Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Artificial Intelligence 152 (2004) 213–234

Artificial Intelligence

www.elsevier.com/locate/artint

Consistency-based diagnosis of configuration knowledge bases [☆]

Alexander Felfernig [a], Gerhard Friedrich [a], Dietmar Jannach [a,*], Markus Stumptner [b]

[a] *Universität Klagenfurt, Produktionsinformatik, A-9020 Klagenfurt, Austria*
[b] *University of South Australia, Advanced Computing Research Centre, SA 5095 Adelaide, Australia*

Received 21 August 2002; received in revised form 23 May 2003

Dedicated to Raymond Reiter, the pioneer of consistency-based diagnosis

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 process 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.

A prototype implementation using commercial constraint-based configurator libraries shows the feasibility of diagnosis within the tight time bounds of interactive debugging sessions. Finally, we discuss the usefulness of the outcomes of the diagnostic process in different scenarios.

© 2003 Elsevier B.V. All rights reserved.

Keywords: Diagnosis; Configuration

[☆] Revised and extended version of A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Consistency-based diagnosis of configurator knowledge bases, in: W. Horn (Ed.), Proc. 14th European Conference on Artificial Intelligence (ECAI-2000), Berlin, Germany, IOS Press, Amsterdam, 2000, pp. 146–150. The authors are listed in alphabetical order.

[*] Corresponding author.
E-mail address: dietmar@ifit.uni-klu.ac.at (D. Jannach).

0004-3702/$ – see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0004-3702(03)00117-6

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 [12, 19], description logics [20], or functional reasoning [24]. 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 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. We also examine the "reverse" use of diagnosis for identifying faults in requirements. In the final sections, we present the results of a prototype implementation, and close with a discussion of related work.

2. Motivating 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 paradigm for modeling configuration problems we rely on the *component-port model* of configuration described in [21] because of its applicability for different domains and because it serves as a basis in many commercial configuration systems [12,17,19]. Following this paradigm, a configurable system is composed of predefined components which are further characterized by attributes. Individual components can be interconnected via predefined connection points (called *ports*). Apart from the description of the available components, their attributes, the permitted range of values for these attributes, and the connection points, a configurator knowledge base typically contains constraints that describe the set of *legal* product constellations. In the following, we employ first order logic as a representation language in order to facilitate a clear and precise presentation. First order logic allows us to describe general concepts for configuration. In practice, decidable variants are applied.

2.1. Defining the faulty knowledge base and the test cases

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. We omit the usage of attributes of components like (CPU clock rate) in the example in order to keep the presentation short (see Fig. 1). Note that this figure only represents the *structure* of the available components,

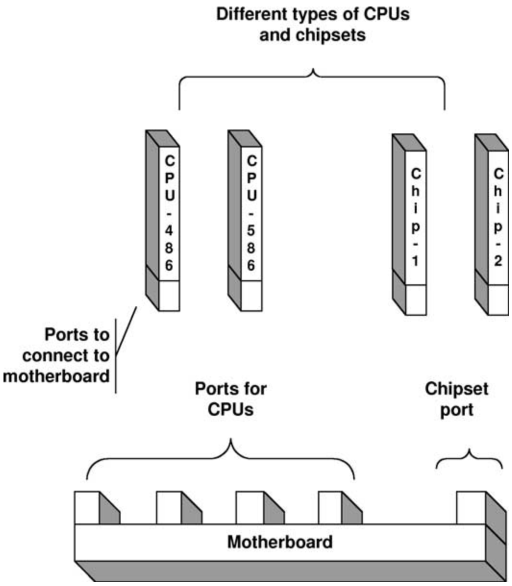

Different types of CPUs and chipsets
CPU-486
CPU-586
Chip-1
Chip-2
Ports to connect to motherboard
Ports for CPUs
Chipset port
Motherboard

Fig. 1. Example problem.

i.e., in a configuration several instances (sometimes called individuals) of a *component type* can occur.

We describe the available component types and their named interconnection points with the functions *types* and *ports*, respectively, i.e., the knowledge base consists of the following definitions:

$$\begin{aligned}
&\mathit{types} = \{\mathit{motherboard}, \mathit{cpu}\text{-}486, \mathit{cpu}\text{-}586, \mathit{chipset}\text{-}1, \mathit{chipset}\text{-}2\},\\
&\mathit{ports}(\mathit{motherboard}) = \{\mathit{chipset}, \mathit{cpu}\text{-}1, \mathit{cpu}\text{-}2, \mathit{cpu}\text{-}3, \mathit{cpu}\text{-}4\},\\
&\mathit{ports}(\mathit{cpu}\text{-}486) = \{\mathit{motherboard}\}, \quad \mathit{ports}(\mathit{cpu}\text{-}586) = \{\mathit{motherboard}\},\\
&\mathit{ports}(\mathit{chipset}\text{-}1) = \{\mathit{motherboard}\},\\
&\mathit{ports}(\mathit{chipset}\text{-}2) = \{\mathit{motherboard}\}.
\end{aligned}$$

Typically, we will use a limited set of predicate symbols for associating types, connections and attributes with individual components: In our example, an individual component instance c is associated with a type t by a literal $\mathit{type}(c, t)$. We represent a connection by a literal $\mathit{conn}(c1, p1, c2, p2)$ where $c1$ and $c2$ are component identifiers and $p1$ and $p2$ are those ports of the involved components where a connection is established.

The constraints that hold in our domain are composed from the above-mentioned predicate symbols and are given below. We use a logic programming notation where uppercase letters represent variable symbols.

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

$$\begin{aligned}
&\forall M, C\colon \mathit{type}(M, \mathit{motherboard}) \wedge \mathit{type}(C, \mathit{cpu}\text{-}486) \wedge\\
&\qquad\quad \mathit{conn}(C, \mathit{motherboard}, M, \_) \Rightarrow\\
&\quad \exists S\colon \mathit{conn}(S, \mathit{motherboard}, M, \mathit{chipset}) \wedge\\
&\qquad\quad (\mathit{type}(S, \mathit{chipset}\text{-}1) \vee \mathit{type}(S, \mathit{chipset}\text{-}2)).
\end{aligned}$$

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

$$\begin{aligned}
&\forall M, C\colon \mathit{type}(M, \mathit{motherboard}) \wedge \mathit{type}(C, \mathit{cpu}\text{-}586) \wedge\\
&\qquad\quad \mathit{conn}(C, \mathit{motherboard}, M, \_) \Rightarrow\\
&\quad \exists S\colon \mathit{conn}(S, \mathit{motherboard}, M, \mathit{chipset}) \wedge \mathit{type}(S, \mathit{chipset}\text{-}2).
\end{aligned}$$

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

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

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., Constraint $C3_{ok}$:

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

In the following, we will 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, e.g., that connections are symmetric, that a port can only be connected to exactly one other port, and that components have a unique type from the set of available types described by the function *types*. Furthermore we employ the unique name assumption and assume that all component attributes are single-valued. Remember that a predefined set of predicate symbols (in our example *type*/2, *conn*/4) is used to describe configurations.

After defining 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 motherboard 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,

$$\begin{aligned} e^+ = \{&\exists M, C1, C2\colon \\ &\mathit{type}(M, \mathit{motherboard}) \wedge \mathit{type}(C1, \mathit{cpu}\text{-}486) \wedge \mathit{type}(C2, \mathit{cpu}\text{-}586) \wedge \\ &\mathit{conn}(C1, \mathit{motherboard}, M, \mathit{cpu}\text{-}1) \wedge \mathit{conn}(C2, \mathit{motherboard}, M, \mathit{cpu}\text{-}2)\}. \end{aligned}$$

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, i.e., we need at least one *chipset* in a complete configuration. Let us suppose that the test engineer provided this positive example in order to test that both types of CPUs can still be used in configurations after he/she had changed the knowledge base.

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. This example was specified by the test engineer to validate the alterations made after introducing the new type of chipsets, knowing that the new chipset-type is required when a CPU of type *cpu*-586 is in the configuration.

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

2.2. Finding an explanation for the unexpected behavior

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.

If we consider the positive example e^+, we 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.,

$$\{C1, C2\} \cup e^+ \cup C_{Basic} \text{ is consistent} \quad \text{and}$$
$$\{C1, C3\} \cup e^+ \cup C_{Basic} \text{ is consistent.}$$

However, if we employ the negative example we recognize that

$$\{C1, C3\} \cup e^- \cup C_{Basic} \text{ is also consistent}$$

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

$$\{C1, C3\} \cup e^- \cup EX \cup C_{Basic} \text{ must be inconsistent}$$

(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 the slot of a motherboard cannot be connected to a chipset of type *chipset*-1 connected whereas $\{C1, C3\} \cup e^+ \cup C_{Basic}$ requires a connection to a chipset of exactly that type.

Consequently, the diagnosis "$C2$ *is faulty*" is rejected. Note that in 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, a view of configuration problems that is well-established both in the academic community [9,21,26] and in practical industrial application [12,17,19].

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 $\mathit{conn}/4$ and $\mathit{type}/2$ are employed without limiting the generality of our approach. Depending on the application domain other predicates can be used, e.g., to specify attribute assignments or ports that have to remain unconnected.

Definition 1 (*Configuration problem*). In general we assume a configuration problem is described by a triple $(\mathit{DD}, \mathit{SRS}, \mathit{CONL})$ where *DD* and *SRS* are logical sentences and *CONL* is a set of predicate symbols.

DD (Domain Description) represents a configuration knowledge base, 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 taken from *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 $\{C1, C2, C3_{ok}\} \cup C_{\mathit{Basic}}$, $\mathit{CONL} = \{\mathit{type}/2, \mathit{conn}/4\}$, 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

$$\begin{aligned}
&\mathit{CONF}_1 = \{\\
&\quad \mathit{type}(m, \mathit{motherboard}).\ \mathit{type}(c1, \mathit{cpu}\text{-}486).\ \mathit{type}(c2, \mathit{cpu}\text{-}586).\\
&\quad \mathit{type}(cs, \mathit{chipset}\text{-}2).\ \mathit{conn}(c1, \mathit{motherboard}, m, \mathit{cpu}\text{-}1).\\
&\quad \mathit{conn}(c2, \mathit{motherboard}, m, \mathit{cpu}\text{-}2).\ \mathit{conn}(cs, \mathit{motherboard}, m, \mathit{chipset}).\}.
\end{aligned}$$

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

Definition 2 (*Consistent configuration*). Given a configuration problem $(\mathit{DD}, \mathit{SRS}, \mathit{CONL})$, a configuration *CONF* is consistent iff $\mathit{DD} \cup \mathit{SRS} \cup \mathit{CONF}$ 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.

As an example, if we consider the needed components, which are described by *type* facts, we want to ensure that all these facts are explicitly listed in the configuration *CONF* and there are no spurious components which are not listed there. In other words, if we want to determine the validity of a configuration, the whole description of the configuration (in our example all the *type* and *conn* predicates) must be contained in *CONF*. A partial configuration like e^+ will be consistent according to our definition, but not a *valid* configuration in reality, because a correct system requires a chipset component (see constraints $C1$ and $C2$).

We need to introduce an explicit formula for each predicate symbol in *CONL*—the set of symbols used to describe configurations—to guarantee this completeness property. Such a set of completeness axioms assure that we can deduce the negation of all possible instances of the literals whose predicate symbol is contained in *CONL* except for those positive ground literals which are explicitly mentioned in *CONF*. Thus *CONF* joined with the completeness axioms defines a complete theory with respect to the predicate symbols contained in *CONL*.

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 following completeness axioms:

$$\mathit{type}(X,Y) \Rightarrow \Bigl(\bigvee_{Z\in \mathit{CONF}} \mathit{type}(X,Y) = Z\Bigr),$$

$$\mathit{conn}(V,W,X,Y) \Rightarrow \Bigl(\bigvee_{Z\in \mathit{CONF}} \mathit{conn}(V,W,X,Y) = Z\Bigr),$$

where $\bigvee_{Z\in \mathit{CONF}}$ is interpreted as a macro which is expanded. The result of this expansion is an or-expression containing all the elements of *CONF*.

These two sentences ensure that if there is a component or a connection described by a (grounded) *type* or *conn* predicate, it has to be contained in *CONF*, i.e., there is a matching predicate Z contained in *CONF*. Stating it conversely, all instances of the *type* and *conn* predicates which are not explicitly described in *CONF* are negated.

For CONF_1 and the *type* predicate the expanded completeness axiom results in

$$\begin{aligned}
&\mathit{type}(X,Y) \Rightarrow \\
&\quad (\mathit{type}(X,Y) = \mathit{type}(m, \mathit{motherboard})) \vee (\mathit{type}(X,Y) = \mathit{type}(c1, \mathit{cpu}\text{-}486)) \vee \\
&\quad (\mathit{type}(X,Y) = \mathit{type}(c2, \mathit{cpu}\text{-}586)) \vee \mathit{type}(X,Y) = \mathit{type}(cs, \mathit{chipset}\text{-}2) \vee \\
&\quad (\mathit{type}(X,Y) = \mathit{conn}(c1, \mathit{motherboard}, m, \mathit{cpu}\text{-}1)) \vee \\
&\quad (\mathit{type}(X,Y) = \mathit{conn}(c2, \mathit{motherboard}, m, \mathit{cpu}\text{-}2)) \vee \\
&\quad (\mathit{type}(X,Y) = \mathit{conn}(cs, \mathit{motherboard}, m, \mathit{chipset})).
\end{aligned}$$

Note that we can omit three of the equalities on the right-hand side of the implication as they are obviously unsatisfiable.

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

Definition 3 (*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 4 (*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^- a set 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 these requirements in the example definition.) In situations where 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 ability to give partial configurations as examples is important since 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 [12].

Taking the view of consistency-based diagnosis, a problem in the knowledge base will be expressed through an inconsistency between *DD* and the positive examples. A diagnosis will then indicate a set of possibly faulty sentences whose removal from *DD* will restore consistency. Conversely, if the removal of these sentences 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 5 (*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 1. *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.*

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

Proof. $(\Rightarrow)$ Since $DD - S \cup EX \cup e^-$ is inconsistent for all $e^- \in E^-$, it follows that $\forall e^- \in E^-$: $DD - S \cup EX \models \neg e^-$. Because of the definition of diagnosis and the monotonicity of standard logic, it holds that $\forall e^+ \in E^+$: $DD - S \cup EX \cup e^+ \models NE$. Consequently, since $DD - S \cup EX \cup e^+$ is consistent for all $e^+ \in E^+$, the consistency of $DD - S \cup EX \cup e^+ \cup NE$ for all $e^+ \in E^+$ follows, which implies the consistency of $\forall e^+ \in E^+$: $e^+ \cup NE$.

$(\Leftarrow)$ Given that $\forall e^+ \in E^+$: $e^+ \cup NE$ is consistent, it follows directly that there exists a diagnosis S where $S = DD$ and $EX = NE$. □

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 into 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 [23]. The algorithm is based on the concept of conflict sets, since these provide an effective mechanism for focusing the search for diagnoses.

Definition 6 (*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. If $e^+ \in E^+$: $CS \cup e^+ \cup NE$ is inconsistent, we also say that e^+ *induces* CS.

In the algorithm we employ a labeling that corresponds to the labeling of the original HS-DAG [16,23], 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 generation of the DAG. 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 denote that set by $preds(n)$ 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 $\{H(n) \mid n \textit{ is a node of } D \textit{ labeled by ok}\}$.

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 (if we neglect the higher effort needed for their specification) preferable for diagnosis 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 2. *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 (*Sketch*). e^+ corresponds to a configuration $\widehat{CONF}$. Because of the completeness axioms this theory is logical complete regarding literals from *CONL*. It follows that there exists exactly one H(erbrand)-model. We partition *DD* into the set of sentences which are inconsistent with e^+, i.e., $USATS = \{s \mid s \in DD, e^+ \cup s \cup NE$ is inconsistent$\}$, and the set of sentences which are consistent with e^+, i.e., $SATS = \{s \mid s \in DD, e^+ \cup s \cup NE$ is consistent$\}$. Each sentence in *SATS* is satisfiable by the same unique H-model. Consequently, $e^+ \cup \{s \mid s \in SATS\} \cup NE$ is satisfiable by this H-model. It follows that this set does not contain a conflict set. Therefore, *USATS* is the set of all minimal conflicts induced by e^+, and therefore all minimal conflicts are singletons. □

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, although in a different workplace situation. Instead of an engineer testing an altered knowledge base, we are now dealing with end users who are using the assumed-to-be-correct knowledge base for configuring actual systems. During their sessions, such users frequently face the problem of requirements being inconsistent because they are infeasible given the 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 it is the user's system requirements *SRS* which contradict the domain description. The validated and therefore consistent domain description is used in the role of an all-encompassing partial example for correct configurations.

Definition 7 (*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 $(SRS, \{DD\}, \{\})$.

Example. Given the correct knowledge base $KB_{ok} = \{C1, C2, C3_{ok}\}$ from Section 2, consider a simple example where the user specifies his/her specific requirements $SRS_1 = \{req_1, req_2, req_2\}$ in terms of a partial configuration:

req_1: one *CPU*-586 on slot *cpu*-1.

req_2: one *CPU*-586 on slot *cpu*-2.

req_3: one chipset of type *chipset*-1 on the *chipset* slot.

Solving the corresponding CREQ-Diagnosis Problem (SRS_1, KB_{ok}) results in two minimal diagnoses $\{req_1, req_2\}$ and $\{req_3\}$, i.e., either remove both CPUs or change the type of the chipset. Given these results, the (interactive) user of the diagnosis system is enabled to discriminate among the diagnoses based on his/her personal preferences.

6. Implementation and practical experiences

In order to test the practical applicability of our diagnosis approach for debugging faulty configuration knowledge bases we have implemented the JCONDIAG system on top of ILOG's *JConfigurator*[1] libraries. These commercially available libraries provide a Java-based, object-oriented framework for the development of configurator applications based on *Generative Constraint Satisfaction* [12,17,19]. Our practical experiences from building configuration systems for various domains (e.g., telecommunication switches, IP-based virtual private networks [11], or facility equipment) showed the practicability of the approach.

In particular, we found that seamless integration of the diagnosis component with the core configurator and its basic explanation- and tracing facilities is a key requirement for successful utilization of the system. In fact, the implementation of JCONDIAG fits the development paradigm of *JConfigurator* in a way that no additional development effort is needed for defining the knowledge base in a format ready for diagnosis and, on the other hand, is generic enough to leave the knowledge engineer sufficient degrees of freedom in defining the test examples. As a result, the test cases can be defined in multiple different ways, e.g., by giving key components or partial configurations, specifying the problem in terms of constraints, or by loading existing configurations from external sources for regression testing. Finally, our experiments showed that the performance of the diagnosis system matches the hard requirements on response times for interactive debugging and maintenance sessions.

In the following, we will discuss some details of the implementation of the system, in particular with respect to conflict generation and will then show the results of benchmark tests that were performed for a typical configuration problem, with the problem parameters being varied along several dimensions.

[1] See ILOG JConfigurator 1.0 Reference Manual, http://www.ilog.com.

6.1. Implementation and computation of conflicts

JCONDIAG is available as a Java library that implements the diagnosis algorithm described in the previous section. The implementation handles multiple test examples. It incorporates the pruning techniques described in [23] and manages conflict reuse. The search depth can be limited to a certain level, thus restricting both the search time, and the maximum cardinality of the diagnoses (to a size still comprehensible for the knowledge engineer). In addition, the design of the library allows the integration of other configuration reasoners than *JConfigurator* by the usage of a defined interface (compare *TP* in previous section).

Although the proposed diagnosis technique does not require the computation of *minimal* conflict sets, the size of the computed conflict sets heavily influences the run-time behavior for the algorithm. In [10], an approach for handling this problem is proposed that relies on iterative focusing based on hierarchical diagnosis for situations where no minimal conflict sets are available. Within JCONDIAG, (minimal) conflict detection is based on Junker's QUICKXPLAIN [18] algorithm. QUICKXPLAIN is a non-intrusive conflict detector that gets its efficiency by recursively partitioning the problem into subproblems of half the size and skipping those that do not contain an element of the propagation-specific conflict. According to [18], this algorithm needs $\mathrm{O}(n \cdot \log(k+1) + k^2)$ checks to compute a minimal conflict of size k out of n constraints, thus improving the ideas given in, e.g., [7].

6.2. Test results

While there are many benchmark problems, metrics, and generators available for standard (static and/or binary) Constraint Satisfaction Problems, up to now there are no benchmark problems available for the configuration domain. Although problem solving is based on Constraint Satisfaction in our problem setting, configuration problems typically exceed the capabilities of standard CSP representations by requiring, e.g., variable generation during the search process and extensible domains for connection constraints. Therefore, we employed a generic and parameterizable example problem that—although simple—captures the main characteristics of configuration problems and is comprehensible enough for analysis purposes.

The configuration and diagnosis problem. The sample problem is similar to the configuration problem described in Section 2, which consists of a *frame* with a predefined set of connection points (ports) onto which *modules* of different types can be mounted, whereby each module can be connected to exactly one of the ports and vice versa. The configuration goal is to place exactly one module of a specific type at each of the available ports. The *configuration knowledge base* consists of a set of constraints that describe legal combinations of modules that are connected to the frame. Consequently, the sample configuration problem can be parameterized by varying

- the number of ports,
- the number of available module types, and
- the number of constraints and possible solutions (i.e., the tightness of the problem).

In the experiments, we varied the tightness of the configuration problem by assigning weights to the individual module types and restricting the upper and lower bounds of the sum of the weights of all modules instances that are included in a valid configuration. Such restrictions are typical in configuration problems and are referred to as *resource* constraints. Consequently, the overall running time needed for consistency checking and solution search during the diagnostic process increases with the time the underlying constraint solver needs for finding a solution or detecting that there is none. Thus, the running time also depends on the (default) search strategy of the constraint solver. Obviously, the time needed for diagnosis only remains constant when increasing the complexity of the configuration problem itself, since an increase of problem complexity will not result in additional diagnoses nor a need for additional consistency checks as long as a solution for the problem exists.

Therefore, in the running time numbers presented below we refer to a setting where resource constraints are excluded. The set of constraints in the knowledge base for the experiments are (a) the generated faulty constraints needed for inducing the diagnoses, and (b) a parameterizable number of constraints that are not contained in any minimal diagnosis but influence the theoretical number of possible diagnoses.

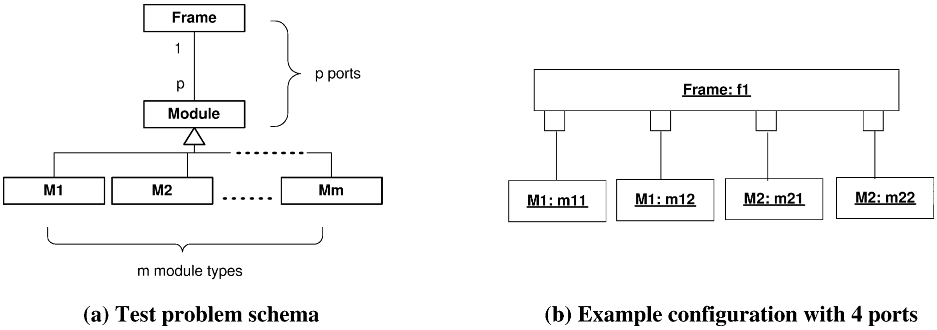
Without any constraints and given m module types and p ports, the number of possible solutions is m^p.

This example problem knowledge base is depicted in Fig. 2(a), an example configuration solution for a problem with four ports is shown in Fig. 2(b).

In order to test the diagnosis algorithms in various problem settings we developed a test case generator that both creates specific configuration problem instances as well as a corresponding diagnosis problem, where this diagnosis problem can be parameterized by varying

- the number of resulting minimal diagnoses,
- the maximum cardinality of the diagnoses, and
- the number of (positive) test examples.

The test case generator both constructs the configuration problem with its constraints as well as one partial positive example (inconsistent positive) that—in combination with



(a) Test problem schema **(b) Example configuration with 4 ports**

Fig. 2. Configuration problem for experiments.

Table 1
Varying the cardinality of diagnoses (time in seconds)

Card.	Conflict [secs]	Config. [secs]	Diagnosis [secs]	Overall [secs]
1	0.21	0.41	1.28	1.90
2	0.36	0.44	1.30	2.10
3	0.50	0.50	1.32	2.32
5	0.76	0.60	1.44	2.80
7	1.15	0.62	1.58	3.35
10	1.51	0.79	1.68	3.98

the constraints—is not accepted by the configurator. Moreover an arbitrary number of other positive examples that are consistent with the knowledge base are generated. The inconsistent positive example and the knowledge base are constructed in a way that at least one of the diagnoses has exactly the specified maximum cardinality. Moreover, the generator can be parameterized to produce *complete* test examples such that only singleton conflicts (see Section 4) will arise.

Note that we do not use negative examples in our experiments since—according to our approach—these examples are included in the knowledge base in negated form, such that only the time needed for configuration and consistency checking will slightly increase but not the time for diagnosis generation (i.e., HS-DAG generation without consistency checking).

Results. Table 1 shows the running times of our experiments in different problem settings. The overall time for diagnosing a problem can be split into

- time needed for consistency checking (i.e., solution search for the configuration problem),
- time for conflict generation, and
- diagnosis time.

The experiments were run on a standard Personal Computer (Pentium III, 256 MB RAM) and Java JDK 1.3.1., with all generated problems being successfully solved.

Table 1 shows the running times for finding a single diagnosis with varying cardinalities. The configuration problem consisted of 50 ports, 100 module types and 50 additional, non-faulty constraints, which is a reasonable number in real-world sales configuration applications. The tests were run using 10 additional (non-faulty) examples. The search problem itself is underconstrained which is a typical situation for configuration problems; decreasing the number of possible solutions will result in an increase of the configuration time. Note that in this test case, partial examples were used, which would be characteristic for an interactive debugging scenario.

We do not list outcomes for diagnosis for complete examples, which is the scenario for regression testing, i.e., the knowledge engineer checks whether former configurations are still working after maintenance activities. According to Section 4, we do not need to generate a hitting set DAG for this problem, but can instead check the knowledge

Table 2
Varying cardinality and number of diagnoses (time in seconds)

MC	AC	#diags	Conflict [secs]	Config. [secs]	Diagnosis [secs]	Overall [secs]
1	1	3	0.11	0.95	1.54	2.60
1	1	5	0.12	0.98	1.58	2.68
3	2	3	0.16	0.87	1.71	2.74
4	2	5	0.51	1.10	1.58	3.19
10	4	3	1.47	1.48	2.35	5.30
6	3	5	2.20	1.85	3.60	7.65

base constraints individually and simply compute the union of the violated constraints of all examples. In a typical debugging scenario the diagnoses will typically both contain constraints that are immediately violated by the test example and constraints whose faultiness can only be detected by propagation or solution search given partial examples.

In Table 2 we will show some excerpts from our test cases with varying numbers of diagnoses and their cardinalities. The configuration problem is the same as for the first set of experiments. Within the table we also show the maximum (MC) and average cardinality (AC) of the computed diagnoses.

6.3. Analysis

JCONDIAG shows promising running times for the described problems as an additional tool that supports the knowledge engineer in focusing his debugging efforts for knowledge bases of realistic problem sizes. The system guides the user in detecting errors of certain types (e.g., overly restrictive or contradictory constraints) that cannot be found using conventional debugging and trace facilities. Note that syntax errors or inconsistent product models, e.g., constraints on non-existing ports or component types, are detected by the underlying configurator software using conventional techniques.

The main influences on the overall diagnosis time are—besides size and number of the diagnoses—the configuration problem itself and the number and size of the test cases: each consistency check for an example means starting a search for a solution to the configuration problem because we allow the user to specify partial examples. We have limited the listed results to cases where the outcomes of the diagnostic process have a comprehensible size: listing too many diagnoses or diagnoses of very high cardinality will not help the engineer very much if no additional hints or error probabilities are attached to the constraints. In practical settings, the resulting diagnoses have a cardinality that is small enough so that an engineer who knows which portions of the knowledge base were recently changed can still discriminate between them.

In fact, an interesting outcome of our experiments is that in typical declarative configuration knowledge bases there are only few interdependencies among constraints, i.e., the size of the minimal conflicts is typically very small (up to three or four constraints). This results in the effect that the number of returned diagnoses is limited and still comprehensible for the knowledge engineer such that further discrimination among the diagnoses is easily possible even without the computational limitations discussed in the previous paragraph.

The diagnosis library is currently implemented independently from the configuration reasoner, and further running time improvements could be reached by integrating it with the core solver. In the current version JCONDIAG can only use the defined API of *JConfigurator* which causes additional computational overhead.

7. Related work

While model-based diagnosis (MBD) techniques were originally developed for finding explanations of unexpected behavior in physical devices such as, e.g., electronic circuits [23], the applicability of the same framework for *debugging software systems* has also been shown for logic programs, declarative knowledge bases as well as—more recently—for functional and imperative languages [5,13,29,31].

In [13], model-based diagnosis was applied to detect and locate faults in large VHDL programs. VHDL is an imperative and widely used hardware description language for the development of large hardware designs. In this approach, the diagnosis model is automatically generated from VHDL code, where an appropriate high-level abstract representation of the problem is chosen that is sufficiently detailed for reducing fault localization costs compared with manual debugging but allows fast enough runtimes for practical problem settings. Abstraction occurs by mapping to a dependency-based model of the system, thus ignoring specific signal values. Diagnosis is performed by analyzing discrepancies between the expected program behavior (given by its specification) and the observed behavior given in terms of a waveform trace from simulation runs; the set of diagnosable components is given by the concurrent statements in the VHDL code. Compared with the approach presented in this paper, one key feature lies in the fact that the diagnosis model can be automatically generated from the VHDL source code and the constraints in the knowledge base, respectively, and model construction causes no additional efforts. It is well known that one of the crucial factors when applying MBD lies in finding the right *model* (see, e.g., [27]). The requirements for such models include correctness, a guarantee that no diagnoses are excluded due to the abstractions, and that ability to compute diagnoses within a reasonable time-frame for practical problem settings. The insertion of error probabilities for components based on their complexity for focusing on relevant diagnoses in [13] is a topic of further work for our approach of diagnosing configurator knowledge bases. Finally, integration of the diagnosis system into the knowledge engineer's original development toolkit is a prerequisite for acceptance of the additional debugging support made available by MBD technology.

Recent work extends the approach of debugging imperative languages with MBD techniques from special-purpose languages to Java programs. In [28,31,32] the dependency-based approach from [13] is extended to a value-based model of program execution which allows for automatic detection of certain error types in Java programs. In contrast to traditional debugging techniques developed in the software engineering community, this approach requires no additional dependencies or annotations for analysis purposes. A value-based approach allows the user to perform fine-grained diagnoses and (partial) identification of repair actions, but the approach results in a higher complexity of the diagnosis task; the authors therefore propose using a hybrid approach where a dependency-based model is used for initial focusing before the value-based model is applied. In particular, the techniques used for mutation diagnosis in [32] and their principal applicability for the computation of repair actions (replacements) will be an interesting piece of future work for our approach: Given the restricted nature of the constraints of configurator knowledge bases, computation of a limited set of repair proposals seems to be feasible with respect to the model complexity compared to general program statements.

In [5], a framework for model-based diagnosis of logic programs was developed using expected and unexpected query results to identify incorrect clauses. When applying the MBD approach, the user of a debugging tool is supposed to know the correct program behavior and answer the questions posed by the debugger correctly. The diagnostic reasoner can consequently use this knowledge in order to distinguish the different error candidates and help the user in focusing his/her efforts. Their work was continued and improved by Bond et al. [2,3] who embedded the diagnosis of logic programs and the concept of Algorithmic Program Debugging [25] in a common underlying framework. Although their general framework is similar to ours, it differs in using queries for logic programs instead of automatic consistency checks as we do for configurations.

A way to diagnose faults in logic programs using a semantics based on SLDNF-resolution was described in [22]. This work led to the development of the non-monotonic reasoning system REVISE [6]. This system showed the applicability of extended logic programming to solving model-based diagnosis problems. The basic logic programming framework was extended with explicit negation and integrity constraints, revision of assumptions is used to remove contradictions from the knowledge base. Explicit negation was introduced to overcome the insufficient treatment of negation as finite failure in Prolog-like systems. With well-defined semantics for the negation construct it was possible to express the default assumption that the components work correctly. In principle, this framework could be extended to achieve results similar to our approach, although one of our main goals was to remain within first order predicate logic with standard semantics, so that generic applicability to declarative configuration knowledge bases could be retained.

The idea to use declarative diagnosis instead of execution tracing was applied to Constraint Logic Programs in [4]. In logic program diagnosis it is assumed that the user is able to answer queries concerning whether or not an intermediate result corresponds to the expectations, which may be a difficult task even for very simple programs. To circumvent this the authors extended the diagnosis framework for logic programs by introducing assertions which can be formulated by the user and are used to indirectly answer the queries posed by the debugger. The approach gives no precise information about the particular assertion language used; they could be expressed as additional constraints or small constraint programs (which in turn may be faulty). In the case of diagnosing configurator knowledge bases however, the definition of examples by listing key components and determining whether it should be a negative or positive configuration example is a comparatively simple task for the knowledge engineer.

In [1], model-based diagnosis was used for finding solutions to overconstrained constraint satisfaction problems. The search process 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 was given in [15]. 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, ab^{del} and ab^{ins}, 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 (i.e., a set of assumptions about database tuples being in either the ab^{del} or ab^{ins} state) serves directly as a specification for the actions that will bring the database into a state consistent with the violated constraints: tuples in ab^{del} need to be deleted, tuples in ab^{ins} need to be inserted into the database. However, the computation of diagnoses is local to 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. The authors present some ideas for an iterative search process to deal with this problem. There is limited correspondence with our work. 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). The database diagnosis approach, however, involves an implicit closure assumption on the database (reasoning about the abnormality of tuples not contained in the relations), whereas it would make no sense to include completeness axioms concerning the set SRS in all CREQ-diagnoses, since SRS would not be expected to completely enumerate all needed components and connections. In addition, the repair actions on a tuple level (which corresponds to adding components, connections, or attribute values to a system requirement specification) have no corresponding application scenario in our domain.

In recent years, alternative algorithms to Reiter's hitting set algorithm for the computation of diagnoses have been developed. In [8] and [30], structural properties of the system to be diagnosed are exploited to speed up the reasoning process. These approaches apply to tree-structured systems where the input and output behavior of diagnosable components can be described in terms of functions. Diagnoses can be computed efficiently by propagating input values and observations forward and backward [30] in the system. The direct applicability of this approach to diagnosis of configuration knowledge bases seems to be limited, given the cyclic structure of dependencies in the knowledge base. However, analysis of the potential to exploit knowledge about the underlying constraint network (consisting of n-ary constraints) will be a part of our future work. In particular it seems to be very promising to explore rewriting techniques to transform configuration knowledge bases into tree-structured systems.

An approach extending the results from the DRUM-II system is described in [14], where diagnosis is based on general purpose automated theorem proving techniques. In the presented work, the underlying logical models are used directly for diagnosis in an iterative repair approach. However, this approach relies on specialized proof procedures, whereas in our work the goal was to diagnose systems that are built with domain-independent commercial constraint solving software which is optimized for efficient solution search.

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. 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 and provided experimental data for our implementation of the algorithm that show the suitability of integrating the diagnosis approach in commercial configurator development environments.

We have also 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 robustness as well as in terms of generality and ease of application.

References

[1] R.R. Bakker, F. Dikker, F. Tempelman, P.M. Wognum, Diagnosing and solving over-determined constraint satisfaction problems, in: Proc. IJCAI-93, Chambéry, France, Morgan Kaufmann, San Mateo, CA, 1993, pp. 276–281.

[2] G.W. Bond, Top-down consistency based diagnosis, in: Proc. DX-96 Workshop on Principles of Diagnosis, Val Morin, Canada, 1996.

[3] G.W. Bond, B. Pagurek, A critical analysis of "Model Based Diagnoses meets Error Diagnosis in Logical Programs", Technical Report SCE-94-15, Carleton University, Department of Systems and Computer Engineering, Ottawa, ON, 1994.

[4] J. Boye, W. Drabent, J. Maluszynski, Declarative diagnosis of constraint programs: An assertion-based approach, in: Proc. AADEBUG-1997, 1997, pp. 123–140.

[5] L. Console, G.E. Friedrich, D.T. Dupré, Model-based diagnosis meets error diagnosis in logic programs, in: Proc. IJCAI-93, Chambéry, France, Morgan Kaufmann, San Mateo, CA, 1993, pp. 1494–1499.

[6] C.V. Damásio, L.M. Pereira, M. Schroeder, REVISE: Logic programming and diagnosis, in: Proc. Fourth Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-97), Springer, Berlin, 1997, pp. 354–363.

[7] N. de Siqueira, J.-F. Puget, Explanation-based generalization of failures, in: Proc. ECAI-88, Munich, Germany, Pitman, London, 1998, pp. 339–344.

[8] Y. El Fattah, R. Dechter, Diagnosing tree-decomposable circuits, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 1742–1748.

[9] A. Felfernig, G. Friedrich, D. Jannach, UML as domain specific language for the construction of knowledge-based configuration systems, Internat. J. Software Engrg. Knowledge Engrg. 10 (4) (2000) 449–469.

[10] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Hierarchical diagnosis of large configurator knowledge bases, in: Proc. 24th German/9th Austrian Conference on Artificial Intelligence (KI-2001), Vienna, Austria, in: Lecture Notes in Artificial Intelligence, Vol. 2174, Springer, Berlin, 2001, pp. 185–197.

[11] A. Felfernig, G. Friedrich, D. Jannach, M. Zanker, Web-based configuration of virtual private networks with multiple suppliers, in: J. Gero (Ed.), Proc. 7th Internat. Conference on Artificial Intelligence in Design (AID-02), Cambridge, UK, Kluwer, Dordrecht, 2002, pp. 41–62.

[12] G. Fleischanderl, G.E. Friedrich, A. Haselboeck, H. Schreiner, M. Stumptner, Configuring large systems using generative constraint satisfaction, IEEE Intelligent Systems 13 (4) (1998) 59–68.

[13] G.E. Friedrich, M. Stumptner, F. Wotawa, Model-based diagnosis of hardware designs, Artificial Intelligence 111 (2) (1999) 3–39.

[14] P. Fröhlich, W. Nejdl, A static model-based engine for model-based reasoning, in: Proc. IJCAI-97, Nagoya, Japan, 1997.

[15] M. Gertz, U.W. Lipeck, A diagnostic approach to repairing constraint violations in databases, in: Proc. DX-95 Workshop on Principles of Diagnosis, Goslar, 1995.

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

[17] U. Junker, Preference programming for configuration, in: Proc. IJCAI-01 Workshop on Configuration, Seattle, WA, 2001, pp. 50–57.

[18] U. Junker, QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms, in: Proc. IJCAI-01, Workshop on Modelling and Solving Problems with Constraints, Seattle, WA, 2001.

[19] D. Mailharro, A classification and constraint-based framework for configuration, Artificial Intel. Engrg. Design Anal. Manufact. 12 (4) (1998).

[20] D.L. McGuinness, J.R. Wright, Conceptual modelling for configuration: A description logic-based approach, Artificial Intelligence for Engrg. Design Anal. Manufact. 12 (4) (1998).

[21] S. Mittal, F. Frayman, Towards a generic model of configuration tasks, in: Proc. IJCAI-89, Detroit, MI, 1989, pp. 1395–1401.

[22] L.M. Pereira, C.V. Damásio, J.J. Alferes, Debugging by diagnosing assumptions, in: Proc. AADEBUG-93, Linköping, Sweden, 1993.

[23] R. Reiter, A theory of diagnosis from first principles, Artificial Intelligence 32 (1) (1987) 57–95.

[24] J.T. Runkel, A. Balkany, W.P. Birmingham, Generating non-brittle configuration-design tools, in: Proc. Artificial Intelligence in Design 94, Lausanne, Kluwer Academic, Dordrecht, 1994, pp. 183–200.

[25] E. Shapiro, Algorithmic Program Debugging, MIT Press, Cambridge, MA, 1983.

[26] T. Soininen, J. Tiihonen, T. Männistö, R. Sulonen, Towards a general ontology of configuration, AI Engineering Design Analysis and Manufacturing 12 (4) (1998) 357–372.

[27] P. Struss, What's in SD? Towards a theory of modeling for diagnosis, in: W. Harmscher, L. Console, J. de Kleer (Eds.), Readings in Model-Based Diagnosis, Morgan Kaufmann, San Mateo, CA, 1992.

[28] M. Stumptner, F. Wotawa, VHDLDIAG+: Value-level diagnosis of VHDL programs, in: Proc. DX-98 Workshop on Principles of Diagnosis, Cape Cod, 1998.

[29] M. Stumptner, F. Wotawa, Debugging functional programs, in: Proc. IJCAI-99, Stockholm, Sweden, Morgan Kaufmann, San Mateo, CA, 1999, pp. 1074–1079.

[30] M. Stumptner, F. Wotawa, Diagnosing tree-structured systems, Artificial Intelligence 127 (1) (2001) 1–29.

[31] C. Mateis, M. Stumptner, F. Wotawa, Modeling Java programs for diagnosis, in: Proc. 14th European Conference on Artificial Intelligence, Berlin, Germany, IOS Press, 2000, pp. 171–175.

[32] F. Wotawa, On the relationship between model-based debugging and program slicing, Artificial Intelligence 135 (1–2) (2002) 125–143.