QUICKXPLAIN: Conflict Detection for Arbitrary Constraint Propagation Algorithms

Ulrich Junker
ILOG
1681, route des Dolines
F-06560 Valbonne
junker@ilog.fr

Abstract

Existing conflict detection methods for CSP's such as [de Kleer, 1989; Ginsberg, 1993] cannot make use of powerful propagation which makes them unusable for complex real-world problems. On the other hand, powerful constraint propagation methods lack the ability to extract dependencies or conflicts, which makes them unusable for many advanced AI reasoning methods that require conflicts, as well as for interactive applications that require explanations. In this paper, we present a non-intrusive conflict detection algorithm called QUICKXPLAIN that tackles those problems. It can be applied to any propagation or inference algorithm as powerful as it may be. Our algorithm improves the efficiency of direct non-intrusive conflict detectors by recursively partitioning the problem into subproblems of half the size and by immediately skipping those subproblems that do not contain an element of the conflict. QUICKXPLAIN is used as explanation component of an advanced industrial constraint-based configuration tool.

Keywords: constraint satisfaction, constraint programming, configuration.

1 Introduction

Many advanced reasoning methods in AI require the detection of conflicts (or nogoods). Conflict-based approaches are common in nonmonotonic reasoning, belief revision, model-based diagnosis, configuration, and planning. Some good examples are hitting-set trees [Reiter, 1987], conditionals [Geffner and Pearl, 1992], TMS-based default provers [Junker and Konolige, 1990]. In recent years, conflict-based reasoning is also gaining interest in the field of constraint satisfaction. Examples are conflict-based backjumping [Prosser, 1993], dynamic backtracking [Ginsberg, 1993], distributed CSP's [Yokoo, 1995], over-constrained CSP's [Régin *et al.*, 2000], Tabu search with propagation [Jussien and Lhomme, 2000], and verification of constraint-based configuration models [Felfernig *et al.*, 2000]. Furthermore, conflict detection is essential for generating explanations in interactive applications. In particular, web-based configuration requires explanation facilities and makes the detection of conflicts for arbitrary constraint propagation algorithms to a topic of raising interest.

Conflicts for CSP's can be elegantly computed by methods such as ATMS [de Kleer, 1989] and dynamic backtracking [Ginsberg, 1993] if we keep track of the employed constraints in the computed conflicts. Unfortunately, those methods do not directly use powerful constraint propagation algorithms that profit from compact representation of variable domains and that exploit the semantics of the given constraint to do the propagations (e.g. for a sum constraint, bounds are calculated directly). Although ATMS is able to achieve k-consistency [de Kleer, 1989], it is not clear how it could exploit the pruning power of n-ary constraints such as the AllDiff [Regin, 1994] which leads to significant speed-ups for problems such as n-queen, sports-league scheduling, quasi-groups.

On the other hand, constraint propagation algorithms lack the ability to compute conflicts in case of failure or to produce explanations for their deductions. The basic problem is that recording justifications during propagation decreases the time and space complexity of many propagation algorithm. For example, Narenda Jussien notes in [Jussien *et al.*, 2000] that AC6 looses its advantages compared to AC4 when recording explanations. This draw-back makes powerful propagation unusable for many advanced AI reasoning methods as well as for interactive applications where explanations guide a Human in his/her decision making.

Since constraint propagation algorithms are usually incomplete and do not discover failures in all cases, we have to distinguish propagation-specific conflicts from global conflicts. A global conflict is a set of constraints that have no solution. Although not each global conflict is a propagation-specific one, there are several reasons why propagation-specific conflicts are interesting:

- They are sufficient if the propagation ensures backtrack-free behaviour.
- They provide explanations for the behaviour of a given propagation engine.
- They can be computed in polynomial time[1] whereas computing global conflicts for CSP's is NP-complete.

[1] supposed the given propagation algorithm needs polynomial time

Thus, they represent a compromise between tractability and completeness.

- They allow to exploit powerful constraint propagation inside conflict-based AI methods and thus bridge the gap between conflict-based reasoning and powerful propagation.
- Each global conflict can be derived from propagation-specific conflicts by using ATMS or intelligent backtracking.

In this paper, we consider conflicts and explanations for a given constraint propagation algorithm or engine. This covers specific propagation algorithms such as the AC-family, as well as the propagation engines employed in constraint programming tools [Fernandez and Hill, 2000], which may employ AC5 as main algorithm and dedicated algorithms for each kind of constraint.

However, even if those constraint propagation algorithms do neither compute conflicts, nor record the relevant information for conflict detection, it is possible to detect conflicts a-posteriori and this for any constraint propagation algorithm as complex as it may be. Given a set of inconsistent constraints, the basic idea is to iteratively test the consistency of subsets of the given constraints until a minimal conflict is found. In this paper, we first present two iterative conflict detections algorithms, which follow ideas given in [de Siqueira N. and Puget, 1988; Bakker *et al.*, 1993] and which need $O(n \cdot k)$ consistency tests to find a minimal conflict of size k out of n constraints. We then show that the efficiency of these iterative methods can be improved by recursively partitioning the conflict detection problem into subproblems of half the size and by immediately skipping those subproblems that do not contain an element of the conflict. The new algorithm needs $O(n{\cdot}log(k{+}1){+}k^2)$ checks. Hence, it wins over the iterative methods if k is small compared to n. Due to this behaviour, we call the new method QUICKXPLAIN.

Although the principle employed in QUICKXPLAIN is quite intuitive, this algorithm is coming as a surprise. We believed for a long time that conflict detection for constraint propagation either has to be intrusive and causing overhead or it has to be very slow. QUICKXPLAIN shows that non-intrusive conflict detection done in a clever way can be sufficiently efficient for real-world problems. In particular, QUICKXPLAIN is used as explanation component of an advanced constraint-based configuration tool [ILOG, 2000] and thus addresses an important issues that has, for example, been raised in [Felfernig *et al.*, 2000]. However, QUICKXPLAIN is not limited to constraint propagation. It can also be used to find conflicts for automated theorem provers and any other inference engines satisfying some common properties.

The paper is organized as follows: In section 2, we illustrate the basic idea of our conflict detection method. Sections 3 and 4 introduce the technical prerequisites for our algorithm. Explanations require notions such as logical consequences of CSP's, which will be defined in section 3. In section 4, we list our assumptions about the constraint propagation algorithms. In section 5, we define conflicts and related notions. Our non-intrusive conflict detection methods (including QUICKXPLAIN) are described in section 6. Finally, we summarize the advantages of QUICKXPLAIN in the conclusion and discuss ongoing and future work.

Added constraints		Deduction	Argument/Conflict
ρ_1	$x_1 = 1$	$y \geq 500$	$\{\rho_1\}$
ρ_2	$x_2 = 1$	$y \geq 1000$	$\{\rho_1, \rho_2\}$
ρ_3	$x_3 = 1$	$y \geq 1500$	$\{\rho_1, \rho_2, \rho_3\}$
ρ_4	$x_4 = 1$	$y \geq 2300$	$\{\rho_1, \rho_2, \rho_3, \rho_4\}$
ρ_5	$x_5 = 1$	$y \geq 4900$	$\{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$
		$\perp$	$\{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$

Table 1: Computing a conflict during propagation.

Step	Activated constraints					Result	Partial conflict
1.	ρ_1					no fail	$\{\}$
2.	ρ_1	ρ_2				no fail	$\{\}$
3.	ρ_1	ρ_2	ρ_3			no fail	$\{\}$
4.	ρ_1	ρ_2	ρ_3	ρ_4		no fail	$\{\}$
5.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	fail	$\{\rho_5\}$
6.	ρ_5					no fail	$\{\rho_5\}$
7.	ρ_5	ρ_1				fail	$\{\rho_1, \rho_5\}$

Table 2: Finding a minimal conflict.

2 Non-intrusive Conflict Detection

In this section, we present the basic ideas of our conflict detection approach and illustrate our algorithm QUICKXPLAIN using following simple example.

Example 1 *Suppose a customer wants to buy a station-wagon with following extras: 1. metal color, 2. ABS, 3. roof racks, 4. one additional seat, 5. special luxury version. We suppose that those extras cost* $c_1 := 500$, $c_2 := 500$, $c_3 := 500$, $c_4 := 800$, *and* $c_5 = 2600$ *dollars. Furthermore, the customer cannot exceed* 3000 *dollar for the options. We model this example with a constrained integer variable* $x_i \in \{0, 1\}$ *for each extra* $i := 1, \ldots 5$, *a variable* $y \in [0, 3000]$ *for the costs of the extras, and following constraint*

$$y = \sum_{i=1}^{5} c_i \cdot x_i \tag{1}$$

Furthermore, we model the customer requests by the constraints $x_i = 1$ *for each* $i = 1, \ldots, 5$. *Let* ρ_i *be a short-hand for* $x_i = 1$.

When a propagation algorithm is supplied with such a set of constraints, it deduces new constraints, which mainly concern a single variable. For example, when an algorithm removes the value v from the domain of a variable x, it deduces the constraint $x \neq v$. If it removes all values strictly smaller (or greater) than m then it deduces the constraint $x \geq m$ (or $x \leq m$). If all values are removed from a variable domain, a failure occurs. We also say that an inconsistency $\perp$ is deduced. In the example, we initially start with the scalar-product constraint and then introduce the requests one by one. In each step, we derive a new constraint of the form $y \geq m$ with increasing m. In the fourth column, we list the requests needed to deduce $y \geq m$. The result is shown in table 1.

If only some of the given constraints are sufficient to deduce this failure (by propagation), we would like to know them. Hence, we are interested in (inclusion-)minimal conflicts, i.e. subsets of the given constraints that are sufficient to deduce the inconsistency (via propagation), but no proper subset of them allows to do so.

Finding conflicts for arbitrary constraint propagation algorithms has been an open question for a long time. Classical approaches based on TMS/ATMS determine a conflict by keeping track of explanations for all intermediate deductions:

1. When a constraint is propagating, a local explanation for each deduced constraint of the form $x \neq v$ has to be determined by the propagation engine.
2. These local explanations have to be synthesized to more global ones by eliminating intermediate deductions. This can either be achieved by storing a single local explanation per deduction as in TMS [Doyle, 1979] or by propagating explanations as done by the "predecessor" of ATMS [Martins and Shapiro, 1988]. Recent work on conflict detection for CSP's also follows this second principle [Jussien *et al.*, 2000].

These approaches are intrusive in the sense that they require that the given constraint propagation algorithm determines and records local explanations during propagation. Although these intrusive approaches are based on elegant deduction principles and have successfully been applied to problems with a specific structure such as numeric CSP's [Jussien and Lhomme, 1998] and open-shop scheduling [Jussien and Lhomme, 2000], they encounter several problems in the general case.

First of all, real-world problems often do not have such a clear structure as numeric CSP's and scheduling problems. For applications such as web-based configuration, crew scheduling, time-tabling, the users can themselves edit the constraints and significantly modify the structure of the problem.

Second, the bookkeeping of explanations during propagation increases the space complexity of many propagation algorithms. If there are n variables and d possible values per variable the cost of this book-keeping is $O(n \cdot d)$ explanations of size $O(n \cdot d)$. Hence, the memory consumption may already be excessive for medium-sized real-world problems where n and d have values around 100. For large-scale problems where n and d have values of at least 1000 the method is inapplicable.

Third, the classical approaches do not guarantee minimality of the computed conflict even if the local explanations are minimal, which itself is non-trivial for global constraints such as AllDiff. In our example, we have listed the explanations computed by Jussien's algorithm for the constraints of the form $y \geq m$ (cf. fourth column in table above). For $y \geq 4900$, we obtain the (local) explanation $X := \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$, which is minimal since all ρ's are needed to deduce the lower bound of 4100 on y. From $y \geq 4100$ and $y \leq 3000$, we then deduce a failure $\perp$ having again the explanation X. However, X is not a minimal conflict. If ρ_5 is chosen we cannot choose any other extra without violating the budget. Hence, we get four possible minimal conflicts $\{\rho_1, \rho_5\}$, $\{\rho_2, \rho_5\}$, $\{\rho_3, \rho_5\}$, and $\{\rho_4, \rho_5\}$, which are significantly smaller than X.

Fourth, if there are multiple conflicts the classical approaches do not provide any control on which of the conflicts is found.

We now consider non-intrusive conflict detection approaches that determine conflicts a-posteriori by checking the consistency of different subsets of the given constraints. These non-intrusive methods do not require any book-keeping, they guarantee minimality (of the propagation-specific) conflicts, and they provide a control on which conflict is found. In fact, the determined conflict is uniquely specified by a total order on the given constraints.

We first introduce an iterative algorithm that is based on [de Siqueira N. and Puget, 1988]. As shown in table 2, we add the given constraints $\rho_1, \ldots, \rho_5$ in a given order to the given constraint propagation engine until a fail is obtained. The first fail occurs after ρ_5 is added, which consequently belongs to some minimal conflict. After that, we backtrack to the initial state and add ρ_5 again. Since this does not result into a fail we again add the remaining constraints $\rho_1, \ldots, \rho_4$. This time we immediately obtain a fail after adding ρ_1. Since ρ_1 and ρ_5 are both needed to obtain a fail we have thus found a minimal conflict $\{\rho_1, \rho_5\}$.

In this case, the detection of a minimal conflict was straightforward. We now consider a variant of our example that is slightly more complex. We consider eight requests $\rho_1, \ldots, \rho_8$ with costs $c_1 = c_3 = c_4 = c_6 = 100$ and $c_2 = c_5 = c_7 = c_8 = 800$. We adapt the scalar product correspondingly, but keep the budget of 3000. The algorithm now needs 26 steps each of which adds a single constraint to the propagation engine. Table 3 shows the detailled trace.

In this example, most elements of the conflict are explored late. Since the algorithm backtracks to the beginning for each detected element of the conflict it has to reexplore most constraints several times.

We now improve this behaviour of the iterative algorithm by recursively partitioning the conflict detection problem into subproblems of half the size. Subproblems that do not contain an element of the conflict can immediately be skipped. Hence, the constraints of those subproblems need not be reexplored. The new algorithm called QUICKXPLAIN needs 21 steps, which are summarized in table 4.

In lines 1-8, QUICKXPLAIN adds the constraints as usual. When the first element of the conflict is detected the overall problem is decomposed into two subproblems. The first subproblem consists in finding a conflict among $\{\rho_5, \ldots, \rho_7\}$ while keeping $\{\rho_1, \ldots, \rho_4\}$ and the already detected elements of the conflict in the background (i.e. as hard constraints). The second subproblem consists in finding a conflict among $\{\rho_1, \ldots, \rho_4\}$ while keeping the conflict elements found by the first subproblem in the background. In order to set up the first subproblem, QUICKXPLAIN backtracks three steps and re-adds the already detected element of the conflict (line 9). It then readds the constraints $\{\rho_5, \ldots, \rho_7\}$ of the first subproblem that is solved in lines 10-15 resulting in the detection of two elements ρ_5 and ρ_7 of the conflict. To set up the second subproblem, QUICKXPLAIN backtracks to the beginning and re-adds all detected elements of the conflict (lines 16-18).

Step	Activated constraints								Result	Partial conflict
1.	ρ_1								no fail	$\{\}$
2.	ρ_1	ρ_2							no fail	$\{\}$
...				...						...
8.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6	ρ_7	ρ_8	fail	$\{\rho_8\}$
9.	ρ_8								no fail	$\{\rho_8\}$
10.	ρ_8	ρ_1							no fail	$\{\rho_8\}$
...				...						...
16.	ρ_8	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6	ρ_7	fail	$\{\rho_7, \rho_8\}$
17.	ρ_8	ρ_7							no fail	$\{\rho_7, \rho_8\}$
18.	ρ_8	ρ_7	ρ_1						no fail	$\{\rho_7, \rho_8\}$
...				...						...
22.	ρ_8	ρ_7	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5		fail	$\{\rho_5, \rho_7, \rho_8\}$
23.	ρ_8	ρ_7	ρ_5						no fail	$\{\rho_5, \rho_7, \rho_8\}$
24.	ρ_8	ρ_7	ρ_5	ρ_1					no fail	$\{\rho_5, \rho_7, \rho_8\}$
25.	ρ_8	ρ_7	ρ_5	ρ_1	ρ_2				fail	$\{\rho_2, \rho_5, \rho_7, \rho_8\}$
26.	ρ_8	ρ_7	ρ_5	ρ_2					fail	$\{\rho_2, \rho_5, \rho_7, \rho_8\}$

Table 3: Example needing frequent replays.

Step	Activated constraints								Result	Partial conflict
1.	ρ_1								no fail	$\{\}$
2.	ρ_1	ρ_2							no fail	$\{\}$
...				...						...
8.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_5	ρ_6	ρ_7	ρ_8	fail	$\{\rho_8\}$
9.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_8				no fail	$\{\rho_8\}$
...				...						...
12.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_8	ρ_5	ρ_6	ρ_7	fail	$\{\rho_7, \rho_8\}$
13.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_8	ρ_5	ρ_7		fail	$\{\rho_7, \rho_8\}$
14.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_8	ρ_7			no fail	$\{\rho_7, \rho_8\}$
15.	ρ_1	ρ_2	ρ_3	ρ_4	ρ_8	ρ_7	ρ_5		fail	$\{\rho_5, \rho_7, \rho_8\}$
16.	ρ_8								no fail	$\{\rho_5, \rho_7, \rho_8\}$
17.	ρ_8	ρ_7							no fail	$\{\rho_5, \rho_7, \rho_8\}$
18.	ρ_8	ρ_7	ρ_5						no fail	$\{\rho_5, \rho_7, \rho_8\}$
19.	ρ_8	ρ_7	ρ_5	ρ_1					no fail	$\{\rho_5, \rho_7, \rho_8\}$
20.	ρ_8	ρ_7	ρ_5	ρ_1	ρ_2				fail	$\{\rho_2, \rho_5, \rho_7, \rho_8\}$
21.	ρ_8	ρ_7	ρ_5	ρ_2					fail	$\{\rho_2, \rho_5, \rho_7, \rho_8\}$

Table 4: Recursive conflict detection reduces replaying.

The second subproblem is solved in lines 19-21 leading to the detection of the remaining element ρ_2. The subproblems themselves are solved in the same way as the main problem and are recursively decomposed when a new element of the conflict is detected.

3 Logical Use of CSP's

CSP's consist of a set $\mathcal{V}$ of variables and a set $\mathcal{C}$ of constraints. Each constraint has the form $c(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are variables in $\mathcal{V}$ or values and c is an n-ary constraint predicate. Each n-ary constraint predicate has an associated n-ary relation $R(c) \subseteq \mathcal{D}^n$ where $\mathcal{D}$ is a global domain. In particular, we introduce constraints of the form $x = v$, $x \neq v$, and $x \in D$ where v is a value in $\mathcal{D}$, D is a subset of $\mathcal{D}$, and the predicates $=, \neq, \in$ are interpret by equality, inequality, and set-membership. A mapping $v : \mathcal{V} \rightarrow \mathcal{D}$ of variables to values *satisfies* a constraint $c(x_1, \ldots, x_n)$ iff $(v(x_1), \ldots, v(x_n)) \in R(c)$. Such a mapping v is a *solution* of the CSP iff it satisfies all constraints in $\mathcal{C}$.

Given a set of constraints C, let $V(C)$ be the set of variables appearing in the constraints of C. We say that a constraint α is a (logical) *consequence* of a set $\mathcal{C}$ of constraints iff all solutions of the CSP with variables $V(\mathcal{C} \cup \{\alpha\})$ and constraints $\mathcal{C}$ also satisfy the constraint α. We write $\mathcal{C} \models \alpha$ in this case.

We say that a set $\mathcal{C}$ of constraints is *consistent* iff the CSP with variables $V(\mathcal{C})$ and constraints $\mathcal{C}$ has a solution. We also write $\mathcal{C} \models \perp$ in this case where $\perp$ is a unary constraint with arbitrary variable and empty relation. Given this, a mapping $v : \mathcal{V} \rightarrow \mathcal{D}$ is a solution iff $\mathcal{C} \cup \{(x = v(x)) \mid v \in V(\mathcal{C})\}$ is consistent. We have thus mapped the problem of finding a solution of a CSP to the problem of checking the consistency of a set of constraints.

4 Deductive View on Propagation

For the purpose of explanation, we need a deductive view on a constraint propagation engine. In this section, we show that this can be achieved supposed the propagation engine satisfies some usual properties.

A standard constraint propagation engine allows to incrementally add a constraint to a given state. Propagation then leads to the deduction of new constraints. For example, if a value v is removed from the domain of a variable x, the constraint $x \neq v$ is (implicitly) derived. Furthermore, if all possible values for a variable are removed from its domain, we say that an inconsistency $\perp$ is derived. We can now characterize a state of a propagation engine by a set C of constraints and the incremental adding by a function *add* that maps a set of constraints and a new constraint to a new set of constraints. We suppose that *add* satisfies following properties:

- P1 **Finiteness:** if C is finite then $add(C, \alpha)$ is finite.
- P2 **Order-independence:** $add(add(C, \alpha), \beta) = add(add(C, \beta), \alpha)$.
- P3 **Inclusion of input:** $\alpha \in add(C, \alpha)$ and $C \subseteq add(C, \alpha)$.
- P4 **Closedness:** if $\alpha \in C$ where $C = add(C', \beta)$ then $add(C, \alpha) \subseteq C$.
- P5 **Correctness:** if $\alpha \in add(C, \beta)$ then $C \cup \{\beta\} \models \alpha$.
- P6 **Completeness w.r.t. checking:** if $\{x_1 = v_1, \ldots, x_n = v_n\} \subseteq C$ and $(v_1, \ldots, v_n) \notin R(c)$ then $\perp \in add(C, c(x_1, \ldots, x_n))$.

Repeated application of the function *add* to the different constraints in a set C allows to define a propagation operator that maps a set of constraints to a new set of constraints. Let $\alpha_1, \ldots, \alpha_n$ be some enumeration of the elements of C. We define the *propagation operator* Π as follows. Let $\Pi(\emptyset) := \emptyset$ and

$$\Pi(\{\alpha_1, \ldots, \alpha_{i+1}\}) := add(\Pi(\{\alpha_1, \ldots, \alpha_i\}), \alpha_{i+1}) \quad (2)$$

Since the result of the function *add* is independent of the order in which constraints are added the result of Π is the same for all possible enumerations of C. We can now show that Π satisfies all properties of a deduction operator.

Theorem 1 *If the function add satisfies the properties P1 to P6 then* Π *satisfies following properties.*

$$\begin{array}{c} C \subseteq \Pi(C) \\ C_1 \subseteq C_2 \textit{ implies } \Pi(C_1) \subseteq \Pi(C_2) \\ \Pi(\Pi(C)) \subseteq \Pi(C) \end{array} \quad (3)$$

According to Tarski, $\Pi(C)$ is the unique minimal fix-point of Π that contains C (supposed C is finite).

Thus, we can characterize the behaviour of any standard constraint propagation algorithm by a Tarski-like deduction operator. We characterize some important constraint propagation algorithms. If Π describes a *constraint checker* then it is entirely characterized by the properties P1-P6. If Π additionally ensures *arc consistency* then it also satisfies following property.

- P7 If $c(x, y) \in \Pi(C)$, $w \in \mathcal{D}$ and $(x \neq v) \in \Pi(C)$ for all $(v, w) \in R(c)$ then $(y \neq w) \in \Pi(C)$.

This property can easily be generalized to non-binary constraints. We can also characterize propagation engines that maintain different notions of consistency for different constraint predicates. We just limit the application of properties such as P7 to the relevant predicates. Finally, we can also characterize complete propagation engines that check global consistency:

- P8 If C is inconsistent then $\perp \in \Pi(C)$.

5 Π-Conflicts and Π-Arguments

We are now able to introduce the central concepts of this paper, namely conflicts and arguments for a given propagation operator Π. Conflicts and arguments are subsets of the given constraints $\mathcal{C}$ that are sufficient to deduce an inconsistency or a given constraint ϕ. Often, we are only interested in a part of a conflict, which contains only selected constraints of $\mathcal{C}$. We therefore split up $\mathcal{C}$ into two disjoint sets, namely a background $C \subseteq \mathcal{C}$ and a set of explainers $\mathcal{X} \subseteq \mathcal{C}$. We call the tuple $(C, \mathcal{X})$ a CSP in *explainable form*. Throughout this paper, we suppose that C and $\mathcal{X}$ are *finite* sets of constraints. We now define:

Definition 1 *Let* Π *be a propagation operator,* $(C, \mathcal{X})$ *be a CSP in explainable form, and* ϕ *be a constraint.*

1. *A subset* X *of* $\mathcal{X}$ *is called a* Π-argument *for* ϕ *and* $(C, \mathcal{X})$ *iff* $\phi \in \Pi(C \cup X)$. X *is a minimal* Π*-argument for* ϕ *and* $(C, \mathcal{X})$ *iff no proper subset of* X *is a* Π*-argument for* ϕ *and* $(C, \mathcal{X})$.
2. *A subset* X *of* $\mathcal{X}$ *is called a (minimal)* Π-conflict *for* $(C, \mathcal{X})$ *iff* X *is a (minimal)* Π*-argument for* $\perp$ *and* $(C, \mathcal{X})$.
3. *A subset* X *of* $\mathcal{X}$ *is called a (minimal)* Π-counterargument *for* ϕ *and* $(C, \mathcal{X})$ *iff* X *is a (minimal)* Π*-conflict for* $(C \cup \{\phi\}, \mathcal{X})$.

If Π describes a constraint checker then the Π-conflicts just correspond to the forbidden tuples of the constraints and we also call them *local conflicts*. If Π supports k-consistency we also refer to the Π-conflicts as *k-conflicts*. If Π supports global consistency checking then the Π-conflicts are equal to the *global conflicts*.

Arguments and counterarguments are needed for explanation purposes. In the following sections, we limit our discussion to the problem of conflict detection. According to our definition, those algorithms can directly be used to determine counterarguments. In order to adapt them to argument detection, tests of the form $\perp \in \Pi(C)$ have to be replaced by tests of the form $\phi \in \Pi(C)$.

In general, an inconsistent set of constraints can have several conflicts. Usually, only a single conflict has to be determined, but we want some control on the conflicts that can be determined. We use a strict partial order $\prec$ on the constraints in $\mathcal{X}$ for this purpose. In analogy to the notion of preferred solutions [Junker, 2000], we define preferred conflicts as follows:

Definition 2 *Let* $(C, \mathcal{X})$ *be a CSP in explainable form and* $\prec \subseteq \mathcal{X} \times \mathcal{X}$ *be a strict partial order. Let* $<$ *be a total order of* $\mathcal{X}$ *that is a superset of* $\prec$. *Let* $\alpha_1, \ldots, \alpha_n$ *be an enumeration of the elements of* $\mathcal{X}$ *in increasing* $<$*-order. We then define* $X_n := \mathcal{X}$ *and*

$$X_i := \begin{cases} X_{i+1} - \{\alpha_i\} & \textit{if } \perp \in \Pi(C \cup X_{i+1} - \{\alpha_i\}) \\ X_{i+1} & \textit{otherwise} \end{cases}$$

X_0 *is a* preferred Π-conflict *for* $\prec$.

Algorithm ROBUSTXPLAIN(C, U)

if $\perp \in C$ **then** return $\emptyset$;
if $U = \emptyset$ **then** throw exception 'no conflict';
let $\alpha_1, \ldots, \alpha_n$ be an enumeration of U;
$k := 0$;
while $\perp \notin C$ and $k < n$ **do**
    $C_k := C$;
    $k := k + 1$;
    $C := \Pi(C \cup \{\alpha_k\})$;
if $\perp \notin C$ **then** throw exception 'no conflict';
$X := \{\alpha_k\}$;
while $k > 1$ **do**
    $k := k - 1$;
    $C := C_{k-1}$;
    $C := \Pi(C \cup X)$;
    if $\perp \notin C$ **then** $X := \{\alpha_k\} \cup X$;
return X.

Figure 1: Robust explanation

Algorithm REPLAYXPLAIN(C, U)

if $\perp \in C$ **then** return $\emptyset$;
if $U = \emptyset$ **then** throw exception 'no conflict';
let $\alpha_1, \ldots, \alpha_n$ be an enumeration of U;
$X := \emptyset$;
while $\perp \notin C$ **do**
    $k := 0$;
    $C_0 := C$;
    while $\perp \notin C$ and $k < n$ **do**
        $k := k + 1$;
        $C := \Pi(C \cup \{\alpha_k\})$;
    if $\perp \notin C$ **then** throw exception 'no conflict';
    $X := X \cup \{\alpha_k\}$;
    $C := C_0$;
    $C := \Pi(C \cup \{\alpha_k\})$;
return X.

Figure 2: Explanation based on replaying

If $\prec$ is a total order there is a unique preferred conflict. Hence, a total order can be used to uniquely specify or characterize the conflict that will be detected by our algorithms.

6 Computing Minimal Π-Conflicts

In this section, we first present two iterative algorithms for non-intrusive conflict detection and analyze their behaviour. Based on this, we then develop a recursive version that improves the efficiency of the iterative versions.

All these algorithms are supplied with a CSP (C, U) in explainable form and determine a conflict X for it if $\Pi(C \cup U)$ contains an inconsistency $\perp$. Otherwise, they throw an exception indicating that there is no conflict. All algorithms consist of an exploration phase in which the unexplored constraints in U are step by step added to C by using the function *add* of the underlying constraint propagation engine. Let $\alpha_1, \ldots, \alpha_n$ be the order in which the constraints are added. The first α_k that causes a failure when added to C belongs to a minimal Π-conflict. It is therefore added to X and its addition to C is undone. Undoing can be achieved by storing intermediate states of C and by reseting C to an appropriate intermediate state[2].

There are different ways to find further elements of the conflict. Algorithm ROBUSTXPLAIN (cf. figure 1) undoes the addition of α_i for $i = k - 1, \ldots, 1$. In each step, it adds the elements of the current X to C by $\mid X \mid$ *add*-operations. If an inconsistency occurs then α_i is not needed to produce a failure and can be dropped. Otherwise, it has to be added to X. This algorithm needs a single exploration phase of $O(n)$ *add*-operations, but adds n times $O(k)$ constraints of X where k is the size of the final conflict. ROBUSTXPLAIN finds a minimal conflict even if *add* is not order-independent since it never changes the order in which the constraints are added.

[2] In practice, constraint programming tools use an efficient trailing-mechanism to undo the addition of the constraints during search.

Algorithm REPLAYXPLAIN (cf. figure 2) undoes the addition of all constraints $\alpha_1, \ldots, \alpha_{k-1}$ before adding α_k to the initial state. If an inconsistency is obtained then X already is a minimal Π-conflict and the algorithm can stop. It thus avoids the large number of test-operations needed by ROBUSTXPLAIN. Otherwise, a new exploration phase is started (without undoing the addition of α_k), which will lead to the detection of a new element of X before α_k is reached. This procedure is repeated until a minimal Π-conflict is detected. This algorithm performs k exploration-phases of $O(n)$ tests, but only k tests after undo operations.

We now present the algorithm QUICKXPLAIN (cf. figure 3) that combines the advantages of ROBUSTXPLAIN and REPLAYXPLAIN. When an element α_k of X is detected it divides the remaining constraints into two disjoint subsets $U_1 := \{\alpha_1, \ldots, \alpha_i\}$ and $U_2 := \{\alpha_{i+1}, \ldots, \alpha_{k-1}\}$ where $i := split(k-1)$. Usually, we split in the middle and choose $split(n) := n/2$. We then recursively apply the algorithm to find a conflict X_2 among U_2 using $C \cup U_1$ as background and a conflict X_1 among U_1 using $C \cup X_2$ as background. The set $X := X_1 \cup X_2 \cup \{\alpha_k\}$ then is a minimal Π-conflict for (C, U). QUICKXPLAIN immediately skips a subproblem if the background of this subproblem is inconsistent. Thus, only subproblems containing elements of X will be explored. Hence, k subproblems will be explored in total, which leads to maximal $n \cdot log(k+1)$ *add*-calls for exploration. For counting the *add*-calls needed for testing the elements of X, we look at the depth of the recursive calls of QUICKXPLAIN. Each time a new element of X is detected the call depth can be increased. Hence, the i-th element of X is detected at depth i or at a smaller depth. It is tested twice (lines 16 and 21) when detected and once (line 21) at smaller depths. We thus obtain maximal $\sum_{i=1}^{k}(i+1) = (k+3) \cdot k/2$ tests. The complexity of the different algorithms is summarized in table 5.

In order to prove the correctness of the three algorithms, we are using following properties of (C, U). Properties X1-X4 are used by all algorithms, whereas X5 is only exploited by QUICKXPLAIN:

X1 If $\perp \in \Pi(C)$ then the empty set is the only minimal Π-conflict of (C, U).

```
Algorithm QUICKXPLAIN(C, U)
if ⊥ ∈ C then return ∅;
if U = ∅ then throw exception 'no conflict';
let α_1, ..., α_n be an enumeration of U;
k := 0;
while ⊥ ∉ C and k < n do
    C_k := C;
    k := k + 1;
    C := Π(C ∪ {α_k});
if ⊥ ∉ C then throw exception 'no conflict';
X := {α_k};
let i be split(k − 1);
U_1 := {α_1, ..., α_i};
U_2 := {α_{i+1}, ..., α_{k−1}};
if U_2 ≠ ∅ then
    C := C_i;
    C := Π(C ∪ X);
    let Δ_2 be the result of QUICKXPLAIN(C, U_2);
    X := Δ_2 ∪ X;
if U_1 ≠ ∅ then
    C := C_0;
    C := Π(C ∪ X);
    let Δ_1 be the result of QUICKXPLAIN(C, U_1);
    X := Δ_1 ∪ X;
return X.
```

Figure 3: Explanation by recursive partitioning

	Explore	**Test**
ROBUSTXPLAIN	n	$n \cdot k - \frac{k \cdot (k-1)}{2}$
REPLAYXPLAIN	$k \cdot n - \frac{k \cdot (k-1)}{2}$	k
QUICKXPLAIN	$n \cdot log(k+1)$	$\frac{(k+3) \cdot k}{2}$

Table 5: Complexity for finding conflicts of size k in terms of #*add*-calls.

X2 If $\perp \notin \Pi(C)$ and $U = \emptyset$ then (C, U) has no minimal Π-conflict.

X3 If $\alpha \in U$ and $\perp \in \Pi(C \cup U - \{\alpha\})$ then each minimal Π-conflict of $(C, U - \{\alpha\})$ is also a minimal Π-conflict of (C, U).

X4 If $\alpha \in U$ and $\perp \notin \Pi(C \cup U - \{\alpha\})$, but $\perp \in \Pi(C \cup U)$, and Δ is a minimal Π-conflict of $(C \cup \{\alpha\}, U - \{\alpha\})$ then $\Delta \cup \{\alpha\}$ is a minimal Π-conflict of (C, U).

X5 If U_1 and U_2 are disjoint, Δ_2 is a minimal Π-conflict of $(C \cup U_1, U_2)$, and Δ_1 is a minimal Π-conflict of $(C \cup \Delta_2, U_1)$ then $\Delta_1 \cup \Delta_2$ is a minimal Π-conflict of $(C, U_1 \cup U_2)$.

We can now show the correctness of our algorithm:

Theorem 2 *The algorithm* QUICKXPLAIN*(C, U) always terminates. If* $\Pi(C \cup U)$ *does not contain an inconsistency* $\perp$ *then it throws an exception 'no conflict'. Otherwise, it returns a minimal* Π*-conflict* X *of* (C, U).

We can also characterize which conflict is found:

Theorem 3 *Let* $< \subseteq U \times U$ *be a total order and suppose that* QUICKXPLAIN *explores elements of* U *in increasing* $<$*-order. If* $\Pi(C \cup U)$ *contains an inconsistency* $\perp$ *then* QUICKXPLAIN*(C, U) returns the unique* $<$*-preferred conflict of* (C, U).

Finally, we mention that QUICKXPLAIN behaves as REPLAYXPLAIN if we choose *split*$(n) := n$.

7 Conclusion

In this paper, we developed a 'non-intrusive' conflict detection algorithm that is able to find minimal conflicts for arbitrary constraint propagation algorithms or engines, as well as any other monotonic inference engine. QUICKXPLAIN improves the efficiency of existing non-intrusive algorithms [de Siqueira N. and Puget, 1988; Bakker *et al.*, 1993] by recursively partitioning the conflict detection problem into subproblems and by skipping subproblems that do not contain an element of the conflict. QUICKXPLAIN appears to be a good choice of a conflict detection method in following cases:

- for explanation detection.
- for black-box propagation engines that are not able to record local explanations as required by (A)TMS.
- for complex or large problems where methods such as ATMS or TMS lead to high overhead.
- for minimizing the conflicts found by a built-in method such as (A)TMS or [Jussien *et al.*, 2000].

In particular, QUICKXPLAIN is used as explanation component of an advanced industrial constraint-based configuration tool [ILOG, 2000] and proved to be feasible for typical configuration problems. Experimental comparisons of the different conflict detection methods are in preparation.

Currently, we are extending our conflict detection method for global conflicts. We investigate several approaches:

1. apply QUICKXPLAIN in each node of a (systematic and complete) backtrack search for a solution. If all sons of a node fail use the hyperresolution principle as employed in ATMS [de Kleer, 1989] in order to compute a conflict that do no longer contain constraints added in the son nodes. These conflicts allow to use conflict-based backjumping [Prosser, 1993] or dynamic backtracking [Ginsberg, 1993] as search method.
2. apply a (systematic and complete) backtrack search for a solution in each step of QUICKXPLAIN. Already detected solutions can be maintained internally and serve as starting points for some of the backtrack searches. They also can make some of the backtrack searches obsolete.
3. interleave or nest both approaches above.

Topics for future work are the decomposition of conflicts C that are too large for explanations and the refinement of conflicts that contain conjunctive constraints. Furthermore, we will adapt PBS [Junker, 2000] to find multiple preferred conflicts. Finally, we will work on heuristics that focus conflict detection on a relevant subset of constraints and thus reduce effort.

Acknowledgements

This work has profited from discussions with Derek Bennett, Gerhard Friedrich, Fred Garrett, Olivier Lhomme, Daniel Mailharro, Thierry Petit, Jean-Francois Puget, Jean-Charles Regin, and many other colleagues. The work has partially been supported by the European Commission under contract EP 28448 GNOSIS-VF.

References

[Bakker *et al.*, 1993] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving overdetermined constraint satisfaction problems. In *IJCAI-93*, pages 276–281, 1993.

[de Kleer, 1989] Johan de Kleer. A comparison of ATMS and CSP techniques. In *IJCAI-89*, pages 290–296, Detroit, MI, USA, 1989. Morgan Kaufmann.

[de Siqueira N. and Puget, 1988] J. L. de Siqueira N. and J.-F. Puget. Explanation-based generalisation of failures. In *ECAI-88*, pages 339–344, Munich, FRG, 1988. Pitman Publishers.

[Doyle, 1979] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Felfernig *et al.*, 2000] A. Felfernig, G. Friedrich, and D. Jannach. Automated generation and validation of configurator knowledge bases using ilog configurator. Technical report, University of Klagenfurt, 2000.

[Fernandez and Hill, 2000] A. Fernandez and P.M. Hill. A comparative study of eight constraint programming languages over the boolean and finite domains. *Constraints*, 5(3), 2000.

[Geffner and Pearl, 1992] H. Geffner and J. Pearl. Conditional entailment: Bridging two approaches to default reasoning. *Artificial Intelligence*, 53:209–244, 1992.

[Ginsberg, 1993] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[ILOG, 2000] ILOG. Ilog Configurator. Reference manual and User manual. V2.0, ILOG, 2000.

[Junker and Konolige, 1990] U. Junker and K. Konolige. Computing the extensions of autoepistemic and default logics with a truth maintenance system. In *AAAI-90*, pages 278–283, Boston, MA, 1990. MIT press.

[Junker, 2000] U. Junker. Preference-based search for scheduling. In *AAAI-2000*, pages 904–909, Austin, Texas, 2000.

[Jussien and Lhomme, 1998] N. Jussien and O. Lhomme. Dynamic domain splitting for numeric CSPs. In *ECAI-98*, pages 224–228, Chichester, 1998. John Wiley & Sons.

[Jussien and Lhomme, 2000] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *AAAI-2000*, pages 169–174, Austin, Texas, 2000.

[Jussien *et al.*, 2000] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP'2000*, pages 249–261, Singapore, 2000. Springer-Verlag.

[Martins and Shapiro, 1988] J.P. Martins and S.C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35:25–79, 1988.

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Régin *et al.*, 2000] J. C. Régin, T. Petit, C. Bessière, and J. F. Puget. An original constraint based approach for solving over constrained problems. In *CP'2000*, pages 543–548, 2000.

[Regin, 1994] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94*, pages 362–367, 1994.

[Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–952, 1987.

[Yokoo, 1995] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *CP'95*, pages 88–102, 1995.