CHAPTER

The Discrete Event Calculus Reasoner

13

This chapter describes the Discrete Event Calculus Reasoner, which uses satisfiability (SAT) to solve event calculus problems. The program can be used to perform automated deduction, abduction, postdiction, and model finding. We discuss the architecture of the program and the encoding of SAT problems. We present some simple examples of how the program is used and then present a more complicated example. We discuss the language used to describe commonsense reasoning problems to the program. The Discrete Event Calculus Reasoner can be downloaded from `decreasoner.sourceforge.net`.

13.1 DISCRETE EVENT CALCULUS REASONER ARCHITECTURE

The architecture of the Discrete Event Calculus Reasoner is shown in Figure 13.1. The program works as follows:

1. A domain description is provided as input to the program.
2. The main program sends the domain description to the SAT encoder.
3. The SAT encoder encodes the domain description as a SAT problem.
4. The SAT encoder sends the SAT problem back to the main program.
5. The main program sends the SAT problem to the SAT solver. The SAT problem is expressed in the standard DIMACS format used by SAT solvers.
6. A complete SAT solver is run in order to find all models. The Discrete Event Calculus Reasoner uses the Relsat SAT solver. If Relsat does not find any models, then the Walksat solver is run in order to find near-miss models.
7. The SAT solver sends models back to the main program.
8. The main program decodes the models from the SAT solver and produces them as output.

13.2 ENCODING SATISFIABILITY PROBLEMS

In order to encode the domain description as a SAT problem, the domain of every sort is limited to a finite set. The timepoint sort is restricted to the integers $\{0, \ldots, n\}$

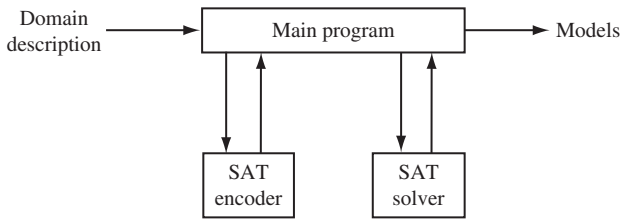

FIGURE 13.1

Discrete Event Calculus Reasoner architecture.

for some $n \geq 0$. An existential quantification $\exists \nu \, \Phi(\nu)$ is replaced by $\bigvee_i \Phi(\nu_i)$, and a universal quantification $\forall \nu \, \Phi(\nu)$ is replaced by $\bigwedge_i \Phi(\nu_i)$, where ν_i are the constants of the sort of ν. The unique names assumption is used.

SAT solving is facilitated by DEC. (The efficient solution of event calculus problems using SAT was the original motivation for the development of DEC.) DEC uses integers for timepoints, unlike EC, which uses real numbers. DEC replaces triply quantified time in many of the EC axioms with doubly quantified time, resulting in smaller SAT encodings.

13.3 SIMPLE EXAMPLES

In this section we present some simple examples of how the Discrete Event Calculus Reasoner is used to perform various types of reasoning.

13.3.1 DEDUCTION

The first example is a deduction problem. We create an input file containing the following domain description:

```
sort agent

fluent Awake(agent)
event WakeUp(agent)

[agent,time] Initiates(WakeUp(agent),Awake(agent),time).

agent James
!HoldsAt(Awake(James),0).
Delta: Happens(WakeUp(James),0).

completion Delta Happens

range time 0 1
```

We define sorts and predicates. We define an agent sort, a fluent *Awake*, and an event *WakeUp*. The fluent and the event both take a single argument that is an agent. Next, we provide an axiom stating that, if an agent wakes up, then that agent will be awake. The symbols `[` and `]` indicate universal quantification. Next we specify the observations and narrative. We define a constant *James* whose sort is agent. We specify that James is not awake at timepoint 0 and that he wakes up at timepoint 0. The symbol `!` indicates negation. For deduction we specify completion of the *Happens* predicate in the formulas labeled `Delta`. Finally, we specify the range of the time sort. There are two timepoints: 0 and 1.

When we run the program on this input, it produces the following output:

```
model 1:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
```

For each model, the fluents that are true at timepoint 0 are shown, followed by differences in the truth values of fluents from one timepoint to the next. Fluents that become true are preceded by + (plus sign), and fluents that become false are preceded by - (minus sign). There is one model, and *HoldsAt*(*Awake*(*James*), 1) is true in that model. Thus, the domain description entails that James is awake at timepoint 1.

An alternative output format is available that shows the truth values of all fluents at all timepoints. This format is obtained by adding the following line to the input file:

```
option timediff off
```

The program then produces the following output:

```
model 1:
0
!HoldsAt(Awake(James), 0).
!ReleasedAt(Awake(James),0).
Happens(WakeUp(James), 0).
1
!Happens(WakeUp(James), 1).
!ReleasedAt(Awake(James), 1).
HoldsAt(Awake(James), 1).
```

13.3.2 ABDUCTION

We can turn the deduction problem into an abduction or planning problem by making several changes. First, we specify a *goal* for James to be awake at timepoint 1. Second, we no longer specify that James wakes up because this is the *plan* we would like the program to find. Third, we do not specify completion of the *Happens* predicate. Our new domain description is as follows:

```
sort agent

fluent Awake(agent)
event WakeUp(agent)

[agent,time] Initiates(WakeUp(agent),Awake(agent),time).

agent James
!HoldsAt(Awake(James),0).
HoldsAt(Awake(James),1).

range time 0 1
```

When we run this input through the program, we get a single model in which James wakes up:

```
model 1:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
```

13.3.3 POSTDICTION

Next, we turn the deduction problem into a postdiction problem. First, we add an action precondition axiom stating that, in order for an agent to wake up, the agent must not be awake. Second, we remove the fact that James is not awake at timepoint 0 because this is the fact we would like the program to find. This gives us the following domain description:

```
sort agent

fluent Awake(agent)
event WakeUp(agent)

[agent,time] Initiates(WakeUp(agent),Awake(agent),time).
[agent,time]
Happens(WakeUp(agent),time) -> !HoldsAt(Awake(agent),time).

agent James
Delta: Happens(WakeUp(James),0).
HoldsAt(Awake(James),1).

completion Delta Happens

range time 0 1
```

From this, the program produces a single model in which James is not awake at timepoint 0:

```
model 1:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
```

13.3.4 MODEL FINDING

We can turn the postdiction problem into a model-finding problem by removing the observations, narrative, and completion of *Happens* in `Delta`:

```
sort agent

fluent Awake(agent)
event WakeUp(agent)

[agent,time] Initiates(WakeUp(agent),Awake(agent),time).
[agent,time] Happens(WakeUp(agent),time) -> !HoldsAt(Awake(agent),
time).

agent James

range time 0 1
```

We then get three models:

```
model 1:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
—
model 2:
0
1
—
model 3:
0
Awake(James).
1
```

That is, James is either awake or not awake at timepoint 0. If he is not awake at timepoint 0, then he may wake up (model 1) or may not wake up (model 2). If he

is awake at timepoint 0 (model 3), then he cannot wake up at timepoint 0. (Event occurrences are disallowed at the final timepoint, which is 1 in this case.)

13.4 EXAMPLE: TELEPHONE

In this section, we run the telephone example in Section 3.1.1 through the Discrete Event Calculus Reasoner. The domain description is as follows:

```
sort agent
sort phone

agent Agent1, Agent2
phone Phone1, Phone2

fluent Ringing(phone,phone)
fluent DialTone(phone)
fluent BusySignal(phone)
fluent Idle(phone)
fluent Connected(phone,phone)
fluent Disconnected(phone)

event PickUp(agent,phone)
event SetDown(agent,phone)
event Dial(agent,phone,phone)

[agent,phone,time]
HoldsAt(Idle(phone),time) ->
Initiates(PickUp(agent,phone),DialTone(phone),time).

[agent,phone,time]
HoldsAt(Idle(phone),time) ->
Terminates(PickUp(agent,phone),Idle(phone),time).

[agent,phone,time]
HoldsAt(DialTone(phone),time) ->
Initiates(SetDown(agent,phone),Idle(phone),time).

[agent,phone,time]
HoldsAt(DialTone(phone),time) ->
Terminates(SetDown(agent,phone),DialTone(phone),time).

[agent,phone1,phone2,time]
HoldsAt(DialTone(phone1),time) &
HoldsAt(Idle(phone2),time) ->
Initiates(Dial(agent,phone1,phone2),Ringing(phone1,phone2),time).
```

```
[agent,phone1,phone2,time]
HoldsAt(DialTone(phone1),time) &
HoldsAt(Idle(phone2),time) ->
Terminates(Dial(agent,phone1,phone2),DialTone(phone1),time).

[agent,phone1,phone2,time]
HoldsAt(DialTone(phone1),time) &
HoldsAt(Idle(phone2),time) ->
Terminates(Dial(agent,phone1,phone2),Idle(phone2),time).

[agent,phone1,phone2,time]
HoldsAt(DialTone(phone1),time) &
!HoldsAt(Idle(phone2),time) ->
Initiates(Dial(agent,phone1,phone2),BusySignal(phone1),time).

[agent,phone1,phone2,time]
HoldsAt(DialTone(phone1),time) &
!HoldsAt(Idle(phone2),time) ->
Terminates(Dial(agent,phone1,phone2),DialTone(phone1),time).

[agent,phone,time]
HoldsAt(BusySignal(phone),time) ->
Initiates(SetDown(agent,phone),Idle(phone),time).

[agent,phone,time]
HoldsAt(BusySignal(phone),time) ->
Terminates(SetDown(agent,phone),BusySignal(phone),time).

[agent,phone1,phone2,time]
HoldsAt(Ringing(phone1,phone2),time) ->
Initiates(SetDown(agent,phone1),Idle(phone1),time).

[agent,phone1,phone2,time]
HoldsAt(Ringing(phone1,phone2),time) ->
Initiates(SetDown(agent,phone1),Idle(phone2),time).

[agent,phone1,phone2,time]
HoldsAt(Ringing(phone1,phone2),time) ->
Terminates(SetDown(agent,phone1),Ringing(phone1,phone2),time).

[agent,phone1,phone2,time]
HoldsAt(Ringing(phone1,phone2),time) ->
Initiates(PickUp(agent,phone2),Connected(phone1,phone2),time).

[agent,phone1,phone2,time]
HoldsAt(Ringing(phone1,phone2),time) ->
Terminates(PickUp(agent,phone2),Ringing(phone1,phone2),time).
```

```
[agent,phone1,phone2,time]
HoldsAt(Connected(phone1,phone2),time) ->
Initiates(SetDown(agent,phone1),Idle(phone1),time).

[agent,phone1,phone2,time]
HoldsAt(Connected(phone1,phone2),time) ->
Initiates(SetDown(agent,phone1),Disconnected(phone2),time).

[agent,phone1,phone2,time]
HoldsAt(Connected(phone1,phone2),time)->
Terminates(SetDown(agent,phone1),Connected(phone1,phone2),time).

[agent,phone1,phone2,time]
HoldsAt(Connected(phone1,phone2),time)->
Initiates(SetDown(agent,phone2),Idle(phone2),time).

[agent,phone1,phone2,time]
HoldsAt(Connected(phone1,phone2),time)->
Initiates(SetDown(agent,phone2),Disconnected(phone1),time).

[agent,phone1,phone2,time]
HoldsAt(Connected(phone1,phone2),time)->
Terminates(SetDown(agent,phone2),Connected(phone1,phone2),time).

[agent,phone,time]
HoldsAt(Disconnected(phone),time) ->
Initiates(SetDown(agent,phone),Idle(phone),time).

[agent,phone,time]
HoldsAt(Disconnected(phone),time) ->
Terminates(SetDown(agent,phone),Disconnected(phone),time).

Delta: Happens(PickUp(Agent1,Phone1),0).
Delta: Happens(Dial(Agent1,Phone1,Phone2),1).
Delta: Happens(PickUp(Agent2,Phone2),2).

[phone] HoldsAt(Idle(phone),0).
[phone] !HoldsAt(DialTone(phone),0).
[phone] !HoldsAt(BusySignal(phone),0).
[phone1,phone2] !HoldsAt(Ringing(phone1,phone2),0).
[phone1,phone2] !HoldsAt(Connected(phone1,phone2),0).
[phone] !HoldsAt(Disconnected(phone),0).

completion Delta Happens

range time 0 3
```

The output is as follows:

```
model 1:
0
Idle(Phone1).
Idle(Phone2).
Happens(PickUp(Agent1, Phone1), 0).
1
-Idle(Phone1).
+DialTone(Phone1).
Happens(Dial(Agent1, Phone1, Phone2), 1).
2
-DialTone(Phone1).
-Idle(Phone2).
+Ringing(Phone1, Phone2).
Happens(PickUp(Agent2, Phone2), 2).
3
-Ringing(Phone1, Phone2).
+Connected(Phone1, Phone2).
```

13.5 DISCRETE EVENT CALCULUS REASONER LANGUAGE

Domain descriptions are described using the Discrete Event Calculus Reasoner language. A domain description consists of zero or more formulas, and zero or more statements. The syntax for formulas, which is based on that of the Bliksem theorem prover, is shown in Figure 13.2. The meanings of the symbols used in formulas are shown in Table 13.1. As in this book, predicate symbols, function symbols, and nonnumeric constants start with an uppercase letter and variables start with a lowercase letter. The sort of a variable is determined by removing trailing digits from

```
  Term ::= Constant | Variable | FunctionSymbol(Term, …, Term) |
           Term + Term | Term - Term | Term * Term | Term / Term |
           Term % Term | - Term | (Term)
  Atom ::= PredicateSymbol(Term, …, Term) | Term < Term |
           Term <= Term | Term = Term | Term >= Term |
           Term > Term | Term != Term
Formula ::=Atom | ! Formula | Formula & Formula |
           Formula | Formula | Formula -> Formula |
           Formula <-> Formula | {Variable, …, Variable} Formula |
           [Variable, …, Variable] Formula | (Formula)
```

FIGURE 13.2

Discrete Event Calculus Reasoner formula syntax.

Table 13.1 Discrete Event Calculus Reasoner Formula Symbols

Symbol	Meaning	Symbol	Meaning
+	Addition	!=	Not equal to
-	Subtraction, negation	!	Logical negation
*	Multiplication	&	Conjunction (AND, ∧)
/	Division	|	Disjunction (OR, ∨)
%	Remainder	->	Implication
<	Less than	<->	Bi-implication
<=	Less than or equal to	{ }	Existential quantifier (∃)
=	Equal to	[ ]	Universal quantifier (∀)
>=	Greater than or equal to	( )	Grouping
>	Greater than	,	Separator

Table 13.2 Discrete Event Calculus Reasoner Statements

Statement	Meaning
`sort` Sort, …, Sort	Define sorts
`function` FunctionSymbol`(`Sort, …, Sort`)`: Sort	Define function
`predicate` PredicateSymbol`(`Sort, …, Sort`)`	Define predicate
`event` EventSymbol`(`Sort, …, Sort`)`	Define event
`fluent` FluentSymbol`(`Sort, …, Sort`)`	Define fluent
`noninertial` FluentSymbol, …, FluentSymbol	Define noninertial fluents
Sort Constant	Define constant
`completion` Label PredicateSymbol	Specify completion
`load` Filename	Load file
`option` OptionName OptionValue	Specify option
`range` Sort Integer Integer	Specify range

the variable. For example, the sort of the variable `time1` is time. Some Discrete Event Calculus Reasoner statements are shown in Table 13.2. A *noninertial fluent* is one that is released from the commonsense law of inertia at all timepoints.

BIBLIOGRAPHIC NOTES

The Discrete Event Calculus Reasoner was introduced by Mueller (2004a, 2004b). DIMACS format is specified by DIMACS (1993). The Relsat SAT solver is discussed by Bayardo Jr. and Schrag (1997), and Walksat is discussed by Selman, Kautz, and Cohen (1993). The technique of restricting the event calculus to a finite universe to enable SAT solving was introduced by Shanahan and Witkowski (2004). The restriction of first-order logic to a finite universe is discussed by Kautz and Selman (1992) and Jackson (2000). To produce a compact conjunctive normal form, the

Discrete Event Calculus Reasoner uses the technique of renaming subformulas, which is described by Plaisted and Greenbaum (1986) and E. Giunchiglia and Sebastiani (1999). The Bliksem theorem prover is described by de Nivelle (1999).

EXERCISES

13.1 Formalize and run the Tweety example in the Bibliographic notes of Chapter 12 in the Discrete Event Calculus Reasoner.

13.2 Using the Discrete Event Calculus Reasoner, formalize and run Shanahan's (1997b) kitchen sink scenario, which is described in Chapter 1.

13.3 Formalize the WALK x-schema of Narayanan (1997), and use the Discrete Event Calculus Reasoner to perform temporal projection for a scenario involving walking.

13.4 (Research problem) Formalize scuba diving, and use the Discrete Event Calculus Reasoner to understand scuba-diving incident reports. See Section 14.2 for a discussion of the use of the event calculus in natural language understanding.

13.5 (Research problem) The Discrete Event Calculus Reasoner currently produces models as output. Extend the Discrete Event Calculus Reasoner and a SAT solver to produce proofs for deduction problems using the techniques discussed by McMillan and Amla (2003, sec. 2).