CHAPTER

14 Applications

The availability of techniques for commonsense reasoning presents an enormous opportunity in that there are many areas that have only begun to be explored that can benefit from the use of commonsense reasoning. Commonsense reasoning provides two important benefits to computer applications:

- Commonsense reasoning can be used to give applications a greater degree of understanding about the human world. Users can be freed from having to state everything explicitly because commonsense reasoning can be used to infer what is obvious to a person.
- Commonsense reasoning can be used to give computer programs more flexibility. An application can adapt to an unexpected real-world problem by reasoning about it in order to come up with a solution.

In this chapter, we describe several applications that have been built using commonsense reasoning in the event calculus, in business systems, natural language understanding, and vision.

14.1 BUSINESS SYSTEMS

An important area of application of the event calculus is business systems. In this section, we review two areas in which the event calculus has been applied: payment protocols and workflow modeling.

14.1.1 PAYMENT PROTOCOLS

Pınar Yolum and Munindar P. Singh have used the event calculus to incorporate commonsense knowledge and reasoning about *commitments* into electronic payment systems. Using Shanahan's (2000) event calculus planner, they have implemented a mechanism that enables customer and merchant applications to (1) keep track of their commitments, (2) determine what actions are currently possible, and (3) plan actions to achieve goals.

Instead of describing a payment protocol as a finite state machine (FSM), as is traditionally done, the commitments inherent in the protocol are extracted and described using the event calculus. Applications that implement the protocol can then

use the event calculus description to reason about their commitments. This makes the applications more flexible than those based on FSMs. An FSM describes only which actions are currently possible, not why they are possible. By using commonsense reasoning to reason about commitments, applications can generate their own plans for achieving their goals and cope with unanticipated situations.

NetBill protocol

The mechanism is implemented for the NetBill protocol, which is used for the online purchase of digital products. This protocol works as follows:

1. A customer requests a price quote for a product from a merchant.
2. The merchant provides a price quote to the customer.
3. The customer sends a purchase request to the merchant.
4. The merchant delivers the product, in an encrypted form, to the customer.
5. The customer sends an electronic payment order (EPO) to the merchant.
6. The merchant sends the EPO to the NetBill server.
7. The NetBill server debits the customer's account and credits the merchant's account.
8. The NetBill server sends a receipt to the merchant.
9. The merchant sends the receipt and a decryption key for the product to the customer.
10. The customer decrypts the product.

Commitments are created at various points in this protocol. When the merchant provides a price quote, the merchant makes a commitment to deliver the product if the customer sends a purchase request. When the customer sends a purchase request, the customer makes a commitment to pay for the product if it is delivered.

Suppose that a customer has not yet purchased a certain product the merchant application wishes to sell. The merchant application can reason that it is possible to send an unsolicited price quote to the customer. There is nothing in the protocol that prohibits this, and it might lead to a sale. Similarly, a customer application can reason that it is possible to send a purchase request without having first received a price quote. In this case, the customer makes a commitment to pay if the product is delivered, which the merchant has not promised to do.

Formalization

We now describe part of the event calculus formalization of these notions. We start with the representation of commitments. We use the following fluents to represent commitments:

$C(a_1, a_2, f)$: Agent a_1 is committed to agent a_2 to bring about fluent f. This is called a *base-level commitment*.
$CC(a_1, a_2, c, f)$: If fluent c is true, then agent a_1 is committed to a_2 to bring about fluent f. This is called a *conditional commitment*.

We have effect axioms for creating and discharging base-level and conditional commitments:

$$Initiates(CreateC(a_1, a_2, f), C(a_1, a_2, f), t) \tag{14.1}$$

$$Initiates(CreateCC(a_1, a_2, c, f), CC(a_1, a_2, c, f), t) \tag{14.2}$$

$$Terminates(DischargeC(a_1, a_2, f), C(a_1, a_2, f), t) \tag{14.3}$$

$$Terminates(DischargeCC(a_1, a_2, c, f), CC(a_1, a_2, c, f), t) \tag{14.4}$$

We have a trigger axiom that, states that, if a_1 is committed to a_2 to bring about f and f is true, then the commitment is discharged:

$$HoldsAt(C(a_1, a_2, f), t) \wedge HoldsAt(f, t) \Rightarrow Happens(DischargeC(a_1, a_2, f), t) \tag{14.5}$$

We also have trigger axioms stating that, if a_1 has a conditional commitment to a_2 to bring about f if c is true and c is true, then a base-level commitment of a_1 to a_2 to bring about f is created and the conditional commitment is discharged:

$$HoldsAt(CC(a_1, a_2, c, f), t) \wedge HoldsAt(c, t) \Rightarrow Happens(CreateC(a_1, a_2, f), t) \tag{14.6}$$

$$HoldsAt(CC(a_1, a_2, c, f), t) \wedge HoldsAt(c, t) \Rightarrow Happens(DischargeCC(a_1, a_2, c, f), t) \tag{14.7}$$

Next we represent part of the NetBill protocol and associated commitments. We use the following events:

RequestQuote(c, m, p): Customer c sends merchant m a request for a price quote for product p.
SendQuote(m, c, p, a): Merchant m sends customer c a quote of amount a for product p.
RequestPurchase(c, m, p, a): Customer c sends merchant m a request to purchase product p for amount a.
Deliver(m, c, p): Merchant m delivers product p to customer c.
SendEPO(c, m, a): Customer c sends merchant m an EPO for amount a.

We use the following fluents:

QuoteRequested(c, m, p): Customer c has sent merchant m a request for a price quote for product p.
QuoteSent(m, c, p, a): Merchant m has sent customer c a quote of amount a for product p.
PurchaseRequested(c, m, p, a): Customer c has sent merchant m a request to purchase product p for amount a.
Delivered(m, c, p): Merchant m has delivered product p to customer c.
EPOSent(c, m, a): Customer c has sent merchant m an EPO for amount a.

Effect axioms relate these events and fluents:

$$Initiates(RequestQuote(c,m,p), QuoteRequested(c,m,p), t) \quad (14.8)$$
$$Initiates(SendQuote(m,c,p,a), QuoteSent(m,c,p,a), t) \quad (14.9)$$
$$Initiates(RequestPurchase(c,m,p,a), PurchaseRequested(c,m,p,a), t) \quad (14.10)$$
$$Initiates(Deliver(m,c,p), Delivered(m,c,p), t) \quad (14.11)$$
$$Initiates(SendEPO(c,m,a), EPOSent(c,m,a), t) \quad (14.12)$$

We now use effect axioms to describe the effects of NetBill events on commitments. If the merchant sends the customer a price quote for a product, then the merchant will have a conditional commitment to deliver the product if the customer promises to pay the quoted price for the product:

$$Initiates(SendQuote(m,c,p,a), \quad (14.13)$$
$$CC(m,c,PurchaseRequested(c,m,p,a), Delivered(m,c,p)), t)$$

If the merchant has yet not delivered a given product to the customer and the customer sends a purchase request for the product to the merchant, then the customer will have a conditional commitment to pay for the product if the merchant delivers it:

$$\neg HoldsAt(Delivered(m,c,p), t) \Rightarrow \quad (14.14)$$
$$Initiates(RequestPurchase(c,m,p,a),$$
$$CC(c,m,Delivered(m,c,p), EPOSent(c,m,a)), t)$$

Scenario

Let us now consider a specific scenario. Suppose that a particular customer has not yet purchased anything from a particular music store:

$$\neg HoldsAt(PurchaseRequested(Jen, MusicStore, p, a), 0) \quad (14.15)$$
$$\neg HoldsAt(Delivered(MusicStore, Jen, p), 0) \quad (14.16)$$
$$\neg HoldsAt(EPOSent(Jen, MusicStore, a), 0) \quad (14.17)$$

Further suppose that the music store has the goal to sell the customer a particular CD:

$$\exists t\,(HoldsAt(Delivered(MusicStore, Jen, BritneyCD), t) \wedge \quad (14.18)$$
$$HoldsAt(EPOSent(Jen, MusicStore, 14.99), t))$$

We can then show that there are several ways this goal can be achieved (see Exercises 14.1–14.3). The music store can start by sending an unsolicited price quote:

$$Happens(SendQuote(MusicStore, Jen, BritneyCD, 14.99), 1) \quad (14.19)$$
$$Happens(RequestPurchase(Jen, MusicStore, BritneyCD, 14.99), 2) \quad (14.20)$$
$$Happens(Deliver(MusicStore, Jen, BritneyCD), 3) \quad (14.21)$$
$$Happens(SendEPO(Jen, MusicStore, 14.99), 4) \quad (14.22)$$

The customer can start by sending an unsolicited purchase request:

$$Happens(RequestPurchase(Jen, MusicStore, BritneyCD, 14.99), 1) \quad (14.23)$$
$$Happens(Deliver(MusicStore, Jen, BritneyCD), 2) \quad (14.24)$$

$$Happens(SendEPO(Jen, MusicStore, 14.99), 3) \tag{14.25}$$

The music store can even start by delivering the product:

$$Happens(Deliver(MusicStore, Jen, BritneyCD), 1) \tag{14.26}$$
$$Happens(SendEPO(Jen, MusicStore, 14.99), 2) \tag{14.27}$$

Thus, the event calculus provides the music store with several possible ways a sale could occur. The music store could perform additional commonsense reasoning to determine which course of action is best.

14.1.2 WORKFLOW MODELING

A method for using the event calculus to specify, simulate, and execute workflows has been defined by Nihan Kesim Cicekli and Yakup Yildirim. A workflow consists of a set of business activities and dependencies among the activities. For example, the workflow in a newspaper might consist of (1) a set of activities $w_1, \ldots, w_n$ in which reporters write stories; (2) a set of activities $e_1, \ldots, e_n$ in which editors edit stories written by reporters, where for each $i \in \{1, \ldots, n\}$, e_i depends on w_i; and (3) a pagination activity p in which the stories are put into place on pages, which depends on all the activities $e_1, \ldots, e_n$.

The Workflow Management Coalition defines several types of dependencies that can be part of a workflow.

In a *sequence*, one activity follows another activity; this is the case for e_i and w_i, just described.
In an *AND-split*, several activities follow one activity, and these activities are performed in parallel.
In an *AND-join*, an activity starts after several other activities have all ended; this is the case for p.
In an *XOR-split*, one of several activities follows an activity, depending on what condition is true.
In an *XOR-join*, an activity starts after any of several other activities have ended.
In an *iteration*, one or more activities are repeatedly performed while a condition is true.

The basic method for modeling workflows in the event calculus is as follows. We use DEC. The fluent $Active(a)$ represents that activity a is active, and the fluent $Completed(a)$ represents that activity a is completed. The event $Start(a)$ represents that activity a starts, and the event $End(a)$ represents that activity a ends. Effect axioms state that, if an activity starts, then it will be active and no longer completed, and, if an activity ends, then it will be completed and no longer active:

$$Initiates(Start(a), Active(a), t)$$
$$Terminates(Start(a), Completed(a), t)$$
$$Initiates(End(a), Completed(a), t)$$
$$Terminates(End(a), Active(a), t)$$

Dependencies are represented using trigger axioms that start activities in appropriate situations. The following trigger axiom represents a sequence in which activity *A* is followed by activity *B*.

$$\begin{gathered}\neg HoldsAt(Active(B), t) \wedge \\ \neg HoldsAt(Completed(A), t-1) \wedge \\ HoldsAt(Completed(A), t) \Rightarrow \\ Happens(Start(B), t)\end{gathered}$$

The following trigger axioms represent an AND-split in which *C1–C3* follow *B*.

$$\begin{gathered}\neg HoldsAt(Active(C1), t) \wedge \\ \neg HoldsAt(Completed(B), t-1) \wedge \\ HoldsAt(Completed(B), t) \Rightarrow \\ Happens(Start(C1), t)\end{gathered}$$

$$\begin{gathered}\neg HoldsAt(Active(C2), t) \wedge \\ \neg HoldsAt(Completed(B), t-1) \wedge \\ HoldsAt(Completed(B), t) \Rightarrow \\ Happens(Start(C2), t)\end{gathered}$$

$$\begin{gathered}\neg HoldsAt(Active(C3), t) \wedge \\ \neg HoldsAt(Completed(B), t-1) \wedge \\ HoldsAt(Completed(B), t) \Rightarrow \\ Happens(Start(C3), t)\end{gathered}$$

The following trigger axiom represents an AND-join in which *D* starts after *C1–C3* end.

$$\begin{gathered}\neg HoldsAt(Active(D), t) \wedge \\ ((\neg HoldsAt(Completed(C1), t-1) \wedge HoldsAt(Completed(C1), t)) \vee \\ (\neg HoldsAt(Completed(C2), t-1) \wedge HoldsAt(Completed(C2), t)) \vee \\ (\neg HoldsAt(Completed(C3), t-1) \wedge HoldsAt(Completed(C3), t))) \wedge \\ HoldsAt(Completed(C1), t) \wedge \\ HoldsAt(Completed(C2), t) \wedge \\ HoldsAt(Completed(C3), t) \Rightarrow \\ Happens(Start(D), t)\end{gathered}$$

The following trigger axioms represent an XOR-split in which *E1* follows *D* if *E1C* is true, in which *E2* follows *D* if *E2C* is true, and in which *E3* follows *D* if *E3C* is true.

$$\begin{gathered}\neg HoldsAt(Active(E1), t) \wedge \\ \neg HoldsAt(Completed(D), t-1) \wedge \\ HoldsAt(Completed(D), t) \wedge\end{gathered}$$

$$HoldsAt(E1C, t) \Rightarrow$$
$$Happens(Start(E1), t)$$

$$\neg HoldsAt(Active(E2), t) \wedge$$
$$\neg HoldsAt(Completed(D), t-1) \wedge$$
$$HoldsAt(Completed(D), t) \wedge$$
$$HoldsAt(E2C, t) \Rightarrow$$
$$Happens(Start(E2), t)$$

$$\neg HoldsAt(Active(E3), t) \wedge$$
$$\neg HoldsAt(Completed(D), t-1) \wedge$$
$$HoldsAt(Completed(D), t) \wedge$$
$$HoldsAt(E3C, t) \Rightarrow$$
$$Happens(Start(E3), t)$$

The following trigger axioms represent an XOR-join in which F starts after $E1$, $E2$, or $E3$ ends.

$$\neg HoldsAt(Active(F), t) \wedge$$
$$((\neg HoldsAt(Completed(E1), t-1) \wedge HoldsAt(Completed(E1), t)) \vee$$
$$(\neg HoldsAt(Completed(E2), t-1) \wedge HoldsAt(Completed(E2), t)) \vee$$
$$(\neg HoldsAt(Completed(E3), t-1) \wedge HoldsAt(Completed(E3), t))) \Rightarrow$$
$$Happens(Start(F), t)$$

The following trigger axioms represent an iteration in which activity F is repeatedly performed while a condition FC is true. When FC is false, activity G is performed.

$$\neg HoldsAt(Active(F), t) \wedge$$
$$\neg HoldsAt(Completed(F), t-1) \wedge$$
$$HoldsAt(Completed(F), t) \wedge$$
$$HoldsAt(FC, t) \Rightarrow$$
$$Happens(Start(F), t).$$

$$\neg HoldsAt(Active(G), t) \wedge$$
$$\neg HoldsAt(Completed(F), t-1) \wedge$$
$$HoldsAt(Completed(F), t) \wedge$$
$$\neg HoldsAt(FC, t) \Rightarrow$$
$$Happens(Start(G), t)$$

Once a workflow is modeled using the event calculus, temporal projection can be used to simulate the workflow. The model can also be used by a workflow manager in a workflow management system to manage the execution of a workflow by the workflow participants. The event calculus enables the past, present, and possible future states of a workflow to be queried at any point in the simulation or execution of the workflow.

14.2 NATURAL LANGUAGE UNDERSTANDING

Understanding natural language text or speech involves building representations of the meaning of that text or speech. The event calculus can be used to perform commonsense reasoning in order to build representations of meaning, and formulas of the event calculus can be used to represent meaning.

For example, we can use a semantic parser to convert the English sentence *Nathan wakes up* into the event calculus formula

$$Happens(WakeUp(Nathan), 0)$$

Then, suppose we have the following representations of commonsense knowledge:

$$Initiates(WakeUp(a), Awake(a), t)$$
$$Happens(WakeUp(a), t) \Rightarrow \neg HoldsAt(Awake(a), t)$$

We can reason using the event calculus in order to build the following representation of the sentence's meaning:

$$\neg HoldsAt(Awake(Nathan), 0)$$
$$Happens(WakeUp(Nathan), 0)$$
$$HoldsAt(Awake(Nathan), 1)$$

In general, the event calculus can be used in a natural language understanding system as follows:

1. The set of domains to be understood by the system is determined.
2. Commonsense knowledge about the domains is represented using the event calculus.
3. For each natural language input:
   - **(a)** The input is parsed by syntactic and/or semantic parsers into predicate-argument structure representations, which resemble event calculus *Happens* and *HoldsAt* formulas.
   - **(b)** The predicate-argument structure representations are converted into event calculus *Happens* and *HoldsAt* formulas.
   - **(c)** The event calculus formulas are fed to an event calculus reasoning program, which uses the commonsense knowledge to produce additional event calculus formulas, or inferences.
   - **(d)** The inferences are produced as output.

The output can be used for various purposes, including dialog management, information retrieval, question answering, summarization, updating a database, and user modeling.

14.2.1 STORY UNDERSTANDING

The event calculus can be used to address the problem of *story understanding*, which consists of taking a story as input, understanding it, and then answering questions

about it. Commonsense reasoning can be used to fill in details not explicitly stated in the input story. The Discrete Event Calculus Reasoner program can be used to build detailed models of a story, which represent the events that occur and the properties that are true or false at various times.

For example, the children's story by Raymond Briggs (1999, pp. 4–8) begins as follows:

> *Hooray! It is snowing! James gets dressed. He runs outside. He makes a pile of snow. He makes it bigger and bigger. He puts a big snowball on top.*

Given a domain description representing this text consisting of (1) a basic axiomatization of space and intentions, (2) observations of the initial state, and (3) a narrative of events, the Discrete Event Calculus Reasoner produces the following model.[1] The first input event is that it starts snowing outside James's house at timepoint 0:

```
0
Happens(StartSnowing(JamesOutside),0).
```

From this input event, it is inferred that it is snowing outside James's house at timepoint 1:

```
1
+Snowing(JamesOutside).
```

The next input event is that James wakes up:

```
Happens(WakeUp(James), 1).
```

It is inferred that he is no longer asleep and is awake:

```
2
-Asleep(James).
+Awake(James).
```

Next, it is inferred that James becomes happy:

```
Happens(BecomeHappy(James), 2).
```

A trigger axiom states that if an agent is awake, likes snow, and is in a room that looks out onto a location where it is snowing, then the agent will become happy. It is also inferred that James intends to play outside his house:

```
Happens(IntendToPlay(James, JamesOutside), 2).
```

A trigger axiom activates an intention for an agent to play when the agent has an unsatisfied need for play, likes snow, is awake, and is in a room that looks out onto an outside area where it is snowing. From these events, it is inferred that James is no longer calm, is happy, and intends to play outside his house:

[1] This model is taken from Section 4 of a paper by Mueller (2003) published by the Association for Computational Linguistics.

```
3
-Calm(James).
+Happy(James).
+IntentionToPlay(James, JamesOutside).
```

The next input events are that James cries for joy and gets up from his bed:

```
Happens(CryForJoy(James), 3).
4
Happens(RiseFrom(James, JamesBed), 4).
```

It is inferred that James is no longer lying on his bed, is no longer lying down, and is standing up:

```
5
-LyingOn(James, JamesBed).
-Lying(James).
+Standing(James).
```

Next, we have the input event that James gets dressed, from which it is inferred that he is dressed:

```
Happens(GetDressed(James), 5).
6
+Dressed(James).
```

The next input event is that James walks through the door of his bedroom, from which it is inferred that he is no longer in his bedroom, is in the hallway, and is near the staircase.

```
Happens(WalkThroughDoor12(James, JamesDoor2Fl), 6).
7
-At(James, JamesBedroom2Fl).
+At(James, JamesHallway2Fl).
+NearPortal(James, JamesStaircase1To2).
```

We have the input event that James walks down the staircase, from which it is inferred that he is no longer in the hallway, is no longer near the door of his bedroom, is in the foyer, and is near the front door and the kitchen door:

```
Happens(WalkDownStaircase(James, JamesStaircase1To2), 7).
8
-At(James, JamesHallway2Fl).
-NearPortal(James, JamesDoor2Fl).
+At(James, JamesFoyer1Fl).
+NearPortal(James, JamesFrontDoor1Fl).
+NearPortal(James, JamesKitchenDoor1Fl).
```

The next input event is that James unlocks the front door, from which it is inferred that the front door is unlocked:

```
Happens(DoorUnlock(James, JamesFrontDoor1Fl), 8).
9
+DoorUnlocked(JamesFrontDoor1Fl).
```

The input is provided that James opens the front door, and it is inferred that the door is open:

```
Happens(DoorOpen(James, JamesFrontDoor1Fl), 9).
10
+DoorIsOpen(JamesFrontDoor1Fl).
```

The input is then provided that James walks through the front door, from which it is inferred that he is no longer in the foyer, is no longer near the kitchen door or the staircase, and is outside:

```
Happens(WalkThroughDoor21(James, JamesFrontDoor1Fl), 10).
11
-At(James, JamesFoyer1Fl).
-NearPortal(James, JamesKitchenDoor1Fl).
-NearPortal(James, JamesStaircase1To2).
+At(James,JamesOutside).
```

The input is provided that James plays at timepoint 11:

```
+ActOnIntentionToPlay(James, JamesOutside).
Happens(Play(James, JamesOutside), 11).
```

It is inferred that the need to play is satisfied:

```
12
-ActOnIntentionToPlay(James, JamesOutside).
-IntentionToPlay(James, JamesOutside).
+SatiatedFromPlay(James).
```

It is provided as input that James picks up some snow, rolls it into a ball, and lets go of it:

```
Happens(HoldSome(James, Snowball1, Snow1), 12).
13
+Holding(James, Snowball1).
Happens(RollAlong(James, Snowball1, Snow1), 13).
14
-Diameter(Snowball1, 1).
+Diameter(Snowball1, 2).
Happens(LetGoOf(James, Snowball1), 14).
15
-Holding(James, Snowball1).
```

James does the same to make another snowball:

```
Happens(HoldSome(James, Snowball2, Snow1), 15).
16
```

```
+Holding(James, Snowball2).
Happens(RollAlong(James, Snowball2, Snow1), 16).
17
-Diameter(Snowball2, 1).
+Diameter(Snowball2, 2).
```

Finally, we have the input event that James places the second snowball on top of the first:

```
Happens(PlaceOn(James, Snowball2, Snowball1), 17).
18
-Holding(James, Snowball2).
+On(Snowball2, Snowball1).
```

14.3 VISION

Murray Shanahan and David Randell use the event calculus to implement the higher-level vision component of Ludwig, an upper-torso humanoid robot. Ludwig has two arms, each with three degrees of freedom, and a stereo camera hooked up to a head with two degrees of freedom (pan and tilt). The low-level vision component uses off-the-shelf edge detection software to map raw images from the camera into a list of edges. The list of edges is fed to the higher-level vision component, which is responsible for recognizing shapes. This component is implemented using the event calculus in Prolog.

The higher-level vision component consists of three layers. The first layer generates hypotheses about what regions are in view, based on the input list of edges. The second layer generates hypotheses about what aspects are in view, based on what regions are in view. The *aspects* of a shape are the various ways it can appear when viewed from different angles; for example, a wedge viewed from above appears to be a rectangle, but a wedge viewed from the side appears to be a triangle. The third layer generates hypotheses about what shapes are in view, based on the aspects that are in view over time.

We present here a simplified version of Ludwig's third layer. The predicate $Arc(s, a_1, a_2)$ represents that, by gradually changing the orientation of a shape s with respect to the camera, it is possible for the appearance of s to change from aspect a_1 to aspect a_2. For example, the appearance of a wedge can change from a rectangle to a rectangle plus an adjacent triangle, but it cannot change immediately from a rectangle to a triangle—it must first appear as a rectangle plus an adjacent triangle. The predicate $Shape(o, s)$ represents that object o has shape s. The fluent $Aspect(o, a)$ represents that the appearance of object o is aspect a. The event $Change(o, a_1, a_2)$ represents that the appearance of object o changes from aspect a_1 to aspect a_2.

We start with state constraints that say that an object has unique shape and aspect:

$$Shape(o, s_1) \wedge Shape(o, s_2) \Rightarrow s_1 = s_2$$
$$HoldsAt(Aspect(o, a_1), t) \wedge HoldsAt(Aspect(o, a_2), t) \Rightarrow a_1 = a_2$$

We have effect axioms that state that if the aspect of an object is a_1, the shape of the object is s, it is possible for the appearance of s to change from aspect a_1 to aspect a_2, and the object changes from a_1 to a_2, then the aspect of the object will be a_2 and will no longer be a_1:

$$\begin{gathered}HoldsAt(Aspect(o, a_1), t) \wedge \\ Shape(o, s) \wedge \\ (Arc(s, a_1, a_2) \vee Arc(s, a_2, a_1)) \Rightarrow \\ Initiates(Change(o, a_1, a_2), Aspect(o, a_2), t)\end{gathered}$$

$$\begin{gathered}HoldsAt(Aspect(o, a_1), t) \wedge \\ Shape(o, s) \wedge \\ (Arc(s, a_1, a_2) \vee Arc(s, a_2, a_1)) \Rightarrow \\ Terminates(Change(o, a_1, a_2), Aspect(o, a_1), t)\end{gathered}$$

Now, suppose we have the following visual knowledge about two shapes *Shape1* and *Shape2*:

$$\begin{gathered}Arc(Shape1, a_1, a_2) \Leftrightarrow \\ (a_1 = Aspect1 \wedge a_2 = Aspect2)\end{gathered}$$

$$\begin{gathered}Arc(Shape2, a_1, a_2) \Leftrightarrow \\ ((a_1 = Aspect1 \wedge a_2 = Aspect3) \vee \\ (a_1 = Aspect3 \wedge a_2 = Aspect2))\end{gathered}$$

Further, suppose we observe at timepoint 0 that *Object1* has *Aspect1* and at timepoint 1 that *Object1* has *Aspect2*:

$$\begin{gathered}HoldsAt(Aspect(Object1, Aspect1), 0) \\ HoldsAt(Aspect(Object1, Aspect2), 1)\end{gathered}$$

We can then show that the shape of *Object1* must be *Shape1* and not *Shape2*:

$$Shape(Object1, Shape1)$$

Note that this inference depends crucially on the sequence of images over time rather than on a single image. These sorts of inferences are supported by the ability of the event calculus to reason about time.

BIBLIOGRAPHIC NOTES

Lenat and Guha (1991b) and Lieberman, Liu, Singh, and Barry (2004) argue for the use of commonsense reasoning in applications and present numerous examples. The payment protocol application of the classical logic event calculus is from Yolum and Singh (2004); our formalization is adapted from theirs. NetBill is described by Sirbu and Tygar (1995). The workflow modeling application is from Cicekli and Yildirim (2000). They use a logic programming version of the event calculus; the axioms we present are adapted to the classical logic event calculus. Terminology relating

to workflows is defined by the Workflow Management Coalition (1999). Others have used the event calculus for workflow modeling; Wilk (2004) discusses the use of the classical logic event calculus in a hospital workflow management system.

Textbooks on natural language processing are by Allen (1995), Manning and Schütze (1999), and Jurafsky and Martin (2009). Deep natural language understanding systems are discussed by Ram and Moorman (1999). Commonsense reasoning is used by a number of deep natural language and discourse understanding systems (Charniak, 1972; Dahlgren, McDowell, & Stabler, 1989; Dyer, 1983; Hobbs, Stickel, Appelt, & Martin, 1993; Mueller, 1998; J. D. Moore, 1995; Norvig, 1989; Schank & Abelson, 1977; Schank, Kass, & Riesbeck, 1994). Halpin, Moore, and Robertson (2004) use a simplified event calculus to represent event occurrences in stories. van Lambalgen and Hamm (2005) use the classical logic event calculus to represent the semantics of tense and aspect in natural language. Predicate-argument structure representations are used in the Proposition Bank (Palmer, Gildea, & Kingsbury, 2005) annotated corpus. A modern syntactic parser is that of M. Collins (1999, 2003). Some semantic parsers are those of Alshawi (1992), Beale, Nirenburg, and Mahesh (1995), Blackburn and Bos (2005), Gildea and Jurafsky (2002), McCord, Murdock, and Boguraev (2012), and Punyakanok, Roth, and Yih (2008).

The problem of story understanding and systems for story understanding are reviewed by Domeshek, Jones, and Ram (1999), Mueller (2002), and Mani (2013). The use of the Discrete Event Calculus Reasoner for modeling children's stories is discussed by Mueller (2003). This program has also been used to model stories involving scripts, such as eating in a restaurant and kidnapping (Mueller, 2004c, 2007b). Nelson and Mateas (2008) use the Discrete Event Calculus Reasoner for executing and reasoning about game designs.

The use of the event calculus for robot perception is discussed by Shanahan (1999c, 2005). The Ludwig robot is discussed by Shanahan and Randell (2004) and Shanahan (2005, pp. 116–125). The axioms we present are simplified versions of axioms in the third layer of Ludwig's vision system. Koenderink and van Doorn (1979) discuss the notion of an aspect of an object and the representation of the shape of an object as a connected graph whose nodes represent aspects and whose links represent visual events.

EXERCISES

14.1 In the scenario in Section 14.1.1, prove that the sequence of events starting with *SendQuote* achieves the goal of selling the customer the CD.

14.2 In the scenario in Section 14.1.1, prove that the sequence of events starting with *RequestPurchase* achieves the goal of selling the customer the CD.

14.3 In the scenario in Section 14.1.1, prove that the sequence of events starting with *Deliver* achieves the goal of selling the customer the CD.

14.4 (Research problem) Use the event calculus to model the semantics of your favorite programming language.

14.5 (Research problem) Use the event calculus to build an automated advisor for your favorite computer program.

14.6 (Research problem) Use the event calculus to model the operation of a stock exchange.