Consistency-based Diagnosis of Configuration Knowledge Bases

Alexander Felfernig, Gerhard E. Friedrich, Dietmar Jannach [1] and **Markus Stumptner** [2]

Abstract. Configuration problems are a thriving application area for declarative knowledge representation that currently experiences a constant increase in size and complexity of knowledge bases. Automated support of the debugging of such knowledge bases is a necessary prerequisite for effective development of configurators. We show that this task can be achieved by consistency-based diagnosis techniques. Based on the formal definition of consistency-based configuration we develop a framework suitable for diagnosing configuration knowledge bases. During the test phase of configurators, valid and invalid examples are used to test the correctness of the system. In case such examples lead to unintended results, debugging of the knowledge base is initiated. Starting from a clear definition of diagnosis in the configuration domain we develop an algorithm based on conflicts. Our framework is general enough for its adaptation to diagnosing customer requirements to identify unachievable conditions during configuration sessions.

1 INTRODUCTION

Knowledge-based configuration systems have a long history as a successful AI application area. These systems have progressed from their rule-based origins to the use of higher level representations such as various forms of constraint satisfaction, description logics [10], or functional reasoning. As a result of the increased complexity and size of configurator knowledge bases, the user of a configuration tool is increasingly challenged to find the source of the problem whenever it is not possible to produce a working configuration, i.e, the configuration process is aborted. Ultimately, the cause of an abort is either an incorrect knowledge base or unachievable requirements.

In this paper, we will focus on the situation of an engineer working on the maintenance of a knowledge base, searching for failures while performing test configurations. Therefore, the goal is to examine what part of the knowledge base itself may have produced the problem and provide adequate automated support to locate the true source of the inconsistency. This validation phase will take place after the initial specification of the knowledge base or later in the lifecycle when the knowledge base is updated to meet new or altered application requirements (e.g., new component types or regulations).

It is the fact that the knowledge base is specified in some high-level, declarative formalism that allows us to employ model-based diagnosis techniques using positive and negative examples for this purpose. This means that positive configuration examples should be accepted by the configurator whereas negative examples should be rejected. The examples therefore play a role much like what is called a test case in software engineering: they provide an input such that the generated output can be compared to the tester's expectations. Once a test has failed, diagnosis can be used to locate the parts of the knowledge base responsible for the failure. Such parts will typically be constraints that specify legal connections between components, or domain declarations that limit legal assignments to attributes. These constraints and declarations, written as logical sentences, will serve as diagnosis components when we map the problem to the model-based diagnosis approach.

A second type of situation where diagnosis can be used is the support of the actual end user where the user's requirements are unfulfillable even though the knowledge base is correct, e.g., because she/he placed unrealistic restrictions on the system to be configured.

The rest of the paper is organized as follows. We first present an example to introduce the problem domain and the employed configuration terminology. We then formalize the configuration task in terms of a domain theory and system requirements, define what we understand by a valid and irreducible configuration, and use the formalization to express the notion of model-based diagnosis as it applies to the configuration domain. After that, we give an algorithm for computing diagnoses based on positive and negative example sets, and explore the influence of different types of examples. In the final sections, we first examine the "reverse" use of diagnosis for identifying faults in requirements, present the results of a prototype implementation, and close with a discussion of related work.

2 A CONFIGURATION EXAMPLE

We introduce our concepts using a small part of a configuration knowledge base from the area of configurable personal computers. We will insert a typical failure in this knowledge base and show how this failure can be diagnosed. As a representation language we employ first order logic in order to facilitate a clear and precise presentation. In our example a PC motherboard can host up to 4 CPUs and exactly one chipset. The physical insertion of the parts is modeled via ports. Additionally, attributes can be used to model further properties of components, e.g., CPU clock rate.

The knowledge base consists of the following definitions:
types={ *motherboard, cpu-486, cpu-586, chipset-1, chipset-2* }.
ports(motherboard)={*chipset, cpu-1, cpu-2, cpu-3, cpu-4*}.
ports(cpu-486)={*motherboard*}. *ports(cpu-586)*={*motherboard*}.
ports(chipset-1)={*motherboard*}.
ports(chipset-2)={*motherboard*}.

Typically, we use three predicates for associating types, connections, and attributes with individual components. A type t is associated with a component c by a literal $type(c,t)$. A connection is

[1] Institut für Wirtschaftsinformatik und Anwendungssysteme, Universitätsstrasse, A-9020 Klagenfurt, email: felfernig@ifi.uni-klu.ac.at
[2] Institut für Informationssysteme, Abteilung für Datenbanken und Expertensysteme, Paniglgasse 16, A-1040 Wien, email: mst@dbai.tuwien.ac.at
Authors are listed in alphabetical order.

represented by a literal $conn(c1, p1, c2, p2)$ where $p1$ (resp. $p2$) is a port of component $c1$ (resp. $c2$). An attribute value v assigned to attribute a of component c is represented by a literal $val(c, a, v)$. In our example, we omit val - predicates to keep the presentation short. (See [4] for examples using val - predicates.)

The following constraints have to hold in our domain:

"If there is a CPU-486 on the motherboard then a chipset of one of the given types must be inserted too." (**Constraint C1**)

$\forall M, C : type(M, motherboard) \wedge type(C, cpu\text{-}486) \wedge$
$conn(C, motherboard, M, \_) \Rightarrow$
$\exists S : conn(S, motherboard, M, chipset) \wedge$
$(type(S, chipset\text{-}1) \vee type(S, chipset\text{-}2))$.

"If there is a CPU-586 on the motherboard, only a chipset of type chipset-2 is allowed". (**Constraint C2**)

$\forall M, C : type(M, motherboard) \wedge type(C, cpu\text{-}586) \wedge$
$conn(C, motherboard, M, \_) \Rightarrow \exists S :$
$conn(S, motherboard, M, chipset) \wedge type(S, chipset\text{-}2)$.

"The chipset port of the motherboard can only be connected to a chipset of type chipset-1". (**Constraint C3**)

$\forall M, C : type(M, motherboard) \wedge$
$conn(C, \_, M, chipset) \Rightarrow type(C, chipset\text{-}1)$

As it turns out, this constraint is faulty because it is too strong. This constellation could have come about because the chipset type *chipset-2* was newly introduced to the knowledge base, and *C3* was not altered to accommodate that. The correct version of this constraint **(**$C3_{ok}$**)** would also permit chipsets of type *chipset-2* , i.e.,

$\forall M, C : type(M, motherboard) \wedge conn(C, \_, M, chipset) \Rightarrow$
$type(C, chipset\text{-}1) \vee type(C, chipset\text{-}2)$.

In the following we denote the faulty knowledge base by $KB_{faulty} = \{C1, C2, C3\}$.

In addition, a set of application independent constraints denoted by C_{Basic} is included in the domain description, specifying that connections are symmetric, that a port can only be connected to one other port, and that components have a unique type. Furthermore we employ the unique name assumption, define that every attribute of a component has a unique value, and that only the predefined *type*, *port*, and *attribute* symbols are used.

After the definition of the knowledge base, the test engineer can validate the returned results of the configurator for different (positive and negative) examples.

The first positive example provided is a mainboard with two CPUs plugged in where one is of type *cpu-486* and one of type *cpu-586*. We denote such a positive example as e^+. More formally,

$e^+ = \{\exists M, C1, C2 : type(M, motherboard) \wedge$
$type(C1, cpu\text{-}486) \wedge type(C2, cpu\text{-}586) \wedge$
$conn(C1, motherboard, M, cpu\text{-}1) \wedge$
$conn(C2, motherboard, M, cpu\text{-}2).\}$

Note that examples can either be partial or complete configurations. The example above is a partial one, as more components and connections must be added to arrive at a finished configuration.

Next, a negative example is provided, comprising a motherboard with two CPUs of type *cpu-486* and *cpu-586* as it was the case in e^+ but in addition a chipset of type *chipset-1* is also connected to the motherboard. We denote such a negative example as e^-, where such an example should be inconsistent with the knowledge base.

$e^- = \{\exists M, C1, C2, CS1 : type(M, motherboard) \wedge$
$type(C1, cpu\text{-}486) \wedge type(C2, cpu\text{-}586) \wedge$
$type(CS1, chipset\text{-}1) \wedge conn(C1, motherboard, M, cpu\text{-}1)$
$\wedge conn(C2, motherboard, M, cpu\text{-}2) \wedge$
$conn(CS1, motherboard, M, chipset).\}$

Testing the knowledge base with e^- results in the expected contradiction, i.e., $KB_{faulty} \cup e^- \cup C_{Basic}$ is inconsistent. However, $KB_{faulty} \cup e^+ \cup C_{Basic}$ is also inconsistent which is not intended. The question is which of the application specific constraints $\{C1, C2, C3\}$ are faulty. When adopting a consistency-based diagnosis formalism, the constraints C1, C2, and C3 are viewed as components and the problem can be reduced to the task of finding those constraints which, if canceled, restore consistency.

Note that $\{C2, C3\} \cup e^+ \cup C_{Basic}$ is contradictory. It follows that C2 or C3 has to be canceled in order to restore consistency, i.e., $\{C1, C2\} \cup e^+ \cup C_{Basic}$ is consistent and $\{C1, C3\} \cup e^+ \cup C_{Basic}$ is consistent. However, if we employ the negative example we recognize that $\{C1, C3\} \cup e^- \cup C_{Basic}$ is also consistent which has to be avoided. Therefore, in order to repair the knowledge base, $\{C1, C3\}$ has to be extended for restoring inconsistency with e^-. To be able to accept *"C2 is faulty"* as a diagnosis we have to investigate whether such an extension EX can exist. To check this, we start from the property that $\{C1, C3\} \cup e^- \cup EX \cup C_{Basic}$ must be inconsistent (note that $\{C1, C3\} \cup EX \cup C_{Basic}$ must be consistent) and therefore $\{C1, C3\} \cup EX \cup C_{Basic} \models \neg e^-$, i.e., the knowledge-base has to imply the negation of the negative example. In addition, this knowledge base has also to be consistent with the positive example: $\{C1, C3\} \cup e^+ \cup EX \cup C_{Basic}$ is consistent. Therefore, $\{C1, C3\} \cup EX \cup e^+ \cup \neg e^- \cup C_{Basic}$ would have to be consistent which is not the case for our example: $e^+ \cup \neg e^-$ implies that there must not be a chipset of type *chipset-1* connected to the slot of a motherboard whereas $\{C1, C3\} \cup e^+ \cup C_{Basic}$ requires that this slot must be connected to a chipset of type *chipset-1*. Consequently, the diagnosis *"C2 is faulty"* must be rejected. Note that the case in which we removed C3, the knowledge base $\{C1, C2\}$ is inconsistent with e^-, i.e., *"C3 is faulty"* can be accepted as a diagnosis.

These concepts will be defined and generalized in the following sections. The resulting consistency-based framework for configuration and diagnosis of configuration knowledge bases will also give us the ability to identify multiple faults given sets of multiple examples.

3 DEFINING CONFIGURATION AND DIAGNOSIS

In practice, configurations are built from a predefined catalog of component types for a given application domain. These component types are described through their properties (attributes) and connection points (ports) for logical or physical connections to other components. We assume this information and additional constraints on legal constellations to be contained in a domain description DD.

An actual configuration problem has to be solved according to some set of specific user requirements SRS describing e.g., additional constraints or initial partial configurations. An individual configuration (result) consists in our example of a set of components, a listing of established connections and their attribute values. Such configurations are described by positive ground literals. In the previous example the predicates $conn/4$ and $type/2$ are employed without limiting the generality of our approach. Depending on the application domain other predicates can be used, e.g., for specifying attribute assignments or ports to be unconnected.

Definition (Configuration Problem): *In general we assume a configuration problem is described by a triple* $(DD, SRS, CONL)$ *where* DD *and* SRS *are logical sentences and* $CONL$ *is a set of predicate symbols.*

DD *represents a configuration knowledge base (domain description), and* SRS *specifies the particular system requirements which define an individual configuration problem instance. A configuration*

$CONF$ is described by a set of positive ground literals whose predicate symbols are in the set of $CONL$. □

Example: In the domain described in the previous section, DD is given by the union of the specification of types and ports with the set of constraints $\{C1, C2, C3_{ok}\} \cup C_{Basic}$, $CONL = \{type/2, conn/4\}$, and the set e^+ can be seen as a particular set of system requirements. In this example the system is specified by explicitly listing the set of required key components. A configuration for this problem is given by

$CONF_1 = \{type(m, motherboard). type(c1, cpu\text{-}486).$
$type(c2, cpu\text{-}586). type(cs, chipset\text{-}2).$
$conn(c1, motherboard, m, cpu\text{-}1).$
$conn(c2, motherboard, m, cpu\text{-}2).$
$conn(cs, motherboard, m, chipset).\}$ □

Note that the above configuration $CONF_1$ is consistent with $SRS \cup DD$. In general, we are interested only in such consistent configurations.

Definition (Consistent Configuration): *Given a configuration problem $(DD, SRS, CONL)$, a configuration $CONF$ is consistent iff $DD \cup SRS \cup CONF$ is satisfiable.* □

This intuitive definition allows determining the validity of partial configurations, but does not require the completeness of configurations. For example, $CONF_1$ above constitutes a consistent configuration, but so would e^+ alone if we view the existential quantification as Skolem constants.

As we see, for practical purposes, the consistency of configurations is not enough. It is necessary that a configuration explicitly includes all needed components (and their connections and attribute values), in order to assemble a correctly functioning system. We need to introduce an explicit formula for each predicate symbol in $CONL$ to guarantee this completeness property. In order to stay within first order logic, we model the property by first order formulae. However, other approaches, e.g., based on logics with nonstandard semantics, are possible.

For our example we have to add the completeness axioms

$type(X, Y) \Rightarrow \bigvee_{Z \in CONF} type(X, Y) = Z.$
$conn(V, W, X, Y) \Rightarrow \bigvee_{Z \in CONF} conn(V, W, X, Y) = Z.$

We denote the configuration $CONF$ extended by completeness axioms with $\widehat{CONF}$.

Definition (Valid Configuration): *Let $(DD, SRS, CONL)$ be a configuration problem. A configuration $CONF$ is valid iff $DD \cup SRS \cup \widehat{CONF}$ is satisfiable.* □

Having completed our definition of the configuration task, we can now try to find the sources of inconsistencies in terms of model-based diagnosis (MBD) terminology. Generally speaking, the MBD framework assumes the existence of a set of components (whose incorrectness can be used to explain the error), and a set of observations that specify how the system actually behaves. Following the exposition given in the introduction, the role of components is played by the elements of DD, while the observations are provided in terms of (positive or negative) configuration examples.

Definition (CKB-Diagnosis Problem): *A CKB-Diagnosis Problem (Diagnosis Problem for a Configuration Knowledge Base) is a triple (DD, E^+, E^-) where DD is a configuration knowledge base, E^+ is a set of positive and E^- of negative configuration examples. The examples are given as sets of logical sentences. We assume that each example on its own is consistent.* □

The two example sets serve complementary purposes. The goal of the positive examples in E^+ is to check that the knowledge base will accept correct configurations; if it does not, i.e., a particular positive example e^+ leads to an inconsistency, we know that the knowledge base as currently formulated is too restrictive. Conversely, a negative example serves to check the restrictiveness of the knowledge base; negative examples correspond to real-world cases that are configured incorrectly, and therefore a negative example that is accepted means that a relevant condition is missing from the knowledge base.

Typically, the examples will of course consist mostly of sets of literals whose predicate symbols are in $CONL$. (If we want to test an example w. r. t. specific user requirements, we include them in the example definition.) In case these examples are intended to be complete the special completeness axioms must be added. If an example is supposed to be a complete configuration, diagnoses will not only help to analyze cases where incorrect components or connections are produced in configurations, but also cases where the knowledge base requires the generation of superfluous components or connections. The reason why it is important to give partial configurations as examples is that if a test case can be described as a partial configuration, a drastically shorter description may suffice compared to specifying the complete example that, in larger domains, may require thousands of components to be listed with all their connections [5].

In the line of consistency-based diagnosis, an inconsistency between DD and the positive examples means that a diagnosis corresponds to the removal of possibly faulty sentences from DD such that the consistency is restored. Conversely, if that removal leads to a negative example e^- becoming consistent with the knowledge base, we have to find an extension that, when added to DD, restores the inconsistency for all such e^-.

Definition (CKB-Diagnosis): *A CKB-diagnosis for a CKB-Diagnosis Problem(DD, E^+, E^-) is a set $S \subseteq DD$ of sentences such that there exists an extension EX, where EX is a set of logical sentences, such that*

$DD - S \cup EX \cup e^+$ *consistent* $\forall e^+ \in E^+$
$DD - S \cup EX \cup e^-$ *inconsistent* $\forall e^- \in E^-$ □

A diagnosis will always exist under the (reasonable) condition that positive and negative examples do not interfere with each other.

Proposition: *Given a CKB-Diagnosis Problem (DD, E^+, E^-), a diagnosis S for (DD, E^+, E^-) exists iff*

$\forall e^+ \in E^+ : e^+ \cup \bigwedge_{e^- \in E^-} (\neg e^-)$ *is consistent.*

Proof (see [4].)

From here on, we refer to the conjunction of all negated negative examples as NE, i.e., $NE = \bigwedge_{e^- \in E^-} (\neg e^-)$

In principle, the definition of CKB-diagnosis *S* is based on finding an extension *EX* of the knowledge base that fulfills the consistency and the inconsistency property of the definition for the given example sets. However, the proposition above helps us insofar as it gives us a way to characterize diagnoses without requiring the explicit specification of the extension EX.

Corollary: *S is a diagnosis iff $\forall e^+ \in E^+ : DD - S \cup e^+ \cup NE$ is consistent.*

The following remark relates configuration and diagnosis for configuration knowledge bases.

Remark: Let e^+ be partitioned in two disjoint sets e^+_{CONF} and e^+_{SRS} where e^+_{CONF} is a set of positive ground literals whose predicate symbols are in the set of $CONL$ and e^+_{SRS} represents system requirements (if some are specified in conjunction with the positive example).

S is a diagnosis for (DD, E^+, E^-) iff $\forall e^+ \in E^+ : e^+_{CONF}$ is a consistent configuration for $(NE \cup DD - S, e^+_{SRS}, CONL)$.

Note that, if the completeness axioms have been added to e^+_{CONF} then e^+_{CONF} is a valid configuration for $(NE \cup DD - S, e^+_{SRS}, CONL)$.

4 COMPUTING DIAGNOSES

The above definitions allow us to employ the standard algorithms for consistency-based diagnosis, with appropriate extensions for the domain. In particular, we use Reiter's Hitting Set algorithm [12] which is based on the concept of conflict sets for focusing purposes.

Definition (Conflict Set) : *A conflict set CS for (DD, E^+, E^-) is a set of elements of DD such that $\exists e^+ \in E^+ : CS \cup e^+ \cup NE$ is inconsistent. We say that, if $e^+ \in E^+ : CS \cup e^+ \cup NE$ is inconsistent, that e^+ induces CS.* □

In the algorithm we employ a labeling that corresponds to the labeling of the original HS-DAG ([8], [12]), i.e., a node n is labeled by a conflict $CS(n)$ and edges leading away from n are labeled by logical sentences $s \in CS(n)$. The set of edge labels on the path leading from the root to n is referred to as $H(n)$. In addition, each node is labeled by the set of positive examples $CE(n)$ that have been found to be consistent with $DD - H(n) \cup NE$ during the DAG-generation. The reason for introducing the label $CE(n)$ is the fact that any e^+ that is consistent with a particular $DD - H(n) \cup NE$ is obviously consistent with any $H(n')$ such that $H(n) \subseteq H(n')$. Therefore any e^+ that has been found consistent in step 1.(a) below does not need to be checked again in any nodes below n. Since we generate a DAG, a node n may have multiple direct predecessors (we refer to that set as $preds(n)$ from here on), and we will have to combine the sets $CE(m)$ for all direct predecessors m of n. The consistent examples for a set of nodes N (written $CE(N)$) are defined as the union of the $CE(n)$ for all $n \in N$.

Algorithm (schema) In: DD, E^+, E^-; **Out:** a set of diagnoses S

(1) Use the Hitting Set algorithm to generate a pruned HS-DAG D for the collection F of conflict sets for (DD, E^+, E^-). The DAG is generated in a breadth-first manner since we are interested in generating diagnoses in order of their cardinality.

(a) Every theorem prover call $TP(DD - H(n), E^+ - CE(preds(n)), E^-)$ at a node n corresponds to a test of whether there exists an $e^+ \in E^+ - CE(preds(n))$ such that $DD - H(n) \cup e^+ \cup NE$ is inconsistent. In this case it returns a conflict set $CS \subseteq DD - H(n)$, otherwise it returns ok.

Let $E_{CONS} \subseteq E^+ - CE(preds(n))$ be the set of all e^+ that have been found to be consistent in the call to TP.

(b) Set $CE(n) := E_{CONS} \cup CE(preds(n))$.

(2) Return $\{H(n) | n$ *is a node of D labeled by ok*$\}$

Complete versus Partial Examples

As mentioned before, examples (negative and positive) can be complete or partial. Previously we stated that complete examples are in principle preferable for diagnosis (neglecting the effort needed for specification) since they are more effective. We will now show that this is so because, under certain assumptions for the language used in the domain description, diagnosing a complete example will always result in only singleton conflicts.

Proposition: *Given an example e^+ (consisting of a configuration and the corresponding completeness axioms) from a set of positive examples E^+ for a CKB-diagnosis problem (DD, E^+, E^-) such that DD uses only predicates from $CONL$, then any minimal conflict set induced by e^+ for (DD, E^+, E^-) is a singleton.*

Proof (see [4].)

The practical implications are that for any given complete positive example, we can limit ourselves to checking the consistency of the elements s of DD with $e^+ \cup NE$ individually, because any s found to be inconsistent constitutes a conflict. Conversely, any s found to be consistent is not in the induced minimal conflict sets of e^+.

5 DIAGNOSING REQUIREMENTS

Even once the knowledge base has been tested and found correct, diagnosis can still play a significant role in the configuration process. Instead of an engineer testing an altered knowledge base, we are now dealing with end users who are using the assumed correct knowledge base for configuring actual systems. During their sessions, such users frequently face the problem of requirements being inconsistent because they exceed the feasible capabilities of the system to be configured. In such a situation, the diagnosis approach presented here can now support the user in finding which of his/her requirements produces the inconsistency. Formally, the altered situation can be easily accommodated by swapping requirements and domain description in the definition of CKB-Diagnosis. Formerly, we were interested in finding particular sentences from DD that contradicted the set of examples. Now we have the user's system requirements SRS, which contradict the domain description. The domain description is used in the role of an all-encompassing partial example for correct configurations.

Definition (CREQ-Diagnosis Problem): *A configuration requirements diagnosis (CREQ-Diagnosis) problem is a tuple (SRS, DD), where SRS is a set of system requirements and DD a configuration domain description. A CREQ Diagnosis is a subset $S \subseteq SRS$ such that $SRS - S \cup DD$ is consistent.* □

Remark: S is a CREQ diagnosis for (SRS, DD) *iff* S is a CKB diagnosis for $(SRS, \{DD\}, \{\})$.

6 EXPERIMENTAL RESULTS

In order to test the applicability of our approach we have implemented a prototype using a commercial configurator product (ILOG Configurator [9]). When using this package of C++ libraries, a configuration problem can be defined in terms of a *Constraint Satisfaction Problem (CSP)* whereby the basic CSP mechanism is enhanced with special features for the configuration domain, e.g., the number of variables (components) the final product consists of may not be known beforehand.

The problem representation used in this tool is based on the general component-port model for configuration as described in Section 2. The domain description comprises the definition of the product structure (available components, attributes and ports) as well as additional relational or arithmetic constraints on the problem variables. A configuration result comprises a set of components and instantiated problem variables (attributes, ports). The user requirements SRS and the example sets E^+ and E^- can be given in terms of additional constraints or in terms of partial or complete examples.

In the context of a CSP, a conflict set for an example is a set of constraints that, if canceled, make the CSP satisfiable, i.e., a solution can be found. The call to the theorem prover as described in the algorithm above then corresponds to a call to the constraint solver, to test – given some positive examples – which constraints are definitely violated, or whether one partial example can not be extended to a complete configuration.

In general the task of finding (minimal) conflict sets may be computationally expensive. Note that the presented diagnosis algorithm does not necessarily need minimal conflict sets, although the performance of the algorithm depends on the size of the conflict sets. In the case of our prototype implementation the calculation of one arbitrary solution for the CSP is done by the solver in a very efficient manner. Therefore, the algorithm can also be employed with reasonable performance, even if the inference engine (constraint solver) has only

limited capabilities of explaining the sources of the inconsistencies.

The single fault from our simple demonstration example can be diagnosed on a standard Pentium-II PC instantaneously. However, in a more realistic setting with about thirty component types and about thirty types of generic constraints on the problem variables and several hundred constraint instances, the system still detects two inserted double faults in a few seconds. Finding diagnoses of cardinality four takes about half a minute in our typical test cases using our non-optimized prototype. We can restrict the search depth to a certain level assuming that in many cases, diagnoses of higher cardinality do not help the user too much anyway.

Apart from the tests on our example domain, we have conducted experiments on real world problems for the configuration of private telecommunication switching systems which have shown the benefits of our method.

7 RELATED WORK

In [3], a framework for model-based diagnosis of logic programs was developed using expected and unexpected query results to identify incorrect clauses, a line of work later continued by Bond [2]. Their framework is similar to ours, but differs in using queries instead of checking consistency as we do for configurations. [2] embedded the diagnosis of logic programs and the concept of Algorithmic Program Debugging [13] in a common underlying framework.

Related work in the area of logic programming was done by [11] using a semantics based on SLDNF-resolution. Simlar results may be achieved by using proper extensions of their framework. Note that our goal was to remain within first order predicate logic with its standard semantics.

Work is currently underway to extend the use of model-based diagnosis to other types of software, in particular those with non-declarative semantics, i.e., imperative languages ([6], [14]). In [1], model-based diagnosis is used for finding solutions for over-constrained constraint satisfaction problems. Search is controlled by explicitly assigning weights to the constraints in the knowledge base that provide an external ordering on the desirability of constraints, an assumption that is generally too strong for our domain.

A model-based scheme for repairing relational database consistency violations is given in [7]. Integrity constraints, though expressed in relational calculus, effectively are general clauses using the relations in the database as base predicates. The interpretation of the constraints in diagnosis terms uses two fault models for each relation, expressing that a particular tuple must either be removed or inserted into the database to satisfy the constraint. Individual violated constraints are used to directly derive conflict sets for the diagnosis process. A particular diagnosis serves directly as a specification for the actions that will bring the database into a state consistent with the violated constraints. However, the computation of diagnoses only considers the current set of violated constraints, i.e., a repair executed according to a diagnosis found by examining the inconsistent constraints may lead to alterations that violate other, previously satisfied, constraints. Given that the goal of the approach is the alteration of the database, the best correspondence is with what we consider requirements diagnosis (and like our definition of CREQ-diagnosis, Gertz and Lipeck do not use negative examples).

8 CONCLUSION

With the growing relevance and complexity of AI-based applications in the configuration area, the usefulness of other knowledge-based techniques for supporting the development of these systems is likewise growing. In particular, due to its conceptual similarity to configuration, model-based diagnosis is a highly suitable technique to aid in the debugging of configurators. We have developed a framework for localizing faults in configuration knowledge bases, based on a precise definition of configuration problems. This definition enables us to clearly identify the causes (diagnoses) that explain a misbehavior of the configurator, and express their properties. Positive and negative examples, commonly used in testing configurators, are exploited to identify possible sets of faulty clauses in the knowledge base. Building on the analogy between the formal models of configuration and diagnosis, we have given an algorithm for computing diagnoses in the consistency-based diagnosis framework. Finally, we have examined how our method can be used for a different task in the same context: identifying conflicting customer or user requirements, that prevent the construction of valid configurations, support the user during configuration sessions. The clear separation between knowledge base and inference engine enables us to deal with knowledge bases in terms of their declarative semantics, and at the same time facilitates their translation to (or incorporation into) the type of model desired for diagnosis purposes. Since the model remains independent of a particular implementation of the inference process, the net result is that the model-based approach scores both in terms of generality and ease of application as well as in terms of robustness.

Acknowledgement
We thank D. T. Dupré for his valuable comments.

REFERENCES

[1] R. R. Bakker, F. Dikker, F. Tempelman, and P.M. Wognum, *Diagnosing and solving over-determined constraint satisfaction problems*, In Proc. IJCAI'93, pp. 276-281, Chambery, Morgan Kaufmann, 1993.

[2] G. W. Bond, *Top-down consistency based diagnosis*, In Proc. DX'96 Workshop, Val Morin, Canada, 1996.

[3] L. Console, G. E. Friedrich, and D. T. Dupré. *Model-based diagnosis meets error diagnosis in logic programs*, Proc. IJCAI'93, pp. 1494-1499, Chambery, Morgan Kaufmann, 1993.

[4] A. Felfernig, G. E. Friedrich, D. Jannach, and M. Stumptner. *Consistency based diagnosis of configuration knowledge bases*, AAAI'99 Workshop on Configuration (WS99-05), AAAI Press, 1999.

[5] G. Fleischanderl, G. E. Friedrich, A. Haselboeck, H. Schreiner, and M. Stumptner, *Configuring Large Systems Using Generative Constraint Satisfaction*, IEEE Intelligent Systems, Vol. 13, No. 4, pp. 59-68, July/August 1998, 1998.

[6] G. E. Friedrich, M. Stumptner, and F. Wotawa: *Model-Based Diagnosis of Hardware Designs*, Artificial Intelligence, 111(2):3-39, 1999

[7] M. Gertz and U. W. Lipeck, *A Diagnostic Approach to Repairing Constraint Violations in Databases*, In Proc. DX'95 Workshop, Goslar, October 1995.

[8] R. Greiner, B. A. Smith, and R. W. Wilkerson, *A correction to the algorithm in Reiter's theory of diagnosis*, Artificial Intelligence, 41(1):79-88, 1989.

[9] D. Mailharro, *A Classification and constraint-based framework for configuration*, Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 12(4), 1998.

[10] D. L. McGuinness and J. R. Wright, *Conceptual Modelling for Configuration: A Description Logic-based Approach*, Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 12(4), 1998.

[11] L. M. Pereira, C. V. Damsio, and J. J. Alferes, *Debugging by Diagnosing Assumptions*, In Proc. AADEBUG'93, Linköping, 1993.

[12] R. Reiter, *A theory of diagnosis from first principles*, Artificial Intelligence, 32(1):57-95, 1987.

[13] E. Shapiro, *Algorithmic Program Debugging*, MIT Press, Cambridge, Massachusetts, 1983.

[14] M. Stumptner and F. Wotawa, *VHDLDIAG+: Value-Level Diagnosis of VHDL Programs*, In Proceedings DX'98 Workshop, Cape Cod, 1998.