


9 Conclusion

In order to provide user support in the case of a detected inconsistency, we first developed a firm theoretical basis and then implemented the ideas in a prototype. We could thus prove that our original goal to develop a system which is able to analyze an inconsistency in a deductive database, and then automatically generate repairs is well attainable.

In the long run, consistency constraints may become an alternative to the ever increasing complexity of new data models that are supposed to capture more of the minilworld semantics. However, to be viable as an alternative, the performance of constraint checking must be improved by orders of magnitudes over what is known today. The methods presented in this paper are no exception. Presently, the performance limits the applicability of the system in practice. Since there is not only one consistency check for each user transaction but as many checks as there are generated causes, the consistency check component is the most critical one. The time spent for query processing during a consistency check cannot be neglected either. Consequently, enhancement of system performance should proceed along two lines:

1. decrease the number of consistency checks, and
2. enhance the performance of the critical components.

The best way to decrease the number of consistency checks is to minimize the number of causes considered by introducing further heuristics. Especially domain-dependent heuristics should be developed because they seem the most powerful ones.

Enhancing the performance of the query answering component can be achieved by introducing algorithms for special cases of rules, e.g., the Henschen-Naqvi algorithm ([16]), or magic set improvements ([31]). To improve the consistency check component we are currently developing a special compilation technique for constraints. This method is based on the similarities of query processing and constraint checking.

Another point worth mentioning is the following: While testing the system on several examples we made the observation that sometimes the system came up with very unplausible repairs. In all those cases we detected afterwards a faulty modeling of the domain. Thus, another problem was recognized: Whereas our tool can be seen as supporting the design of transactions very little has been done to support the design of deductive databases schemas (for exceptions see [14, 4]). We think that both kinds of design aids must become more active areas of research in order to provide the user with sufficient support. Otherwise he/she won’t be able to efficiently use the new database technologies developed.

Acknowledgment: We are grateful to the anonymous referees for the many helpful comments.

References


investigations to access path techniques such as quinary trees ([20]), multidimensional binary
trees ([3]), and combined attributes ([24]) with AVL trees or hashing used for indexing. Here,
quinary trees and combined attributes showed the best performance. Since the storage costs
for quinary trees are very high, combined attributes with hashing were implemented. Once we
extend the prototype to peripheral storage databases, modern multi-attribute index techniques
such as grid files ([34]) should be investigated as well.

Transaction management is comparatively simple. Since we developed an experimental main
memory database system we did not concern ourselves with recovery or multiuser synchroniza-
tion. Thus the 'ACID' properties isolation and durability were not considered. Atomicity and
consistency remain valid but have a new interpretation: consistency is checked at commit as
usual, but is then enforced, if necessary, by generating a repair, and/or by dialogue with the user.
Atomicity becomes irrelevant because transactions are never backed out (unless, of course, one
still allows for situations where a suitable repair cannot be determined). Consequently, transac-
tion management is a very simple component which just interprets the given transaction, and
performs the additions and deletions of facts.

The query processing component implements the deduction within the completion of the
database, i.e., \( C(DB) \). Further, there are three requirements we wanted to meet:

1. general recursion,
2. cyclic databases, and
3. simultaneous reasoning within multiple fact bases.

The third requirement is necessary since we have to maintain and access information about the
original database, the database as it results from the user’s transaction, and the database states
as predicted by the repairs.

To meet these requirements we have developed a modification of the QSQ, or frozen subquery
algorithm (cf. [41, 32]) and proven its correctness. To enhance efficiency this algorithm is
implemented using the rule-goal-graph as introduced in [40]. Reasoning with multiple fact bases
is realized through several entries in each goal node. Each entry then corresponds to those facts
which are deducible from one fact base.

In section 3.1 we introduced a very simple algorithm to check the consistency of a database.
A straightforward implementation of this method would inspect the entire database and check
all consistency constraints. This is quite inefficient. Performance can be enhanced if only
those constraints and facts in the database are considered which are affected by the performed
transaction. Under the traditional assumption of database consistency at transaction start, the
main idea is to reason forward from the transaction to those instantiations of the constraints
which necessarily have to be considered ([33]). As a realisation of this idea the subcomponent
'Dependent Instantiation' implements the algorithm given in [28] which unifies the approaches
of [22] and [19]. To avoid the search for unifiable literals within the rules and constraints,
both are organized in a graph where unifiable literals are connected by links of different kinds.
This graph bears similarity to connection graphs ([18]) and is introduced in [28]. However,
in the presence of \( \exists \)-quantifiers these methods are no longer applicable. Therefore, a general
instantiation procedure reflecting the :EX and :ALL method (cf. sec. 3.1, compare to [6, 5]) has
been developed and implemented ('General Instantiation'). The consistency check component
integrates all these methods and adds the methods that treat the Boolean connectors. For fast
checking the :ATOM method uses the query answering component.

The 'Extraction of Causes' component essentially realizes the \( REDUCE \) operator (cf. sec.
5) and the iteration procedure (cf. sec. 6.1). Sorting of repairs is left to the dialog component.
8 The System Architecture

In this section we briefly outline the architecture of a system for checking and repairing inconsistencies. The architecture was explored within a prototype implementation. It is depicted in figure 6. The bold arrows indicate the main data flows, whereas the broken arrows indicate the functionality of the user interface with the arrows pointing to modules servicing it. The prototype was implemented as a main memory database system.

The data management component stores and administrates all explicit data, i.e., facts, rules, and consistency constraints. At present, facts are stored in predicate form (for work on deductive databases that store and access facts in traditional, say relational, form see, e.g., [2]). Very important is the efficient retrieval of facts for partial match queries. Since the instantiations may occur at different positions within the atom (as in is($x$, vehicle) and loc(bmw, $x$)) several binding patterns must be supported. This is done in the classical way by introducing a number of (redundant) access paths. Because of our restriction to main memory databases, we limited our
2. \( \text{del}(\text{dom} (\text{loc, vehicle})), \text{add}(\text{dom} (\text{loc, mob.obj})): 2,2,1 \)

3. \( \text{del}(\text{isa}(\text{automobile}, \text{mob.obj})), \text{add}(\text{isa}(\text{automobile}, \text{vehicle})): 2,5,1 \)

4. \( \text{del}(\text{is}(\text{bmw.car})), \text{add}(\text{is}(\text{bmw.vehicle})): 2,5,3 \)

5. \( \text{del}(\text{isa}(\text{car, automobile})), \text{add}(\text{isa}(\text{car, vehicle})): 2,6,3 \)

6. \( \text{del}(\text{isa}(\text{vehicle, mob.obj})), \text{add}(\text{isa}(\text{mob.obj, vehicle})): 2,8,3 \)

Actually, in our opinion the second repair seems the most elegant one. Note that it did not make it to the first place only because there are not enough individual cars with defined location, but the evaluation of each criterion for the first repair directly reflects the number of individual cars with defined location.

### 7.3 Explanations to the User

The user interface consists of a set of windows, one for each main module of the implemented system. Besides the usual features to choose a certain database, to answer queries, to execute transactions there is an additional collection of functions presented to the user by the analyzer window. Their purpose is to provide the user with the means to examine in more detail the process and the results of the analysis of the consistency check. The associated window becomes active in the case of a detected consistency violation. The pop up menu of this window has the following entries.

| try full analysis |
| complete global trace |
| browse global trace |
| extract candidates |
| browse candidates |
| resume discarded entries |

The first entry is useful for the beginner. Here the system automatically generates a list of potential and definite repairs. To look at each generated candidate the entry 'browse candidates' has to be selected. While this solution might be good for a beginner a more experienced user might want to have explicit control over the search process for potential and definite repairs. He/she even might want to walk through the extracted trace. In order to support these requirements the user can explicitly order the completion of the trace in the way explained in sections 4.3. Looking at the first levels of the trace the symptoms become visible to him/her. When viewing the application of the :EX method the user is able to look at the list of constants considered by the method, and, if necessary, to modify this list.

After the extraction of causes from the trace the user can walk through the list of potential and definite repairs. During browsing there is another menu active which allows the browsing through the list of potential and definite repairs. Repairs can also be presented in a sorted order. While looking at the entries in the candidate list the user has the possibility to discard an entry, i.e., to exclude this entry from further analysis. In the case the user made a mistake he/she is able to recover the discarded entries. If the user favors a particular entry he/she may selectively complete the local trace of this entry, select the potential causes, and browse through them.

Of course, the automatic analysis may be resumed at any time, and will use the user's information concerning favored and discarded entries to govern its search.
than symptoms, only the former are gathered in a list. In this section we argue that presenting to the user an unmodified list of definite causes is a bad idea, and suggest further steps of processing to provide further help. More specifically, there are three further requirements we will meet:

- Eliminating unwanted potential/definite causes.
- Sorting the list of definite causes due to plausibility.
- Enable user influence on the derivation process.

Below we expand on each of these points.

### 7.1 Reduction

Reduction of a list of causes may take place on the level of potential and/or definite causes. Remember the heuristic in the previous section where a limit of 1 or 2 iterations has been set on the development of definite and intermediate potential causes. Also applicable to both is the observation that adding facts to satisfy derived predicates does not make much sense. Thus, causes which try to add such facts to the fact base should be eliminated. Since these reductions do not depend on the domain we refer to those as domain-independent reductions.

Of course, there are also domain-dependent reductions. One possibility for our example is: Never touch the isa-hierarchy, i.e. do not manipulate facts concerning the isa predicate. Another example is the labeling of constraints which should never be affected by any cause. To express these domain dependent and domain independent reductions, reduction predicates can be formulated which are evaluated on symptoms, and/or causes.

### 7.2 Elegance

Even in our simple example the number of causes derived was as high as six (see ch. 6.2), and in rich application domains there may be tens of generated causes. The goal of this section is to go beyond reduction and to derive some criterion in order to sort the generated repairs by their “elegance” such that the most elegant repair is at the top, or at least among the first elements of the sorted list. More specifically, we formalize the following three criteria:

1. A repair should be short.
2. A repair should be simple.
3. The resulting repair should delete as little information as possible.

The first criterion is easy to formalize. We just have to count the number of operations in the repair. Thus, if \( TA \) is a repair then \(| TA |\) is the measure used for the first criterion. The second criterion should not simply use the number of changes to \( DB^a \) but should include the changes effected in \( C(DB) \). We define, therefore, for a given inconsistent database \( DB \) and some repair \( TA \) \( \Delta(DB, TA) := \{a|DB \models a, TA(DB) \not\models a\} \cup \{a|DB \not\models a, TA(DB) \models a\} \). The second measure for a repair then is \(|\Delta(DB, TA)|\). For the third criterion we just look at the number of elements in the first set in the definition of \( \Delta(DB, TA) \), i.e. \( |\Delta^-(DB, TA)| := \{a|DB \models a, TA(DB) \not\models a\} \).

Since quite often the three criteria do not coincide, some priority scheme among them is needed. We start first on the second criterion, followed by the third and the first.

For the example the system generates the following ordering on the list of repairs above:

1. del(loc(bmw,bmwpos)): 1,1,1
6. \( pe^6 = \neg is(a(mob, obj, vehicle)} \)

Cause 1 is a definite cause and will thus become an element of DC. All others are in PC. As we saw earlier \( pe^2 \) (cause 2) has to be amplified by \( \{ \neg dom(loc, mob, obj) \} \) in order to give a definite cause. The other causes also violate other consistency constraints concerning the semantics of is (cause 3) , or the semantics of the isa-hierarchy (cause 4,5,6) respectively. Thus they are also potential causes only. To summarize, in the first iteration the following causes are derived:

1. \( pe^1 = \{ loc(bmw, bmwpos), \}
2. \( pe^{1,2} = \{ dom(loc, vehicle), \neg dom(loc, mob, obj) \}, \)
3. \( pe^{1,3} = \{ \neg is(bmw, vehicle), is(bmw, car) \}, \)
4. \( pe^{1,4} = \{ \neg isa(car, vehicle), isa(car, automobile) \}, \)
5. \( pe^{1,5} = \{ \neg isa(automobile, vehicle), isa(automobile, mob, obj) \}, \)
6. \( pe^{1,6} = \{ \neg isa(mob, obj, vehicle), isa(vehicle, mob, obj) \} \)

Experience has shown that the higher the number of iterations, the more awkward the generated causes become. Thus, in practice the number of iterations can be limited. A good choice seems to be a limit of 1 or 2 iterations. As a side effect, the number of generated definite causes is reduced.

6.2 Transformation of Causes into Repairs

In this short section the transformation of causes into repairs is illustrated. Remember the definition of the mapping \( \alpha \) which transforms a transaction into a set of literals. Here, we need the reverse case \( (\alpha^{-1}) \) of mapping a set of literals into a transaction. This is done for each literal separately. Starting out with the empty transaction, we append to it for each positive literal \( a \) the operation \( del(a) \), and for each negative literal \( \neg a \) the operation \( add(a) \).

For our extended example, this yields the following transactions:

1. \( del(loc(bmw, bmwpos)) \)
2. \( del(dom(loc, vehicle)), add(dom(loc, mob, obj)) \)
3. \( del(is(bmw, car)), add(is(bmw, vehicle)) \)
4. \( del(isa(car, automobile)), add(isa(car, vehicle)) \)
5. \( del(isa(automobile, mob, obj)), add(isa(automobile, vehicle)) \)
6. \( del(isa(vehicle, mob, obj)), add(isa(mob, obj, vehicle)) \)

7 Cleaning up the Repairs

Under the general header of clean-up, we should consider two issues: first, how to limit the repair suggestions to the user to "sensible" ones, and second, since the user may still have to choose between various repair alternatives, how and in which form to provide him/her with enough information on the process of checking the consistency constraints and determining the causes and repairs. We consider the two issues in turn.

The functionality of the system described so far covers the extraction of potential/definite symptoms and potential/definite causes. Since the user obviously is more interested in causes
**Algorithm 6.1** This algorithm generates the set of definite causes for a given inconsistency. It uses the following definition of a union over $pc^F$ denoted by $\bigcup pc^F$ and defined by $\bigcup pc^i := \emptyset$, and $\bigcup pc^i, pc^\alpha := pc^i, pc^\alpha \cup \bigcup pc^\alpha$. This union corresponds to summing up the historical development of the cause.

**Input:**

$DB$: consistent database state  
$TA$: the user’s transaction, s.t. $TA(DB)$ inconsistent

**Initialization**

$$PC := \{pc^i | pc^i \text{ is a cause for } TA(DB)\},$$  
$$DC := \emptyset,$$  
$$IC := \emptyset.$$

**Loop:** As long as there exists some $pc^F \in PC$ execute the following steps:

1. $PC := PC \setminus \{pc^F\}$,  
2. If $TA \circ \alpha^{-1}(\bigcup pc^F)(DB^0)$ consistent then  
   $$DC := DC \cup \{\bigcup pc^F\}$$  
   else  
   if $pc^F$ inconsistent then  
   $$IC := IC \cup \{pc^F\},$$  
   else  
   Let $pc_1, \ldots, pc_n$ all potential causes for $TA \circ \alpha^{-1}(\bigcup pc^F)(DB)$.  
   $$pc^i, pc^\alpha := pc_i, \text{ for all } i = 1, \ldots, n.$$  
   $$PC := PC \cup \{pc^i, pc^\alpha | 1 \leq i \leq n\}.$$

**Output:** Print all elements in $DR$.

To demonstrate this definition, we extend our running example. We add some more constraints which reflect the semantics of an isa-hierarchy without multiple inheritance. The added consistency constraints are:

- $\forall x_1 \forall x_2 \forall z \text{is}(z, x_1), \text{is}(z, x_2) \implies x_1 = x_2$
- $\forall x_1 \forall x_2 \forall z \text{isa}(z, x_1), \text{isa}(z, x_2) \implies x_1 = x_2$
- $\forall x_1 \forall x_2 \text{isa}^+(x_1, x_2) \implies \neg x_1 = x_2$

Remember the initial causes:

1. $pc^1 = \{\text{loc(bmw, bmwpos)}\}$,
2. $pc^2 = \{\text{dom(loc, vehicle)}\}$,
3. $pc^3 = \{\neg \text{is(bmw, vehicle)}\}$,
4. $pc^4 = \{\neg \text{isa(car, vehicle)}\}$,
5. $pc^5 = \{\neg \text{isa(automobile, vehicle)}\}$,
associated repair will restore all violated constraints. The following procedure will iteratively compute such a "correct" combination (or set) from the potential causes.

Consider again our complete example in sec. 4.2 which contains two consistency constraints. Start with the constraint

$$\forall x_1 \forall x_2 \forall z \text{loc}(x_1, x_2) \land \text{dom}(loc, z) \Rightarrow is^*(x_1, z)$$

which is violated by the execution of the transaction TA, now renamed because it constitutes the first iteration

$$TA_0 := \text{del}(\text{isa(automobile, vehicle)}) \text{add}(\text{isa(automobile, mob_obj)})$$

One potential cause is \{\text{dom}(loc, vehicle)\}. Appending \(TA_0\) by this cause is denoted by \(TA_1 = TA_0 \circ \alpha^{-1}(\{\text{dom}(loc, vehicle)\})\), and results in

$$TA_1 := \text{del}(\text{isa(automobile, vehicle)}) \text{add}(\text{isa(automobile, mob_obj)}) \text{del}(\text{dom}(loc, vehicle))$$

As we have already seen, executing this transaction violates the constraint

$$\forall x \exists z \text{is}(x, at) \Rightarrow \text{dom}(x, z)$$

with a potential cause \{-\text{dom}(loc, mob_obj)\}. The database state resulting from the execution of

$$TA_1 \circ \alpha^{-1}(\{-\text{dom}(loc, mob_obj)\})$$

is consistent. Thus,

\{\text{dom}(loc, vehicle), \neg \text{dom}(loc, mob_obj)\}

is a new potential cause (we call it an amplification of the previous potential cause). Since there are no further violations, it also is a definite cause.

This idea is reflected in the following algorithm where \(pc^F\) denote sets of literals for a finite sequence \(F\) of natural numbers, \(PC\) is the set of potential causes, and \(DC\) is the set of definite causes. We use sequences of natural numbers to keep track of the history of each cause. Let the initial potential causes be \(pc^1\) and \(pc^2\). Of course, the amplifications of each of them are in general different. Assume the existence of three possible amplifications for each potential cause. We then denote by \(pc^{1,1}, pc^{2,1}, pc^{3,1}\) the amplifications of the first cause \(pc^1\), and by \(pc^{1,2}, pc^{2,2}, pc^{3,2}\) the amplifications of \(pc^2\).

In general, we have the following three possibilities for a potential cause \(pc\):

1. \(pc\) is a definite cause,
2. there exists a definite cause \(dc\), and \(pc \subseteq dc\), or
3. there is no definite cause subsuming \(pc\).

Of course, we have to detect those causes which can not be amplified to a definite cause. This is done by checking them for consistency, i.e., if a 'cause' contains a fact and its negation, it is eliminated from the set of potential causes and added to a set of inconsistent causes (\(IC\)). As long as a potential cause is consistent there is some hope to amplify it to a definite cause. In the case of contradicting consistency constraints the algorithm will terminate without having generated any definite cause, i.e., \(DC = \emptyset\). Otherwise at least the cause corresponding to the inverse transaction of \(TA_0\) is a member of the final set \(DC\).
\[\text{REDUCE}(N_i) =
\{\{\text{loc}(bMW, bMWpos)\}\}
\{\{\text{dom}(\text{loc, vehicle})\}\}
\{\{\neg\text{isa}(bMW, vehicle)\}\}
\{\{\neg\text{isa}(\text{car, vehicle})\}\}
\{\{\neg\text{isa}(auto, vehicle)\}\}
\{\{\neg\text{isa}(\text{mob_obj, vehicle})\}\}\]

For the node \(N\) labelled by \(\text{isa} \ast (auto, vehicle)\) we have
\[\text{REDUCE}(N) = \{\{\neg\text{isa}(auto, vehicle)\}, \{\neg\text{isa}(\text{mob_obj, vehicle})\}\}\]

For the top node of figure 3 we have:
\[\text{REDUCE}(N) =
\{\{\neg\text{isa}(bMW, vehicle)\}, \neg\text{isa}(\text{car, vehicle})\}, \neg\text{isa}(auto, vehicle)\}, \neg\text{isa}(\text{mob_obj, vehicle})\}\}
\]

and for the top node of figure 2 the resulting set contains
\[
\{\text{loc}(bMW, bMWpos)\},
\{\text{dom}(\text{loc, vehicle})\},
\{\neg\text{isa}(bMW, vehicle)\},
\{\neg\text{isa}(\text{car, vehicle})\},
\{\neg\text{isa}(auto, vehicle)\},
\{\neg\text{isa}(\text{mob_obj, vehicle})\}\\]

Note, that the resulting causes in this example are singletons. This is due to the absence of redundancy in the derivation.

6 From Potential to Definite Causes and Repairs

In this section, we discuss the problem of constructing definite causes from potential ones, and of translating causes into repairs.

6.1 Iterative Construction of Definite Causes

Up to now, we have only been concerned with the extraction of potential causes. But as seen in section 4.3 the repair of a violated consistency constraint according to a potential cause may cause a violation of another constraint which so far has been valid. Consequently, we need a procedure which generates definite causes, if they exist, from potential causes. Note the connection between potential and definite causes.

- \(R\) is a definite cause \(\succ\) there exists a potential cause \(R'\) s.t. \(R' \subseteq R\).

In other words, usually the same literal will cause only the violation of one constraint. If several constraints are violated this will be due to different causes. Consequently, a definite cause will usually be a combination of potential causes, limited however to those combinations whose
1. \( \{ r_1(a) \} \),
2. \( \{ r_1(a), r_2(c) \} \),
3. \( \{ r_2(b), r_1(a) \} \), and
4. \( \{ r_2(b), r_2(c) \} \).

Obviously, the sets 2 and 3 are not minimal. To resolve this problem a special operator \( RED \) is introduced. This operator has to deal with several causes which must considered at the same time. Viz. the example where the first derivation and the second derivation had to be disabled. The same occurs if we deal with several consistency constraints since they must hold concurrently, too. In general, this situation arises whenever the subnodes of a trace node are connected by :AND. The \( RED \) operator then combines sets of causes connected by :AND in a way that the resulting causes are minimal.

This operator works as follows. Let \( C_1, \ldots, C_n \) be causes connected by 'and'. For example, the \( C_i \) are sets of literals each responsible for the disabling of a different derivation. For these sets the \( RED \) operator builds up a set containing all possible minimal sets which are supersets of at least one set in each \( C_i \). If we have built all sets this way we are sure that the collection contains the minimal causes.

**Definition 5.1 (RED)** Let \( \Sigma \) be a signature. Then the mapping \( RED : \mathcal{P}(\mathcal{P}(\text{Lit}_\Sigma)) \rightarrow \mathcal{P}(\text{Lit}_\Sigma) \) is defined as follows:

- \( RED(\emptyset) := \{\emptyset\} \)
- \( RED(C) := \{ c \mid \text{f.a. } c_i \in C_i, \exists c_i \in C_i : c_i \subseteq c \text{ and } c \text{ minimal } \} \)

The \( REDUCE \) operator, defined next, can be applied to the whole (completed) trace to extract all potential causes concerning all the violated constraints.

**Definition 5.2 (REDUCE)** The operator \( REDUCE \) is defined for a trace node \( N \) as follows:

- If \( N \) is a leaf node of :LOOK-UP with fully instantiated goal \( b \) which failed with respect to the aim then define the \( REDUCE \) operator depending on the aim. If the aim is to fail then \( REDUCE(N) := \{ \{ b \} \} \) else if the aim is to succeed \( REDUCE(N) := \{ \{ \neg b \} \} \).
- If \( N \) is a non-leaf node with successor nodes \( N_1, \ldots, N_k \), and connector :OR then
  \[
  REDUCE(N) := RED(\{ REDUCE(N_1) \cup \ldots \cup REDUCE(N_k) \}).
  \]

- If \( N \) is a non-leaf node with successor nodes \( N_1, \ldots, N_k \), and connector :AND then \( N_1, \ldots, N_k \) then
  \[
  REDUCE(N) := RED(\{ N_1, \ldots, N_k \}).
  \]

To demonstrate this definition consider again our running example. We apply the \( REDUCE \) operator to the trace shown in figures 2 and 3. For the leaf nodes \( N_l \) (in fig. 3 labelled with a small triangle) ordered from left to right we have
our experiments the heuristic in deed showed very good performance. In fact, we have meanwhile extended the technique to the generation of test data bases, with highly satisfactory results. In [29] a method for generating consistent test data for a variable set of consistency constraints, which is mainly based on this heuristic is presented.

5 Extraction of Potential Causes

Whereas a symptom is a set of literals which shows the deficiency of $C(DB)$ a cause is a set of literals showing the deficiencies of $DB^a$, i.e., the set of facts. It can easily be seen from the definitions of symptom and cause that in order to derive all potential causes it is sufficient to try to complete all symptoms to causes, i.e., for each derived symptom one looks for a set of literals by which the fact base must be modified in order to derive a consistent completion. (For an example see section 6.) More formally, there exists a potential symptom $S$ for each potential cause $C$ such that $C(DB) \circ_S C = C(DB \circ_C C)$, and vice versa (this is not true for definite symptoms and causes). Thus the task of finding all potential causes could start with the already detected potential symptoms.

We note, though, that contrary to our main idea in section 4.1 we cannot simply stop with a :ATOM node when developing the trace, because it might have been called for a derived predicate. Hence, more of the trace is needed by the time we look for potential causes. As a consequence, we introduce a general method that computes all potential causes for any kind of symptom for all violated consistency constraints at once without explicitly generating the symptoms first. This is done by introducing a special operator which directly works on the trace. Besides efficiency, the main reason for this approach is that by working on several violated consistency constraints all at once, the operator can deal with situations in which the simple union of their causes is not minimal.

If the symptom is a single positive literal this means that this literal is contained in the completion of the database and is additionally responsible for the consistency violation. Thus, this literal has to be deleted from the completion in order to regain consistency. There are two possibilities for a positive literal to be contained in the completion: First, if it is in the fact part of the database, or second if it is deducible by at least one rule. Hence, we have to remove the literal from the fact base, or have to remove at least one literal of each premise for each rule by which the literal is deducible. This can be done recursively by looking at the premise to be deleted as a positive symptom. Since causes are defined to be minimal, problems arise in the presence of redundant derivations. In general, there are two kinds of redundancy possible. First, there may exist different rules to derive the same fact, as demonstrated by the following example. Let $c$ be a symptom and the two rules under consideration are $a, b \implies c$, and $a, d \implies c$. If no attention is paid to this kind of redundancy non-minimal 'causes' are derived as follows. By choosing $\{a\}$ to be deleted in order to disable the applicability of the first rule, by choosing $\{d\}$ for the second rule, and finally by taking the union of those singletons, one comes up with the 'cause' $\{a\} \cup \{d\} = \{a, d\}$. Obviously, deleting $\{a\}$ from the fact base would be enough to disable the applicability of both rules. Thus, the derived 'cause' is not minimal. The second kind of redundancy occurs if it is possible to derive the same fact several times by the same rule using different instantiations. For example having the derived facts $p(a, b)$ and $p(a, c)$, the rule $p(x, y) \implies q(x)$ can be used to derive the fact $q(a)$ by using either of the facts. Now, assume the existence of the rule $r_1(x), r_2(y) \implies p(x, y)$, and the facts $r_1(a), r_2(b)$, and $r_2(c)$. In order to disable the derivation of $q(a)$ we have to disable the derivation of $p(a, b)$, and the one of $p(a, c)$. To delete $p(a, b)$ from the completion we can either delete $\{r_1(a)\}$, or $\{r_2(b)\}$, and to delete $p(a, c)$ we can either delete $\{r_1(a)\}$, or $\{r_2(c)\}$. By taking the union of each possible combination yields the following four sets:

17
Obviously, adding \( \text{dom}(\text{loc}, c) \) for any of these constants restores consistency concerning the new constraint. Note, however, that only adding \( \text{dom}(\text{loc}, \text{mob.obj}) \) to the fact base regains overall consistency, hence the heuristic just provides a set of constants.

We are now ready to state the heuristic in general terms. If there is a violated consistency constraint for which no instantiation for some existentially quantified variable \( z \) can be found we consider the restriction literals \( r_1, \ldots, r_h \) for this variable. Among the other consistency constraints with a matrix of the form \( a_1, \ldots, a_n \Rightarrow a_{n+1}, \ldots, a_{n+m} \) we select those for which there exists the most general unifier of some of its premise literals with some \( r_i \). Let \( \sigma = \text{mgu}(r_i, a_j) \).

In a second step those literals in the premise which are not unifiable with \( r_i \) are collected and, after the application of \( \sigma \), queried to the database. This yields substitutions \( \tau_i \). Last, all literals \( a_{n+j} \tau_i \) which contain \( z \sigma \) are treated as new restriction literals.

Suppose that constants \( c_1, \ldots, c_l \) for the (new) restriction literals \( r_1, \ldots, r_h \) have been derived. We rank these in decreasing order of the number of restriction literals which they satisfy. The ranking may be used in various ways, for example by considering only the first \( n \) constants, or by presenting the order to a user to let him/her choose, by dialog, his/her favorite among the constants.

Applying this heuristic in our example yields the following order on the derived constants

1. \( \text{mob.obj} \)
2. \( \text{automobile}, \text{car}, \text{vehicle} \)

where the first constant satisfies both new restriction literals whereas the last three satisfy only one of them. Note, that the first one also happens to be the one which restores overall consistency.

Returning to the trace node of the :EX method, if this method fails we add the trace nodes for \( f'[x - c] \) for each constant we find by the heuristic explained above. Figure 5 shows the extended trace of figure 4. Here, the four constants give rise to the corresponding subnodes of the :EX method.

It is in the nature of heuristics that they cannot be judged in terms of formal criteria like correctness and completeness. Instead, they should be evaluated on the basis of experience. In
\begin{align*}
\forall x \exists z \; is(x, at) \implies dom(x, z) \\
\therefore succeeds \quad : \text{AND} \\
\exists z \; is(loc, at) \implies dom(loc, z)
\end{align*}

Figure 4: Simplified trace for failing :EX

\[ TA := del(isa(automobile, vehicle)) \; add(isa(automobile, mob_obj)) \]

the consistency is violated. As we saw, one potential symptom is \( \{ dom(loc, vehicle) \} \). Following definition 2.6, consistency is recovered by removing this symptom. Unfortunately, \( dom(loc, vehicle) \) is no longer a definite symptom. Suppose now that nonetheless we extend the above transaction by \( del(dom(loc, vehicle)) \) in order to study the effect of a violated existentially quantified constraint. In fact, after the execution of this modified transaction the new constraint is violated since there is no fact \( dom(loc, z) \) for some \( z \) contained in the fact base. Further, \( dom(x, z) \) is the only restriction literal for \( z \) in the violated constraint. Thus, we cannot derive any constant as a possible candidate for instantiating the matrix of the new constraint.

The trace (see fig. 4) consists of two nodes. The first node corresponds to the goal of the :ALL method, the second to that of the :EX method. The attribute list of the latter contains the entry :no_instantiations_found (not in the figure). There are no further subnodes from which symptoms could be extracted. The problem, then, is to derive constants which could serve as candidates for instantiations so that the trace can be completed accordingly. Of course, it does not make much sense to really consider all constants to generate symptoms. To give an example, it would be quite awkward to treat \( bike \) as a possible candidate, and insert \( dom(loc, bike) \) into the fact base. Even worse are constants which are not present in the database. To overcome this difficulty we will present a powerful heuristic to derive meaningful instantiations.

### 4.3 A Heuristic Solution

The general idea is to derive by forward chaining within the constraints new restriction literals. This seems reasonable since one would like to derive instantiations which are plausible in the context of the given consistency constraints.

Our original constraint

\[ \forall x_1 \forall x_2 \forall z \; loc(x_1, x_2) \land dom(loc, z) \implies is^*(x_1, z) \]

has a premise containing a literal (trivially unifiable with) \( dom(loc, z) \) which in turn is our restriction literal derived from the new constraint. Hence, we cannot simply recover from the violation by adding some fact \( dom(loc, c) \) to the database. If we did, there is the danger of violating the original constraint since the premise may become true for those constants \( c \) for which already \( loc(x_1, x_2) \) holds. Here, there exist two such "dangerous" facts in our fact base, namely, \( loc(bmw, bmwpos) \), and \( loc(bike, bikepos) \). Thus, for the variable \( x_1 \) the substitutions \( bmuw \) and \( bike \) have to be considered.

The conclusion of the original constraint is \( is^*(x_1, z) \). Avoiding another violation requires that for each constant \( c \) for which \( dom(loc, c) \) might be added to the fact base, \( is^*(bmw, c) \) and \( is^*(bike, c) \) must hold. We call \( is^*(bmw, z) \) and \( is^*(bike, z) \) the new restriction literals. Querying them individually against the database yields \( c \in \{ car, automobile, vehicle, mob_obj \} \).
We immediately recognize that each of its constituents corresponds to one potential symptom. We just mention the formal justification for this derivation procedure: for a universally quantified constraint determine the restriction literals, instantiate the matrix and transform it into disjunctive normal form. Then, each conjunction within the form corresponds to one potential symptom, and all potential symptoms can be extracted this way.

4.2 The Problem of Existential Quantification

Unfortunately, for existentially quantified constraints the procedure is less straightforward. For an :EX method which failed due to the lack of possible constants to be used for instantiation the trace is incomplete and must be completed by the following heuristic before extracting the potential symptoms. The method to detect violations for existentially quantified formulas tries to find an instantiation for the subformula s.t. it evaluates to true. More formally, if the :EX method is applied to a formula of the form $f = \exists x f'$ where $f'$ is a formula with $x$ being the only free variable it tries to find a constant $c$ s.t. $DB \models f[x \leftarrow c]$. In order to do so the method inspects the restriction literals $r_1, \ldots, r_k$ for $x$ to yield the constants $c$ s.t. $DB \models r_i[x \leftarrow c]$ for all $1 \leq i \leq k$. All these constants are then evaluated on $f'$. If one of the constants succeeds we are already done and $f$ evaluates to true.

Suppose there exists no constant $c$ s.t. $DB \models r_i[x \leftarrow c]$ for all $i$. Of course, we then know that $f$ evaluates to false but this information is not sufficient for producing all potential symptoms for $f$. But what, then, are the potential symptoms? We know of no restrictions at this point; each constant in our database is a potential candidate. What we need, therefore, is a heuristic that reduces this set of candidates.

To illustrate the problem, consider again our example. Since so far it does not contain an existentially quantified constraint we augment it by adding a fact stating that $loc$ is an attribute ($\text{is}(\text{loc}, \text{at})$), and a constraint requiring the existence of a domain for every attribute ($\forall x \exists z \text{is}(x, z) \rightarrow \text{dom}(x, z)$). Further, we add a second individual, a bike ($\text{is}(\text{bike, vehicle})$), with location $\text{bikepos}$.

$$DB = \{\text{is(car, automobile),}$$
$$\text{is(automobile, vehicle),}$$
$$\text{is(vehicle, mob_obj),}$$
$$\text{is(loc, at),}$$
$$\text{dom(loc, vehicle),}$$
$$\text{ram(loc, coord),}$$
$$\text{is(bmw, automobile),}$$
$$\text{is(bmwpos, coord),}$$
$$\text{is(bike, vehicle),}$$
$$\text{is(bikepos, coord),}$$
$$\text{loc(bmw, bmwpos),}$$
$$\text{loc(bike, bikepos)}\},$$

$$\{\forall x \forall y \text{is}(x, y) \rightarrow \text{isa}^*(x, y),$$
$$\forall x \forall y \forall z \text{is}(x, y), \text{isa}^*(y, z) \rightarrow \text{isa}^*(x, z),$$
$$\forall x \forall y \text{is}(x, y) \rightarrow \text{isa}^*(x, y),$$
$$\forall x \forall y \forall z \text{is}(x, y), \text{isa}^*(y, z) \rightarrow \text{isa}^*(x, z)\},$$

$$\{\forall x \exists z \text{is}(x, at) \rightarrow \text{dom}(x, z),$$
$$\forall x \forall x_2 \forall z \text{loc}(x_1, x_2) \wedge \text{dom}(\text{loc}, z) \rightarrow \text{isa}^*(x_1, z)\}$$

After the execution of our earlier transaction

14
Figure 3: Simplified Trace
Take instead the second rule application. Here, the :ATOM method is called with the atom \( is^*(bmw, Z) \) which is not fully instantiated. Using the :LOOK-UP method one finds the instantiation \( car \). This in turn gives rise to two subnodes for which the :LOOK-UP method (left subnode \( is(bmw, car) \)) and the :APPLY-RULE method (right subnode \( isa^*(car, vehicle) \)) are called, and so forth until the call of the :APPLY-RULE method for the goal \( isa(mob.obj, vehicle) \). The trace terminates at this point because for neither of the body literals \( isa(mob.obj, W) \), and \( isa^*(W, \text{vehicle}) \) an instantiation can be found.

4 Deducing Potential Symptoms

This section is devoted to the deduction of potential symptoms for the violated consistency constraints. More accurately its concern is to extract potential symptoms from the trace as introduced in the last section.

4.1 Main Idea

Intuitively speaking, the symptoms are obtained by examining the failing instantiation of the matrix of the violated consistency constraint as it appears in the trace. Let us consider our example again, and look at the evolving trace as depicted in figure 2. The violated consistency constraint is

\[
\forall x_1 \forall x_2 \forall z \text{ loc}(x_1, x_2) \land \text{dom}(\text{loc}, z) \implies is^*(x_1, z).
\]

The first step in deducing the potential symptoms for the consistency violation is to get the violated instantiation of the matrix. If we apply the :ALL method as described in section 3.1 we obtain

\[
\text{loc}(bmw, bmwpos) \land \text{dom}(\text{loc}, \text{vehicle}) \implies is^*(bmw, vehicle).
\]

The next application is by the implication method. In turn, the implication method calls the :ATOM method for all the literals in the premise and the literal in the conclusion with the associated aims :fail, or :succeed, respectively. Now, according to definitions 2.5 and 2.6 the goals of these three :ATOM methods could qualify as symptoms because all three are literals, and constitute a minimal set of these. We thus conclude that symptoms are derived by developing the trace up to the point where a :ATOM method is to be applied (provided we knew the outcome of that method). For our example this yields the trace as shown in figure 2.

To be more specific, the derivation takes place as follows. Starting at the root of the trace we collect the information given in the trace in the following manner. We connect the subnodes of each node with the connector given, and introduce negation if the aim is :succeed. When arriving at an :ATOM node we just write down its goal. For the trace (fig. 2) this results in:

\[
(:\text{AND} (:\text{OR} \quad \text{loc}(bmw, bmwpos) \\
\text{dom}(\text{loc}, \text{vehicle}) \\
\neg is^*(bmw, \text{vehicle})).)
\]

Simplification of this expression yields

\[
(:\text{OR} \quad \text{loc}(bmw, bmwpos) \\
\text{dom}(\text{loc}, \text{vehicle}) \\
\neg is^*(bmw, \text{vehicle})).
\]
\[
\forall x_1 \forall x_2 \forall z \\
loc(x_1, x_2) \land \text{dom}(loc, z) \implies is^*(x_1, z)
\]

:AND

\[
loc(bmw, bmwpos) \land \text{dom}(loc, vehicle) \implies is^*(bmw, vehicle)
\]

:OR

\[
loc(bmw, bmwpos) \quad \text{dom}(loc, vehicle) \quad is^*(bmw, vehicle)
\]

... 

Figure 2: Simplified Trace

If the application of the :ALL method fails for some instantiations we will find a list of these in the attribute list. The attribute list is particularly important in the case of the :EX method. This method must find an instantiation such that the instantiated subformula evaluates to true. In principle one constant is as good as another. Remember that the method restricts the instantiations to those which are deducible from all the restriction literals. Since any constant is a priori a useful candidate, the trace tree at this stage is incomplete in that it does not contain all possible constants. As we will see later, it might be worthwhile to check on each single restriction literal to find more possible instantiations for the 3-quantified variable. To avoid recomputing the restriction literals and the checking of constants already checked both are kept in the attribute entry of the trace node. The derivation of more constants to get a broader spectrum of symptoms is discussed in the section 4.

To give an example for a trace consider again the database given in our example. The simplified trace for this example is depicted in figure 2. The top node stems from the application of the :CHECK-CONSTRAINT method to the (violated) consistency constraint \(\forall x_1 \forall x_2 \forall z loc(x_1, x_2) \land \text{dom}(loc, z) \implies is^*(x_1, z)\). There is only one instantiation of this constraint because there is only one vehicle with a defined location, and consequently there is only one subnode. This node represents the application of the implication method. In turn, it results in calling the :ATOM method for each premise and the conclusion.

The application of the two premises succeeds. Hence, we must follow the rightmost branch \((is^*(bmw, vehicle))\). The details are shown in figure 3. The applications of the :ATOM method are drawn as ellipses. The rectangle representing a :APPLY-RULE method only shows the body of the applied rule. The connectors are shown at the right margin of the figure. We examine how this trace evolves.

We start with the goal \(is^*(bmw, vehicle)\). There are two ways to derive it. First, we could try to call the :LOOK-UP method but this does not make much sense since \(is^*\) is a derived predicate. This attempt is not depicted in the figure. Second, we can apply the :APPLY-RULE method. Since there are two candidate rules, namely, \(is(X, Y) \implies is^*(X, Y)\), and \(is(X, Y), isa^*(Y, Z) \implies is^*(X, Z)\) there exist two subnodes for our goal. Each is labeled with the body of the applied rule. The rule nodes in turn call the :ATOM method for each literal in the body of the rule. It fails for the goal \(is(bmw, vehicle)\). (The further subnodes of :ATOM calls for base predicates have been omitted.)
and the method :LOOK-UP also fails we know that a is not deducible from the facts in the database, otherwise if at least one rule or the :LOOK-UP method succeeds the fact is deducible. To simplify the check, a method :ATOM integrates both methods, first calling :LOOK-UP, and if necessary then calling :APPLY-RULE for each rule applicable to the fact to be derived. Since the deduction process may loop forever in the case of recursive rules there must be a halting criterion. Thus the :ATOM method checks whether there has been the same atom tested before.

**Further Methods** For reasons of uniform treatment we introduce two further methods. These are :CHECK-OVERALL-CONSISTENCY which calls the second method :CHECK-CONSTRAINT for each consistency constraint contained in the database. Since a database is consistent only if all consistency constraints hold, the connector of the :CHECK-OVERALL-CONSISTENCY method is set to :AND.

### 3.2 The Trace

Checking consistency starts with applying :CHECK-OVERALL-CONSISTENCY (or :CHECK-CONSTRAINT in trivial cases such as our example), which gives rise to calling the other methods in a recursive fashion. A trace that records all these calls necessarily takes the form of a tree, where each node corresponds to a method application and the subnodes correspond to methods called. However, not every method application has a bearing on the consistency violation. Just consider the simple example where the method :CHECK-CONSTRAINT is applied to a consistency constraint which is not violated. In order to determine symptoms it clearly suffices to include in a trace only those parts of the deduction process which detect the consistency violation. Each method thus must know when to add a trace node. In turn this depends on the context in which the method is called. Consider implication as an example. In order for an implication to evaluate to true the premise must fail or the conclusion must evaluate to true. Thus if an atom a occurs in the premise the method :ATOM must fail, if it occurs in the conclusion the method :ATOM must succeed. A trace node must be generated in the inverse situations. To describe the context we include with each method a parameter "aim" whose value is either :fail or :succeed. As a second example, consider the search for applicable rules in the method :ATOM. If the aim is to succeed one of the rules must succeed. If the aim is to fail all rules must fail. Thus in the first case the rules can be thought of as logically connected by "or" and in the second case by "and".

All this information is collected into a single trace node represented by the following record structure:

```plaintext
defstruct(tracenode
    aim : {fail, succeed}
    goal : FOR
    method : {CHECK-OVERALL-CONSISTENCY, :CHECK-CONSTRAINT,
             :ALL, :EX, :IMPL, :AND, :OR, :NOT,
             :ATOM, :LOOK-UP, :APPLY-RULE}
    attributes : list of attributes
    connector : {AND, OR}
    subnodes : list of trace nodes)
```

goal refers to the formula to be evaluated, aim specifies whether this formula should evaluate to true or false, method determines the method to be applied, subnodes points to the successor nodes, and connector establishes how the result is determined from the truth values for the subgoals. In the attribute entry we gather information concerning the result of the method application. For example, we will find there a value :not-found if the method :LOOK-UP failed.
Negation merits a further comment. The simple truth table of negation looks as follows:

<table>
<thead>
<tr>
<th>( \neg a )</th>
<th>( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( t )</td>
</tr>
<tr>
<td>( t )</td>
<td>( f )</td>
</tr>
</tbody>
</table>

The negation is then implemented as testing for each closed literal \( \neg a \) whether \( DB \models a \) or not. If \( DB \models a \) then true is returned otherwise \( (DB \not\models a) \) false is returned. This kind of procedural treatment of negation is usually called "negation-as-failure" ([8]). In the case where the rules are restricted to Horn clauses this procedural interpretation of negation fits perfectly to the closed world assumption assumed here (see for example [38]).

The above methods will be referred to by :IMPL and :NOT, respectively. Methods for :AND and :OR are defined in a comparable way.

**Methods for Quantifiers** The treatment of quantifiers makes heavy use of the restriction literals that serve to derive the instantiations for the \( \forall \)-quantified variables to be tested during the validity check of the consistency constraint under consideration. The main idea is illustrated by using again the formula \( \forall x \forall y \, p(x, y) \Rightarrow q(x) \). Remember that \( p(x, y) \) is the restriction literal for \( x \) and \( y \). Let us think of this formula as a consistency constraint which has to be verified. Obviously we can conclude \( DB \models \forall x \forall y \, p(x, y) \Rightarrow q(x) \) if there is no substitution \( a \) for \( x \) and \( b \) for \( y \) such that \( DB \models p(a, b) \) since then the premise of the formula cannot become true. Otherwise, if for all possible instantiations \( a \) for \( x \) and \( b \) for \( y \) of \( p(x, y) \) which are deducible from \( DB \) \( q(a) \) holds as well the constraint also evaluates to true.

Following these lines we define a general method :ALL which is responsible for deducing the instantiations to be tested for a given \( \forall \)-quantified constraint. It works as follows. For a group of successive \( \forall \)-quantifiers the variables are collected. In a second step the restriction literals for all these variables are collected. Let’s say these are \( r_1, \ldots, r_n \). A third step then requires the deduction of all answers to the query \( r_1, \ldots, r_n \). These can be derived by means of an ordinary query answering mechanism. The last step then checks for each of the answers the accordingly instantiated partial constraint where the considered \( \forall \)-quantifiers have been removed. If all these instantiations then evaluate to true the result of the :ALL method applied to the constraint is also true. Otherwise if at least one instantiation evaluates to false the overall result is false.

The :EX method works for \( \exists \)-quantifiers in an analogous fashion. Just note that the restriction literals for \( \exists \)-quantified variables are defined differently from those for \( \forall \)-quantified variables, and that the :EX method returns true if at least one instantiation of the considered constraint evaluates to true.

A side effect of range restrictedness for consistency constraints is the following. After the treatment of all quantifiers of the constraint the matrix is fully instantiated, i.e., does not contain any variables. The connectors can then be resolved by the appropriate connector methods.

**Methods for Atoms** Because of the full instantiation just mentioned, the check of atoms reduces to a check of facts, i.e., to decide whether \( DB \models a \) for a fact \( a \). Consider a fact \( p(a, b) \). Of course the validity can be checked by a standard query processing algorithm. But since we are interested in a trace of the deduction process we introduce the following methods which taken together are similar to SLD resolution (see e.g. [21]). In order to check whether some fact \( a \) holds we should first check whether \( a \) is contained in the fact base (this method is called :LOOK-UP); if so we are done. If not we are forced to search for a rule \( l_1, \ldots, l_n \Rightarrow l_{n+1} \) such that the head of the rule unifies with \( a \). For this rule we then have to check the body after the most general unifier of \( a \) and the head of the rule has been applied to it. If we can deduce the body for some ground substitution \( \sigma \) applied the application of the rule succeeds and \( a \) is deduced. If the application of this method (:APPLY-RULE) fails for each possible rule,
\{\text{dom}(\text{loc, vehicle})\}, and \{\neg \text{is}^*(\text{bmw, vehicle})\} are symptoms since if the database would not contain either \text{loc}(\text{bmw, bmwpos}) or \text{dom}(\text{loc, vehicle}), or if the fact \text{is}^*(\text{bmw, vehicle}) were deducible from the fact base, i.e., be in the completion, then the consistency constraint would not be violated. Since \text{loc}(\text{bmw, bmwpos}) and \text{dom}(\text{loc, vehicle}) can be deleted from the extension by simply removing them from the fact base these symptoms are also causes. On the other hand, since \text{is}^* is a derived predicate it would not make much sense to add \text{is}^*(\text{bmw, vehicle}) to the fact base in order to restore consistency. The solution is to add facts concerning base predicates to the fact base such that \text{is}^*(\text{bmw, vehicle}) becomes derivable. Possible facts are: \text{isa(automobile, vehicle)}, or \text{is(bmw, vehicle)}. The causes thus generated from the symptom \{\neg \text{is}^*(\text{bmw, vehicle})\} are
\{\neg \text{isa(automobile, vehicle)}\}, and \{\neg \text{is(bmw, vehicle)}\}.

For the rest of the paper we are concerned with the detection, first, of potential symptoms, then causes, and finally definite causes. These topics could be treated in a rigorous fashion, with an appropriate theory and systematically developed algorithms. The corresponding formalism is the subject of [27] and, in shorter form, of a companion paper ([30]). The objective of the present paper is to give an intuitive understanding of the solution. Since the presented method highly relies on the trace of the proof of inconsistency this will be the subject of the next section.

3 Tracing the Consistency Check

At transaction commit the consistency constraints are checked. As we noticed, this may already involve inference processes: \text{is}^* is a derived predicate, i.e., atoms of the form \text{is}^*(x_1, d) cannot directly be retrieved from \text{DB}. Once a constraint has been invalidated, all possible (potential) symptoms must be derived from it (these, in turn, will give rise to the causes). The question we approach here is the following: Does the inference process during the consistency check yield enough information to imply the symptoms? Before we can answer this question we discuss how such information can be collected into a trace.

3.1 The Consistency Check

In order to demonstrate the principles involved in creating the trace we adopt at this point a rather straightforward, easy to understand method for consistency checking. Since it is a highly inefficient method, we return to the issue of efficient checking in section 8. It presumes that for each logical operator there exists one procedure (or method) which checks the validity of an expression with that operator as the outer connector. Additionally, there is a method which derives the truth value of a queried fact.

Methods for Boolean Connectors One can think of each method as a realization of the truth table of the connector implemented. Consider, for example, the implication. Its truth table is:

\begin{center}
\begin{tabular}{ccc}
a & \Rightarrow & b \\
t & t & t \\
f & t & f \\
t & f & t \\
t & f & f \\
\end{tabular}
\end{center}

Given a matrix \(f' \Rightarrow f''\) the method first checks \(f'\) for validity by calling the appropriate method. If \(f'\) then evaluates to false the whole formula evaluates to true. If \(f'\) evaluates to true then the truth value of the whole formula is that of \(f''\) which, therefore, must now be evaluated. The other connectors are implemented in an analogous fashion according to their truth table.
Figure 1: Illustration of the fact base

\[
\text{loc}(\text{bmw}, \text{bmwpos}),
\{\text{isa}(x, y) \Rightarrow \text{isa}^*(x, y),
\text{isa}(x, y), \text{isa}(y, z) \Rightarrow \text{isa}^*(x, z),
\forall x \forall y \text{isa}(x, y) \Rightarrow \text{is}^*(x, y),
\forall x \forall y \forall z \text{isa}^*(y, z) \Rightarrow \text{is}^*(x, z),
\forall x_1 \forall x_2 \forall z \text{loc}(x_1, x_2), \text{dom}(\text{loc}, z) \Rightarrow \text{is}^*(x_1, z)\}
\]

Figure 1 a) represents most of the fact base in the following sense. The facts concerning the \textit{isa} predicate are illustrated by arrows labeled with \textit{isa} from the first argument to the second argument of the represented facts. The facts for the \textit{is} predicate are represented the same way. The domain of the \textit{loc} attribute (\textit{dom}(\text{loc}, \text{vehicle})) and the range of it (\textit{ran}(\text{loc}, \text{coord})) is depicted by an arrow labeled by \textit{loc} from \textit{vehicle} to \textit{coord}. The consistency constraint requires that if some object has a location, the object must directly or indirectly be in the domain of \textit{loc}. Using the graphical representation, the constraint can be interpreted as the requirement for a path from the individuals like \textit{bmw} to the domain \textit{vehicle}. According to the rules it must start with an arrow labeled \textit{is} succeeded by arrows labeled \textit{isa}. Of course this must only be true for individuals which do have a defined location in the database. This is true for \textit{bmw} which has the location \textit{bmwpos}. (This fact \{\textit{loc}(\text{bmw}, \text{bmwpos})\} contained in the database is not represented in the figure.)

For the remainder of the paper we consider the transaction

\[
T_A := \text{del}(\text{isa}(\text{automobile, vehicle})) \text{ add}(\text{isa}(\text{automobile, mob_obj})).
\]

The state of \(DB^a\) after the execution of \(T_A\) is depicted in figure 1 b. Obviously, the consistency constraint is violated since

\[
\text{loc}(\text{bmw}, \text{bmwpos}), \text{dom}(\text{loc}, \text{vehicle}) \Rightarrow \text{is}^*(\text{bmw, vehicle})
\]

does not hold any longer. Since \(DB^c\) contains just a single consistency constraint we do not have to distinguish between potential and definite symptoms and causes. Thus every potential symptom (cause) is a definite symptom (cause). It is further obvious that \{\textit{loc}(\text{bmw, bmwpos})\},
Definition 2.7 (cause) Let $C \subseteq \text{Lit}_{\Sigma}$, $DB = (DB^a, DB^d, DB^c)$ be a database, and $c \in DB^c$. If $C(DB) \cup \{c\}$ is inconsistent then $C$ is called

1. (potential) cause : $\Leftrightarrow$ $C(DB \circ_C C) \cup \{c\}$ consistent and $C$ minimal
2. definite cause : $\Leftrightarrow$ $C(DB \circ_C C) \cup DB^c$ consistent and $C$ minimal

Hence, a cause is a set of literals that must be corrected by applying $\circ_C$ in order to regain consistency.

Clearly, one of our subsequent objectives must be to establish a connection between symptoms and causes. Ultimately we wish to provide the user with a transaction, i.e., a sequence of operations which resolve the inconsistency. This kind of transaction will be called repair.

Definition 2.8 (repair) Let $TA$ be a transaction, $DB = (DB^a, DB^d, DB^c)$ a database and $c \in DB^c$ a consistency constraint. If $C(DB) \cup \{c\}$ is inconsistent then $TA$ is called

1. (potential) repair : $\Leftrightarrow$ $C(TA(DB)) \cup \{c\}$ consistent and $TA$ minimal
2. definite repair : $\Leftrightarrow$ $C(TA(DB)) \cup DB^c$ consistent and $TA$ minimal

Obviously, a transaction $TA$ is a repair iff $\alpha(TA)$ is a cause, e.g., if $\{p(a) , \neg q(b)\}$ is a repair then, taking into account the definition of $\circ_C$, $del(p(a)) , add(q(b))$ clearly is a cause, and vice versa. Thus, the paper will concentrate on the steps of extracting symptoms from the inconsistency, i.e. violated consistency constraints, and of generating the causes from the symptoms.

As can easily be seen, the following implications hold:

1. $S$ definite symptom $\supset$ ex. $S'$ potential symptom: $S' \subseteq S$
2. $C$ definite cause $\supset$ ex. $C'$ potential cause: $C' \subseteq C$
3. $R$ definite repair $\supset$ ex. $R'$ potential repair: $R' \subseteq R$

Less obvious may be the falsity of the following implications:

1. $S$ potential symptom $\not\subset$ ex. $S'$ definite symptom: $S \not\subseteq S'$
2. $C$ potential cause $\not\subset$ ex. $C'$ definite cause: $C \not\subseteq C'$
3. $R$ potential repair $\not\subset$ ex. $R'$ definite repair: $R \not\subseteq R'$

In other words, for a given potential symptom (cause, repair) there may not exist a definite symptom (cause, repair), i.e., the symptom may not be extended to other constraints.

In order to illustrate the above definitions of symptom, cause, and repair we introduce the following example which will serve us throughout the rest of the paper. Consider the following database. To simplify the following presentations we initially restrict ourselves to a single consistency constraint. Examples with several constraints will follow later on.

Example 2.9

$DB = \{\{\text{isa(car, automobile)} ,$
$\text{isa(automobile, vehicle)} ,$
$\text{isa(vehicle, mob_obj)} ,$
$\text{dom(loc, vehicle)} ,$
$\text{ran(loc, coord)} ,$
$\text{is(bmw, automobile)} ,$
$\text{is(bmwpos, coord)} ,$
$\}$
We will use a very simple notion of a transaction. For $a_i$ being facts a **transaction** is defined as a finite sequence $T A := op_1(a_1), \ldots, op_n(a_n)$ where $op_k \in \{\text{add}, \text{del}\}$. Further, a mapping $\alpha$ from the set of all transactions to a subset of all closed literals is defined as $\alpha(T A) := \{a|\text{del}(a) \in T A\} \cup \{-a|\text{add}(a) \in T A\}$. Only transactions with $\alpha(T A)$ consistent will be considered. Thus we do not allow one transaction to add and delete the same fact.

Now, the main definitions for symptom, cause, and repair follow. For each of these definitions two different cases are considered. First, the notions are defined for the case of a single consistency constraint, and subsequently for the case of the entire $DB^c$. The former will be referred to as potential symptoms (causes, repairs), the latter as definite symptoms (causes, repairs). For the definitions two mappings are needed. The first one ($\circ_S$) will be used to modify the completion $C(DB)$ of a database whereas the second is used to denote modifications to the fact base $DB^c$.

**Definition 2.5** If $A, B \subseteq \text{Lit}_\Sigma$ then $\circ_S, \circ_C : \mathcal{P}(\text{Lit}_\Sigma) \times \mathcal{P}(\text{Lit}_\Sigma) \rightarrow \mathcal{P}(\text{Lit}_\Sigma)$ are defined as follows:

1. $A \circ_S B := (A \setminus B) \cup \overline{B}$
2. $A \circ_C B := (A \setminus \{b|b \in B, b \in At_\Sigma\}) \cup \{b|\neg b \in B, b \in At_\Sigma\}$

where $\overline{B}$ for a set of literals is defined as $\overline{B} := \{a|\neg a \in B, a$ is an atom$\} \cup \{-a|a \in B, a$ is an atom$\}$, and $\mathcal{P}$ denotes the powerset.

Let $a, b, c, d, e, \ldots$ be facts. Consider the following database $DB = (\{a, b\}, \{a \Rightarrow c\}, \emptyset)$. Then $C(DB) = \{a, b, c, \neg d, \neg e, \ldots\}$. If we now wish the extension of $DB$ to exclude $a$ but include $d$ this can be denoted by $C(DB) \circ_S \{a, \neg d\}$ which results in $\{\neg a, b, c, d, \neg e, \ldots\}$. That $d$ is added to the fact base and $a$ deleted from it can be denoted by $DB^c \circ_C \{a, \neg d\}$. The completion of the thus modified database is slightly different, namely $\{\neg a, b, \neg c, d, \neg e, \ldots\}$.

The example nicely demonstrates that the completion of a database can be seen as the contents of the database as it appears to the user. Hence, it makes sense from a pragmatic standpoint as well to use the completion to define the consistency of the database. Consequently, the analysis of a consistency violation is first performed at the level of the completion. This yields symptoms which are allowed to be derived facts. In a second step causes are derived from these symptoms, which directly concern $DB^c$.

One problem arises if several consistency constraints are present. It is quite possible that the "repair" of one consistency constraint may violate another. Therefore we distinguish between a potential symptom which is concerned with a single consistency constraint, and a definite symptom which takes into consideration the entire set of consistency constraints.

**Definition 2.6 (symptom)** For a database $DB = (DB^a, DB^d, DB^c)$, and $c \in DB^c$ with $C(DB) \cup \{c\}$ inconsistent, a subset $S \subseteq \text{Lit}_\Sigma$ of facts is called

1. **(potential) symptom** : $\leftarrow (C(DB) \circ_S S) \cup \{c\}$ consistent and $S$ minimal
2. **definite symptom** : $\leftarrow (C(DB) \circ_S S) \cup DB^c$ consistent and $S$ minimal

In other words, a symptom is a set of literals that must be corrected by applying $\circ_S$ in order to restore consistency.

Having detected the symptoms does not by itself help to alleviate the consistency violation, since the facts in the completion need not necessarily be really present in the database. This is especially true for the negated facts. On the other hand, a repair can certainly be applied only to the set $DB^a$ of stored facts. Hence, we need a notion that relates repairs to such facts. This notion, referred to as cause, should be differentiated in a way similar to symptoms.
all formulas is denoted by \( \text{For}_\Sigma \). If \( f \) is a formula \( m(f) \) is defined to be the matrix of the formula. A formula is defined to be in conjunctive (disjunctive) normal form iff the matrix is 
\[
(\ell_{1,1} \lor \ldots \lor \ell_{1,n_1}) \land \ldots \land (\ell_{n,1} \lor \ldots \lor \ell_{n,m_n}) \land (\ell_{1,1} \land \ldots \land \ell_{1,m_1}) \lor \ldots \lor (\ell_{n,1} \land \ldots \land \ell_{n,m_n})
\]
for some literals \( \ell_{i,j,k} \).

In order to prove the validity of a consistency constraint solely on the basis of the contents of the database we need the notion of domain independence. It was introduced in [11], and captures the intuition that the semantics of a database depends only on its contents and not on the underlying domain. We use it in the syntactic form of range restrictedness as introduced in [33]. Another characterization are the allowed formulas of [12]. We use the following definition of range restricted formulas.

**Definition 2.1 (range restrictedness)** A formula \( f \) is called range-restricted iff it satisfies the following conditions:

- For every \( \forall \)-quantified variable there exists at least one negative literal in each disjunction of the conjunctive normal form of the matrix of \( f \) in which it occurs. The negative literals in which the variable occurs are called restriction literals for this variable.

- For every \( \exists \)-quantified variable occurring in a negative literal in the conjunctive normal form of the matrix of \( f \) exists a disjunction containing only positive literals each containing the variable. Those positive literals are called the restriction literals for the variable.

Consider the example formula \( \forall x \forall y \ p(x,y) \implies q(x) \) which in conjunctive normal form is \( \forall x \forall y \ 
eg p(x,y) \lor q(x) \). It is range-restricted, and the restriction literal for \( x \) and \( y \) is \( \neg p(x,y) \). Contrarily, the formula \( \forall x \forall y q(x) \implies p(x,y) \) is not range-restricted since there there is no negative literal in \( \forall x \forall y \neg q(x) \lor p(x,y) \) in which \( y \) occurs. Replacing the \( \forall \) quantifiers by \( \exists \) quantifiers we have \( \exists x \exists y p(x,y) \implies q(x) \) is not range-restricted whereas \( \forall x \exists y q(x) \implies p(x,y) \) is. We note in passing that range-restrictedness includes the notion of typing which is being added to logic more and more frequently. Consequently, the results of this paper remain applicable in the presence of typing.

A database is defined to consist of facts, rules, and consistency constraints.

**Definition 2.2 (Database)** A database is a triple \( DB := (DB^a, DB^d, DB^c) \) where \( DB^a \) is a set of facts, \( DB^d \) is a set of rules, and \( DB^c \) is a set of closed range-restricted formulas called consistency constraints.

The next definition is concerned with the contents of a database. We assume that it consists of all facts which are deducible from the facts and rules in the database, and also of the negations of those facts which are not consequences of the facts and rules. The latter assumption corresponds to the closed world assumption ([36]).

**Definition 2.3 (Extension)** For a database \( DB := (DB^a, DB^d, DB^c) \) we define the extension \( M(DB) := \{ a | a \ is \ a \ fact, DB \models a \} \), and the complete extension or for short completion \( C(DB) := M(DB) \cup \{ \neg a | a \ is \ a \ fact, DB \not\models a \} \).

\( DB^a \cup DB^d \models f \) which is equivalent to \( C(DB) \models f \) for formulas \( f \) is abbreviated by \( DB \models f \). We are now ready to define the consistency of a database.

**Definition 2.4 (Consistency)** A database \( DB := (DB^a, DB^d, DB^c) \) is called consistent iff \( C(DB) \cup DB^c \) does not contain any contradiction, i.e., is consistent in the logical sense.
presented where the designer of the consistency constraints specifies for each constraint a set of repair actions. Once a consistency violation is detected the system automatically selects one of the repair actions for one of the violated constraints, performs it, and restarts the consistency check. Besides the necessity of specifying repair actions for each constraint the approach exhibits two further limitations: constraints are restricted to imperative form, and no treatment exists for violations in which existentially quantified variables are involved. Quite a different approach is presented in [39]. On the basis of a Boyer-Moore theorem prover the authors develop a tool to support transaction design. In order to do so they model the relational theory in terms of recursive functions which, together with certain lemmata, provide the basis for the deduction process within the prover. Using this knowledge the prover tries to prove without querying the database that a given transaction will not violate the consistency. If such a proof is not possible consistency checks are automatically generated from open subgoals and inserted into the transaction. A highlight of the approach is the automatic generation of postconditions for the user-specified transaction which reflect the user’s intent.

Our intentions are somewhat different, though, and to some extent go beyond those in [7]: to phrase it in a nutshell, generate repairs automatically for general constraints in deductive databases. In order to do so, the paper is organized as follows. In section 2 the basic notation and the main definitions are introduced, together with an example which will serve to illustrate the main ideas throughout the paper. The third section is concerned with the consistency check and the trace of inference steps developed during the check. The fourth section shows how to deduce the potential symptoms for an observed consistency violation. The extraction of potential causes underlying the symptoms is the subject of section 5. Not all of the potential causes qualify as definite causes, because the former are associated with the violation of a single constraint whereas the latter take the total of all violated constraints into account. Therefore, section 6 examines how one obtains the definite causes from the potential ones. Section 7 discusses what a comfortable user interface should look like and how flooding the user with information can be avoided. Section 8 describes in some detail the overall system architecture and the interplay of its components, and outlines the algorithmic solutions employed in the various components. First experiences of the performance observed in our first prototype will be related. The conclusions will discuss the needs for future work desirable to improve the acceptance of the presented technology by the end user. The paper will not go into formal completeness proofs or discussions of the correctness and computational complexity of the algorithmic solutions; these can be found in other publications by one of the authors ([27, 30]).

2 Symptoms, Causes, and Repairs

In order to give precision to the remaining discussion we establish a formal basis in the form of some basic definitions.

A signature $\Sigma$ is a triple $(K_{\Sigma}, P_{\Sigma}, \alpha_{\Sigma})$ where $K_{\Sigma}$ is a set of constants, $P_{\Sigma}$ is a set of predicates, and $\alpha_{\Sigma}$ is a mapping $\alpha_{\Sigma}: P_{\Sigma} \rightarrow \mathbb{N}$ called arity. Variables are denoted by $x, y, z, \ldots$, possibly indexed. A term is a variable symbol or a constant symbol (we restrict ourselves to function-free terms such as in DATALOG). The set of all terms is denoted by $T_{\Sigma}$. The set of atoms is defined as $At_{\Sigma} := \{ p(t_1 \ldots t_n) \mid t_i \in T_{\Sigma}, \alpha_{\Sigma}(p) = n, p \in P_{\Sigma} \}$. An atom not containing any variable is called a fact. A literal is defined as either an atom or, if $a$ is an atom then $\neg a$ is a literal. The set of all literals is denoted by $Lit_{\Sigma}$. If $l_1, \ldots, l_n, l_{n+1}$ are literals then $l_1, \ldots, l_n \implies l_{n+1}$ is a rule. A matrix is either a literal, or for $m_1, m_2$ matrices an expression of the form $\neg m_1$, $(m_1 \lor m_2)$, $(m_1 \land m_2)$, $(m_1 \implies m_2)$, or $(l_1, \ldots, l_n \implies l_{n+1})$. If possible we omit parenthesis using the usual precedence ordering of the boolean connectors. Every matrix is a formula, and if $f$ is a formula then $\forall x f$ and $\exists x f$ are formulas. The set of
CASE, mechanical CAD, plant layout are particularly rich in miniworld semantics that are all in some way due to physical and technological constraints or to special properties attached to a design object. Nonetheless, even the multitude of experimental DBMS (for a survey see [17]) and the few commercial DBMS (e.g., [13, 35]) in this area avoid external consistency and, instead, attempt to capture the miniworld semantics by providing more complicated data models. The causes are only too clear: checking for external consistency is very expensive because the entire database or at least a restricted but often still large section of it must be inspected. For example, early attempts to include assertions and trigger mechanisms in relational DBMSs (in System R, see [1]) never found their way into corresponding products. Only more recently have approaches been published to a more efficient consistency checking ([33, 15, 23, 10, 19, 37, 28, 9]).

Besides the data model and consistency constraints, deductive rules provide a means for expressing miniworld semantics. This latter approach has commanded attention over a good many years and has given rise to deductive databases ([26]). In fact, there is a certain trade-off between all three. Design databases are prejudiced towards complicated data models such as those based on object-orientation. Diagnostic databases or those supporting expert systems stress simple data models augmented by rules. In the long run, both should include consistency constraints. The recent emphasis, however, has been towards examining consistency in deductive databases.

To avoid excessive checking, or to allow for intermediate inconsistent database states that can only be rectified after a sequence of several database updates, consistency checks are usually deferred to specific points in time. The standard solution is the use of transactions, with checks limited to transaction commit time.

This paper concentrates on external consistency and its checking at the execution end of (flat) transactions in the presence of rules. It assumes that a fair number of consistency constraints have been declared, and that usually several of them are affected by the update operations within the transaction. The classical response to the failure of any one of these is to roll back the transaction in its entirety. The problem this paper addresses is the following. Rollback may leave the user at a loss as to the potential causes for the failure and, hence, with few clues as to the needed changes to the transaction. Also, in long-duration engineering transactions rollback is an entirely inappropriate response; rather the user should be able to perform a number of compensating actions to restore consistency. The objective of our work is to inform the user of a failure, and then to provide him with enough information to take an intelligent decision, e.g., give him a line of reasoning why there is any inconsistency and how it can be resolved. Consequently, our work is primarily oriented towards a DBMS that directly interacts with a human user.

We claim that augmenting consistency control in a DBMS by such a diagnostic and therapeutic component is particularly important in deductive databases. In these, one cannot assume that the user truly understands the interplay of all, facts, rules, and consistency constraints. This would be especially true in a multi-user environment where s/he is not the only person responsible for the contents of the deductive database. We present a system that supports the user in the situation where s/he executed a transaction which violated one or more consistency constraints. The goal is to automatically generate repairs, i.e., transactions which must be appended to the user transaction in order to regain consistency. Intuitively speaking, in a first phase one observes the symptoms through which the inconsistency makes itself known, i.e., (possibly derived) facts that violate the existing constraints. In a second phase one derives the causes underlying the symptoms, i.e., those facts existing in or missing from the database that give rise to a symptom. In a third step the causes are transformed to repairs by syntactic modification. We demonstrate that all steps can indeed be performed in an automated fashion.

Comparative intentions have been reported for relational databases. In [7] an approach is
Reactive Consistency Control 

 in 

 Deductive Databases *

Guido Moerkotte and Peter C. Lockemann

Fakultät für Informatik
Universität Karlsruhe
D-7500 Karlsruhe 1

Abstract

Classical treatment of consistency violations is to back out a database operation or transaction. In applications with large numbers of fairly complex consistency constraints this clearly is an unsatisfactory solution. Instead, if a violation is detected the user should be given a diagnosis of the constraints that failed, a line of reasoning on the cause that could have led to the violation, and suggestions for a repair. The problem is particularly complicated in a deductive database system where failures may be due to an inferred condition rather than simply a stored fact, but the repair can only be applied to the underlying facts. The paper presents a system which provides automated support in such situations. It concentrates on the concepts and ideas underlying the approach and an appropriate system architecture and user guidance, and sketches some of the heuristics used to gain in performance.

1 Introduction

A database is called consistent if it is a truthful model of a given miniworld. However, with the potential exception of process control databases which receive their inputs from process-based sensors, the data are acquired through human interlocutors so that a database management system (DBMS) has no way to verify whether the database indeed agrees with a given state of the miniworld. Instead, a weaker notion of consistency is introduced. A database is called consistent if its current state, and perhaps the transition that led to it, obey a given set of conditions, where the conditions reflect laws that govern states and transitions in the miniworld. Consistency is in part achieved by utilizing the concepts offered by the DBMS data model, and by proper design of the database schema that determines the use of these concepts (data model consistency and schema consistency, respectively, collectively referred to as internal consistency). Those laws that could not be covered in this way must be formulated explicitly in the form of conditions called consistency constraints (external consistency).

Consistency constraints are a means for capturing miniworld semantics that cannot be expressed by a given data model. Business applications which form the mainstay of DBMS applications seem to have been satisfied with the data models of commercially available DBMS such as the relational and the network models, with little need to resort to external consistency. On the other hand, so-called non-standard applications as exemplified by design databases for VLSI,