

Final Thesis


A CLP(FD) based model checker for CTL

by

Marcus Eriksson

LITH-IDA-EX—05/056—SE

2005-06-08

 LINKÖPINGS UNIVERSITET	Avdelning, Institution Division, Department Institutionen för datavetenskap 581 83 LINKÖPING	Datum Date 2005-06-08
--	--	--

Språk Language Svenska/Swedish X Engelska/English	Rapporttyp Report category Licentiatavhandling X Examensarbete C-uppsats D-uppsats Övrig rapport _____	ISBN ISRN LITH-IDA-EX--05/056--SE Serietitel och serienummer ISSN Title of series, numbering
---	--	---

URL för elektronisk version http://www.ep.liu.se/exjobb/ida/2005/dd-d/056/

Titel Title	En CLP(FD)-baserad modellverifierare för CTL A CLP(FD)-based model checker for CTL
Författare Author	Marcus Eriksson

Sammanfattning Abstract <p>Model checking is a formal verification method where one tries to prove or disprove properties of a formal system. Typical systems one might want to prove properties within are network protocols and digital circuits. Typical properties to check for are safety (nothing bad ever happens) and liveness (something good eventually happens).</p> <p>This thesis describes an implementation of a sound and complete model checker for Computation Tree Logic (CTL) using Constraint Logic Programming over Finite Domains (CLP(FD)). The implementation described uses tabled resolution to remember earlier computations, is parameterised by choices of computation strategies and can with slight modification support different constraint domains. Soundness under negation is maintained through a restricted form of constructive negation.</p> <p>The computation process amounts to a fixpoint search, where a fixpoint is reached when no more extension operations has any effect. As results show, the choice of strategies does influence the efficiency of the computation. Soundness and completeness are of course independent of the choice of strategies. Strategies include how to choose the extension operation for the next step and whether to perform global or local rule instantiations, resulting in bottom-up or top-down computations respectively.</p>
--

Nyckelord Keywords fixpoint engine, model checking, tabled resolution, Constraint Logic Programming, strategies
--

Final Thesis

A CLP(FD) based model checker for CTL

by

Marcus Eriksson

LITH-IDA-EX—05/056—SE

2005-06-08

Supervisor: **Vladislavs Jahundovics**

Department of Computer and Information Science
at Linköpings universitet

Examiner: **Ulf Nilsson**

Department of Computer and Information Science
at Linköpings universitet

Abstract

Model checking is a formal verification method where one tries to prove or disprove properties of a formal system. Typical systems one might want to prove properties within are network protocols and digital circuits. Typical properties to check for are safety (nothing bad ever happens) and liveness (something good eventually happens).

This thesis describes an implementation of a sound and complete model checker for Computation Tree Logic (CTL) using Constraint Logic Programming over Finite Domains (CLP(FD)). The implementation described uses tabled resolution to remember earlier computations, is parameterised by choices of computation strategies and can with slight modification support different constraint domains. Soundness under negation is maintained through a restricted form of constructive negation.

The computation process amounts to a fixpoint search, where a fixpoint is reached when no more extension operations has any effect. As results show, the choice of strategies does influence the efficiency of the computation. Soundness and completeness are of course independent of the choice of strategies. Strategies include how to choose the extension operation for the next step and whether to perform global or local rule instantiations, resulting in bottom-up or top-down computations respectively.

Acknowledgements

I would like to thank Ulf Nilsson and Vladislavs Jahundovics for explaining some of the less obvious parts of the theory involved, for their guiding comments during the thesis work and for being patient with my inability to follow the time schedule. I would also like to thank Mikael Asplund for his input in our discussions on syntax and semantics of the source program and Simon Elén for being my opponent.

Contents

1	Introduction	1
1.1	Problem	1
1.2	Solution	2
1.3	Reading instructions	2
2	Preliminaries	3
2.1	Constraint Logic Programming (CLP)	3
2.2	Computation Tree Logic (CTL)	5
2.2.1	CLP representation of CTL	6
2.3	Overview of the computation model	10
2.3.1	Resolution operations	11
2.3.2	Instantiation operations	12
2.3.3	Updating operations	13
3	The computation model	15
3.1	Input specification	15
3.1.1	CLP program	15
3.2	The scanner	16
3.3	The parser	17
3.4	The fixpoint engine	17
3.4.1	Structures	17
3.4.2	Update mechanism	18
3.4.3	Extension operations	20
3.4.4	Soundness of the computation model	23
3.4.5	Completeness of the computation model	24
3.4.6	Computation process	27
3.5	The output	27
4	Strategies	31
4.1	Extension picking strategy	31
4.1.1	Preference ordering on entry types	31
4.1.2	Weighted random selection	32
4.2	Entry picking strategy	32
4.3	EBC clause picking strategy	32
4.4	EBC locality strategy	32

4.4.1	Global EBC	33
4.4.2	Local EBC	33
5	Experimental evaluation	35
5.1	Input program	35
5.2	Input interpretation	36
5.3	Discussion about the results	38
6	Conclusions	41
6.1	Summary	41
6.2	Results	41
6.3	Future improvements	42
	Bibliography	43
A	BNF for input program	45
B	Configuration file example	47

Chapter 1

Introduction

A formal system, according to Encyclopedia Britannica, is a *formal language together with a deductive apparatus by which some well-formed formulas can be derived from others*. Formal verification is then the process of proving or disproving properties within this formal system.

An instance of formal verification is model checking [4], which is an intriguing problem in computer science today. The model is often derived from a hardware specification or protocol (e.g. network communication protocols) and it is of great value to be able to check properties such as liveness or safety of the model in a mathematically rigorous way.

Since the invention of symbolic model checking [8], where a set of states can be implicitly represented by a logic formula as opposed to listing all states explicitly, model checking has been considerably more useful.

Nilsson and Lübcke [9] presented a way of doing model checking using Constraint Logic Programming [6], which is currently being revised by Jahundovics and Nilsson [7]. Their approach was to use *tabled resolution* [14] to cache results of computations and a restricted form of *constructive negation* [13] to maintain soundness for general programs.

The caching also made it possible to solve a goal partially and move on to solving another goal in between. This permitted a great number of possible computation paths and picking which computation path to pursue was the basis on which to define computation strategies, such as top-down or bottom-up computation or a hybrid of the two.

1.1 Problem

When implementing a model checker based on Constraint Logic Programming, one would naturally attempt to use Prolog or some other similar system which already has support for Constraint Logic Programming and merely implement the missing parts, but in this case it would be like implementing a Prolog interpreter in Prolog, since some of the missing parts is basic system functionality.

Perhaps the most recognised problems with Prolog's control strategy are infinite loops and redundant computations. This is because Prolog does not remember earlier computations and therefore is deemed to repeat them every time they are needed. Often infinite loops can be avoided by rewriting the program, but as is the case with the fixpoint calculations described in this thesis, this is just not practical. Another problem with most Prolog implementations is the lack of a sound negation strategy.

The goal of this thesis and the underlying implementation is to provide a facility for testing different choices of computation strategies when performing model checking of CTL specifications and to see how well this can be done using constraint logic programs over finite domains. The implementation should also permit extensive parameterisation through choice of computation strategies and different finite domains. Soundness and completeness are paramount, of course. This requires, among other things, a sound negation strategy. The model checker described herein uses the same kind of restricted constructed negation as in [7].

1.2 Solution

To deal with redundant computations and infinite loops, we employ *tabled resolution*. This enables us to store previous computations in an internal abstract table and use them when a compatible call is made later on. An introduction to the concept of tabled resolution (albeit without constraints) can be found in [2].

Having an internal table of stored computations can be quite space-consuming for large programs, since each entry may spawn several other entries, which theoretically makes for an exponential growth of the table. A solution for this is to merge similar entries. This will keep the number of entries in the table linear to the number of clauses in the program.

Efficient merging also requires an update mechanism that can be applied when an existing entry is augmented with a new one. This mechanism is described in-depth in Section 3.4.2.

1.3 Reading instructions

This thesis report consists of three main parts. The first is the introductory part, which is covered in this chapter and Chapter 2, which explains the underlying concepts. The second part describes the implementation of the fixpoint engine, wherein Chapter 3 describes the computation model and Chapter 4 describes various strategies used to influence the computation process. The last part shows an experimental evaluation (Chapter 5) and draws conclusions (Chapter 6).

Chapter 2

Preliminaries

In this chapter, we present the underlying definitions and theory used in Chapter 3. This mostly includes definitions from the field of Constraint Logic Programming (CLP).

2.1 Constraint Logic Programming (CLP)

Constraint Logic Programming [6] aims to determine under which constraints a goal in a logic program holds. A constraint logic program looks like any logic program, with the exception that each program rule has an associated constraint, saying “This rule is applicable if this constraint is satisfied.”

Definition 2.1 (constraint logic program) *A constraint logic program consists of a set of universally quantified program rules (clauses)*

$$A_0 \leftarrow C \mid B_1, \dots, B_n. \quad (n \geq 0)$$

where A_0 is an atomic formula (the head), B_1, \dots, B_n are body literals (or subgoals of the rule) and C is a constraint. The meaning of such a rule is that A_0 is satisfied whenever C and B_1, \dots, B_n are satisfied.

Definition 2.2 (literal) *A literal is a possibly negated atomic formula. The terms positive literal and negative literal will be used to refer to non-negated and negated atomic formulae.*

Definition 2.3 (call, answer) *A program rule with one or more body literals is called a call or a non-unit clause. A program rule with no body literals is called an answer or a unit clause.*

Definition 2.4 (renaming substitution) *A renaming substitution θ is a bijective mapping $VAR \rightarrow VAR$ on the set VAR of variables.*

Definition 2.5 (variant) Two formulae F_1 and F_2 are called variants if there exists a renaming substitution θ such that $F_1\theta = F_2$.

A special case of CLP is $CLP(FD)$, that is *Constraint Logic Programming over a Finite Domain*. In $CLP(FD)$ the set of constraint variables VAR range over some finite set VAL of values. A typical finite set is the set $\{true, false\}$ of Boolean values, which would permit us to have Boolean expressions as constraints.

The set of solutions $sol(C)$ of a constraint C are all variable mappings $\sigma : VAR \rightarrow VAL$ that render the constraint C true.

The implementation described in this thesis uses the binary decision diagram package BuDDy (see [1] for an explanation of Binary Decision Diagrams) over Boolean values, but can with some simple modifications support any set CON of constraints supporting the basic operations $\oplus : CON \times CON \rightarrow CON$, $\otimes : CON \times CON \rightarrow CON$, $\Pi : VAR^* \times CON \rightarrow CON$, $\odot : CON \times CON \rightarrow CON$, $\neg : CON \rightarrow CON$ and $\diamond : VAR \times VAL \rightarrow CON$, satisfying the following:

- $sol(C_1 \oplus C_2) = sol(C_1) \cup sol(C_2)$
- $sol(C_1 \otimes C_2) = sol(C_1) \cap sol(C_2)$
- $\theta \in sol(\Pi_{\bar{x}}C)$ iff there exists $\theta' \in sol(C)$ such that $\forall x_i \in \bar{x} : \theta(x_i) = \theta'(x_i)$
- $sol(\neg C) = sol(true)^1 \setminus sol(C)$
- $sol(C_1 \odot C_2) = sol((C_1 \otimes \neg C_2) \oplus (\neg C_1 \otimes C_2))$
- $\theta \in sol(x \diamond v)$ iff $\theta(x) = v$

That is, we require disjunction, conjunction, projection, complementation, symmetric difference and binding of values to variables.

Below are some examples. A valuation $\{V_1 \rightarrow n_1, V_2 \rightarrow n_2, V_3 \rightarrow n_3\}$ is written (n_1, n_2, n_3) . A value of $*$ denotes that any value in VAL is permitted, so $\{(1, 2, *)\}$ acts as a short form for $\{(1, 2, X) \mid X \in VAL\}$.

$$\begin{aligned}
 & \frac{VAR = \{V_1, V_2, V_3\} \quad VAL = \{1, 2, 3\}}{sol((V_1 \diamond 1) \otimes \neg(V_2 \diamond 2))} \\
 &= sol(V_1 \diamond 1) \cap sol(\neg(V_2 \diamond 2)) \\
 &= \{(1, *, *)\} \cap (\{(*, *, *)\} \setminus \{(*, 2, *)\}) \\
 &= \{(1, *, *)\} \cap \{(*, 1, *), (*, 3, *)\} \\
 &= \{(1, 1, *), (1, 3, *)\} \\
 &= sol(\Pi_{\{V_2, V_3\}}((V_1 \diamond 1) \otimes \neg(V_2 \diamond 2))) \\
 &= \{(*, 1, *), (*, 3, *)\}
 \end{aligned}$$

Unifiers are extensively used throughout the model description. The unifiers used here are *directed most general unifiers*, or *dmgus* for short. The motivation for such a construct is that we will need to control which variables occur in the terms we create.

¹*true* represents the constraint that is always satisfied

Definition 2.6 (directed most general unifier (dmgu)) A directed most general unifier $dmgu(s, t)$ of two terms s and t is a most general unifier of s and t in which all variable-to-variable mappings are from variables in t to variables in s .

Whenever there exists an mgu , there also exists a $dmgu$. The example below should show this intuitively.

$s = f(X, Y)$		$t = f(S, g(Z))$
<hr/>		
unifier	$mgu(s, t)?$	$dmgu(s, t)?$
$\{X \rightarrow S, Y \rightarrow g(Z)\}$	YES	NO
$\{S \rightarrow X, Y \rightarrow g(Z)\}$	YES	YES

2.2 Computation Tree Logic (CTL)

Only a brief survey of CTL is supplied here. For further insights on CTL and other temporal logics, see [5].

CTL is a temporal logic. This means that it can express properties about present and future states. It also supports quantifying existentially or universally on the possible paths to get to these states. A state in CTL is a valuation of the state variables (which might be B_1, \dots, B_n when we operate in the Boolean domain and need no more than 2^n states). Modifying one or more state variables moves us into another state.

For expressing the CTL operators, we need a binary transition relation R that describes which pairs of states that express possible transitions in our model. $\langle \sigma_1, \sigma_2 \rangle \in R$ (or, alternatively expressed, $\sigma_1 R \sigma_2$) iff it is possible to go from state σ_1 to σ_2 in a single step. It is assumed that every state permits at least one transition (possibly into itself). The transition relation then implicitly defines a set of infinite transition paths $\sigma_0 \sigma_1 \dots$, given an initial state σ_0 . In Table 2.1, such an underlying static transition relation is assumed. $\sigma_0 \models F$ means that the property F holds in the state σ_0 .

$\sigma_0 \models x = v$	iff $x \in VAR, v \in VAL$ and $\sigma_0(x) = v$
$\sigma_0 \models F_1 \wedge F_2$	iff $\sigma_0 \models F_1$ and $\sigma_0 \models F_2$
$\sigma_0 \models F_1 \vee F_2$	iff $\sigma_0 \models F_1$ or $\sigma_0 \models F_2$ (or both)
$\sigma_0 \models \neg F$	iff $\sigma_0 \not\models F$
$\sigma_0 \models ex(F)$	iff there is a path $\sigma_0 \sigma_1 \dots$ such that $\sigma_1 \models F$
$\sigma_0 \models eg(F)$	iff there is a path $\sigma_0 \sigma_1 \dots$ such that $\sigma_i \models F$ for every $i \geq 0$
$\sigma_0 \models eu(F_1, F_2)$	iff there is a path $\sigma_0 \sigma_1 \dots$ and an $i \geq 0$ such that $\sigma_j \models F_1$ for every $0 \leq j < i$ and $\sigma_i \models F_2$
$\sigma_0 \models ef(F)$	iff there is a path $\sigma_0 \sigma_1 \dots$ such that $\exists i \geq 0 : \sigma_i \models F$

Table 2.1. CTL operators

In addition to the operators listed in Table 2.2, there are also operators ax , ag , au and af . Their meaning differ from ex , eg , eu and ef only in that “if there is

a path $\sigma_0\sigma_1\dots$ such that” is changed into “for all paths $\sigma_0\sigma_1\dots$ ”. For example, $\sigma_0 \models ax(F)$ means “ F holds in every state that can be reached from σ_0 in one step”.

To remember the meaning of the CTL operators, one could associate the letters in the following manner.

e	there Exists
a	for All
x	in the neXt
g	is Globally satisfied (now and in the future)
u	is satisfied Until
f	is satisfied some time in the Future

2.2.1 CLP representation of CTL

When defining the CLP representation of the CTL operators defined in Table 2.1, one needs to define a few predicates.

holds/2 takes a (ground) CTL formula and a state description as arguments. The meaning of *holds(F, S)* is “ F holds in the state S ”.

step/2 takes two state descriptions as arguments and returns true iff there is a transition from the first state to the second.

sat/1 is the *satisfies*-predicate. It takes a formula of constraint variables as its argument and expresses a constraint. This predicate is treated in a special way, since its semantics are different from the other predicates’.

The complete list of CTL operators may then be expressed as in Table 2.2. Some of the operators, are expressed in terms of other operators. For example, *eg(F)* is equivalent to $\neg af(\neg F)$, since “there exists an infinite path where F is always satisfied” is equivalent to “it is not true that all paths sooner or later dissatisfy F ”.

The transition relation *step/2* is easiest described by having a single *sat*-formula on the right hand side. An example of a transition relation over states expressed by two state variables (a maximum of four states) that permits transitions from $\langle 1, 1 \rangle$ to $\langle 1, 0 \rangle$ and from $\langle 1, 0 \rangle$ to $\langle 1, 1 \rangle$ and to all other states to themselves² is then

$$\begin{aligned} \text{step}([S_1, S_2], [S'_1, S'_2]) \leftarrow & \text{sat}(\text{or}(\text{and}(S_1 = 1, \text{and}(S_2 = 1, \text{and}(S'_1 = 1, S'_2 = 0))), \\ & \text{or}(\text{and}(S_1 = 1, \text{and}(S_2 = 0, \text{and}(S'_1 = 1, S'_2 = 1))), \\ & \text{or}(\text{and}(S_1 = 0, \text{and}(S_2 = 0, \text{and}(S'_1 = 0, S'_2 = 0))), \\ & \text{and}(S_1 = 0, \text{and}(S_2 = 1, \text{and}(S'_1 = 0, S'_2 = 1)))))). \end{aligned}$$

or, in infix notation

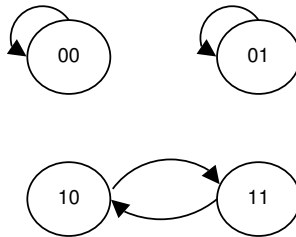
²As defined in Section 2.2, all states must have a transition to some state.

$holds(or(F, G), S)$	$\leftarrow holds(F, S).$
$holds(or(F, G), S)$	$\leftarrow holds(G, S).$
$holds(and(F, G), S)$	$\leftarrow holds(F, S), holds(G, S).$
$holds(not(F), S)$	$\leftarrow \sim holds(F, S).$
$holds(xor(F1, F2), S)$	$\leftarrow holds(F1, S), \sim holds(F2, S).$
$holds(xor(F1, F2), S)$	$\leftarrow holds(F2, S), \sim holds(F1, S).$
$holds(ex(F), S)$	$\leftarrow step(S1, S2), holds(F, S2).$
$holds(eg(F), S)$	$\leftarrow holds(not(af(not(F))), S).$
$holds(eu(F, G), S)$	$\leftarrow holds(G, S).$
$holds(eu(F, G), S)$	$\leftarrow holds(F, S), holds(ex(eu(F, G))), S).$
$holds(ef(F), S)$	$\leftarrow holds(eu(true, F), S).$
$holds(ax(F), S)$	$\leftarrow holds(not(ex(not(F))), S).$
$holds(ag(F), S)$	$\leftarrow holds(not(ef(not(F))), S).$
$holds(au(F1, F2), S)$	\leftarrow $holds(and(not(eu(not(F2), and(not(F1), not(F2)))), not(eg(not(F2)))), S).$
$holds(af(F), S)$	$\leftarrow holds(F, S).$
$holds(af(F), S)$	$\leftarrow holds(ax(af(F)), S).$
$holds(true, S).$	

Table 2.2. CLP version of CTL operators

$$\begin{aligned}
step([S_1, S_2], [S'_1, S'_2]) \leftarrow & sat((S_1 \wedge S_2 \wedge S'_1 \wedge \neg S'_2) \\
& \vee (S_1 \wedge \neg S_2 \wedge S'_1 \wedge S'_2) \\
& \vee (\neg S_1 \wedge \neg S_2 \wedge \neg S'_1 \wedge \neg S'_2) \\
& \vee (\neg S_1 \wedge S_2 \wedge \neg S'_1 \wedge S'_2))
\end{aligned}$$

or, in graph notation



The expressions above is written as a disjunction of steps from the state (S_1, S_2) to the state (S'_1, S'_2) , but any equivalent Boolean constraint expression would also do. The constraints on the state variables tells under which conditions a state transition is allowed.

Negation concerns

The usage of negation in CTL requires us to ask questions of the form “Which variable substitutions do *not* satisfy this formula?”. This raises a concern that we have to deal with.

What if we have the illustrative example program in Figure 2.1?

$p(X) \leftarrow \sim q(X).$
$q(1).$
$r(2).$
$r(3).$

Figure 2.1. Illustrative example program

Here, we would like to get the answer $X \neq 1$ or $X \in \{2, 3\}$ if we seek an answer to the question $p(X)$. The latter means calculating the Herbrand universe and then complementing the answers to $q(X)$ within this universe.

The kind of negation we need is called *constructive negation* [13] and basically amounts to taking the complement of the disjunction of all answers to a positive goal.

Definition 2.7 (alphabet, functor) *The alphabet of a program P is the set of functors (symbols denoting objects) and predicate symbols (symbols denoting relations between objects) of P . A functor takes zero or more arguments from the object domain to construct an object. A functor with arity zero is called a constant.*

Definition 2.8 (Herbrand universe) *The Herbrand universe U_A of an alphabet A is the set of all possible ground terms possible to express using the functors of A .*

$P_1 :$	$holds(and(F_1, F_2), S) \leftarrow holds(F_1, S), holds(F_2, S).$
$P_2 :$	$holds(a, [S_1, S_2]) \leftarrow sat(S_1 = 1).$
$P_3 :$	$holds(b, [S_1, S_2]) \leftarrow sat(S_2 = 1).$
$P_4 :$	$query([B_1, B_2]) \leftarrow holds(and(a, b), [B_1, B_2]).$
<hr/>	
predicate symbols:	$query/1, holds/2, sat/1$
functors:	$and/2, a/0, b/0$

Figure 2.2. Example program

Part of the Herbrand universe of the program in Figure 2.2 would be

$$\{a, b, and(a, b), and(and(a, and(b, a))), and(and(a, a), and(b, b)), \dots\}$$

We see that as soon as we have a functor taking at least one argument, we will have a Herbrand universe of infinite size. This is then clearly also the case with our representation of CTL (Table 2.2), since it contains several such functors.

Luckily, by imposing a very sensible restriction on our queries, we can avoid this problem. The solution is to ground all CTL formulae in our queries. This actually makes perfect sense, since we are hardly interested in asking “From which states can we go to another state where *any* formula is true?”. By inspecting Table 2.2, we see that as long as we keep our queries semi-ground, the first argument to *holds/2* in all subsequent calls will also be ground and our negation concerns reduced to negating so-called *semi-ground* formulae. This is enough, since the constraint variables are only placeholders for the constraint values we return.

Definition 2.9 (semi-ground) *A formula is semi-ground if it contains no variables except for constraint variables.*

Definition 2.10 (linear) *A formula is linear if no variable in it occurs more than once.*

Definition 2.11 (state vector) *A state vector is a sequence of state variables $[V_1, \dots, V_n]$ that together with a constraint on the same state variables expresses a set of states of the system we are simulating.*

Definition 2.12 (semi-ground canonical form) *A formula is in semi-ground canonical form if it is semi-ground and linear and all state variables occur only in state vectors.*

Examples of formulae and properties follow. $VAR = \{V_1, V_2\}$.

	Semi-ground	Linear	State variables only in state vectors
$h([V_1, V_2, V_1]).$	YES	NO	YES
$h([V_1]) \leftarrow b_1(V_1).$	YES	NO	NO
$h([V_1, V_2]) \leftarrow b_1(X).$	NO	YES	YES
$h_1(a, [V_1, V_2]).$	YES	YES	YES

Definition 2.13 (answer tuple) *An answer tuple is a tuple $\langle H, A \rangle$ where*

H is a positive literal in semi-ground canonical form.

A is a constraint that represents a set of answers associated with H .

Depending on the context in which it is used, an answer tuple will be an alternative answer tuple or a bounding answer tuple. The difference between these will be explained in Section 2.3.

Definition 2.14 (call tuple) *A call tuple is a tuple $\langle H, B, C, P \rangle$ where*

H is a positive literal in semi-ground canonical form, representing the head of a call.

B is a non-empty sequence of literals, the body literals of a call.

C is the constraint so far accumulated for this call tuple.

P is the set $\{P_1, \dots, P_n\}$ of program clauses that can be used for EBC (see Section 2.3) with this call tuple.

Using the program in Figure 2.2 as our input program, some of the possible call tuples and answer tuples that may arise are:

Call tuples			
H	B	C	P
$query([B_1, B_2])$	$(holds(and(a, b), [B_1, B_2]))$	$true$	$\{P_1\}$
$holds(and(a, b), [B_1, B_2])$	$(holds(a, [B_1, B_2]),$ $holds(b, [B_1, B_2]))$	$true$	$\{P_2\}$
$holds(and(a, b), [B_1, B_2])$	$(holds(b, [B_1, B_2]))$	$true \otimes (B_1 = 1)$	$\{P_3\}$

Answer tuples	
H	A
$holds(a, [B_1, B_2])$	B_1
$holds(b, [B_1, B_2])$	B_2
$holds(and(a, b), [B_1, B_2])$	$true \otimes (B_1 = 1) \otimes (B_2 = 1)$

This thesis covers only the temporal logic CTL, but any temporal logic expressed in CLP(FD) meeting the requirements of a correct input program (see Section 3.1.1) will do.

2.3 Overview of the computation model

This section will give an informal explanation to the computation model, in order to make the exact definition later on easier to understand.

All extension operations operate on an *abstract table*. The actual implementation may use a set or another data structure. Lookup and insert operations are used frequently, and each entry is guaranteed to be unique³ so the implementation described in this thesis uses several sets (giving logarithmic time complexity for both insertion and lookup). The term used for this abstract data structure is simply *table*.

The entries that populate the table are call tuples and answer tuples. The answer tuples come in two flavours—alternatives and (upper) bounds—which differ only in the way that they are used.

³Entries whose clauses are variants are merged, while keeping information about renaming substitutions to separate them.

The main idea behind the computation model is that the order in which the extension operations are applied does not matter, as far as soundness and completeness are concerned. By using different computation strategies, one can influence different aspects of the computation, which will in turn influence the amount of resources used. The strategies will be discussed in Chapter 4.

Before any of the extension operations take place, the table needs to contain something to operate on. This will be our query (a semi-ground CLP program clause, e.g. P_4 in Table 2.2), given along with the rest of the CLP program. Extensive definitions of the input program syntax and semantics are given in Section 3.1.

When none of the possible extension operations would change the table (by adding new entries or modifying existing ones) we have reached our sought fixpoint and the answers to the initial query can be extracted from the final table.

There are three types of extension operations on the table, namely resolution operations, instantiation operations and updating operations.

2.3.1 Resolution operations

The resolution operations are Extension By Answer Alternative (EBAA) and Extension By Closed World Assumption (EBCWA).

EBAA

Given one call tuple C and one alternative answer tuple A in the table, a new entry is created by intersecting the set of answers for C with the set of answers of A and removing the first body literal of C . This new entry is then inserted into the table. The requirement is of course that the head of A is relevant for (unifies with) the first body literal of C . The following is an example of the EBAA operation. The entries above the arrow are present in the table and the entry below the arrow is added. $C_{2[Z \rightarrow X]}$ in the example is the constraint C_2 with all occurrences of Z replaced by X .

$h(X, Y) \leftarrow C_1 \mid b_1(X), b_2(Y), b_3(Z).$	∈ table
$b_1(Z) \leftarrow C_2.$	∈ table
\Downarrow	
$h(X, Y) \leftarrow C_1 \otimes C_{2[Z \rightarrow X]} \mid b_2(Y), b_3(Z).$	∈ table

The update mechanism (described in Section 3.4.2) needs to know which entries that were added by EBAA operations and which call tuples that were used, so we need to store these *answer extensions*.

Definition 2.15 (answer extension) An answer extension is a tuple $\langle E_i, E_k, \sigma \rangle$ where E_m is the entry obtained through an EBAA on E_i , E_k is an entry stored in the table and $\sigma = \text{dmgu}(E_k, E_m)$ exists. Sometimes E_k is also called the answer extension of E_i (by σ).

All answer extensions are stored as a labeled DAG (Directed Acyclic Graph) on the entries, where σ constitutes the label. This DAG will be used by the update mechanism described in Section 3.4.2. Figure 2.3 shows how part of such a DAG could look like.

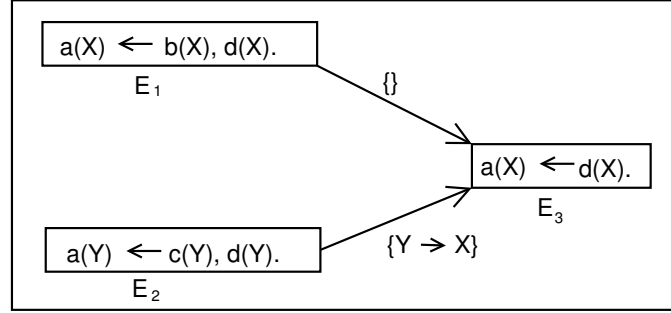


Figure 2.3. DAG for the answer extensions $\{\langle E_1, E_3, \{\} \rangle, \langle E_2, E_3, \{Y \rightarrow X\} \rangle\}$, obtained from two EBAA operations—first using E_1 and then using E_2 .

EBCWA

Many logic programming systems operate under the closed world assumption, in which lack of evidence of a property is taken as evidence to the contrary. The closed world assumption made in our implementation is that if a body literal L of a call tuple in the table does not unify with the head of any clause in the input program, we infer that there is no way to solve L .

If L is a positive literal, there exists no solution to the whole body, since any solution must satisfy all body literals.

But if L is a negative literal we have the case that the negation of what we want to prove is false. Then we simply remove L and continue.

2.3.2 Instantiation operations

Instantiating operations create new entries to put in the table. The only instantiation operation is Extension By Call (EBC).

EBC

Given one call tuple in the table and a clause from the input program, a new entry is created by unifying the head of a copy of the program clause and the first body literal of the call tuple. An example of the EBC operation is shown below.

$a(X_1) \leftarrow C_1 \mid h(X_1, X_2), b_3(X_2).$	$\in \text{table}$
$h(Z, X_1) \leftarrow C_2 \mid b_1(Z, X_1), b_2(X_1).$	$\in \text{input program}$
\Downarrow	
$h(X_1, X_2) \leftarrow C_1 \otimes C_2[Z \rightarrow X_1, X_1 \rightarrow X_2] \mid b_1(X_1, X_2), b_2(X_2).$	$\in \text{table}$

The meaning of this operation is to make sure a subgoal (the first body literal) of the chosen call tuple is solved eventually. When this subgoal is actually solved depends on the choice of strategies (see Chapter 4).

2.3.3 Updating operations

Updating operations do not actually extend the table, but propagate it towards the fixpoint. The only updating operation is Extension By Answer Bound (EBAB).

EBAB

Through the course of computation, we might end up with bounding answer tuples in our table. A bounding answer tuple with a head H states that the answers for H must be within those specified by the bound. The final set of answers will be $(B_1 \otimes \dots \otimes B_n) \otimes (A_1 \oplus \dots \oplus A_m)$ where $\{\langle H, A_1 \rangle, \dots, \langle H, A_m \rangle\}$ are the alternative answer tuples for H and $\{\langle H, B_1 \rangle, \dots, \langle H, B_n \rangle\}$ are the bounding answer tuples for H .

To make sure that this property holds even for entries previously processed, an update mechanism is used on the table. The update mechanism will be covered in the detailed computation model description in the next chapter.

Chapter 3

The computation model

The computation model described in this chapter is an implementation of the abstract computation model described in [7] and outlined in the previous chapter. A few changes have been made to Jahundovics and Nilsson's abstraction, but as demonstrated later, soundness and completeness are still achieved. The implementation consists of several components (Figure 3.1) which will be explained in detail in this chapter.

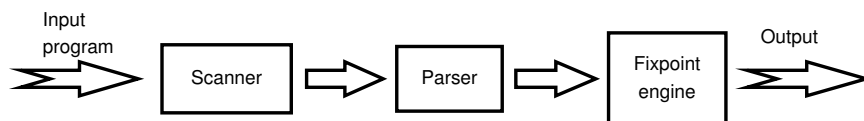


Figure 3.1. Components of the implementation

3.1 Input specification

The input to the program consists of two parts. Firstly, a configuration file specifying at least which strategies to use (see Chapter 4) and which output to produce. The second type of input is the complete CLP program and the query for which to seek all the answers.

3.1.1 CLP program

The syntax of the CLP program mimics that of the logic programming language Prolog [10], with some restrictions. A BNF for the grammar used can be found in Appendix A. The last clause of the input program, as shown in the BNF, is the query to be answered.

Some additional requirements on the input program are required for the unifier and entry merging (see Section 3.4.1) to work.

linearity All literals must be linear. This is because the merging of two entries (see Section 3.4.1) presupposes that there is always a renaming substitution unifying the stored entry and the one added.¹

preserved typed arity If two functors or predicate symbols in the program share the same name, they also share the same arity and type of their arguments.²

semi-ground query The head of the query clause must be semi-ground and all its body literals must be in semi-ground canonical form.

instance semi-groundness All instances of program clauses must be in semi-ground form. This implies that all variables of a program rule that is not mentioned in the head are state variables (in state vectors).

variable representation The head of a clause (including the query) must contain all state variables that are to be included in the answer to the head of that clause. This is because variables mentioned in the body but not in the head act as existentially quantified in the body.

The *semi-ground query* requirement and the *instance semi-groundness* requirement together ensure that all added entries will be in semi-ground canonical form.

Constraint expressions

A focal point of constraint logic programming is the constraints themselves. To express a constraint associated with a clause, a *sat*/1-literal is put in the body (see Appendix A). These constraint literals do not count as ordinary body literals and are thus not referred to as such in the future.

Below is an example of a rule with the associated constraint “ S_1 is not satisfied”. Earlier examples of *sat* in Chapter 2 used the notation $S_1 = 0$, but what is written below is how it actually looks like in the input program, since only Boolean constraints are supported.

```
holds(prop, [S1, S2]) :- sat(not(S1)), holds(subprop, [S1, S2]).
```

3.2 The scanner

The scanner reads the program text, consisting of the step relation, the semantically complete fragment of CTL and the query to be evaluated. It then outputs semantically marked symbols for the parser to use in the next step.

The scanner generator Flex [11] was used to create the scanner.

¹This imposes a slight restriction on the program, since it is not possible to express a predicate that takes two equal arguments (e.g. $p(X, X)$). It is still possible with state list arguments, since it is possible to equate two state variables by using *sat*-clauses.

²Thus there can be no case where a functor takes a state list as its first argument and another case where it takes a constant as its first argument. The program has to be constructed in such a way that this never occurs. Variables do not count as a type in this regard, since they can represent any type.

3.3 The parser

The parser reads the symbols given by the scanner and produces an abstract syntax tree according to an LR(1) grammar (see Appendix A). In this process, constraint expressions are interpreted (so it is only done once). After parsing is done, the fixpoint search begins.

The parser generator Bison [3] was used to create the parser.

3.4 The fixpoint engine

Once the program and the supplied query has been fully parsed, the actual model checking can begin. By repeatedly applying one of several extension operations, which modify the table, until no more rules can be applied, a fixpoint will be reached.

The fixpoint engine and auxiliary modules were entirely implemented in C++ using the Standard Library. [12]

3.4.1 Structures

The table³ is a tuple $\langle AnsA, ObAnsA, AnsB, ObAnsB, Calls, ObCalls, Links \rangle$ of disjoint sets where

AnsA is the set of alternative answer tuples derived but not yet used in an EBAA operation.

ObAnsA is the set of alternative answer tuples already used in an EBAA operation.

AnsB is the set of bounding answer tuples that have not yet influenced any used answer alternatives in *ObAnsA*. EBAB operations remove entries from this set.

ObAnsB is the set of bounding answer tuples that have influenced all matching answer alternatives in *ObAnsA*.

Calls is the set of call tuples that still have a possibility for EBC or EBCWA.

ObCalls is the set of call tuples where neither EBC nor EBCWA are possible anymore.

Links is the set of answer extensions (may be perceived as a DAG) created by EBAA operations.

Also included in the computation is the program *Prog*, which is a static list containing all the clauses except the query from the input.

Before computation starts, *Calls* consists of the query and the rest of the table is empty.

³Discussed in Section 2.3.

Adding to the table

In the following sections, we will use the expression “put $\langle E, C \rangle$ into the table”, where E is a clause $H \leftarrow B_1, B_2, \dots, B_n$ ($n \geq 0$) and C is the constraint to be associated with that clause. This convenient notation covers both call tuples and answer tuples. The constraint C is projected onto the state variables occurring in E before being added.

This does not necessarily mean that an entry is added to the table, as $\langle E, C \rangle$ could instead merge with an existing entry $\langle E', C' \rangle$. A prerequisite for merging them is that E and E' are variants. The actual merging amounts to changing the existing constraint C' into $C' \oplus C\theta$, where $\theta = dmgu(E, E')$, and discarding $\langle E, C \rangle$.

Adding an answer tuple If $n = 0$, then E is an (alternative) answer tuple. In this case, we check if there exists a renaming substitution λ and an answer tuple $\langle H\lambda, A_H \rangle \in AnsA$. If it exists, A_H is replaced by $A_H \oplus C\lambda$. Otherwise, a new answer tuple $\langle H, C \rangle$ is inserted into $AnsA$.

Adding a call tuple If $n > 0$, we are dealing with a call tuple. In this case, we first check $Calls$ for a call tuple $\langle H', (B'_1, B'_2, \dots, B'_n), C', P' \rangle$. If such a call tuple exists, and $\theta = dmgu((H' \leftarrow B'_1, B'_2, \dots, B'_n), E)$ exists, then C' is replaced by $C' \oplus C\theta$. If there exists no such call tuple, a new call tuple $\langle H, (B_1, B_2, \dots, B_n), C, P \rangle$ is inserted into $Calls$, where P is a set containing the program clauses from $Prog$ whose heads unifies with H .

When adding a call, we also need to instantly apply an answer extension by any previously used answers for the first body literal B to it. This is necessary since the answers already used will not reoccur and we need to incorporate them also into our added entry. The answer to use (σ_a, σ_b unifiers) in the extension is $C_a \otimes C_b$, where C_a is the answer alternative ($A_a\sigma_a^{-1}$ if $\langle B\sigma_a, A_a \rangle \in ObAnsA$, otherwise *true*) and C_b is the answer bound ($A_b\sigma_b^{-1}$ if $\langle B\sigma_b, A_b \rangle \in ObAnsB$, otherwise *true*). If no earlier answer (alternative or bound) for B existed, nothing is done.

3.4.2 Update mechanism

When working with a table in which some entries depend on others, modifying one entry might require modifying others. In this computation model, we sometimes find additional answer constraints, which we must incorporate into the table. The main idea of the update mechanism’s functionality is that the final result should be the same independent of the order in which the operations were applied.

This amounts to having an update mechanism consisting of two mutually recursive algorithms, which handle disjunctive and conjunctive constraint updates respectively. The algorithms traverse along the answer extensions defined in *Links*. Different actions are performed depending on whether the literal consumed in the answer extension was negative or not.

The update mechanism guarantees that all necessary entries in the table are affected by all previously processed answers before any given application of an

extension operation. This is required for soundness, as explained in Section 3.4.4.

Disjunctive constraint update

A disjunctive constraint update on a table entry Ent by a constraint C means changing the constraint C_{Ent} currently associated with Ent into $C_{Ent} \oplus C$. We also need to recursively modify the answer extension of Ent , if it exists. This is done in the following manner.

If Ent is an answer tuple If Ent is an answer tuple $\langle H, X \rangle$, it means we have found an answer alternative for H . If we already have an entry $\langle H\lambda, A \rangle \in AnsA$, we change it into $\langle H\lambda, A \oplus C\lambda \rangle$. Otherwise, we insert $\langle H, C \rangle$ into $AnsA$.

By definition, answer tuples have no answer extensions.

If Ent is a call tuple with a negative first body literal Ent 's constraint C_{Ent} is changed into $C_{Ent} \oplus C$. This means that an answer extension Ext from Ent needs to have its constraint changed from $C_{Ent} \otimes \neg C'$ to $(C_{Ent} \oplus C) \otimes \neg C'$, where $\neg C'$ is the answer used for Ent 's first body literal.

Since this is the same as $(C_{Ent} \otimes \neg C') \oplus (C \otimes \neg C')$, what we do is simply to recursively make a disjunctive constraint update by $C \otimes \neg C'$ on Ext .

If Ent is a call tuple with a positive first body literal By a similar reasoning as for a negative first body literal, we arrive at the conclusion that we should make a disjunctive constraint update by $C \otimes C'$ on the answer extension Ext (if it exists) instead.

Conjunctive constraint update

A conjunctive constraint update on a table entry Ent by a constraint C means changing the constraint C_{Ent} currently associated with Ent into $C_{Ent} \otimes C$. We also need to modify all entries that are answer extensions of Ent . This is done in the following manner.

If Ent is an answer tuple If Ent is an answer tuple $\langle H, X \rangle$, it means we have found an answer bound for H . If we already have an entry $\langle H\lambda, A_b \rangle \in AnsB$, we change it into $\langle H\lambda, A_b \otimes C\lambda \rangle$. Otherwise, if we have an entry $\langle H\theta, A_a \rangle \in ObAnsA$, we insert $\langle H, C \rangle$ into $AnsB$.

If we neither have $\langle H\lambda, A_b \rangle \in AnsB$ nor $\langle H\theta, A_a \rangle \in ObAnsA$, it means that no answer alternative has been used for any variant of H yet. In this case, we may just change $ObAnsB$ directly, since no EBAA have had use for the bounds in $ObAnsB$ anyway. This slightly ad-hoc solution is necessary to fulfill the requirement that an alternative has been already applied when applying EBAB (see Section 3.4.3). If $\langle H\omega, A' \rangle \in ObAnsB$, we change it into $\langle H\omega, A' \otimes C\omega \rangle$, otherwise we insert $\langle H, C \rangle$ into $ObAnsB$.

By definition, answer tuples have no answer extensions.

If Ent is a call tuple Ent 's constraint C_{Ent} is changed into $C_{Ent} \otimes C$. This means that an answer extension Ext from Ent needs to have its constraints changed from $C_{Ent} \otimes C'$ to $(C_{Ent} \otimes C) \otimes C'$, where C' is the answer used for Ent 's first body literal.

Since this is the same as $(C_{Ent} \otimes C') \otimes C$, what we do is simply to recursively make a conjunctive constraint update by C on Ext .

3.4.3 Extension operations

There are four possible extension operations on the table. These extension operations may in turn use the update mechanism defined above. Which one of these extension operations to apply in a given situation is determined by the chosen extension strategy. The strategies are covered in detail in Chapter 4.

Extension by call (EBC)

In applying EBC, we hope to find some answers for the next body literal in a call in the table. We do this by unifying the head of an input program clause with the first body literal in the chosen call. After this extension, we remove the used input program clause from the set of matching clauses P to avoid trying to use it again.

Preconditions

- $Call = \langle H_1, (B_{11}, B_{12}, \dots, B_{1n}), C, P \rangle \in Calls$
- $P_i = \langle (H_2 \leftarrow B_{21}, B_{22}, \dots, B_{2m}), C' \rangle \in P$.
- $\theta = dmgu(B_{11}, H_2)$

Postconditions

- $P_i \notin P$
- $sol(C'') \subseteq sol(C')$ and $(sol(C') \cap sol(C)) \subseteq sol(C'')$ and $sol(C'') \neq \emptyset$.
- $\langle (H_2 \leftarrow B_{21}, B_{22}, \dots, B_{2m})\theta, C'' \rangle$ is put into the table.

The loose definition in the second postcondition leaves room for choice through the EBC locality strategy described in Section 4.4.

If P turns into the empty set, then $Call$ is removed from $Calls$ and inserted into $ObCalls$.

Extension by answer alternative (EBAA)

When applying EBAA, we pick an answer alternative $\langle H, A \rangle$ from $AnsA$. If $\langle H\omega, A_{prev} \rangle \in ObAnsA$ and $sol(A) \subseteq sol(A_{prev}\omega^{-1})$ then we simply remove $\langle H, A \rangle$ from $AnsA$ and abort the EBAA operation. This is because the new alternative

is already covered by the disjunction of previous alternatives and would not affect the set of answers when applied.

If we have not aborted already, we take different actions depending on whether this is the first time an answer (alternative or bound) for H is applied. This is because we might have to update call and answer tuples that have been created from the previous answer extensions. We also always have to adhere to any used answer bounds for the answer alternative we are applying.

Preconditions

- $Ans = \langle H, A \rangle \in AnsA$
- If $\langle H\lambda, A_{old} \rangle \in ObAnsB$, then $A_b = A_{old}\lambda^{-1}$. Otherwise $A_b = true$.
- S is the set of all call tuples $Call_i = \langle H_i, B_i, C_i, P_i \rangle$ in $Calls \cup ObCalls$ whose first body literal B_{i1} unifies with H .
- $\theta_i = dmgu(B_{i1}, H)$

Postconditions ($\forall Call_i \in S$) if no earlier answer for H existed

- Ext_i is equal to $Call_i$ with B_{i1} removed and the constraint changed to C' .
- If B_{i1} is negative, $C' = C_i \otimes \neg(A \otimes A_b)\theta_i$. Otherwise $C' = C_i \otimes (A \otimes A_b)\theta_i$.
- Ext_i is put into the table.
- If a new entry is added, $Ext'_i = Ext_i$ and $\lambda_i = \emptyset$. Otherwise Ext'_i is the already existing entry that was modified and $\lambda_i = dmgu(Ext'_i, Ext_i)$.
- A link $\langle Call_i, Ext'_i, \lambda_i \rangle$ is also added to $Links$.
- $\langle H, A \rangle$ is removed from $AnsA$ and inserted into $ObAnsA$.

Postconditions ($\forall Call_i \in S$) if an earlier answer for H existed

- If the first body literal of $Call_i$ is negative, a conjunctive constraint update by $(\neg(A \otimes A_b)\theta_i)\lambda_i$ is performed on Ext_i , where $\langle Call_i, Ext_i, \lambda_i \rangle \in Links$. Otherwise, a disjunctive constraint update by $(C_i \otimes (A \otimes A_b)\theta_i)\lambda_i$ is performed on Ext_i .
- $\langle H, A \rangle$ is removed from $AnsA$ and $\langle H\omega, A_{prev} \rangle \in ObAnsA$ is replaced by $\langle H\omega, A_{prev} \oplus A\omega \rangle$.

Extension by answer bound (EBAB)

When applying EBAB, we pick an answer bound $\langle H, A \rangle$ from $AnsB$. If $\langle H\omega, A_{prev} \rangle \in ObAnsB$ and $sol(A_{prev}\omega^{-1}) \subseteq sol(A)$ then we simply remove $\langle H, A \rangle$ from $AnsB$ and abort the EBAB operation. This is because the new bound is weaker than the conjunction of the previous bounds and would not affect the set of solutions when applied. If we do not abort, we continue with the procedure below.

The only way of adding an answer bound to $AnsB$ is by the update mechanism. That procedure ensures that if we have an answer bound for H in $AnsB$, we have applied an alternative for H some time before (see Section 3.4.2).

Preconditions

- $Ans = \langle H, A \rangle \in AnsB$
- S is the set of all call tuples $Call_i = \langle H_i, B_i, C_i, P_i \rangle$ in $Calls \cup ObCalls$ whose first body literal B_{i1} unifies with H .
- $\theta_i = dmgu(B_{i1}, H)$

Postconditions ($\forall Call_i \in S$)

- If the first body literal of $Call_i$ is negative, a disjunctive constraint update by $(C_i \otimes \neg A\theta_i)\lambda_i$ is performed on Ext_i , where $\langle Call_i, Ext_i, \lambda_i \rangle \in Links$. Otherwise, a conjunctive constraint update by $(A\theta_i)\lambda_i$ is performed on Ext_i .
- $\langle H, A \rangle$ is removed from $AnsB$. If $\langle H\omega, A_{prev} \rangle \in ObAnsB$ then A_{prev} is replaced by $A_{prev} \otimes A\omega$. Otherwise, $\langle H, A \rangle$ is inserted into $ObAnsB$.

Extension by Closed World Assumption (EBCWA)

When applying EBCWA, we look for a call tuple in $Calls$ which has no possibility for EBC. If its first body literal is negative, we may simply remove it. This is because we operate under CWA, where $\neg F$ is true if F cannot be proven to be true.

Preconditions

- $Call = \langle H, (B_1, B_2, \dots, B_n), C, \emptyset \rangle \in Calls$

Postconditions

- $Call$ is removed from $Calls$ and inserted into $ObCalls$.
- If B_1 is negative, and $Call$ has not been used for EBC, $\langle (H \leftarrow B_2, B_3, \dots, B_n), C \rangle$ is put into the table.

3.4.4 Soundness of the computation model

For showing soundness, we employ an induction proof showing that the table is correct in every reachable state. By correct, we mean that a few specific properties are true. The properties we want to uphold are

Consistency Every constraint associated with a call tuple or an answer tuple in the table is as it would have been if all tuples from $ObAnsA$ and $ObAnsB$ had been known from the beginning and the corresponding operations had been applied at once. That is to say that all constraints in the table has been affected by every appropriate constraint in $ObAnsA \cup ObAnsB$ and that the application order of the operations is irrelevant.

No stray calls There is no $Call = \langle H, B, C, P \rangle \in (Calls \cup ObCalls)$ such that $\langle H\theta, A \rangle \in (ObAnsA \cup ObAnsB)$ and $\langle Call, Call', \lambda \rangle \notin Links$. This means that we have the same result independent of whether the call tuple was added before an answer for its first body literal was applied, or vice versa.

The base case, where

$$\begin{aligned} &\langle AnsA, ObAnsA, AnsB, ObAnsB, Calls, ObCalls, Links \rangle \\ &= \langle \emptyset, \emptyset, \emptyset, \emptyset, \{Query\}, \emptyset, \emptyset \rangle \end{aligned}$$

obviously satisfies the properties. Thus it suffices to show that every extension operation transforms the table into a new table where the properties still hold. So we examine the extension operations.

The update mechanism

This is technically not an extension operation, but acts as a major part of EBAA and EBAB.

The soundness of the update mechanism as far as consistency is concerned should be apparent from its definition in Section 3.4.2.

The property of no stray calls is apparently preserved, since no new calls are added.

EBC

When performing EBC, we may either add a call tuple or an answer tuple. In both of these cases, the consistency property holds. This is because the answers stored in $ObAnsA$ and $ObAnsB$ are not modified in any way.

As for no stray calls—if no answer for the first body literal of the call has been used, we clearly are not adding a stray call. If an answer for the first body literal was used before, the property still holds. The latter follows from the definition of adding a call to the table (see Section 3.4.1).

EBAA

If it is the first answer alternative found for the literal in question, the consistency property is naturally satisfied. A new call might be added from this first answer extension though, but the stray call property is covered in the same way as for EBC.

The trickier part comes when there have been earlier answer extensions for this literal. Here we must consider two cases, depending on whether the literal in question is positive or negative. The commonality in these cases is that we operate on all call tuples $Call = \langle H, (B_1, \dots, B_n), C, P \rangle$ where B_1 is a variant of the head of our answer to be applied. First renaming the variables to those used already in C is a requirement for merging the answers at all. We are also sure that $\langle Call, Call', \theta \rangle \in Links$. It is $Call'$ that we focus on in this proof. C_{new} is the new alternative we are adding.

In the positive case, $C' = C \otimes (C_a \otimes C_b)$ must be changed into $C \otimes ((C_a \oplus C_{new}) \otimes C_b) \equiv C' \oplus (C \otimes C_{new} \otimes C_b)$. This of course amounts to making a disjunctive update by $(C \otimes C_{new} \otimes C_b)$.

In the negative case, the constraint C' associated with $Call'$ can be expressed as $C' = C \otimes \neg(C_a \otimes C_b)$, where C_a and C_b are the answer alternatives and answer bounds used for H so far (not including the alternative to be incorporated). C_b is defined to be *true* if no earlier answer bound for B_1 has been applied. The new constraint for $Call'$ must be logically equivalent to $C \otimes \neg((C_a \oplus C_{new}) \otimes C_b) \equiv C' \otimes \neg(C_b \otimes C_{new})$. This clearly represents the same outcome as making a conjunctive update by $\neg(C_b \otimes C_{new})$.

EBAB

As explained in the EBAB description of Section 3.4.3, we are sure that every time we perform an EBAB we have used a corresponding answer alternative before. Just as for EBAA above, we have the negative and the positive case. The variables used are the same as in the EBAA case, except of course that C_{new} now represents the new answer bound.

In the negative case, $C' = C \otimes \neg(C_a \otimes C_b)$ must be changed into $C \otimes \neg(C_a \otimes (C_b \otimes C_{new})) \equiv C' \oplus (C \otimes \neg C_{new})$, a disjunctive update by $(C \otimes \neg C_{new})$.

In the positive case, $C' = C \otimes (C_a \otimes C_b)$ must be changed into $C \otimes (C_a \otimes (C_b \otimes C_{new})) \equiv C' \otimes C_{new}$, a conjunctive update by C_{new} .

ECWA

An ECWA extension amounts to adding the derived call or answer to the table so the same reasoning as for EBC applies.

3.4.5 Completeness of the computation model

For completeness, we need to show that the computation process terminates and that all possible answers have been found.

Termination

For proving termination, we create a rank function on the possible states of the table and prove that each operation decreases the rank towards a finite minimal value.

The valuation function F on the table state is defined as the sum of X_1, \dots, X_6 .

X_1 Total number of non-satisfying valuations for the constraints in *ObAnsA*.

X_2 Total number of satisfying valuations for the constraints in *ObAnsB*.

X_3 The product of X_{31} and X_{32} .

X_{31} The number of possible valuations for all state variables in any literal + 1. This is the same as the product of the number of possible values for each state variable. We only deal with finite domains for the state variables, so we know this is a finite product.

X_{32} The number of additional answer tuples that can be added to *ObAnsA* or *ObAnsB* without merging with existing ones. The set of all possible entries that can be added is uniquely determined (modulo renaming of variables) by the input and is clearly finite.

X_4 The sum of all set sizes of P in all call tuples in *Calls*.

X_5 The product of X_{51} and X_{52} .

X_{51} The number of rules in *Prog* + 1.

X_{52} The number of additional call tuples we can add to the table that would not merge with already existing ones. The set of all possible entries that can be added is uniquely determined (modulo renaming of variables) by the input and is clearly finite.

X_6 The number of call tuples in *Calls* that permit EBCWA.

The minimal value of F is 0, since all six terms have 0 as their minimal value. So if we can decrease the value of F in every step, we know that we cannot go on forever.

EBC Suppose that the number of rules in *Prog* is n . That means that the size of P in any call tuple is at most n .

By making an EBC, we first decrease X_4 by one. At the same time, we might add a new call tuple, which would increase X_4 by m ($0 \leq m \leq n$) and/or increase X_6 by 1. But by adding a call, we also decrease X_5 by $n + 1$. The net change of $X_4 + X_5 + X_6$ is then a decrease of at least 1.

EBAA There are two cases here.

We merge with an existing entry in *ObAnsA* Since we only use⁴ answer alternatives that are not already covered by the disjunction of previous alternatives used for the same head we are sure to decrease X_1 . A call might be added from this EBAA operation (if it is the first answer), but just as in the case of EBC, $X_4 + X_5 + X_6$ will then have a net decrease of at least 1.

We create a new entry in *ObAnsA* If we add a new entry to *ObAnsA*, X_1 will probably increase. X_{32} will decrease by 1, however, which in turn will decrease X_3 by more than X_1 increased. The net decrease of $X_1 + X_3$ is at least 1.

EBAB There are two cases here too, and they are similar to those in the EBAA case.

We merge with an existing entry in *ObAnsB* Since we only use⁴ answer bounds that are more discriminating than the conjunction of bounds used for the same head before we are sure to decrease X_2 .

We create a new entry in *ObAnsB* If we add a new entry to *ObAnsB*, X_2 will probably increase. X_{32} will decrease by 1, however, which in turn will decrease X_3 by more than X_2 increased. The net decrease of $X_2 + X_3$ is at least 1.

EBCWA This will decrease X_6 by 1. As with EBC and EBAA, this EBCWA may have added a call, but the same reasoning applies yet again. We get a net decrease of at least 1.

Update mechanism It is clear that a finite number of invocations of the update mechanism performs a finite number of steps. More to the point, it will perform at most n steps per invocation, where n is the length in literals of the longest body of a rule in *Prog*.

Furthermore, it is evident from the EBAA and EBAB parts above that the update mechanism is only invoked a finite number of times, since they are the only invokers of the update mechanism.

The cases where the update mechanism puts entries into *ObAnsA* and *ObAnsB* are to be considered the same as the EBAA and EBAB cases above. Both will give a net decrease of at least 1.

Completeness discussion

As shown above, the computation always terminates in the end. When it does terminate, all possible extension operations have been applied, meaning that all

⁴Aborting before commencing the actual EBAA/EBAB is equivalent to doing nothing, since the answer might as well not have been there.

possible call tuples have been added through EBC and that all answers have been applied through EBAA.

It is not as clear, however, that all answer bounds are properly incorporated into the table, since they are added to *ObAnsB* directly if no associated EBAA has been performed. This will only be a problem if no EBAA is performed for that head at all. As shown below, we are sure that an EBAA will always be performed for these heads.

The update mechanism is the only way of adding entries into *ObAnsB* in this way. We have two cases where an (upper) answer bound is added for a head H .

- We have $\langle H\theta, (B_1), C, P \rangle \in \text{Calls} \cup \text{ObCalls}$, where B_1 is the only body literal of the call tuple and is negative. If we find an *additional* alternative answer for B_1 , the update mechanism will impose a new bound for H . In this case, however, we have already added an answer alternative for H to *AnsA*, which have been or will be applied.
- We have $\langle H\theta, (B_1), C, P \rangle \in \text{Calls} \cup \text{ObCalls}$, where B_1 is the only body literal of the call tuple and is positive. If we find a bound for B_1 , the update mechanism will want to add a new bound for H . By recursive reasoning, this would require that an EBAA for B_1 has been applied, in which case we would have already added an answer alternative for H to *AnsA*, which have been or will be applied.

3.4.6 Computation process

The initial state of the computation is when $\text{AnsA} = \text{AnsB} = \text{ObAnsA} = \text{ObAnsB} = \text{Calls} = \text{ObCalls} = \emptyset$ and $\langle Q, C \rangle$ is added to the table. Q is the query supplied as the last clause of the input and C is the constraint initially associated with that rule.

After that, the computation proceeds in accordance with the flow chart in Figure 3.2. Actions marked with “(S)” are influenced by the chosen strategy (see Chapter 4).

3.5 The output

The amount of output from the program can be regulated to include selected parts of the following. See Appendix B for an example of how this is specified in the configuration file. Most users will only use *status*, *report*, *res* and *err*.

Each output type is explained and illustrated below.

status This option enables the printing of status messages showing the configuration options and informing of the different stages of the execution.

```
Configuration options are
-----
GLOBAL_STRATEGY = weighted
```

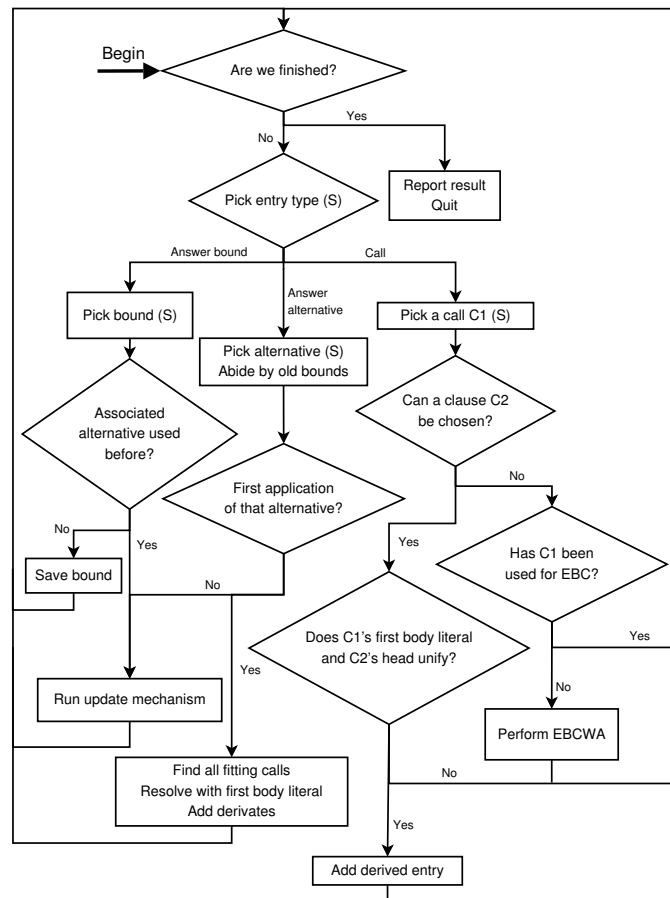


Figure 3.2. Flow chart of fixpoint computation.

```

PERCENT_LOCAL_EXTENSIONS = 50
STRATEGY_ANS_A = random
STRATEGY_ANS_B = random
STRATEGY_CALLS = random
STRATEGY_POSSIBLES = random
Logging: report status res err

```

```

Initialising constraints...
Parsing... Parsed.

```

report This option enables the printing of the final answer from the fixpoint calculation.


```

query([A1, A2]) is satisfied under the following conditions:
{ A1=0, A2=1 }
{ A1=1, A2=0 }

```

res This option enables the printing of resource measurements, including number of constraint macro operations, different types of extension operations, etc. as well as the total time used for each resource.

```

Resource statistics
-----
Constraint macro operations (67): 0.002 s
Constraint projection (27): 0.000999 s
EBAA (8): 0.004 s

```

table This option enables the printing of the internal table after each computation step. The number of steps is usually very large and a single description of the table will often be very big, so this logging option will increase the output substantially.

```

Step 2
----
--Answer alternatives (1)
holds(crit, [A1,A2]). { }
{ A1(8) A2(9) }
ROOT: 82
[ 20] 9 : 0 1
[ 82] 8 : 0 20

--Answer bounds (0)

--Calls (1)
query([A1,A2]) <- holds(or(ex(crit), crit), [A1,A2]). { 4 }
{ }
ROOT: 1

--Obsolete answer alternatives (0)

--Obsolete answer bounds (0)

--Obsolete calls (1)
holds(or(ex(crit), crit), [A1,A2]) <- holds(crit, [A1,A2]). { }
{ }
ROOT: 1

```

ext This option enables the printing of messages for the actions taken during extensions.

```
Extending by first answer
Trying to add to set: holds(or(ex(crit), crit), [A1,A2]).
Adding answer
```

err This option enables the printing of error messages. The only harmless case where these show up is if the input program is syntactically incorrent. Other occasions include corrupt table entries which indicate program failure.

```
An error occured during parsing.
```

```
Bad alternative answer present.
```

dbg This option enables the printing of debug messages. These are strictly for debugging purposes and may not make much sense other than to developers.

```
Applying (1) { A1=S1#5, A2=S2#5 } to holds(ex(crit), [A1,A2]).
Projecting constraint in: holds(ex(crit), [A1,A2]). { }
```

Chapter 4

Strategies

As the attentive reader may have noticed in the preceding chapter, not every aspect of the computation is unambiguously defined. What is not defined is left as a choice of *strategy* and is the non-determinism of the model. Deciding on the strategies does not alleviate all non-determinism though, since a fully plausible strategy would be to randomly pick an alternative among the alternatives presented.

There are a few different types of strategies that can be defined, and they will be covered in detail in this chapter. All of these strategies are stateless and can thus be changed before any computation step.

The choice of strategies is specified in the configuration file. For an example of how a configuration file might look like, see Appendix B.

4.1 Extension picking strategy

In many cases, several of the extension operations (EBC, EBAA, EBAB, EBCWA) are possible. In this case, it would be preferable to have a strategy that decides which extension type to use in the next step.

4.1.1 Preference ordering on entry types

This strategy permits six (3!) different preference orderings on the table entry types (calls, answer alternatives and answer bounds). In a given preference ordering, the entry type that is most preferred and has at least one corresponding entry in the table will be chosen.

The letters *A*, *B* and *C* are used to denote *Alternative answer tuples*, *Bounding answer tuples* and *Call tuples*. The preference ordering *ABC* then means “Choose alternative answer tuples before bounding answer tuples and bounding answer tuples before call tuples.”

4.1.2 Weighted random selection

Using this strategy, a random entry type will be chosen. The probability of choosing a certain entry type is the number of such entries in the table divided by the sum of all three types.

4.2 Entry picking strategy

When an entry type is chosen, we have implicitly also chosen which extension operation to try. This is almost true, since call tuples can be used for both EBC and EBCWA. This ambiguity is resolved after the entry picking strategy has had effect.

Through the entry picking strategy, we decide which entry of a given type to use. Once we have picked an entry we are bound to a specific extension operation. If we have decided on alternative answer tuples or bounding answer tuples, this should be obvious, since they may only be used for one extension operation each. When presented with a specific call tuple, the choice is still clear, since EBCWA is only possible if the set P of possible call extensions is empty.

The possible choices for this strategy type is *first*, *last* and *random*, where *first* and *last* use an arbitrary (but predetermined) ordering which is approximately alphabetical and *random* just picks an element at random.

4.3 EBC clause picking strategy

When performing EBC, a clause in the program is chosen and unified with the first body literal of the chosen call tuple. Although we can only guess which clauses in the program unifies with the literal, we are sure that we are choosing from a superset of the possible choices. The EBC clause picking strategy decides in which order to try clauses from this superset. If the chosen clause's head does not unify with the chosen call tuple's first body literal, the possibility is removed from the superset.

Just as in the entry picking strategy, our possible strategy choices are *first*, *last* and *random* and the meaning is the same, with the exception that the ordering is now the order in which the clauses appear in the program.

4.4 EBC locality strategy

In the definition of EBC in Section 3.4.3, one can notice that " $sol(C'') \subseteq sol(C')$ and $(sol(C') \cap sol(C)) \subseteq sol(C'')$ " is one of the postconditions. The reason for this being so loosely defined is that one may choose C'' differently here and affect the efficiency (keeping soundness and completeness of course). We call this the *EBC locality strategy*, and the two obvious choices would be to use $C'' \equiv C'$ (global EBC) and $C'' \equiv (C \cap C')$ (local EBC).

The strategy permits randomly choosing between these two extreme values by weighted probability. Setting the probability to 0 or 1 then means using a pure EBC locality strategy.

4.4.1 Global EBC

By using the global strategy, we mean that the entries added during EBC do not care about the constraint context in which they were derived. This is evident from the fact that the constraint for the added entry is only the constraint defined for that program rule.

The global strategy will in effect result in calculating the constraints for the constituents first and then conjoining them when calculating the constraint for the parent call.

4.4.2 Local EBC

Local EBC differs from global in that we know the calling context of the entry added. We see that because the solutions of the constraint of the added entry is guaranteed to be a subset of the solutions to the caller's constraint. In a way, this is like working top-down.

Chapter 5

Experimental evaluation

This chapter shows the result of running a test program using different strategies. The test program represents a system of processes with critical sections. For each tested strategy configuration, the number of calculation steps and various resources used are shown. Each test is run three times and the values represented are average values. The hardware used during testing was a Pentium 3 of 450 MHz with 256 MB of memory.

5.1 Input program

The program listed here is for three processes, but the tests below are made of programs with three, five and ten processes. The two larger programs are written in an analogous way.

This particular input program is chosen for several reasons.

scalability It can be scaled up to as many processes as wanted, while still keeping the overall structure.

negation It will provide several layers of nested negation during computation. Negation also provides the possibility for EBAB operations.

branching It contains many branching points, where the computation can take different paths, because of the involvement of the CTL predicate *or*.

comprehensibility The program is comprehensible to humans. We also know which answers we are supposed to get and can thus easily verify the result.

The input program contains the CTL representation listed in Table 2.2 and the following code (at the end). The meaning of the code is described in Section 5.2.

```

holds(init, [S1,S2,S3]).

holds(crit1, [S1,S2,S3]) :- sat(S1).
holds(crit2, [S1,S2,S3]) :- sat(S2).
holds(crit3, [S1,S2,S3]) :- sat(S3).

holds(doublecrit, [S1, S2, S3]) :-
    holds(or(or(and(crit1, crit2), and(crit1, crit3)),
             and(crit2, crit3)), [S1, S2, S3]).

step([S1, S2, S3], [T1, T2, T3]) :-
    action([S1, T1, T2, T3]),
    action([S2, T2, T1, T3]),
    action([S3, T3, T1, T2]).

action([S, T, TT1, TT2]) :-
    sat(or(and(not(S), not(T)),
           or(and(not(S), and(not(TT1), and(not(TT2), T))),
           S))).

spec([S1, S2, S3]) :- holds(ag(not(doublecrit)), [S1, S2, S3]).

query([S1, S2, S3]) :- holds(init, [S1, S2, S3]),
                        spec([S1, S2, S3]).

```

5.2 Input interpretation

We use one state variable to represent the state of each process. They are very simple processes that are only allowed to be in either the non-critical or the critical state (values 0 and 1 respectively). The whole system of three processes then contain three state variables.

We see that the program defines five properties.

init holds in every state. By restricting this, we can restrict which states we are interested in. The name *init* suggests that it is our initial configuration, but the name is of course arbitrary.

critN ($N \in \{1, 2, 3\}$) holds iff process N is in its critical state (i.e. the N:th state variable is 1).

doublecrit holds iff at least two of the properties $\{crit1, crit2, crit3\}$ hold. This will be called an *invalid* state, as opposed to a *valid* state, where at most one process is in its critical section.

Also noteworthy is that when scaling this example up to five and ten processes, the *doublecrit* predicate is defined as a balanced binary tree with depth $\lceil \lg \binom{5}{2} \rceil$ and $\lceil \lg \binom{10}{2} \rceil$ respectively.

The restriction put on the transitions within the system is that each process must perform one of three actions.

stay Stay in the state it is currently in.

enter Enter its critical session in the next step. This is only allowed if no other process will be in its critical section in the next step.

leave Leave the critical state.

The *step* relation represented by the *step* predicate in the program will look like in Figure 5.1. State *XYZ* means that process 1 is in state *X*, process 2 in state *Y* and process 3 in state *Z*.

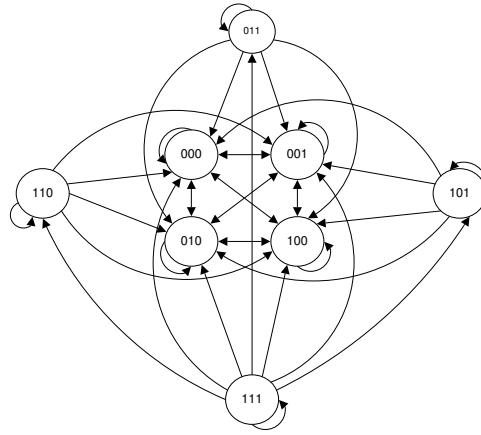


Figure 5.1. The *step* relation.

We then ask which states among the *init* states (all possible states in this example) that supports the specification “No path ever reaches an invalid state.” (for paths originating from that state). We expect that all valid states and no other states will be returned as the set of answers. This is indeed what will be returned.

The interesting part then becomes how much resources will be used to determine this. This is shown using two different extension picking strategies in Table 5.1 (using CBA) and in Table 5.2 (using ABC). Each of the tables show resource usage when running the programs with three, five and ten processes, each using 0%, 50% and 100% local EBC extensions. The results will be discussed in Section 5.3.

The rows of the tables have the following meaning.

Steps Number of extension operations done.

Ops Number of operations on constraints. (\otimes, \oplus, \dots)

EBAA Number of EBAA operations.

EBAB Number of EBAB operations.

Tot time Total time in seconds used during computation.

The number of EBC and EBCWA operations performed is not shown, since it does not vary with choice of strategy. Figure 5.2 shows the average time used per constraint operation for different percentages of EBC locality with the CBA extension picking strategy and 10 processes.

EBC locality Processes	0%			50%			100%		
	3	5	10	3	5	10	3	5	10
Steps	64	135	458	69	126	483	57	134	466
Ops	131	314	1062	148	312	1187	120	324	1171
EBAA	30	62	198	34	56	219	26	61	204
EBAB	2	5	13	4	5	16	0	5	13
Tot time	0.04	0.12	0.92	0.04	0.12	0.88	0.04	0.12	0.82

Table 5.1. Resource usage with the CBA extension picking strategy.

EBC locality Processes	0%			50%			100%		
	3	5	10	3	5	10	3	5	10
Steps	82	240	1050	76	232	1089	86	235	1063
Ops	180	665	3758	173	677	3887	204	677	3569
EBAA	43	132	616	40	127	633	45	128	617
EBAB	8	40	187	5	37	208	10	38	198
Tot time	0.05	0.23	2.26	0.05	0.23	2.28	0.06	0.23	2.06

Table 5.2. Resource usage with the ABC extension picking strategy.

5.3 Discussion about the results

The interpretation provided in this section applies only to the tested input programs. Other programs may behave in a different way.

As seen in Table 5.1 and Table 5.2, the only remarkable difference is that *CBA* outperforms *ABC* in every way. The main reason for this is that *ABC* uses answer alternatives as soon as they are available, which greatly reduces the number of occasions where two answer alternatives are merged before application. That makes the update mechanism run much more frequently. It also introduces far more bounding answer tuples into the table.

The EBC locality percentage evidently has an impact on the time used for constraint operations (see Figure 5.2). This is because the added constraint may become *false* during a local EBC operation. A constraint operation involving *true*

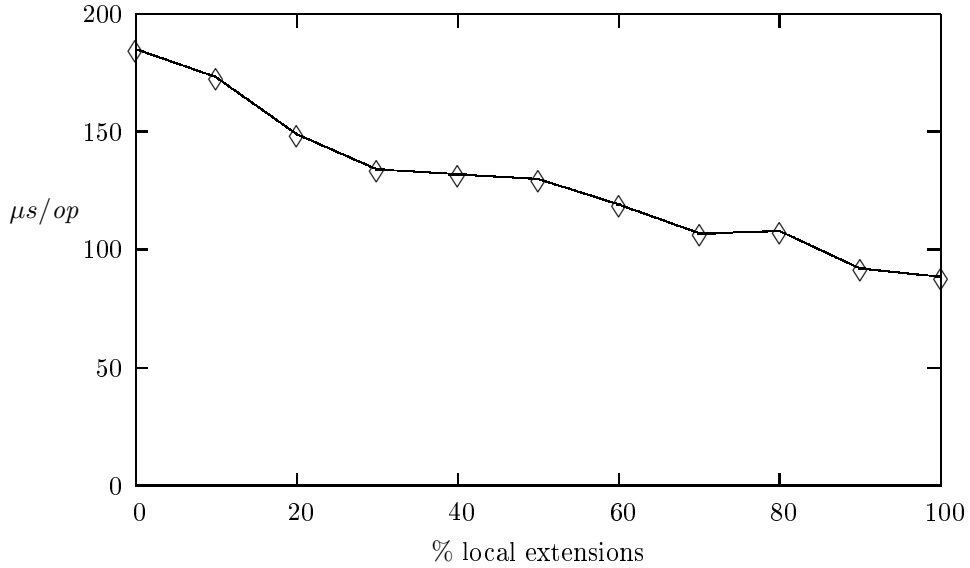


Figure 5.2. Running time per constraint operation.

or *false* is easier to handle. In this case, the average time for a constraint operation using a purely local EBC locality strategy was about half of that using only global EBC extensions. The reason why EBC locality did not matter significantly in the test cases is that the constraint operations in these examples only contributed with 5-15 percent of the total time. This increases when more complicated constraints (e.g. with more variables) are used.

It is clear, however, that the choice of computation strategies influence the efficiency of the computation.

Chapter 6

Conclusions

This chapter concludes by providing a thesis summary and suggesting some possible improvements.

6.1 Summary

In this thesis, we have shown how ideas from Constraint Logic Programming (CLP) can be used in model checking of CTL. Tabled resolution is used as a technique for avoiding infinite computation loops and constructive negation is used to maintain soundness under negation.

The implementation behind this thesis is intended to work as a test bed for future study on how to best use CLP(FD) for model checking. This necessitated the possibility of specifying computation strategies to influence different aspects of the computation.

6.2 Results

As shown in Section 5.3, the choice of computation strategies do influence the use of resources. This is to be expected, of course.

In the experimental evaluation, the same configuration of strategies (CBA extension picking strategy, only local EBC extensions) worked best in every regard. This need not be the case in every situation, though.

The ABC extension picking strategy might be preferred e.g. if we have a program without negation and want to extract partial solutions mid-way through the computation. When we have no negation in the program, every answer reported on the way will be correct, since no bounding answer tuples will be produced to restrict those answers. The impact of EBC locality naturally has much to do with the implementation of the constraints.

6.3 Future improvements

One of the main goals with the implementation have been to provide for extensibility—e.g. by new strategies and different finite domains. This therefore naturally provides many possibilities of future improvements.

Another improvement would be to rewrite the unification process and the process merging two table entries to also permit non-linear terms. The current implementation uses a very simple unification process since it is run very frequently.

The current implementation only supports reading the entire input program at the same time together with the query. It would be preferable to have the possibility of adding new queries interactively. As long as the head of the new query does not unify with any previously existing formula, this will not be a problem.

When problems get larger and more complicated, so does the internal representation and the computation process. To remedy this problem, at least partly, the implementation should support parallel computation on several machines. The concept of *divide and conquer* used, where subgoals of a call tuple are solved independently from most of the rest of the table, is normally well suited for parallelism.

Bibliography

- [1] H.R. Andersen. An Introduction to Binary Decision Diagrams. *Lecture notes*, 1997.
- [2] L. Degerstedt. *Tabulation-based logic programming*. PhD thesis, Linköping Institute of Technology, 1996.
- [3] C. Donnelly and R. Stallman. *Bison*. Free Software Foundation Inc., 59 Temple Place - Suite 330, Boston, MA 02111, USA.
- [4] O. Grumberg E. Clarke and D. Peled. *Model Checking*. MIT Press, 2000.
- [5] E. Allen Emerson. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072. MIT Press, 1990.
- [6] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
- [7] V. Jahundovics and U. Nilsson. On the Efficiency of Local and Global Symbolic Model Checking of CTL. (Draft), 2004.
- [8] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [9] U. Nilsson and J. Lübcke. Constraint Logic Programming for Local and Symbolic Model-Checking. *Lecture Notes in Computer Science*, 1861:384+, 2000.
- [10] U. Nilsson and J. Maluszyński. *Logic, Programming and Prolog*. John Wiley & Sons Ltd., 2nd edition, 2000.
- [11] V. Paxson. *Flex*. Free Software Foundation Inc., 59 Temple Place - Suite 330, Boston, MA 02111, USA.
- [12] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
- [13] P. J. Stuckey. Constructive Negation for Constraint Logic Programming. In *Proceedings of the 14th Australian Computer Science Conference*, pages 328–341, 1991.
- [14] D. Warren. Memoing for logic programs. *CACM*, 35(3):93–111, 1992.

Appendix A

BNF for input program

This is the BNF of the grammar used for input programs (start symbol is *program*). Every symbol may be separated by any number of *<whitespace>* characters.

```
program := program-clauses query

program-clauses := program-clause program-clauses
                  | program-clause

query := program-clause

program-clause := head ' .'
                  | head ' :- ' body ' .'
head := functor

body := sat
       | sat literals
       | literals

sat := 'sat' '(' sat-expr ') '

sat-expr := 'and' '(' sat-expr ',' sat-expr ') '
           | 'or' '(' sat-expr ',' sat-expr ') '
           | 'xor' '(' sat-expr ',' sat-expr ') '
           | 'not' '(' sat-expr ') '
           | <var>

literals := literal
           | literal literals

literal := functor
          | '\+' functor
```

```

functor := <id>
         | <id> '(' args ')'

args := arg
      | arg ',' args

arg := functor
     | list
     | variable

list := '[' ']'
      | '[' args ']'
      | '[' args ']' arg ']'

variable := <var>

<id> = [a-z][a-zA-Z_0-9]+

<var> = [A-Z][a-zA-Z_0-9]*

<whitespace> = [space]
              | [tab]
              | [newline]

```

To support constraints over other finite domains than the Boolean, an extra interpretation of *sat* needs to be added, as well as an interpretation of a value of that domain. There is currently no support for it in the implementation, but it is easily added. Here is an example of a domain of integer numbers 0...99. *eq* corresponds to the \diamond operator defined in Chapter 2.

```

sat := 'eq' '(' <var> ',' <val> ')'

<val> = [0-9][0-9]

```

Appendix B

Configuration file example

A configuration file is specified when running the program. Below are the general configuration settings and their descriptions.

Logging This option lists which types of textual output we want from the program. The different output types are described in Section 3.5.

```
LOG report status err res
```

EBC locality strategy This option defines the probability of performing local EBC operations when EBC operations are performed. See Section 4.4 for a description of the EBC locality strategy.

```
PERCENT_LOCAL_EXTENSIONS 100
```

Extension picking strategy This option defines our extension picking strategy. A value of *WEIGHTED* denotes a weighted probability and a permutation of *ABC* denotes a preference ordering where EBAA is performed before EBAB and EBAB is performed before EBC and EBCWA. See Section 4.1 for a description of the extension picking strategy.

```
GLOBAL_STRATEGY cba
```

Entry picking strategies These options define how an entry is picked when the operation has been determined. Listed in order are the strategy choices for picking alternative answer tuples, bounding answer tuples, call tuples and program clauses (for EBC). Possible values are *first*, *last* and *random*. See Section 4.2 for a description of the entry picking strategy.

```
STRATEGY_ANS_A random
STRATEGY_ANS_B random
STRATEGY_CALLS random
STRATEGY_POSSIBLES random
```

Random seed Setting the random seed to a specific value ensures that the pseudo-random number generator will generate the same sequence of random numbers from time to time. This is useful for debugging. Setting this option to *random* or leaving it out completely results in a time-specific random seed and unknown pseudo-random values.

```
RANDOM_SEED 1108844205
```