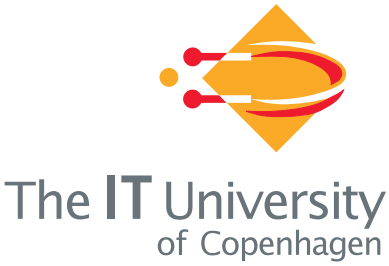

Interactive Reconfiguration in Power Supply Restoration

Tarik Hadzic
Henrik Reif Andersen

IT University Technical Report Series

TR-2005-68

ISSN 1600–6100 **June 2005**

ISSN 1600–6100

ISBN 87-7949-098-0

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web www.itu.dk

Interactive Reconfiguration in Power Supply Restoration

Tarik Hadzic and Henrik Reif Andersen

Department of Innovation, IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{tarik,hra}@itu.dk

Abstract. Interactive configuration is the problem of assisting a user in selecting values for parameters that respect given constraints. The problem was introduced as a problem of *product configuration* with the emergence of the mass-customization paradigm in product manufacturing but has also been applied to other application areas. Examples include specifying a product (a PC or a car), a service (a plane ticket or an insurance) or setting up equipment (a VCR or heating controller). Intuition is that in these situations, there is no definable unique best solution, and therefore a user should instead be guided in selecting the appropriate values for the parameters while at the same time obeying the constraints and meeting user preferences. The guidance takes the form of immediate feedback on the consequences of choices. There are three main important features required of an implementation of interactive configuration: It should be complete, backtrack-free, and provide real-time performance. It is a computational challenge to obtain all three simultaneously.

In this paper we look at *interactive reconfiguration*, where the starting point is a full valid configuration, which for external reasons becomes inconsistent and therefore has to be changed back to a consistent configuration. We take the approach of determining a small set of parameters that need to be changed and on these perform interactive configuration to get back to a consistent configuration. We present two BDD-based precompilation algorithms for solving the problem. One based on a monolithic BDD-representation of the solution space and another using a set of BDDs. We carry out experiments on a set of *power supply restoration* benchmarks and show that the set-of-BDDs algorithm scales well. In fact, we are able to perform interactive reconfiguration on examples where interactive configuration is not possible due to explosions in the size of the corresponding monolithic BDDs. This shows that even systems that are too large for full interactive configuration could be amenable to reconfiguration.

1 Introduction

Interactive configuration problems are special applications of CSP problems where a user is assisted by supporting software to interactively assign values to variables. This software, called a configurator, assists a user by calculating and displaying valid choices for each unassigned variable in what is called *valid domains computations*. Application areas include customizing physical products (such as PCs and cars) and services (such as airplane tickets and insurances).

There are three main important features required of an implementation of interactive configuration: it should be complete (all valid configurations should be reachable through user interaction), backtrack-free (a user is never forced to change an earlier choice due to incompleteness in the logical deductions), and provide real-time performance (feedback should be fast enough to allow real-time interactions). The requirement of obtaining backtrack-freeness and at the same time keep completeness makes the problem of calculating valid domains NP-hard. The real-time

performance requirement enforces further that runtime calculations are bounded in polynomial time and preferably in practice within approximately 250 milliseconds according to user-interface design criteria for experiencing real-time interaction [1]. Therefore the current approaches that meet all three conditions use offline precomputation to generate an efficient runtime data structure representing the solution space [2–5]. The challenge for the data structure is that the solution space is almost always exponentially large and it is NP-hard to find. Despite the bad worst-case bounds, it has nevertheless turned out in real industrial applications that the data structures can often be kept small [6, 4, 3].

The interactive configuration algorithms work from an initially empty assignment of values for the parameters and proceed towards a full assignment of values to all the parameters in interactions with the user. However, in this paper we take the viewpoint of starting from a complete configuration that becomes inconsistent because of a change to one or more of the parameters forced upon the configuration for external reasons. For example, in configuring a PC on a web-site, an earlier saved configuration could be revisited and it could be discovered that a certain component is no longer available for sale and another consistent configuration would have to be found.[1] In power supply distribution, a fault could cause a power distribution line to be shut down and a new configuration of the distribution network must be found.

Our approach is that in the reconfiguration situation, a subset of the parameters have to be changed in order to restore consistency. Often it will be desirable to change as few parameters as possible but other criteria that were also important in creating the original configuration will probably also be relevant to consider. Our point of view is that the user should be given control to interactively reassign variables in real-time in a complete and backtrack-free manner, and thus effectively explore the trade-offs between different criteria, such as selecting in his opinion the second best PC component instead of the out-of-stock component, or find a configuration of the power distribution network that tries to maximize the number of customers regaining electricity without shutting down distribution to important nodes such as hospitals and at the same time taking into account the plans for next week's planned maintenance on some of the supply lines.

We introduce *interactive reconfiguration* as a formulation of the problem of going from a full valid configuration, to a forced inconsistent configuration and back to a full valid configuration using interactive configuration in obtaining it. We describe two different BDD-based reconfiguration algorithms. A *monolithic* algorithm, which can be used when the entire problem can be compiled into a single BDD. A *set-of-BDDs* algorithm, which works with a set of BDDs and scales to much larger instances where a single BDD cannot be found, thus allowing us to perform interactive reconfiguration even when interactive configuration is not possible. We evaluate the performance of the algorithms on a newly introduced set of benchmarks from the field of *Power Supply Restoration (PSR)*, where reconfiguration is the daily mode of operations.

Interactive reconfiguration is related to the computation of *explanations* and *restorations* [7, 8, 2]. However, these concepts are usually defined in order to support the interactive configuration process and not in order to deal with an initial invalid configuration. The relationship is further elaborated in section 5.

The remainder of the paper is organized as follows. In section 2, we formally define interactive reconfiguration and describe the reconfiguration algorithms. The Power Supply Restoration benchmarks are described in section 3. In section 4,

[1] At the time of writing this happens at one of the main PC vendor's website, except that reconfiguration is not supported. The configuration is simply dropped as being invalid.

we present the empirical evaluation of our approach. Related work is presented in section 5, while section 6 concludes the paper.

2 Interactive Reconfiguration

The basis for our discussion of interactive reconfiguration is closely related to constraint satisfaction problems with the only difference being that instead of constraints as explicit sets of tuples, we use (the more compact) propositional formulae for representing constraints.

Definition 1. *A* configuration problem (CP) *C is a triple (X, D, F) where X is a set of variables $\{x_1, \ldots, x_n\}$, $D = D_1 \times \ldots \times D_n$ is the cartesian product of their finite domains $D_1, \ldots, D_n$ and $F = \{f_1, ..., f_m\}$ is a set of propositional formulae over atomic propositions $x_i = v$, where $v \in D_i$, specifying conditions on the values of the variables.*

A *total configuration* is an assignment ρ of values $v_1, \ldots, v_n$ to each of the variables represented as a set of pairs (x_i, v_i) such that $v_i \in D_i$. A *partial configuration* ρ is an assignment to a subset of the variables. We let $dom(\rho)$ denote the domain of the assignment ρ, i.e., $dom(\rho) = \{x_i \mid \exists v_i \in D_i.(x_i, v_i) \in \rho\}$. With these definitions a partial assignment for a configuration problem has $dom(\rho) \subseteq X$ and is total if $dom(\rho) = X$. A *total configuration ρ is valid* if it satisfies all the formulae, i.e. $\rho \models f_j$ for $j = 1, \ldots, m$, which we also abbreviate as $\rho \models F$. A *partial configuration ρ is valid*, abbreviated as $\rho \models_p F$, if it can be extended to a total valid configuration $\rho' \supseteq \rho$. Notice, that while determining a total configuration is valid is easy by a straightforward evaluation of all the formulae, determining whether a partial configuration is valid, involves solving an NP-hard satisfiability problem.

Interactive Configuration Given a configuration problem $C = (X, D, F)$ and a partial configuration ρ, *interactive configuration* is the process of assisting a user in interactively reaching a total valid configuration starting from ρ. The key operation in the interaction is that of computing for each unassigned variable $x_i \in X \setminus dom(\rho)$ the *valid domain* $D_i^\rho \subseteq D_i$ such that it contains those and only those values that can be extended to a total valid configuration, i.e. $D_i^\rho = \{v \in D_i \mid \exists \rho' : \rho' \models F \wedge \rho \cup \{(x_i, v)\} \subseteq \rho'\}$. We call this the *valid domains computation*. For ease of presentation, we shall take $D_i^\rho = \{v_i\}$ for $(x_i, v_i) \in \rho$ and refer to the tuple of all the D_i^ρ's as D^ρ.

At each step of the interaction, the configurator reports the valid domains to the user based on the current partial assigment of his earlier choices ρ. The user then picks an unassigned variable $x_j \in X \setminus dom(\rho)$ and selects a value from the calculated valid domain $v_j \in D_j^\rho$. The partial assignment is then extended to $\rho \cup \{(x_j, v_j)\}$ and another interaction step is initiated.

In previous work [4, 9] this functionality was obtained by representing the set of solutions to C, all the total valid configurations, as a Binary Decision Diagram (BDD) [10]. The BDD represents the set $Sol = \{\rho \mid \rho \models F\}$ through a proper encoding of the finite domains with Boolean variables. For a given valid partial assigment ρ, we denote by $Sol^\rho = \{\rho' \in Sol \mid \rho \subseteq \rho', \rho' \models F\}$ the set of total solutions consistent with ρ.

The interactive configuration algorithm $InCo$ in figure 1 summarizes the interactive process starting from the compiled representation of the set of solutions Sol and a valid partial assignment ρ, which in typical applications initially is empty.

The $InCo$ algorithm helps us to reach a valid total configuration as an extension of the argument ρ.

```
InCo(Sol, ρ)
1:      while |Sol^ρ| > 1
2:            compute D^ρ = ValidDomains(Sol, ρ)
3:            report D^ρ to the user
4:            the user chooses (x_i, v) for some x_i ∉ dom(ρ), v ∈ D_i^ρ
5:            ρ ← ρ ∪ {(x_i, v)}
6:      return ρ
```

Fig. 1. Interactive configuration algorithm working on a monolithic representation of the solutions Sol as a BDD

Interactive Reconfiguration For reconfiguration, we need to model externally forced changes to the current total assignment ρ. We model the external events as a partial assignment ρ_f (f for fixed: it must not be changed in the process of obtaining a new valid configuration). An assignment ρ can be updated with another assignment ρ_f by overwriting assignments in ρ with assignments on overlapping variables in ρ_f. We denote this operation by

$$\rho[\rho_f] = \{(x_i, v_i) \mid (x_i, v_i) \in \rho_f \text{ or } (x_i \notin \mathrm{dom}(\rho_f) \text{ and } (x_i, v_i) \in \rho)\}.$$

Reconfiguration comes into play when $\rho[\rho_f]$ is invalid. We want to keep most of the existing assignments unchanged and therefore we try to compute a small *release set* of variables $R \subseteq X \setminus \mathrm{dom}(\rho_f)$ that need to be unassigned from $\rho_1 = \rho[\rho_f]$ in order to reach a valid partial configuration $\rho_2 \supseteq \rho_f$. The partial configuration ρ_2 is found as $\rho_1 \upharpoonright R = \{(x_i, v_i) \in \rho_1 \mid x_i \notin R\}$.

Definition 2 (Interactive Reconfiguration). *Given a configuration problem $C(X, D, F)$, a starting valid total configuration $\rho \models F$ and a forced partial assignment ρ_f such that the updated total configuration $\rho_1 = \rho[\rho_f]$ is invalid. The* reconfiguration problem *is to find a (small)* release set *$R \subseteq X \setminus dom(\rho_f)$ such that the partial assignment $\rho_2 = \rho_1 \upharpoonright R$ is valid if such a set exists or report that it does not exist.*

Notice, that if ρ_f by itself is invalid it will be impossible to find a release set. After computing R, we provide the user with interactive configuration functionalities on ρ_2, in order to again arrive at a total configuration.

We now explain the two different approaches to interactive reconfiguration: *monolithic* and *set-of-BDDs*.

2.1 Monolithic Approach

The algorithm in figure 2 presents interactive reconfiguration in the monolithic approach.

```
InRecoMono(Sol, ρ, ρ_f)                    /* ρ is valid and total */
1:  ρ_1 ← ρ[ρ_f]                           /* ρ_1 is invalid and total */
2:  R ← PickReleaseSetMono(Sol, ρ, ρ_f)
3:  ρ_2 ← ρ_1 ↑ R                          /* ρ_2 is valid and partial */
4:  ρ' ← InCo(Sol, ρ_2)                    /* ρ' is valid and total */
5:  return ρ'

PickReleaseSetMono(Sol, ρ, ρ_f)
1:      R ← ShortestPath(Sol^{ρ_f}, ρ, cost)
2:      return R
```

Fig. 2. Interactive reconfiguration algorithm

The key part of the algorithm is the *PickReleaseSetMono* function (line 2) which computes a release set R given the BDD for the full solution space Sol. We first restrict Sol to Sol^{ρ_f} as BDD operations. We then find the set of variables corresponding to the path of lowest cost using the cost function that has zero cost on BDD edges corresponding to assignments in ρ, and a positive cost on all other assignments. The algorithm works as a depth-first traversal of the BDD and can thus be implemented to run in linear time.

A similar algorithm for finding minimum cost paths is explained in more details in [11].

2.2 Set-of-BDDs Approach

Sometimes the monolithic approach is not feasible because the intermediate or resulting BDD for representing the solutions Sol becomes too big. We therefore develop an algorithm based on a *set of BDDs*. There will be a BDD for each of the formulae $f_j \in F$. We denote the j'th BDD by Sol_j and the full set of BDDs by $SSol$. In the precompilation step, $SSol$ will be computed. When reconfiguration has to take place, we will find a release set R in an incremental fashion and compute a single BDD $RelSol$ of the relevant part of the solution space to be used for reconfiguration. The BDD $RelSol$ is found as a conjunction of the BDDs $Sol_j^{\rho_2}$ corresponding to the BDDs Sol_j restricted with the assignment ρ_2. The algorithm in figure 3 illustrates this approach.

```
InRecoSoB(SSol, ρ, ρ_f)                    /* ρ is valid and total */
1:  ρ_1 ← ρ[ρ_f]                           /* ρ_1 is invalid and total */
2:  R ← PickReleaseSetSoB(SSol, ρ, ρ_f)
3:  ρ_2 ← ρ_1 ↑ R                          /* ρ_2 is valid and partial */
4:  RelSol ← ⋀_{j=1}^{m} Sol_j^{ρ_2}
5:  ρ' ← InCo(RelSol, ρ_2)                 /* ρ' is valid and total */
6:  return ρ'
```

Fig. 3. Interactive reconfiguration algorithm for sets of BDDs

A key element in the algorithm, is the incremental computation of the release set (line 2). There are many ways in which this could be done. We formalize it as a function $next(Y)$ which from a set of variables Y finds a next larger set of variables to be tried for a release set. In each incremental step, we verify whether the current candidate for a release set will provide a valid partial configuration by checking that if the set of variables is indeed released, the resulting partial configuration has at least one extendable solution through a satisfiability check. The satisfiability check is performed with the algorithm SoBSAT($SSol, \rho$) which determines whether there exists a total $\rho' \supseteq \rho$ fulfilling all the BDDs in $SSol$.

The algorithm SoBSAT is implemented as a Propositional Constraint Solver that is based on a BDD representation of individual (propositional) constraints, using the learning and conflict-resolution mechanisms of modern SAT solvers [12, 13]. It is implemented on top of the BDD-package Buddy [14].

The computation of the release set is presented in figure 4.

For the function $next(Y)$ (line 7) we require that it always adds at least one variable unless Y is the full set of variables: $next(Y) \supseteq Y$ and $next(Y) = Y$ implies $Y = X$. Otherwise the function can be chosen depending upon the application. It could for instance use structural information in the problem in order to try to select variables that are close to the variables given as arguments so that there is some

```
PickReleaseSetSoB(SSol, ρ, ρ_f)
1:      Δ ← dom(ρ_f)
2:      ρ_1 ← ρ[ρ_f]
3:      R ← ∅
4:      while not SoBSAT(SSol, ρ_1 ↑ R) do
5:         if R ∪ Δ = X then
6:             halt /* all variables tried, no solution: ρ_f ⊭ F */
7:         R ← next(R ∪ Δ) \ Δ
8:      end
9:     return R
```

Fig. 4. Pick release set algorithm in the set-of-BDDs approach

locality in the spreading of release set approximations. The function could even be given a priori by a user describing preferred sequences of release sets to try out or interactively by adding in variables on the fly. In the present experiments we used two domain specific heuristic algorithms for computing $next(Y)$ as described in the experiments section.

2.3 Complexity Issues

The complexity of checking whether ρ_f is restorable (i.e. answering whether $\rho_f \models_p F$) depends on the underlying data representation. In the monolithic case this task is trivial. It is equivalent to checking wether the BDD for Sol is the terminal node corresponding to false. If all the rules are represented separately as a set of BDDs it is NP-hard since the Boolean satisfiability problem [15] can be reduced to it.

The task of finding the optimal release set in the monolithic case is linear in the size of the BDD representation of solutions Sol. The complexity of computing a release set in the set-of-BDDs approach is NP-hard since in every incremental step, the algorithm checks restorability of a partial configuration $\rho_1 \uparrow R$ (Fig. 4, line 4). It is expected however that the NP-hard part of the problem will not inflict too heavy a computational overhead in the real world instances where the unassigning heuristic ($next(Y)$ function) exploits the domain specific structure encoded into the instance and works only on a small subset of the entire configuration. This intuition is supported by our experiments with Power Supply Restoration.

3 Power Supply Restoration

Distribution Network The supply of electrical power involves *power generation* at power plants, transmission over larger areas via high-voltage *transmission grids*, and transmission to consumers via local *distribution networks*.

A distribution network is organized in *substations*, local power sources that supply a number of *radial feeders* which are tree structures of powered electric *lines*. Each line has two *switching devices* at both ends (which when both turned off effectively isolates the line from the electric system). *Nodes* are connection points between the lines, and can be just a branching point or also a connection to a *sink*, a transformer station that consumes electricity from the network and delivers it to final consumers. For a more detailed description, please refer to [16–19].

Electric lines can become faulty, for example during bad weather conditions, when trees fall down, in which case the feeder supplying the line is turned off. This affects the entire area supplied from the feeder, and the problem is in reconfiguring the network by opening and closing lines, to resupply the maximum number of consumers in the affected areas while addressing a number of other domain specific goals (minimize the number of intrusive actions [20], minimize the change of the

standard network topology, do not overload lines or do that for just a short period, prioritize in resupplying specific areas).

Model For each line L we introduce two variables: L_dir and L_load. The variable L_dir represents a flow direction in a line which can be: off, forward, or backward (depending on whether the line is off or on and from which direction it is being supplied). The current load in the line (L_load) is a finite domain integer (in our case the domain is integers from 0 to 13). Every sink that is powered consumes one unit of current. A sink is modelled via a Boolean variable S_powered which represents a sink being either off or on. A power source (a substation) can also be supplying energy or not, which is also modelled as a Boolean variable P_powered.

One of the main requirements in modelling the PSR network is to forbid feeding a node from two different directions, and to model physical laws regulating current loads in the lines. To do this, we generated explicit set of direction rules for each node and used Kirchoff's laws stating that the current going into a node must equal the coming out of the node. For a detailed description refer to [19].

Instances We use a number of new instances from the Power Supply Restoration domain. They were created by Stuart Henney, Tine Bak, Rene Jensen and Lars Sonne [18, 19] in collaboration with NESA - the Danish power distributor in the Copenhagen area [21]. All the instances are made available for download at [22]. Structural properties of these instances are reported in Table 1.

Table 1. Benchmark properties

Benchmark name	#Lines	#Sinks	#Rules	#Variables	Size(KB)
Std-diagram	13	7	72	36	215
1-6+22-32	21	17	122	61	455
Complex-P2	24	18	138	75	3.282
Complex-P3	28	19	159	88	61.240
1-32	38	32	222	110	-
Large	66	56	386	190	-
Complex-P1	102	82	592	299	-
Complex	146	119	849	414	-

The first column in the table lists the name of the benchmark. Subsequent 2 columns list the number of lines and sinks in the network topology of the instance. The following 2 columns (#Rules and #Variables) indicate the number of declared variables and rules in our encoding of the instance. Finally, in case the compilation of entire network was possible, the last column indicates the size of resulting BDD in Kilobytes.

4 Experimental Evaluation

We implemented the reconfiguration algorithms, on top of CLab [23], a library of interactive configurator based on a single BDD representation. All the experiments were carried out on a Pentium-Xeon machine with 4GB RAM and 1MB L2 Cache, running Linux.

Two Set-of-BDDs heuristics We implemented two domain specific heuristics in the set-of-BDDs approach for the next-computations, and measured their performance.

The first heuristic H_1 unassigns all the variables (L_dir, L_load) representing the lines that are *downstream* the fault line (i.e. the lines that were supplied with electricity coming through the fault line), and unassigns only the load variables for the lines *upstream* the fault line (the lines on the path from the power source to the fault line). The heuristic additionally unassigns all the *border lines*, the healthy lines that were turned off in the original configuration since their neighboring lines were supplied from different feeders. Heuristic H_1 allows a maximum number of 50 variables to be unassigned (referred to as the *threshold*).

The second heuristic H_2, additionally, through the border lines, identifies neighboring feeders, and selects the ones with the biggest capacity (i.e. feeders that can additionally supply the biggest number of sinks) and unassigns load variables for all the lines on the path from the border line to the power source for the neighboring feeder. The heuristic increases the threshold to 70 variables.

The first heuristic puts more emphasis on response time while the second one invests more in the restoration quality. Namely, the number of released variables influences the time needed to compile the BDD representing the restricted solution space $RelSol$, and the smaller the threshold, the less time is used and lesser quality of restoration is reached.

The intuition behind these heuristics is that we usually need to unassign only few variables compared to the total number of parameters, since the real-world instances have a built in structure that can be effectively exploited (by appropriate heuristics) to restrict the effects of external updates ρ_f to the locally affected area.

Simulation In each simulation, we loaded a precalculated valid initial configuration ρ, randomly picked a powered line from ρ and made it a fault line which resulted in an invalid total configuration ρ_1. For each heuristic H ($H \in \{H_1, H_2\}$) we ran the reconfiguration algorithm and measured the time t needed to calculate a release set R, and to compile a resulting BDD $RelSol$ (Fig. 3, page 5). The compilation was not executed and statistics was not recorded when the release set was greater then the designated threshold ($|R_1| > 50$, $|R_2| > 70$), a choice made according to the heuristic goals as described above. In these cases, we believe it is reasonable to use the valid total configuration found by the satisfiability check $SoBSAT$ to reassign some of the released variables until the number is reduced below the threshold.

After getting the restoration $\rho_2 = \rho_1 \uparrow R$ and compiling the solution space $RelSol$, we measured the average maximum number of variables representing line directions (L_dir) that could be left unchanged during configuration with $InCo(RelSol, \rho_2)$. Although this number is maximal only with respect to $RelSol$ (i.e. there might be a larger maximum for a different R), it still gives us an estimate of the quality of restoration when comparing against the number of starting powered lines in ρ.

We also measured the average maximum number of resupliable sinks as another indication of the restoration quality. All statistics are reported separately for heuristics H_1 and H_2. In Table 2, columns labelled S_1, S_2 indicate maximum resupliable sinks, $RDir_1$, $RDir_2$ maximum unchanged line directions, and t_1, t_2 the corresponding running times.

When a single BDD representing the solutions Sol of the entire instance was compilable, we measured the time t needed to restrict the starting BDD with the fault line restriction (ρ_f). We used the resulting representation of Sol^{ρ_f} to measure the same statistics as in the set-of-BDDs approach: the maximum unaffected number of line directions $RDir$ and the maximum number of resupliable sinks S. In this

case, the calculated numbers are global optimums, i.e. it is not possible to keep more lines unchanged, or to resupply more sinks, whatever the release set R.

For each PSR instance, 100 seed-based pseudo-random simulations were generated. The same random sequence was used for both approaches. In case of the instances Complex-P2 and Complex-P3 only 10 simulations were run, due to the slow response times in the monolithic case. The restoration quality numbers underapproximate the actual quality in the sense that they compare against the total number of defined lines and sinks, whether it is possible to reach them or not. This is best illustrated in the monolithic case, when although we have global optimums, the numbers indicate less then 100% quality of restoration. In case of the "Complex" instance the initial number of powered sinks (104) is smaller then the number of defined sinks (119) and the above remark does not necessarily apply. We used the best known variable ordering for the BDD compilation throughout the experiments (fan-in).

The results are shown in Table 2. The first column lists the name of the benchmark. The remaining columns indicate the measured statistics as already described: the second, third and fourth column list the estimate of the maximum number of resupliable sinks (one for the monolithic approach and the other two for heuristics H_1 and H_2). The following three columns indicate the maximum number of unchanged line directions while the last three columns indicate the corresponding running times.

Table 2. Combining Experimental Results

Benchmark	**Restoration quality**						**Avg. RT (sec)**		
	$S(\%)$	$S_1(\%)$	$S_2(\%)$	$RDir(\%)$	$RDir_1(\%)$	$RDir_2(\%)$	t	t_1	t_2
Std-diagram	98	96.00	96.00	75.54	75.54	75.54	0.17	0.87	1.31
1-6+22-32	100	99.47	99.47	77.33	77.33	77.33	0.50	0.16	0.25
Complex-P2	100	85.19	97.22	84.17	77.31	84.17	3.88	0.14	0.36
Complex-P3	100	90.00	98.42	91.07	91.07	91.07	132.02	0.12	4.44
1-32	-	91.53	99.00	-	91.82	91.82	-	0.10	0.28
Large	-	93.98	98.73	-	94.89	94.89	-	0.27	1.43
Complex-P1	-	79.26	96.94	-	78.96	96.86	-	0.77	15.58
Complex*	-	86.5	91.92	-	85.52	91.67	-	3.11	12.05

*There are 119 defined sinks, but only 104 powered sinks in starting assignment. The numbers reported are comparisons against the latter number.

The numbers in Table 2 indicate that the set-of-BDDs approach scales dramatically better. The biggest instance where the monolithic approach was applicable was the instance Complex-P3 (28 lines and 19 sinks) with response time of 132.02 seconds, compared to the five times bigger instance Complex (146 lines and 119 sinks) that was handled in 42 times shorter time (3.11 seconds).

The quality of restoration depends on how far we are willing to search throughout the distribution network. H_2 achieves better quality than H_1 but worse response time. Namely, H_2 also unassigns lines belonging to neighboring feeders as it tries to resupply the faulty line. This is best illustrated in case of the instance Complex-P1 where the H_2 heuristics achieves approximately 17% better quality at the price of 20 times longer response time. There is an obvious tradeoff between the quality of the restoration and the running time. As another illustration for this we ran 100 simulations on the Complex instance with a third heuristic H_3, where the H_1 heuristic was combined with an increased threshold of 100 variables. We got

$S_3 = 96.23$, $RDir_3 = 96.39$ and $t_3 = 55.35$, compared to H_1 a 10% gain in quality for 17 times longer response time.

In comparison to monolithic approach, the H_2 heuristic achieves the same (globally optimal) quality w.r.t. *RDir* indicator, and achieves the quality within the 2% of the global optimum w.r.t. to the number of ressupliable sinks.

The high percentage of ressupliable sinks and unaffected lines (most under-approximated quality estimates are above 90%) supports the intuition about the locality of external effects in the real world instances (recovery within the 10% of change in network topology). We hope to further confirm this by running experiments on instances coming from other domains.

5 Related Work

Computing *explanations* is related to our work, but there is no unique formal definition of that problem. Most research is done within the framework of interactive configuration, i.e. providing a user with useful reasons for the (un)available options or inconsistencies during interaction. This affects the underlying implementation algorithms which are closely interleaved with *constraint propagation* mechanisms invoked *during search* for solutions during user interaction [7, 8, 24, 25]. We however, start from the finished configuration and cannot reuse the knowledge synthesized during search.

More related is the problem of *restoring a constraint* in assumption-based CSPs [2] where the algorithms determine the subset of the current user assignments that can be kept while the given constraint is satisfied. In fact, computing the release set R can be seen as the generalized restoration of the non-unary constraint ρ_f. However, besides the general definition, the published algorithms work only on the unary constraints and base it on the single-automata compiled representation (similar to our monolithic approach) that suffers from the same scalability problems as our monolithic approach of sometimes encountering that compilation is not possible.

In [11] the author introduces a scalable tree-of-BDDs approach for computing restorations of unary constraints. The tree-of-BDDs is a data structure implementing a tree decomposition of the original cyclic constraint problem and when this can be done, it offers globally optimal restorations. Our set-of-BDDs approach provides greater scalability, since it works even when tree decomposition is not possible, although at the price of potentially suboptimal restoration quality.

The term reconfiguration has also been used to describe the problem of adjusting an existing product configuration to meet new constraints [26]. The focus there is however on conceptually managing the consistent representation of evolving configuration knowledge and not on concrete algorithms for adjusting product individuals by restoring valid configurations under the assumption of unchanged configuration knowledge.

There is a number of other, domain-specific approaches for restoring services in electric power distribution systems [27]. However, we have presented general algorithms exemplified only by power supply restorations examples. The planning community also uses benchmarks from the PSR domain as a real-world instances for evaluation of planning algorithms [16, 20]. However, they are solving a different problem (planning under uncertainty) and to the best of our knowledge use benchmarks of much smaller size than those introduced in this paper.

6 Conclusions and Future Work

In this paper we introduced a concept of interactive reconfiguration that addresses the problem of recovering consistency once the existing valid total configuration has been invalidated with the external effects over which a user has no control.

We proposed two implementation approaches, first an exact approach based on a monolithic BDD representation of the entire problem, and second a set-of-BDDs approach that heuristically computes release sets when the compilation to a single BDD is not feasible.

We demonstrated that the set-of-BDDs approach scales dramatically better, handling instances that are far out of reach for the monolithic approach, with sorter response time. By implementing two unassignment heuristics we further identified and highlighted the tradeoff between the quality of restoration and response time that can be adjusted to domain specific requirements. The high percentage of reachable restoration quality confirms our intuition about the local nature of external effects in the real-world instances.

In the future we plan to work on developing general domain-independent unassignment heuristics and compare them against the domain-specific ones. We intend to further extend the scalability and quality of the set-of-BDDs approach by offering interactive configuration of released variables directly on set-of-BDDs for instance when they can be decomposed to a tree-of-BDDs representation [11] instead of the more restrictive single BDD representation of the relevant solution set.

Acknowledgments

We would like to thank Rene Jensen, Lars Sonne, Tine Bak and Stuart Henney for their contribution to the PSR benchmarks.

References

1. Raskin, J.: The Humane Interface. Addison Wesley (2000)
2. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. Artificial Intelligence **1-2** (2002) 199–234 `ftp://fpt.irit.fr/pub/IRIT/RPDMP/Configuration/`.
3. Madsen, J.N.: Methods for interactive constraint satisfaction. Master's thesis, Department of Computer Science, University of Copenhagen (2003)
4. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: PETO Conference, DTU-tryk (2004) 131–138
5. Møller, J., Andersen, H.R., Hulgaard, H.: Product configuration over the internet. In: Proceedings of the 6th INFORMS Conference on Information Systems and Technology. (2004)
6. Configit Software A/S. `http://www.configit-software.com` (online)
7. Jussien, N.: e-constraints: explanation-based constraint programming. In: CP01 Workshop on User-Interaction in Constraint Satisfaction, Paphos, Cyprus (2001)
8. Jussien, N., Barichard, V.: The PaLM system: explanation-based constraint programming (2000)
9. Subbarayan, S., Jensen, R.M., Hadzic, T., Andersen, H.R., Hulgaard, H., Møller, J.: Comparing two implementations of a complete and backtrack-free interactive configurator. In: CP'04 CSPIA Workshop. (2004) 97–111
10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **8** (1986) 677–691
11. Subbarayan, S.: Integrating CSP decomposition techniques and BDDs for compiling configuration problems. (To appear in Springer LNCS volume of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CP-AI-OR, May 2005)
12. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Design, Automation, and Test in Europe (DATE '02). (2002) 142–149
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001)

14. Lind-Nielsen, J.: BuDDy - A Binary Decision Diagram Package. `http://sourceforge.net/projects/buddy` (online)
15. Cook, S.A.: The complexity of theorem proving procedures. ACM Symposium on Theory of Computing (1971)
16. Thiébaux, S., Cordier, M.O.: Supply restoration in power distribution systems – a benchmark for planning under uncertainty. In: Pre-Proceedings of the 6th European Conference on Planning (ECP-01). (2001) 85–96
17. Bertoli, P., Cimatti, A., Slanley, J., Thiébaux, S.: Solving power supply restoration problems with planning via symbolic model checking. In: Proceedings of the 15th European Conference on Artificial Intelligence ECAI'02. (2002)
18. Bak, T., Henney, S.: Power Supply Restoration - a Constraint-based Model for Reconfiguration of 10kv Electrical Distribution Networks. Student project at IT University of Copenhagen (2004)
19. Sonne, L., Jensen, R.: Power Supply Restoration. Master's thesis, Department of Innovation, IT University of Copenhagen (2005)
20. Bonet, B., Thiébaux, S.: GPT meets PSR. In: 13th International Conference on Automated Planning and Scheduling (ICAPS-03). (2003)
21. Nesa. `http://www.nesa.dk` (online)
22. Power Supply Restoration Benchmarks. `http://www.itu.dk/people/tarik/psr` (online)
23. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. `http://www.itu.dk/people/rmj/clab/` (online)
24. Junker, U.: QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI'01 Workshop on Modelling and Solving problems with constraints, Seattle, WA, USA (2001)
25. Freuder, E.C., Likitvivatanavong, C., Wallace, R.J.: Explanation and implication for configuration problems. In: Workshop on Configuration, 17th International Joint Conference on Artificial Intelligence (IJCAI-01). (2001)
26. Tiihonen, J., Sulonen, R., Soininen, T., Mnnist, T.: Framework and conceptual model for reconfiguration (1999)
27. Rudnick, H., Harnisch, I., Sanhueza, R.: Reconfiguration of electric distribution systems. In: Revista de la Facultad de Ingeniera, Universidad de Tarapac (1997) 41–48