predicate Sides(room,room,door)
predicate Accessible(room,room,time)

range room 1 9
range offset 1 1
range time 0 11

_asp {
       sides(1,2,d12). sides(2,1,d12). sides(2,3,d23). sides(3,2,d23).
       sides(1,4,d14). sides(4,1,d14). sides(2,5,d25). sides(5,2,d25).
       sides(3,6,d36). sides(6,3,d36). sides(4,5,d45). sides(5,4,d45).
       sides(5,6,d56). sides(6,5,d56). sides(4,7,d47). sides(7,4,d47).
```

```
       sides(5,8,d58). sides(8,5,d58). sides(6,9,d69). sides(9,6,d69).
       sides(7,8,d78). sides(8,7,d78). sides(8,9,d89). sides(9,8,d89).
}.

; 1) 'go' event
; [Effect axioms]
[room,time]
Initiates(Go(room),InRoom(room),time).

[room1,room2,time]
(HoldsAt(InRoom(room1),time)
-> Terminates(Go(room2),InRoom(room1),time)).

; [Action preconditions]
; The robot cannot go to the same room in which it stays.
[room,time]
(Happens(Go(room),time) -> !HoldsAt(InRoom(room),time)).

; The robot can go through only an unlocked door.
[room2,time]
(Happens(Go(room2),time) ->
 {door,room1}(Sides(room1,room2,door) &
              !HoldsAt(Locked(door),time) &
               HoldsAt(InRoom(room1),time))
```

```
).

; 2) ‘lock’ and ‘unlock’ event
; [Effect axioms]
[door,time]
Initiates(Lock(door),Locked(door),time).

[door,time]
Terminates(Unlock(door),Locked(door),time).

; [Action preconditions]
; The robot can lock the door between room1 and room2
; only if the robot is in either room1 or room2.
[door,time]
(Happens(Lock(door),time) ->
  {room1,room2}(Sides(room1,room2,door) &
               (HoldsAt(InRoom(room1),time) | HoldsAt(InRoom(room2),time)))
).

; The robot can unlock the door between room1 and room2
; only if the robot is in either room1 or room2.
[door,time]
(Happens(Unlock(door),time) ->
  {room1,room2}(Sides(room1,room2,door) &
```

```
                (HoldsAt(InRoom(room1),time) | HoldsAt(InRoom(room2),time)))
).

; Event occurrence constraints: No concurrent events
[event1,event2,time]
( Happens(event1,time) & Happens(event2,time) -> event1=event2 ).

; [State constraints]
; The robot can be in one room at a timepoint.
[room1,room2,time]
(HoldsAt(InRoom(room1),time) & HoldsAt(InRoom(room2),time)
-> room1=room2).

; Initial condition
; Robby is in the room #5
HoldsAt(InRoom(5),0).

; All the doors are locked.
[door]HoldsAt(Locked(door),0).

; Any two different rooms are not accessible to each other.
[room1,room2]
(room1!=room2 -> !Accessible(room1,room2,0)).
```

```
[door]!ReleasedAt(Locked(door),0).

[room]!ReleasedAt(InRoom(room),0).


; A room is accessible to itself.

[room,time]Accessible(room,room,time).


; _ASP // ECASP allows both upper and lower case letters.

; Transitive closure

_asp {

       accessible(Room1,Room2,Time) :- not holdsAt(locked(Door),Time),

                                       sides(Room1,Room2,Door).


       accessible(Room,Room2,Time) :- accessible(Room,Room1,Time),

                                      accessible(Room1,Room2,Time).

     }.


; Our goal: every room is accessible to every other.

_ASP {

       :- not accessible(Room1,Room2,11).

     }.
```