Discrete Event Calculus Reasoner
Documentation

Erik T. Mueller

IBM Thomas J. Watson Research Center

P.O. Box 704

Yorktown Heights, NY 10598

USA

March 2, 2008

1 Introduction

The Discrete Event Calculus Reasoner is a program for performing automated commonsense reasoning using the discrete event calculus [Mueller, 2006], a version of the classical logic event calculus [Shanahan, 1997, Miller and Shanahan, 2002]. The program supports such types of reasoning as deduction, temporal projection, abduction, planning, postdiction, and model finding.

The Discrete Event Calculus Reasoner supports the following commonsense phenomena:

- The *commonsense law of inertia*, which states that an event typically changes only a small number of things and everything else in the world remains unchanged. For example, moving a glass does not mysteriously cause a glass in another room to move.

- *Conditional effects of events.* For example, the results of turning on a television set depend on whether or not it is plugged in.

- *Release from the commonsense law of inertia.* For example, if a person is holding a glass, then the location of the glass is released from the commonsense law of inertia so that the location of the glass is permitted to vary.

- *Event ramifications* or indirect effects of events. The tool supports *state constraints*. For example, a glass moves along with the person holding it. The tool supports *causal constraints*, which deal with the instantaneous propagation of interacting indirect effects, as in idealized electrical circuits.

- *Events with nondeterministic effects.* For example, flipping a coin results in the coin landing either heads or tails.

- *Gradual change* such as the changing height of a falling object or volume of a balloon in the process of inflation.

- *Triggered events* or events that are triggered under certain conditions. For example, if water is flowing from a faucet into a sink, then once the water reaches a certain level the water will overflow.

- *Concurrent events with cumulative or canceling effects.* For example, if a shopping cart is simultaneously pulled and pushed, then it will spin around.

Here is how you use the Discrete Event Calculus Reasoner. First, you place a *domain description* into a file. (This file may load other files.) The domain description consists of

- an *axiomatization* describing a commonsense domain or domains of interest,

- *observations* of world properties at various times, and

- a *narrative* of known event occurrences.

The domain description is expressed using the *Discrete Event Calculus Reasoner language*. Then, you invoke the Discrete Event Calculus Reasoner on the domain description. The program transforms the domain description into a *satisfiability* (SAT) problem. (The SAT problem is expressed in the standard format used by most SAT solvers [DIMACS, 1993].) The program then runs a SAT solver, which produces zero or more solutions, called *models*. The program decodes these models and displays them.

1.1 Software Requirements

The Discrete Event Calculus Reasoner runs under Windows and Linux. To run under Windows, you will first need to install Cygwin, which is available from `http://www.cygwin.com/`. When installing Cygwin, make sure you install Python, gcc, and g++. Most Linux installations already have Python, gcc, and g++. (If you need to get Python, you can get it from `http://www.python.org/`. If you need to get gcc and g++, you can get them from `http://gcc.gnu.org/`.) Under both Windows and Linux, you will also need to download the following:

- One or more SAT solvers. The following are recommended: Relsat [Bayardo Jr. and Schrag, 1997], which is available from `http://www.bayardo.org/resources.html`, and Walksat [Selman et al., 1993], which is available from `http://www.cs.rochester.edu/u/kautz/walksat/`. Walksat can be used to display near miss models (see Section 1.6). You can also use Minisat [Eén and Sörensson, 2004], which is available from `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html`.

- PLY (Python Lex-Yacc), which is available from `http://www.dabeaz.com/ply/`.

1.2 Installing the Program

Download the tar file `decreasoner.tar.gz` and issue the following command:

```
tar -zxvf decreasoner.tar.gz
```

Then continue by following the installation instructions in the file `decreasoner/README`.

To verify that the program is functioning properly, `cd` to the `decreasoner` directory and type the following:

```
$ python
>>> import decreasoner
>>> decreasoner.test()
```

If the program is working properly, it will print something like the following:

```
loading examples/Shanahan1997/Yale.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
Happens(Shoot(), 2).
3
-Alive().
-Loaded().
EC: 7 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Root: 0 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Yale: 0 predicates, 0 functions, 2 fluents, 3 events, 8 axioms
encoding 0.0s
solution 0.0s
total 0.1s
>>>
```

1.3   A Simple Example: Deduction

Here is a simple example of how to use the Discrete Event Calculus Reasoner. We use a text editor to create a file containing the following domain description:

```
load foundations/Root.e
load foundations/EC.e

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
range offset 1 1
```

The first line of the file is a comment. The `;` indicates that it and the rest of the characters on the line should be ignored. We always start by loading `Root.e`, which defines the boolean, integer, predicate, and function sorts, and `EC.e`, which defines the sorts, functions, predicates, and axioms for the event calculus.

Next we create a simple axiomatization of waking up: We define a new sort `agent`, to represent a person or animal. We define a new fluent `Awake` and a new event `WakeUp`. We add an effect axiom that states that if an agent wakes up, then that agent will be awake. Every sentence must be terminated with a period.

Now we code a particular commonsense reasoning problem that uses the axiomatization. We start by defining a constant `James`, who is an agent. We state that James is not awake at timepoint 0 and that he wakes up at timepoint 0. We state that reasoning should be limited to timepoints 0 through 1.

We save the file as `examples/Manual/Example1.e` and run it by typing the following:

```
$ python
>>> import decreasoner
>>> decreasoner.run('examples/Manual/Example1.e')
```

The following output is produced:

```
loading examples/Manual/Example1.e
loading foundations/Root.e
loading foundations/EC.e
6 variables and 10 clauses
relsat solver
1 model
---
model 1:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
P
!Happens(WakeUp(James), 1).
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
EC: 7 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Example1: 0 predicates, 0 functions, 1 fluents, 1 events, 3 axioms
Root: 0 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
encoding 0.0s
solution 0.0s
total 0.1s
```

This shows the fluents that are true at timepoint zero, and the differences in what fluents are true from one timepoint to the next. Fluents that become true are indicated with a plus sign ("+"), and fluents that become false are indicated with a minus sign ("-").

An alternative output format is available that shows all the fluents that are true and false at every timepoint. This format is obtained by adding the line:

```
option timediff off
```

With this option set, we get the following output:

```
loading examples/Manual/Example1a.e
loading foundations/Root.e
loading foundations/EC.e
6 variables and 10 clauses
relsat solver
1 model
---
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).
P
EC: 7 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Example1a: 0 predicates, 0 functions, 1 fluents, 1 events, 3 axioms
Root: 0 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
encoding 0.0s
solution 0.0s
total 0.1s
```

1.4   Abduction and Model Finding

The above example showed how the Discrete Event Calculus Reasoner can be used to perform deduction. Specifically the example demonstrated temporal projection—given an initial situation and a sequence of events, we would like to determine the resulting situation. Instead, suppose we are given an initial situation and a resulting situation, and would like to reason about what must have happened. This is an example of abduction. We can perform this type of reasoning by setting up the commonsense reasoning problem as follows:

```
load foundations/Root.e
load foundations/EC.e

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
range offset 1 1
```

That is, we assert that James was not awake at timepoint 0 and he was awake at timepoint 0, but we do not say anything about James waking up. If we run this, the program abduces that James woke up:

```
loading examples/Manual/Example2.e
loading foundations/Root.e
loading foundations/EC.e
5 variables and 9 clauses
relsat solver
1 model
---
model 1:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
P
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
EC: 7 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Example2: 0 predicates, 0 functions, 1 fluents, 1 events, 3 axioms
Root: 0 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
encoding 0.0s
solution 0.0s
total 0.1s
```

What if we add another agent `Jessie`, but do not specify anything further? Then we get four models:

```
loading examples/Manual/Example3.e
loading foundations/Root.e
loading foundations/EC.e
10 variables and 16 clauses
relsat solver
4 models
---
model 1:
0
Happens(WakeUp(James), 0).
Happens(WakeUp(Jessie), 0).
1
+Awake(James).
+Awake(Jessie).
P
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
!ReleasedAt(Awake(Jessie), 0).
!ReleasedAt(Awake(Jessie), 1).
---
model 2:
0
Awake(Jessie).
Happens(WakeUp(James), 0).
```

```
Happens(WakeUp(Jessie), 0).
1
+Awake(James).
P
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
!ReleasedAt(Awake(Jessie), 0).
!ReleasedAt(Awake(Jessie), 1).
---
model 3:
0
Awake(Jessie).
Happens(WakeUp(James), 0).
1
+Awake(James).
P
!Happens(WakeUp(Jessie), 0).
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
!ReleasedAt(Awake(Jessie), 0).
!ReleasedAt(Awake(Jessie), 1).
---
model 4:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
P
!Happens(WakeUp(Jessie), 0).
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
!ReleasedAt(Awake(Jessie), 0).
!ReleasedAt(Awake(Jessie), 1).
EC: 7 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Example3: 0 predicates, 0 functions, 1 fluents, 1 events, 3 axioms
Root: 0 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
encoding 0.0s
solution 0.0s
total 0.1s
```

The program simply finds all the models consistent with the stated facts and axioms. (An exception is that occurrences of events at the last timepoint are ruled out.) We haven't said whether Jessie was initially awake or not, and we haven't said that in order for a person to wake up, the person must have previously been asleep. If we add Jessie's initial state and an action precondition axiom for WakeUp:

```
!HoldsAt(Awake(Jessie),0).
[agent,time] Happens(WakeUp(agent),time) -> !HoldsAt(Awake(agent),time).
```

then we get only two models:

```
loading examples/Manual/Example4.e
loading foundations/Root.e
```

```
loading foundations/EC.e
10 variables and 19 clauses
relsat solver
2 models
---
model 1:
0
Happens(WakeUp(James), 0).
1
+Awake(James).
P
!Happens(WakeUp(Jessie), 0).
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
!ReleasedAt(Awake(Jessie), 0).
!ReleasedAt(Awake(Jessie), 1).
---
model 2:
0
Happens(WakeUp(James), 0).
Happens(WakeUp(Jessie), 0).
1
+Awake(James).
+Awake(Jessie).
P
!ReleasedAt(Awake(James), 0).
!ReleasedAt(Awake(James), 1).
!ReleasedAt(Awake(Jessie), 0).
!ReleasedAt(Awake(Jessie), 1).
EC: 7 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
Example4: 0 predicates, 0 functions, 1 fluents, 1 events, 5 axioms
Root: 0 predicates, 0 functions, 0 fluents, 0 events, 0 axioms
encoding 0.0s
solution 0.0s
total 0.1s
```

That is, Jessie might or might not have woken up.

1.5 Propositional Calculus Reasoning

The Discrete Event Calculus Reasoner can also be used to solve propositional logic problems using a first-order syntax. That is, you do not have to load the event calculus, which is normally loaded by doing `load foundations/EC.e`. Here is an example:

```
load foundations/Root.e

sort object

predicate P(object)
predicate Q(object)

[object] P(object) -> Q(object).
```

```
object A, B

P(A).

; keep Discrete Event Calculus Reasoner happy
sort fluent
sort time: integer
range time 1 1
```

When this example is run, the Discrete Event Calculus Reasoner produces three models. `Q(A)` is true in all of them. If you add the formula `!Q(A)`, then the program reports that there are no models—the problem is unsatisfiable.

1.6 Near Miss Models

An *n-near miss model* of a SAT problem is a truth assignment that satisfies all but n clauses of the problem. Walksat provides the command-line option:

```
-target N = succeed if N or fewer clauses unsatisfied
```

If relsat produces no models, the Discrete Event Calculus Reasoner invokes walksat with `-target 1`. If this fails, it invokes walksat with `-target 2`. If this fails, it gives up. One or two unsatisfied clauses may be helpful for debugging. Three or more unsatisfied clauses tend to be less useful. (To change this in the program, change the line `while unsatisfied<3`.)

If you get a near miss model, it is often useful to rerun the Discrete Event Calculus Reasoner. Because walksat is stochastic, you may get back a different near miss model, and that near miss model may be more informative than the previous one. Or by looking at several near miss models, you may build a mental picture of what is wrong.

2 Discrete Event Calculus Reasoner Language

In this section, we describe the Discrete Event Calculus Reasoner in more detail.

2.1 Sorts

Sentences are expressed in the language of many-sorted first-order predicate calculus with subsort orders [Walther, 1987]. This means that:

- sorts can be subsorts of other sorts,

- every variable, constant, and function symbol has an associated sort,

- every argument position of every function and predicate symbol has an associated sort, and

- for a term to fill an argument position of a function or predicate symbol, the sort associated with the term must be a subsort of the sort associated with the argument position.

We define a sort called `object` as follows:

```
sort object
```

We define a sort `agent` that is a subsort of `object` as follows:

```
sort agent: object
```

The sort associated with a variable is determined by removing digits from the variable. For example, the sort of the variable `snowflake72` is `snowflake`. A constant's sort is specified when the constant is defined. For example, the following defines three constants whose sort is `agent`:

```
agent Fred, Annie, Thea
```

Each object in the world is assumed to be named by a unique constant. That is, the constants `Annie` and `Thea` do not refer to the same person. This is known as the *unique names assumption*.

Here is a definition of a function symbol `Floor` whose sort is `integer` and whose first and only argument position is of sort `room`:

```
function Floor(room): integer
```

Here is a definition of a predicate symbol `PartOf` that takes two arguments of sort `physobj` and `object`:

```
predicate PartOf(physobj,object)
```

2.2 Formulas

The following grammar, which is based on that of the Bliksem theorem prover [de Nivelle, 1999], in conjunction with the sort constraints described above, specifies how sentences are constructed:

term ::= *variable* | *constant* |
       functionsymbol(*arguments*) |
       term + *term* | *term* − *term* | *term* * *term* | *term* / *term* |
       term % *term* | − *term* |
       (*term*) | *reifiedformula*
formula ::= *predicatesymbol*(*arguments*) |
        term < *term* | *term* <= *term* | *term* = *term* |
        term >= *term* | *term* > *term* | *term* != *term* |
        formula | *formula* | *formula* & *formula* | ! *formula* |
        formula -> *formula* | *formula* <-> *formula* |
        {*variables*} *formula* | [*variables*] *formula* |
        (*formula*)
reifiedformula ::= *formula*
arguments ::= *term* | *arguments*, *term*
variables ::= *variable* | *variables*, *variable*

A *variable* consists of one or more lowercase letters followed by zero or more digits. Examples of variables are *agent* and *cat1*. A *constant* consists of (a) one or more digits or (b) an uppercase letter followed by zero or more letters or digits. Examples of constants are 63, *James*, and *Snowball1*. *functionsymbols* and *predicatesymbols* consist of an uppercase letter followed by zero or more letters or digits. Examples of *functionsymbols* are *BuildingOf* and *SkyOf*; examples of *predicatesymbols* are *PartOf* and *Adjacent*. Fluent and event symbols are predicate symbols.

The meaning of the symbols is as follows:

Symbol	Meaning	Symbol	Meaning
+	addition	!=	not equal to
-	subtraction, negation	|	disjunction (OR, ∨)
*	multiplication	&	conjunction (AND, ∧)
/	division	!	logical negation
%	modulus	->	implication
<	less than	<->	bi-implication
<=	less than or equal to	{ }	existential quantification (∃)
=	equal to	[ ]	universal quantification (∀)
>=	greater than or equal to	( )	grouping
>	greater than	,	separator

2.3 Completion

The `completion` statement specifies that a predicate symbol should be subject to predicate completion. The syntax of this statement is:

completion [*label*] *predicatesymbol*

If the optional *label* is not present, then *predicatesymbol* is completed in all formulas. If a *label* is present, then *predicatesymbol* is completed only in formulas with the specified label. For example, consider the following:

```
[x] P(x) & !Ab(x) -> Q(x).
Theta: [x] R(x) -> Ab(x).
Theta: [x] S(x) -> Ab(x).

completion Theta Ab
```

The predicate `Ab` is completed in the second and third formulas, but not the first.

2.4 Ignore

The `ignore` statement specifies that one or more predicate symbols and all sentences containing those predicate symbols should be ignored. The syntax of this statement is:

ignore *predicatesymbol*, *predicatesymbol*, . . .

2.5 Load

The `load` statement can be used to load other Discrete Event Calculus Reasoner language files. The syntax of this statement is:

load *filename*

2.6 Manualrelease

The `manualrelease` statement inhibits automatic generation of assertions that particular fluent symbols are not released at time point 0. The syntax of this statement is:

manualrelease *fluentsymbol*, *fluentsymbol*, . . .

2.7 Mutex and Xor

The `mutex` statement specifies that one or more event or fluent symbols are mutually exclusive at each timepoint. The syntax of this statement is:

`mutex` *symbol,* *symbol,* ...

Exclusive or (xor) is also provided:

`xor` *symbol,* *symbol,* ...

2.8 Noninertial

The `noninertial` statement specifies that one or more fluent symbols should not be subject to the commonsense law of inertia, across all timepoints. The syntax of this statement is:

`noninertial` *fluentsymbol,* *fluentsymbol,* ...

2.9 Options

The `option` statement can be used to specify the values of certain program options. The syntax of this statement is:

`option` *optionname optionvalue*

The following *optionname*s are supported:

debug If the *optionvalue* is `on`, detailed debugging output is produced. The default value of this option is `off`.

encoding The value of this option specifies the event calculus-to-SAT encoding method. Option 2 is the method described in the FLAIRS paper [Mueller, 2004b], and option 3 is the slightly different method described in the *Journal of Logic and Computation* article [Mueller, 2004a]. The default value of this option is 2.

finalstatefile The value of this option specifies the name of a file to which the final state of (the first model of) the run will be written.

manualrelease If `on`, automatic generation of assertions of the form `!Released(fluent,0)` is inhibited, so that such assertions can be added manually on a case-by-case basis. This is typically used when pasting together several runs—the final state of each run, including whether each fluent is `Released` or not, serves as the initial state of the next run. The default value of this option is `off`.

modeldiff If `on`, differences from one model to the next are shown, instead of complete models. The default value of this option is `off`.

renaming If `on`, the technique of renaming subformulas [Plaisted and Greenbaum, 1986, Giunchiglia and Sebastiani, 1999] is used to convert to a compact conjunctive normal. The default value of this option is `on`.

showpred If `on`, the truth values of all predicates other than fluents and events are shown. The default value of this option is `on`.

solver The value of this option specifies the name of the solver to use. The default value of this option is `relsat`.

timediff If **on**, differences from one timepoint to the next are shown, instead of complete states. The default value of this option is **on**.

tmpfilerm If **on**, temporary files, which are stored in the **/tmp** directory, are removed at the end of each run. The default value of this option is **on**.

trajectory If **on**, **Trajectory** axioms are supported. The default value of this option is **off**.

2.10   Ranges

The **range** statement specifies the range of sorts, such as the **time** sort, that are subsorts of the **integer** sort. The syntax is:

range *sort minvalue maxvalue*

References

[Bayardo Jr. and Schrag, 1997] Bayardo Jr., R. J. and Schrag, R. C. (1997). Using CSP look-back techniques to solve real world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, pages 203–208, Menlo Park, CA. AAAI Press.

[de Nivelle, 1999] de Nivelle, H. (1999). Bliksem 1.10 user manual.

[DIMACS, 1993] DIMACS (1993). Satisfiability suggested format. Technical report, Center for Discrete Mathematics and Theoretical Computer Science.

[Eén and Sörensson, 2004] Eén, N. and Sörensson, N. (2004). An extensible SAT-solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Berlin. Springer.

[Giunchiglia and Sebastiani, 1999] Giunchiglia, E. and Sebastiani, R. (1999). Applying the Davis-Putnam procedure to non-clausal formulas. In *Proceedings of the Sixth Congress of the Italian Association for Artificial Intelligence*, Bologna.

[Miller and Shanahan, 2002] Miller, R. and Shanahan, M. (2002). Some alternative formulations of the event calculus. In Kakas, A. C. and Sadri, F., editors, *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski, Part II*, volume 2408 of *Lecture Notes in Computer Science*, pages 452–490. Springer, Berlin.

[Mueller, 2004a] Mueller, E. T. (2004a). Event calculus reasoning through satisfiability. *Journal of Logic and Computation*, 14(5):703–730.

[Mueller, 2004b] Mueller, E. T. (2004b). A tool for satisfiability-based commonsense reasoning in the event calculus. In Barr, V. and Markov, Z., editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference*, pages 147–152, Menlo Park, CA. AAAI Press.

[Mueller, 2006] Mueller, E. T. (2006). *Commonsense Reasoning*. Morgan Kaufmann/Elsevier, San Francisco.

[Plaisted and Greenbaum, 1986] Plaisted, D. A. and Greenbaum, S. (1986). A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304.

[Selman et al., 1993] Selman, B., Kautz, H. A., and Cohen, B. (1993). Local search strategies for satisfiability testing. In Johnson, D. S. and Trick, M. A., editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementatio n Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Scienc e*. American Mathematical Society, Boston, MA.

[Shanahan, 1997] Shanahan, M. (1997). *Solving the Frame Problem*. MIT Press, Cambridge, MA.

[Walther, 1987] Walther, C. (1987). *A Many-Sorted Calculus Based on Resolution and Paramodulation*. Pitman, London.