Computing Minimal Conflicts
for Rich Constraint Languages

Jakob Mauss[1] and Mugur Tatar[1]

Abstract. We address here the following question: Given an inconsistent theory, find a minimal subset of it responsible for the inconsistency. Such conflicts are essential for problem solvers that make use of conflict-driven search (cf. [2, 4, 9]), for interactive applications where explanations are required (cf. [16, 22]), or as supporting tools for consistency maintenance in knowledge-bases (cf. [11]). Conflict computation in AI applications was usually associated with dependency recording as performed by TMSs (cf. [2, 3, 18]). This techniques, however, have a rather limited applicability for languages that go beyond the expressiveness power of propositional logic. For more powerful languages and solvers constraint suspension appeared, until now, to be the only available alternative for the computation of minimal conflicts.

We present here an algorithm for computing minimal conflicts that can be used with powerful constraint languages, e.g. possibly including finite and non-finite variable domains, algebraic and FD constraints, etc. The conflicts are extracted post mortem from the proof (a tree with inferences of the form $A \wedge B \Rightarrow C$) that lead to the derivation of the inconsistency by an informed search that computes and generalizes conflicting relations. The algorithm is based on a simple but powerful principle that allows to recursively decompose the minimization problem into smaller sub-problems. This principle can also lay the foundation for efficient constraint suspension algorithms that can be used in case no intermediary results are cached during the constraint solving, i.e. in case no proof structures are available.

1 INTRODUCTION

For problems expressed using propositional logic or using finite-domain (FD) constraints there exist some efficient solutions for the computation of conflicts and explanations (cf. [13, 16, 18]). Unfortunately, this is not the case for more expressive constraint languages. Due to the scope of our application interests, namely supporting engineering tasks such as safety and diagnosability analysis and also design and configuration (cf. [12, 15, 20, 22]), we are especially interested in modeling languages adequate for engineering problems. Such languages have to mix logical and FD constraints with (more or less) classical systems of linear and non-linear algebraic or even differential equations. The general purpose techniques that can be applied in this case for the (minimal) conflict computation are constraint suspension (cf. [7]) and TMS-like dependency recording (cf. [3]). Constraint suspension can guarantee conflict minimality, but it is in many cases too inconvenient due to the large amount of time required to recompute many subsets of the initial problem. When applied to

[1] DaimlerChrysler AG, Research and Technology, RIC/EK
Alt-Moabit 96a, D-10559 Berlin, Germany.
Email: {jakob.mauss, mugur.tatar}@DaimlerChrysler.com

systems of equations where local value propagation is not enough for solving, TMS-based architectures usually become a heavy machinery that consumes considerable amounts of time and memory (see also [17]) and, in the end, still do not have any guarantees for conflict minimality – the minimality is (at most) guaranteed with respect to the propositional clauses that represent the dependencies and not with respect to the semantic of the original constraint language. The following example is an attempt to illustrate this. Consider a system of five algebraic constraints

$$A_1 \equiv x > 4 \qquad A_3 \equiv y \geq 2 \qquad A_5 \equiv x > 2\,y + 1$$
$$A_2 \equiv x < 5 \qquad A_4 \equiv y \leq 2$$

A solver may process these constraints in 4 steps as shown in Figure 1. In step ④, they are discovered inconsistent. A minimal conflict among the given constraints is { A_2, A_3, A_5 }. If the solver were using dependency recording it would not find the above minimal conflict – just the trivial {A_1, A_2, A_3, A_4, A_5} in this case!
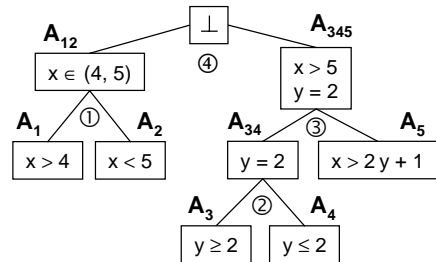

Figure 1. Tree for proving the inconsistency of 5 constraints.

Of course, this was a just simple example where no symbolic variable elimination was required and, for the above example, one can easily define a strategy to handle correctly the conflict computation – for instance by maintaining separate dependencies for lower and upper bounds of intervals as in [6]. However, this unnecessarily overloads the solving process in case of consistency and, still, would not solve the problem in general.

In contrast, the key idea of this paper is to do a (guided) post mortem analysis of the context in order to compute the minimal conflicts. The algorithm uses the information that A_{345} is conflicting with A_{12} (we say that A_{345} is a conflicting relation for A_{12}) and propagates and updates these conflicting relations through the proof tree in order to select only those parts of it that are really contributing to the conflict. The paper is organized as follows: in section 2 we present the basic procedure for extracting a minimal conflict from a binary proof tree. In section 3 we describe how a constraint solver can control the inferences in order to easily provide such trees. In section 4 we report some first empirical results regarding the performance of the algorithms. Section 5 concludes the paper with a comparison to related work.

2 COMPUTING MINIMAL CONFLICTS

We assume in the following a relational framework, i.e. constraints are noted as *relations* over variables with finite or continuous domains. These relations may be represented extensionally (as in Figure 4), or intensionally using formulas (as in Figure 1). In relational terms, '$\wedge$' represents the join (intersection) of relations, falsity '$\bot$' is represented by the empty relation, and the implication $A \Rightarrow B$ is interpreted as subset relation $A \subseteq B$. A set of constraints forms a *conflict* if it is not satisfiable, i.e. in relational terms, if the join of the relations representing the constraints is the empty relation. Given an initial set of inconsistent constraints, we are interested in extracting a *minimal conflict*, i.e. a minimal subset that is still inconsistent. Of course, there can be more than one minimal conflict in an inconsistent context, but we focus for the moment on finding just one such minimal conflict. In the following, we show how to extract a minimal conflict from a binary proof tree such as the one shown in Figure 2. The initial constraints appearing as leaves in the proof tree are also called *assumptions* in the following.
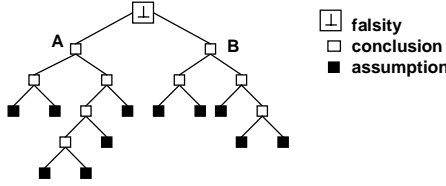
Figure 2. Tree proving the inconsistency of 11 assumptions.

Assume that we have two conflicting relations A, B, none of them being empty, i.e. $A \neq \bot$, $B \neq \bot$, and $A \wedge B \Rightarrow \bot$. Then we have to consider two cases.

1. A and B are both assumptions. In this case, {A, B} is the minimal conflict.

2. At least one of A, B is not an assumption. Assume without loss of generality that A has been derived from A_1 and A_2, i.e. $A_1 \wedge A_2 \Rightarrow A$. Let now: $C_1 := A_1 \wedge B$ and $C_2 := A_2 \wedge B$. We can then distinguish 4 cases, as shown in Figure 3.
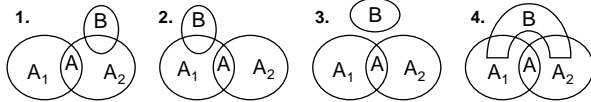
Figure 3. Four cases distinguished by computing intersections. Each relation is depicted as a set of variable assignments.

2.1: $C_1 = \bot \wedge C_2 \neq \bot$ In this case, the assumptions leading to the derivation of A_2 do not contribute to the conflict with B. Consequently, we can prune the whole sub-tree A_2 and continue the conflict search in A_1.

2.2: $C_1 \neq \bot \wedge C_2 = \bot$ Analog to case 2.1. A_1 can be ignored.

2.3: $C_1 = \bot \wedge C_2 = \bot$ There are at least two independent conflicts with B, at least one in the sub-tree A_1, and at least one in A_2. If we want to find just one conflict then we can non-deterministically decide to skip one of the sub-trees.

2.4: $C_1 \neq \bot \wedge C_2 \neq \bot$ All minimal conflicts are spread across both sub-trees. A minimal conflict has to be composed from a partial solution retrieved from the sub-tree A_1 and an appropriate completion retrieved from the sub-tree A_2. If B was a conflicting relation for A, then C_1 is a conflicting relation for A_2 and C_2 is a conflicting relation for A_1. With these new conflicting relations we can descend recursively in the A_1, A_2 sub-trees and collect the sub-conflicts.

This case analysis leads to the following procedure for extracting a minimal conflict from a proof tree.
Specification: Let

R	be a non-empty set of relations (assumptions)
A	the root of a binary proof tree with the leaves given by R
B	a conflicting relation for A, i.e.: $B \neq \bot$ and $A \wedge B = \bot$.

The proof tree satisfies the requirement that, for any non-leaf node A: $\quad$ left(A) $\wedge$ right(A) $\Rightarrow$ A.

The procedure XC1(A, B) returns one minimal and non-empty set $M \subseteq R$ such that $(\wedge M) \wedge B = \bot$. As a consequence, if A is the root of a refutation tree then XC1(A, T) returns a minimal conflict from the tree - where T represents the universal relation i.e. the complement of $\bot$.

```
XC1(A, B)
⓪ if (isLeaf(A)) return { A }
   A₁ ← left(A)
   A₂ ← right(A)
   C₁ ← A₁ ∧ B
   C₂ ← A₂ ∧ B
① if (C₁ = ⊥ and C₂ ≠ ⊥)   return XC1(A₁,B)
② if (C₁ ≠ ⊥ and C₂ = ⊥)   return XC1(A₂,B)
③ if (C₁ = ⊥ and C₂ = ⊥)   return XC1(A₁,B)
                     or    return XC1(A₂,B)
④ if (C₁ ≠ ⊥ and C₂ ≠ ⊥)
            M₁ ← XC1(A₁, C₂)
            M₂ ← XC1(A₂, (∧M₁) ∧ B)
            return M₁ ∪ M₂
```

In case ④ the procedure first descends in the sub-tree A_1 with C_2 as conflicting relation. Before it descends in the sub-tree A_2 we, however, have to generalize A_1 to $\wedge M_1$ and C_1 to $(\wedge M_1) \wedge B$. This is necessary in case we have *several* minimal conflicts that span over the sub-trees A_1 and A_2 in order to select from A_2 a sub-conflict that is part of the same conflict as the sub-conflict that was non-deterministically chosen (case ③) from the sub-tree A_1. Such a case is also illustrated by the following example.

Example Consider the set R = {A_1, …A_5} shown in Figure 4. The constraints are extensionally defined relations in this example. E.g. $A_1 \equiv$ '$(x = a \wedge y = 1) \vee (x = b \wedge y = 0)$'. R is inconsistent, actually it contains two minimal conflicts. Figure 5 shows how XC1 computes one of them. Circled numbers correspond to the five cases marked in the pseudo code above.
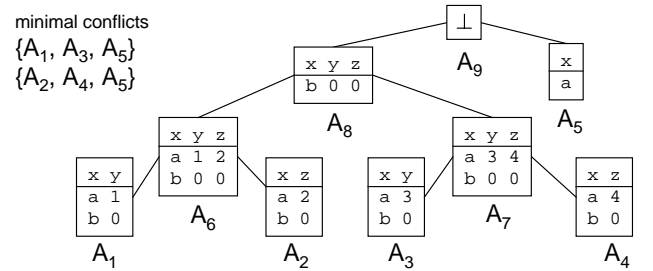
Figure 4. A proof tree for R = {A_1, A_2, A_3, A_4, A_5}

The crucial part of the procedure is handled in case ④, where a minimal conflict is composed as a disjoined union of two sets M_1 and M_2 computed using the left and right sub-tree. Note that the second set M_2 depends on the first set M_1. During the recursive call at A_6 the procedure non-deterministically decides to select the conflict containing A_1. This decision is reflected in the arguments of the succeeding call at A_7 in order to select the right sub-conflict - i.e A_3 and not A_4 which could be erroneously selected if we did not update the conflicting relation for A_7!

Some properties of XC1 that are worth discussing are:

(1) During top-down traversal of the proof tree, only direct fathers of the nodes contained in the returned minimal conflict are visited. Sub-trees not involved in the minimal conflict are pruned without investigating their nodes. The worst-case appears when the pruning is not effective and we have to inspect the whole tree (always in case ④). For a tree with n leaves there are no more than $4(n-1)$ joins for the worst case (see also the incremental computation of $\wedge M_1$ later on). However, the complexity of the conflict minimization crucially depends on the complexity of the basic join operations.
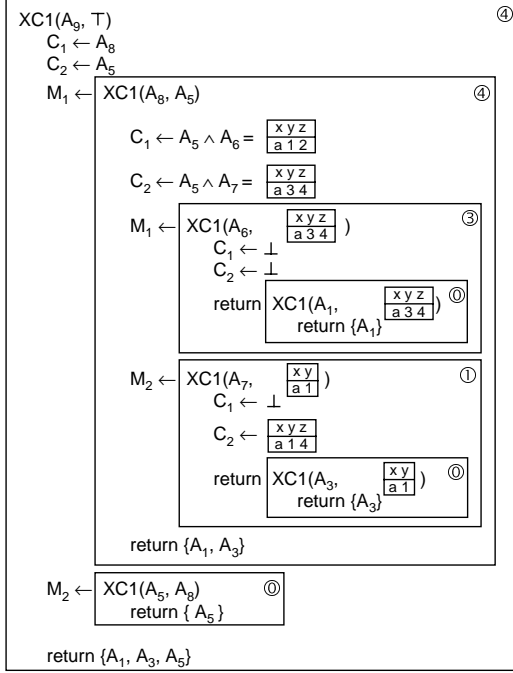

Figure 5. Trace of computation of a minimal conflict

(2) An inference engine will be unable in general to provide complete implementations of the join and empty-check operations - for instance in case we are dealing with systems of non-linear equations. When used in conjunction with a correct but *incomplete* inference engine, XC1 may return a non-minimal conflict. The conflict 'minimallity' is only relative to the completeness degree of the inference services supplied by the solver.
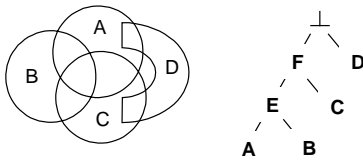

Figure 6. Two minimal conflicts {B, D} and {A, C, D}

(3) The procedure can easily be extended to return several minimal conflicts instead of only one. Basically, in case ③, one can continue search in both sub-trees, instead of non-deterministically choosing one of them. However, this simple extension of XC1 will not always return all of the minimal conflicts. See Figure 6 for an example. The second conflict {A, C, D} is missed, when using the given proof tree. Anyway, the computation of all minimal conflicts from a context can require significantly more effort and is seldom justified in practice.

(4) There are several obvious improvements of the efficiency of XC1 as given above. If the proof structure is a tree then $M_1 \cup M_2$ can be computed as a disjoined union in case ④. If case ③ is always mapped to (say) case ① then the computation of C_2 is required only if $C_1 \neq \perp$. The repeated $\wedge$ computations $\wedge M_1$ can be avoided if XC1, in addition to returning the set M, also returns the join $\wedge M$, which allows for an incremental computation of the conjunction in case ④. Moreover, the generalization of the conflicting relation C_1, i.e. $(\wedge M_1) \wedge B$ in case ④, is required only if it is a strict generalization, i.e. if M_1 is a strict subset of the leaves of A_1.

3 DERIVING PROOF TREES

In the previous section we have seen how to extract one or several minimal conflicts from a proof structure. In this section we sketch how a constraint solver operating on a set R of input relations can control the inference in order to

1. check whether R is consistent, i.e. whether $\wedge R \neq \perp$
2. solve R for any variable
3. provide the proof structure required for conflict computation.

Conflict computation using XC1 works however with any well-formed proof structure, irrespective if the proof was generated by a solver like the one described in this section or not[2].

We note with V(A) the set of *variables* constrained by a relation A. $\pi(A, X)$ denotes the *projection* of A onto a variable set X. The projection $\pi(A, X)$ results from eliminating all variables $V(A) \setminus X$ from the relation A. For example, if

$$A \equiv x^2 + y^2 < 1 \qquad B \equiv (x-1)^2 + y^2 < 1$$

then $\pi(A \wedge B, \{x\}) = '0 < x \wedge x < 1'$

and $\pi(A \wedge B, \{y\}) = '-1 < y \wedge y < 1'$.

The projection operation is an abstraction (generalization) operation, i.e. $A \Rightarrow \pi(A, X)$. Hence, the computation $C := \pi(A \wedge B, X)$ can be seen as an inference of the form $A \wedge B \Rightarrow C$. We call such an inference, i.e. projecting the join of two relations A and B onto a variable set X, an *aggregation*.

The computed proofs will contain aggregations as the only kind of inference. The proof structures will be used to derive minimal conflicts, or minimal explanations of variable solutions.

The consistency check, may seem trivial to specify. We could simply ask the solver to compute $\wedge R$ to check whether $\wedge R \neq \perp$. However, in the practical applications with which we are commonly confronted, R may contain hundreds of algebraic and logical constraints with thousands of variables. In this case, the intermediate relations created during the computation of $\wedge R$ would be huge. Instead, following [1], after computing a single conjunction $A \wedge B$, we eliminate all those variables from the result that do not occur in the remaining relations. Consequently, the intermediate relations remain 'small' – the size depending, of course, on the degree of connectivity of the constraint network. This works fine, as long as a variable is shared by a relatively small number of constraints. If the connectivity degree increases (cf. induced width w* in [5]), then many of the aggregations degrade to simple joins and the approach is likely to become inappropriate.

---

[2] Such proof structures can be recovered, for instance, also from the well-founded-support recorded by a TMS (cf. [18]) - in which case XC1 could be used for further conflict minimization (recall that a TMS guarantees minimality only with respect to *propositional* dependencies and not with respect to the more expressive constraint language).

The creation of a proof tree for the consistency check is given by the following procedure.

Specification: Let R be a non-empty set of assumptions, $\perp \notin R$. Then the procedure isConsistent(R) returns true, iff R is satisfiable.

```
isConsistent(R)
  if ( |R|= 1 )
     return true
  else
     choose {A, B} ⊆ R
     S ← R \ {A, B}
     X ← (vars(A) ∪ vars(B)) ∩ (∪ vars(S))
     C ← π(A ∧ B, X)
     if (C = ⊥) return false
     return isConsistent(S ∪ { C })

vars(A)
  if (A is a leaf)
     return V(A)
  else
     return vars(left(A)) ∪ vars(right(A))
```

Obviously, the procedure isConsistent computes a proof tree containing aggregations as the only kind of inference. Therefore, we call this an *aggregation tree*. If A is the root of an aggregation tree for an inconsistent set R of assumptions then, as shown in section 2, XC1(A, T) returns a minimal conflict. To keep the conflicting relation B small, we may add a projection step $B \leftarrow \pi(B, vars(A))$ as first instruction in XC1. The strategy used to choose a pair of relations for aggregation may for example minimize the variable set X, or try to achieve a balanced tree.

For checking the consistency of n assumptions, isConsistent computes $n - 1$ aggregations. A significant feature of proof trees as derived above is their ability to support *incremental context analysis*. Assume we have performed a consistency check for a set R of n assumptions, and we want to analyze a second context R', constructed by replacing an assumption A in the proof tree for R by a new assumption B with the same variable set. In order to check the new context $R \cup \{B\} \setminus \{A\}$, we only have to re-compute the inferences on the path from A to the root of the proof tree, i.e. if the proof tree is balanced, we only have to compute $log(n)$ aggregations. As we see next, the computation of variable solutions can be performed using aggregations as well.

Specification: Let R be a non-empty consistent set of assumptions, and A the root of an aggregation tree computed by the procedure isConsistent. Then the procedure solve(A, T) computes for every variable x in R the solution $S[x] := \pi(\wedge R, \{x\})$.

```
solve(A, B)
   if (A is a leaf) return
   B ← π(B, vars(A))
   A₁ ← left(A);   A₂ ← right(A)
   A₁₂ ← A₁ ∧ A₂
   X ← vars(A₁) ∪ vars(A₂) \ vars(A)
   for each x ∈ X
      S[x] ← π( A₁₂ ∧ B, {x})
   solve(A₁, A₁₂ ∧ B)
   solve(A₂, A₁₂ ∧ B)
```

If we take a closer look at the procedure solve, we note that each S[x] is the root of a proof structure defined by a sequence of aggregations. In this case the proof is not purely a tree, it is actually a DAG because some nodes are used more than once. Still, the proof is well-formed, i.e. there are no cyclic justifications. XC1 can be modified to cope with the DAG structure. The resulting procedure XE1(S[x], ¬S[x]) returns a *minimal supporting set of assumptions* for the solution of x, i.e. a minimal subset $E \subseteq R$ such that $S[x] = \pi(\wedge E, \{x\})$.

4 APPLICATION AND EMPIRICAL RESULTS

We have recently finished a prototype implementation of a Relational Constraint Solver (RCS) that follows the principles described in this paper, including the computation of explanations and conflicts. RCS is already integrated in our environment for engineering knowledge management, and its integration in MDS [12] is planned to follow.

In this section we compare XC1 with the conflict computation based on naive constraint suspension. Let R be an inconsistent set of relations. Then the procedure MC(R, {}) returns a minimal conflict, computed by constraint suspension.

```
MC(R, M)
   if R = {} return M
   else choose A ∈ R
      if ∧(R ∪ M \ {A}) = ⊥
         return MC(R \ {A}, M)
      else return MC(R \ {A}, M ∪ {A})
```

The procedure MC resembles Junker's ROBUSTXPLAIN [8], which may use a trailing-mechanism not described in [8] to perform incremental (i.e. fast) consistency checking. If $|R| = n$ and a consistency check for R requires n aggregations, then MC(R, {}) needs $O(n^2)$ aggregations for computing a minimal conflict. In contrast, XC1 requires only $O(n)$ aggregations for the same task, given an arbitrary, not necessarily balanced tree. Our implementation of MC uses an incremental consistency check as explained in section 3 – thus, a check requires only $O(log(n))$ instead of $O(n)$ aggregations in the best case.
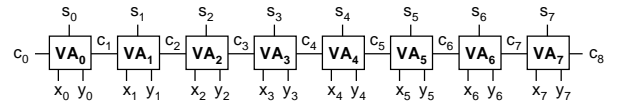

Figure 7. An 8-bit full adder

For the empirical comparison, we used a set R of 137 relations, representing eight 1-bit full adders connected in series as shown in Figure 7, and the assignments $c_0 = 1$, and for $0 \leq k \leq 7$: $x_k = 0$, $y_k = 1$. If we add one more relation of the form $c_k = 0$, then R becomes inconsistent and contains a minimal conflict M of size $|M| = 2 + 3 k$. This gives us 8 different sets R_k of size 138, containing a minimal conflict M of size $2 + 3 k$.

conflict detection			MC		XC1		t(MC)
|M|	n	$\wedge, \pi$	$\wedge$	π	$\wedge$	π	t(XC1)
2	8.5	59.1	95.7	91.6	8.8	3.5	446
5	32.6	120.8	76.2	59.4	31.4	13.9	12
8	52.4	129.1	108.3	80.4	52.5	22.9	6.9
11	70.5	131.5	140.4	103.6	72.8	31.5	6.4
14	93.0	133.6	178.5	130.1	93.9	40.4	5.4
17	107.7	134.9	208.5	151.1	113.5	48.3	4.1
20	122.6	135.8	247.5	179.8	131.3	55.7	2.5
23	134.8	136.7	278.3	201.0	151.8	63.7	1.3

Figure 8. Empirical results

For each k isConsistent(R_k) is run for consistency check and it returns a refutation tree that is used as input by both MC and XC1. The leaves of this tree represent an initial (not necessarily minimal) conflict of size n - see Figure 8. The table gives the average results obtained for running both algorithms 100 times for all eight R_k. For each run, we permutated the order of the input relations which resulted in different structures of the derived aggregation trees. The columns in the table denote the average number of join and project

operations needed for conflict detection in isConsistent and by the subsequent minimization call to MC or to XC1. The last column gives the ratio of the measured runtimes for MC and XC1. For example, for the case of a minimal conflict of size 2, the average initial conflict provided by isConsistent has size 8.5 and it takes 59.1 aggregations (join followed by project) to detect the conflict. MC needs then 95.7 more joins and 91.6 projections to minimize the initial conflict by suspension, while XC1 is 446 times faster than MC and needs only 8.8 joins and 3.5 projections for the same task. The performance gain of XC1 relative to MC depends strongly on the structure of the proof trees supplied by the solver – i.e. whether the conflicting assumptions are uniformly spread among the leaves of the tree or whether they are clustered in a few sub-trees.

5 RELATED WORK AND DISCUSSION

Dependency recording, like the one performed by TMSs (cf. [2, 3, 18]) works relatively fine as long as we stay in a propositional framework (or, anyway, in a finite world). In more expressive frameworks these techniques gradually become both

- very resource consuming (in time and space)
- incomplete with respect to the more expressive framework.

Constraint suspension is another technique used for conflict computation. It is in general expensive because it relies on performing the consistency check many times for different subsets of the initial problem. A recent enhancement to constraint suspension is the one reported in [8]. The performance of the conflict computation is improved there in two ways:

(a) by adding the constraints to the solver's store one after the other and performing each time a complete consistency check, one knows that the last constraint added that caused the store to become inconsistent is part of all conflicts from the already considered subset; and

(b) by employing an intelligent search, where sets of constraints are simultaneously suspended and then are binary split if necessary.

The proof structure corresponding to the control strategy assumed by Junker is a linear tree. We do not need to enforce the sequential consistency check as assumed by (a). We can assume any clustering technique, such as the ones resulting after structure analysis, e.g. cycle-cutset, hypertree decomposition, etc. (cf. [10]), and thus take advantage of the performance improvements for constraint solving enabled by these methods. Our solution suits better the solvers employing such decompositions or the solvers that are recording (at least partially) their proof structures in order to support incremental operation. Although developed independently and using quite differing notations, the *principle* underlying the decomposition of the conflict minimization problems *is the same* for our XC1 and for Junker's QUICKXPLAIN algorithm. After several years of trying to improve constraint solving and dependency recording (cf. [15]), the existence of such simple and general algorithms for minimal conflict computation came for us as a surprising positive result.

One of our main application areas is model-based diagnosis. We do not argue here that one should perform diagnosis by always first computing conflicts and then generating minimal / preferred / etc. diagnoses. Several authors point out that the direct computation of diagnoses can be more efficient (cf. [16, 19, 21, 23]). The ideas from an algorithm such as TREE* (cf. [21]) can be probably easily adapted to a general relational framework such as

the one of RCS. One weak point, however, of the available computation techniques that are not based on conflicts is that they basically address static problems. It would be interesting to see if the ideas of the *temporal decomposition* that can be applied for computing minimal conflicts (cf. [14]) can be also applied for the direct computation of diagnoses or interpretations.

Although we discussed here about the computation of *minimal* conflicts, in practice minimality and completeness have to be traded against efficiency. Nevertheless, sometimes the definition of the application (minimisation, compilation, explanation, etc) require a higher degree of completeness that is more important than the computation times.

REFERENCES

[1] Y. El Fattah: An Elimination Algorithm for Model-based Diagnosis. *Dx98*, Cape Cod, USA, pp. 47-54, 1998.

[2] K. Forbus, J. de Kleer: Building Problem Solvers. MIT Press, 1993.

[3] J. de Kleer: An Assumption-based truth maintenance system. *Artificial Intelligence*, 28, pp. 127-162, 1986.

[4] J. de Kleer, B. Williams: Diagnosing Multiple Faults. *Artificial Intelligence*, 32, pp. 97-130, 1987.

[5] R. Dechter: Bucket Elimination: a Unifying Framework for Reasoning. *Artificial Intelligence*, 113, pp. 41 - 85, 1999.

[6] D. J. Goldstone: Controlling inequality reasoning in a TMS-based analog diagnosis system. *9th Nat. Conf. On AI*, pp. 512-517, 1991.

[7] R. Bakker, F. Dikker, F. Tempelman, P. Wognum: Diagnosing and solving over-determined CSP. *Proc. IJCAI-93*, 1993

[8] U. Junker: QUICKXPLAIN: Conflict Detection for Arbitrary Constraint Propagation Algorithms. *IJCAI'01 Workshop on Modelling and Solving Problems with constraints*, pp. 75-82, 2001.

[9] N. Muscettola, P. Nayak, B. Pell, B. Williams: Remote Agent: To boldly go where no AI system has gone before. *Artificial Int.*, 103, pp. 5-47, 1998.

[10] G. Gottlob, N. Leone, F. Scarcello: A comparison of structural CSP decomposition methods. *Artificial Int.*, 124(2), pp. 243-282, 2000.

[11] A. Fleming, G. Friedrich, D. Jannach, M. Stumptner: Consistency-based Diagnosis of Configuration Knowledge Bases. *ECAI-2000*, Berlin, 2000.

[12] J. Mauss, V. May, M. Tatar: Towards Model-based Engineering: Failure Analysis with MDS. *ECAI-2000 Workshop W31*, 2000. http://www.dbai.tuwien.ac.at/event/ecai2000-kbsmbe/papers.html

[13] F. Bouquet, P. Jegou: Solving over-constrained CSP using weighted OBDDs. Proc. Over-Constrained Systems, Lecture Notes in Computer Science, Vol. 1106, Springer, Berlin, 1996.

[14] M. Tatar : Diagnosis with cascading defects. *ECAI-1996*, 1996.

[15] M. Tatar : Model-based failure analysis in engineering – an experience report. *Invited talk at Dx 2001*. Available at request.

[16] J. Amilhastre, H. Fargier, P. Marquis: Consistency restoration and explanations in dynamic CSPs. *Artificial Intelligence* 135, 2002.

[17] G. Katsillis, M. Chantler: Can Dependency-based Diagnosis Cope with Simultaneous Equations? *Dx97*, France, 1997.

[18] D. McAllester: Truth Maintenance. *AAAI-90*, pp. 1109-1116, 1990.

[19] W. Nejdl, B. Giefer: DRUM: Reasoning without conflicts and justifications. *Dx94* , pp. 226-233, New Paltz, NY, 1994.

[20] M. Tatar, P. Dannenmann: Integrating Simulation and model-based Diagnosis into the Life Cycle of Aerospace Systems. *Dx99*, Loch Awe, Scotland, 1999.

[21] M. Stumptner, F. Wotawa: Diagnosing tree-structured systems. *Artificial Intelligence*, 127, pp. 1-29, 2001.

[22] F. Feldkamp, M. Heinrich, K.-D. Meyer-Gramann: SyDeR - System Design for Reusability. *AI-EDAM Special Issue on Configuration Design*. Sept. 1998.

[23] A. Darwiche: Decomposable Negation Normal Form. *Journal of ACM*, July 2001.