Consistency based diagnosis of configuration knowledge-bases

Alexander Felfernig[*], Gerhard Friedrich[*], Dietmar Jannach[*], Markus Stumptner[+1]
[+]Technische Universität Wien
Institut für Informationssysteme
Abteilung für Datenbanken und Expertensysteme
Paniglgasse 16
A-1040 Wien, Austria

[*]Universität Klagenfurt
Institut für Wirtschaftsinformatik und Anwendungssysteme
Produktionsinformatik
Universitätsstraße 65-67
A-9020 Klagenfurt, Austria

Abstract

Configuration problems are a driving application area for declarative knowledge representation that 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. The examples used for testing are combined to identify faulty chunks of knowledge. Starting from a clear definition of diagnosis in the configuration domain we develop an algorithm based on conflicts and exploit the properties of positive examples to reduce consistency checks. Our framework is general enough for its straightforward adaptation to diagnosing customer requirements. Given a validated knowledge base our approach can then be used to identify unachievable conditions during configuration sessions.

Introduction

Knowledge-based configuration systems have a long history as a successful AI application area and today form the foundation for a thriving industry. Comparable to the switch from rule-based to model-based diagnosis as the state of the art since the mid-1980s, configuration systems have likewise progressed from their successful rule-based origins [BO89] to the use of higher level representations such as various forms of constraint satisfaction [SH+98], description logics [MW98], or functional reasoning [RBB94], due to the significant advantages offered: more concise representation, higher maintainability, and more flexible reasoning.

The increased use of knowledge-based configurators as well as the increasingly complex tasks tackled by such systems ultimately lead to both the knowledge bases and the resulting configurations becoming larger (i.e., comprising more component types) and more complex constraints (i.e., representing more involved component behavior). As a result, the user of a configuration tool, whether an engineer working on maintaining the knowledge base, or an end user producing actual configurations, is increasingly challenged, when the configuration system does not behave as expected, to find out what is actually wrong. For example, the configuration process could be aborted with a message that some inconsistency has arisen, i.e., it is not possible to produce a finished, working configuration. The configurator may find that all attempts to find a solution end with a constraint violation, or that a finished configuration leaves some parts unaccounted for. Ultimately, the inconsistency has to be caused either by an incorrect knowledge base or by unachievable requirements. In this paper, we will focus primarily on the engineer working on the maintenance of a large configuration knowledge base, searching for failures performing test configurations.

The focus on checking the knowledge base is reasonable since after the specification of knowledge bases, one of the most important steps for real world applications is (analog to traditional software development) the validation phase. It will also be necessary to validate the knowledge base, which may contain hundreds of component types and be used for problems with thousands of components, again later in its lifecycle whenever it is updated to meet new or altered application requirements (e.g., new component types or new regulations).

[1] Authors appear in alphabetical order.

Using a declarative approach we have the possibility to test our configuration system with positive and negative examples. This means that positive configuration examples should be accepted by the configurator whereas negative examples should be rejected. Note that the examples can be partial configurations, i.e., it may be that a complete configuration has to be computed to determine whether the example is consistent with the knowledge base or not. The examples therefore play a role much like what is called a test case in software engineering: they provide an input so the generated output can be compared to the tester's expectations. For simplicity, we will still refer to them as examples throughout this paper.

Our general notion of support for the validating engineer is that, once a test has failed, diagnosis can be used to locate the parts of the knowledge base responsible for the failure. Given the usual nature of configuration knowledge, 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 for configuring that we will briefly discuss later is the support of the actual end user. This support is necessary where a configuration problem is specified that, even though the knowledge base is correct, is unsolvable, 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 configuration terminology used in the rest of the paper. We then formalize configuration in terms of a domain theory and system and use the formalization to express the notion of model-based diagnosis as it applies to the configuration domain. (Proofs are omitted for space reasons.) We give an algorithm for computing diagnoses based on positive and negative example sets, and explore the influence of different types of examples. Finally, we examine the "reverse" use of diagnosis for identifying faults in requirements.

A Configuration Example

For the introduction of our concepts we use a small part of a configuration knowledge base from the area of telecommunications, in particular telephone switching systems. 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.

The most numerous components in such systems are switching modules which implement flexible connections of input and output lines. Modules are inserted into frames, which provide a higher (physical and functional) level of organization. The physical insertion of a module into a frame, as well as any other kind of connection between components, are modelled via *ports*. In our example, a frame possesses 4 ports whereas modules just have one port. In addition to ports, attributes are used to model properties of components, e.g., each controller module has a version number. Each attribute is further characterized by its domain. In this exposition we limit ourselves to discrete, enumerable domains.

We distinguish between the following component types:

types = {frame, digital_module_version1, digital_module_version2, control_module_version1, control_module_version2}.

Each of the types above possesses a set of ports, which are listed using the function *ports*:

ports(frame) = {control, slot1, slot2, slot3}.
ports(digital_module_version1) = {mounted_on}.
ports(digital_module_version2) = {mounted_on}.
ports(control_module_version1) = {mounted_on}.
ports(control_module_version2) = {mounted_on}.

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 represented by a literal *conn(c1,p1,c2,p2)* where *p1 (p2)* is a port of component *c1 (c2)*. An attribute value *v* assigned to attribute *a* of component *c* is represented by a literal *val(c,a,v).* In addition, the predicate *unconn(c,p)* is used as a shorthand to express that port *p* of component *c* is not connected to any other port.

Attributes and connections between components must obey the following application specific constraints:

If there is a digital module of **version 1** connected to a frame then a control module of **version 1 or version 2** must be connected to the control port.

Constraint C1:
$\forall$ *M,F: type(M, digital_module_version1)* $\wedge$
conn(M,mounted_on,F,_) $\wedge$ *type(F,frame)*
$\Rightarrow \exists$ *C:conn(C,mounted_on,F,control)* $\wedge$
(type(C,control_module_version1) $\vee$
type(C,control_module_version2)).

If there is a digital module of **version 2** connected to a frame then a control module of **version 2** must be connected to the control port.

Constraint C2:
$\forall$ *M,F: type(M, digital_module_version2)* $\wedge$
conn(M,mounted_on,F,_) $\wedge$ *type(F,frame)*
$\Rightarrow \exists$ *C: conn(C,mounted_on,F,control)* $\wedge$
type(C,control_module_version2).

The control port of a frame can only be connected to a control module of version1.

Constraint C3:
$\forall$ F,C: type(F,frame) $\wedge$ conn(F,control,C,_)
$\Rightarrow$ type(C,control_module_version1).

As it turns out, this constraint is faulty because it is too strong. This constellation could have come about, because digital modules of version 2 were newly introduced to the knowledge base, and *C3* was not altered to accommodate them. The correct version of this constraint would also permit

control modules of version2 as allowed connections to the control port of a frame, i.e.,

Constraint $C3_{ok}$*:*
$\forall$ *F,C: type(F,frame)* $\wedge$ *conn(F,control,C,_)* $\Rightarrow$ *type(C,control_module_version1)* $\vee$ *type(C,control_module_version2).*

In the following we denote the faulty knowledge base by KB_{faulty} = {*C1, C2, C3*} and the correct one by $KB_{correct}$= {*C1, C2,* $C3_{ok}$}

In addition, we have to model some application independent configuration constraints specifying that connections are symmetric,

Constraint C4:
$\forall$ *C1, C2, P1, P2 : conn(C1,P1,C2,P2)* $\Rightarrow$ *conn(C2,P2,C1,P1).*

that a port can only be connected to one other port:

Constraint C5:
$\forall$ *C1, C2, C3, P1, P2, P3 : conn(C1,P1,C2,P2)* $\wedge$ *conn(C1,P1,C3,P3)* $\Rightarrow$ *C2=C3* $\wedge$ *P2=P3.*

and that components have a unique type:

Constraint C6:
$\forall$ *C, T1, T2 : type(C,T1)* $\wedge$ *type(C,T2)* $\Rightarrow$ *T1=T2.*

We denote these constraints by C_{Conn} = *{C4,C5,C6}* and employ the unique name assumption.

In our application domain, we define in addition that every attribute of a component has a unique value.

As described in the introduction, the typical situation envisioned for using diagnosis on this knowledge base is that of validation. First, having the correct configuration model in mind, the test engineer provides a positive example, a frame with two digital modules plugged in where one digital module has version 1 and the other version 2. We denote such a positive example as e^+. More formally,

e^+={$\exists$ *F,D1,D2*: *type*(*F,frame*).
type(*D1,digital_module_version1*).
type(*D2,digital_module,version2*).
conn(*F,slot1,D1,mounted_on*).
conn(*F,slot2,D2,mounted_on*).}

Note that examples can be either 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.

A negative example would be a frame where two digital modules are plugged in with version 1 and 2 as it was the case in e^+ but in addition a control module of version 1 is also connected to the frame. We denote such a negative example as e^-, where such an example should be inconsistent with the configuration knowledge base.

e^- ={$\exists$ *F,D1,D2,C1*: *type*(*F,frame*).
type(*D1,digital_module_version1*).
type(*D2,digital_module,version2*).
type(*C1,control_module_version1*).
conn(*F,slot1,D1,mounted_on*).
conn(*F,slot2,D2,mounted_on*).
conn(*F,control,C1,mounted_on*).}

Testing the knowledge base with e^- results in the expected contradiction, i.e., $KB_{faulty} \cup e^- \cup C_{Conn}$ is inconsistent. However, $KB_{faulty} \cup e^+ \cup C_{Conn}$ is also inconsistent which is not intended. The question is which of the application specific constraints {*C1,C2,C3*} are faulty.

As we will see the question can be answered by adopting a consistency-based diagnosis formalism. The constraints *C1*, *C2*, and *C3* are then 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*} $\cup e^+ \cup C_{Conn}$ is contradictory. It follows that *C2* or *C3* has to be canceled in order to restore consistency, i.e., {*C1, C2*} $\cup e^+ \cup C_{Conn}$ is consistent and {*C1, C3*} $\cup e^+ \cup C_{Conn}$ is consistent. However, if we employ the negative example we recognize that {*C1, C3*} $\cup e^- \cup C_{Conn}$ 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^-. For accepting "*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*} $\cup e^- \cup EX \cup C_{Conn}$ must be inconsistent (note that {*C1, C3*} $\cup EX \cup C_{Conn}$ must be consistent) and therefore {*C1, C3*} $\cup EX \cup C_{Conn} \models \neg e^-$, 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*} $\cup EX \cup e^+ \cup C_{Conn}$ is consistent. Therefore, it must hold that {*C1, C3*} $\cup EX \cup e^+ \cup \neg e^- \cup C_{Conn}$ is consistent which is not the case for our example: $e^+ \cup \neg e^-$ implies that there must not be a control_module_version1 connected to the control slot of a frame whereas {*C1, C3*} $\cup e^+ \cup C_{Conn}$ requires that this control slot must be connected to a control_module_version1. 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^- as desired, i.e., "*C3* is faulty" can be accepted as a diagnosis. Therefore, "*C3* is faulty" is the only single fault diagnosis for our example.

These concepts will be defined and generalized in the following sections. In particular, we will formally describe the configuration task and the mapping from configuration test cases to diagnosis problems. Using the consistency-based diagnosis framework will also give us the ability to identify faults given sets of multiple examples (negative and positive) and diagnose knowledge bases with multiple faults.

Defining Configuration and Diagnosis

In practice, configurations are built from a catalog of component types. This catalog (which in industrial applications often corresponds to an actual catalog of available component types), is fixed for a given class of problems, e.g., it contains frames and modules for our previous example, offices, passageways, and stairwells when configuring buildings, or motherboards, controllers, and memory for configuring computers.

The catalog specifies the basic properties and the set of logical or physical connections that can be established between different components. We therefore assume the existence of a domain description *DD* that contains the definition of a set of types.

To define the properties and connections associated with the types in the catalog, *DD* must describe functions *ports, attributes,* and *dom*. The function *ports* maps each type to the set of constants that represent the ports provided by components of that type, i.e., the possible connections for each component type. Examples for both were given in the introductory example. The function *attributes* defines the set of attributes, and the function *dom* defines the domain of an attribute for a particular type. The rest of the domain description specifies the behavior of components, e.g., its sentences describe valid value assignments to ports as well as any other conditions that influence the correctness of the configuration.

In addition to the catalog and its domain specific restrictions, an individual configuration problem usually has to be solved according to some set of user requirements. The requirements are not restricted syntactically; they can consist of special cases, describe initial partial configurations, legacy components that should be incorporated, in fact generally any desired functionality.

An individual configuration consists of a set of components, a listing of the connections between the components, and their attribute values.

Definition: Configuration Problem

In general we assume a configuration problem is described by two sets of logical sentences: *DD*, the domain description, and *SRS*, the particular system requirements which specify an individual configuration problem instance. A configuration then is a triple *(COMPS,CONNS,ATTRS)* of components, connections, and attributes. *COMPS* is a set of ground literals *type(c,t)* , *CONNS* is a set of ground literals *conn(c1,p1,c2,p2)* where *c1,c2* are components and *p1,p2* are ports, and *ATTRS* is a set of ground literals of the form *val(c,a,v)* where *c* is a component, *a* is an attribute name, and *v* is the value of the attribute. ❑

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 $\{C1,C2,C3_{ok}\} \cup C_{Conn}$ and the set e^+ is a particular set of system requirements. (Note that although this example is strongly simplified, it does often occur in practice that a system is specified by explicitly listing the set of required key components.) A configuration for this problem would be given by $CONF_1 = (COMPS_1 , CONNS_1, ATTRS_1)$, where

$COMPS_1$={*type*(*f,frame*). *type*(*d1,digital_module_version1*).
type(*d2,digital_module,version2*)
type(*c1,control_module_version2*).},
$CONNS_1$={*conn*(*f,slot1,d1,mounted_on*),
conn(*f,slot2,d2,mounted_on*),
conn(*f,control,c1,mounted_on*).},
$ATTRS_1$={}. ❑

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)*, a configuration *(COMPS,CONNS,ATTRS)* is consistent iff $DD \cup SRS \cup COMPS \cup CONNS \cup ATTRS$ 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, it is necessary that a configuration explicitly includes all needed components (as well as their connections and attribute values), in order to manufacture and assemble a correctly functioning system. We need to introduce an explicit formula 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 ease of notation we write *Comps*(*t,COMPS*) for the set of all components of type *t* mentioned in *COMPS*, i.e., *Comps*(*t,COMPS*)={*c* | *type*(*c,t*) ∈ *COMPS*} and *Types*(*COMPS*) for the set of all types *t* mentioned in *COMPS*, i.e., *Types(COMPS*)={*t* | *type*(*c,t*) ∈ *COMPS*}.

We describe the fact that a configuration uses not more than a given set of components *Comps*(*t,COMPS*) of a type *t* ∈ *Types(COMPS)* by the literal *complete*(*COMPS*) and the addition of

complete(*COMPS*) ⇔ (∀ *C,T: type*(*C,T*) ⇒ *T* ∈ *Types*(*COMPS*) ∧ *C* ∈ *Comps*(*T,COMPS*)).

We will write $COMPS^{comp}$ to denote *COMPS* ∪ {*complete*(*COMPS*)}.

Similar to the completion of the type literals, we have to make sure that all conn facts are included in *CONNS*. We use the name

CCONNS = {∀ *X,Y*: ¬ *conn*(*c,p,X,Y*) | *type*(*c,t*) ∈ *COMPS* ∧ *p* ∈ *ports*(*t*) ∧ *conn*(*c,p,_,_*) ∉ *CONNS* ∧ *conn*(*_,_,c,p*) ∉ *CONNS*}

and write $CONNS^{comp}$ to denote $CONNS \cup CCONNS$.

Given a particular configuration according to the above definition, the most important requirement is that it satisfies the domain description, and if so, then, in concordance with the criteria specified in the example section, we do not want it to contain any components or connections which are not required by the domain description. Note that this merely means that there are no spurious parts in the solution; it does not imply that a valid configuration is unique or has to have minimum cost.

For the sake of brevity, we will define *CONF* to refer to a configuration consisting of components *COMPS*, connections *CONNS*, and attributes *ATTRS*, and

$CONF = COMPS \cup CONNS \cup ATTRS$
$CONF^{comp} = COMPS^{comp} \cup CONNS^{comp} \cup ATTRS^{comp}$

where $ATTRS^{comp}$ is the assumption made in the previous section that the value assignment for attributes is both required (depending on the type) and unique for each component, i.e.,

$\forall C,T,A : type(C,T) \in COMPS \wedge A \in attributes(T)$
$\Rightarrow \exists V : V \in dom(T,A) \wedge val(C,A,V).$

$\forall C1,A1,V1,V2 : val(C1,A1,V1) \wedge val(C1,A1,V2)$
$\Rightarrow (\exists T : type(C1,T) \in COMPS \wedge A1 \in attributes(T)) \wedge$
$(V1=V2).$

Definition: Valid and Irreducible Configuration

Let (*DD, SRS*) be a configuration problem. A configuration *CONF* is valid iff $DD \cup SRS \cup CONF^{comp}$ is satisfiable. *CONF* is irreducible if there exists no other valid configuration $CONF^{sub}$ such that $CONF^{sub} \subset CONF$. ❑

Having finished our definition of the configuration task, it is now relatively easy to express our goal, finding the sources for inconsistencies in configurations, 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 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 *type*, *conn*, and *val* literals. In case these examples are complete special completeness axioms can 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 [FFH+98].

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* 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 the 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.

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.

Computing Diagnoses

The above definitions allow us to employ the standard algorithms available for consistency-based diagnosis, with appropriate extensions for the domain. In particular, we use Reiter's Hitting Set algorithm [Rei87] 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 [Rei87, GSW89], i.e., a node *n* is labeled by a conflict *CS*(*n*), 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 $DD\text{-}H(n) \cup NE$ during the DAG-generation. The reason for introducing the label $CE(n)$ is the fact that any e^+ that is consistent with a particular $DD\text{-}H(n) \cup NE$ 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 for all direct predecessors 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)

Input: DD, E^+, E^-
Output: 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 $\{H(n) \mid n$ is a node of D labeled by $ok\}$

Note that, in the case that e^+ is a partial configuration, each call to TP (since it involves checking the consistency of $\forall e^+ \in E^+: DD–H(n) \cup e^+ \cup NE$) corresponds to the extension of e^+ to a complete configuration.

In order to guide the breadth first search, we have to define a preference criterion between diagnoses. Obviously, we will prefer those diagnoses which result in minimal changes for the existing knowledge base. The removal of parts of the knowledge base is given by the diagnosis itself, but this does not tell us anything about the size of the extension. Since we avoid to compute the extension itself here, we need a different measure. It is natural to use the negative examples for this purpose, since maintaining the inconsistency with the negative examples was the reason for introducing the notion of extensions in the first place.

Definition: Ordering among diagnoses

A diagnosis S is preferred to S' ($S < S'$) iff $|S| < |S'|$ or, if $|S| = |S'|$ then the diagnosis is preferred where the number of negative examples $e^- \in E^-$ that are consistent with $DD–S$ is minimized. ❑

Note that this preference criterion is one among many that could in principle be chosen. It is selected here because of its simplicity and effective computability and because of its direct correspondence to the application context, namely, those diagnoses are preferred where the smaller number of clauses are regarded as faulty. More complex criteria cannot be discussed here for space reasons.

Complete and 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 *type*, *conn*, and *val* predicates, then any minimal conflict set induced by e^+ for (DD,E^+,E^-) is a singleton.

The practical implications of this result 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^+.

Diagnosing Requirements

Even once the knowledge base has been tested and found correct, diagnosis can still play a significant role in the configuration process, but the scenario has changed. Instead of an engineer testing an altered (extended or updated) knowledge base, we are now dealing with an end user (e.g., customer or sales rep) who is using the tested (and 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 (although it does not, of course, fit our earlier characterization of examples as consisting mostly just of *type*, *conn*, and *val* literals).

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 ($SRS,\{DD\},\{\}$).

Related Work

[CFT+93] develop a frame work for model-based diagnosis of logic programs using expected and unexpected query results to identify incorrect clauses, a line of work later continued by Bond [Bon96]. Their framework is similar to ours, but differs in using queries instead of checking consistency as we do for configurations, and their use of horn clauses (as is usual in logic programming) versus our use of general clauses. [Bond96] embedded the diagnosis of logic programs and the concept of Algorithmic Program Debugging [Sha83] in a common underlying framework. 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) [SW98].

In [BDTW93], model-based diagnosis is used for finding solutions for overconstrained 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 [GL95]. 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. 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.

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 [FS98], 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.

Acknowledgement

The authors would like to thank Daniele Theseider-Dupré for his valuable comments.

Bibliography

[BDTW93] R.R. Bakker and F. Dikker and F. Tempelman and P.M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In Proceedings IJCAI, pages 276–281, Chambery, August 1993.

[BO89] Virginia E. Barker and Dennis E. O'Connor. Expert systems for configuration at Digital: XCON and beyond. Comm. ACM, 32 (3): 298–318, 1989.

[Bon96] Gregory W. Bond. Top-down consistency based diagnosis. In Proceedings DX'96 Workshop, Val Morin, Canada, 1996.

[CFT+93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In Proceedings IJCAI, pages 1494–1499, Chambery, August 1993.

[FFH+98] Gerhard Fleischanderl, Gerhard E. Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. IEEE Intelligent Systems, Vol. 13, No. 4, July/August 1998, pp. 59–68.

[FS98] Gerhard Friedrich and Markus Stumptner, Consistency-Based Configuration, University of Klagenfurt, Technical Report KLU-IFI-98-5, 1998.

[GL95] Michael Gertz and Udo W. Lipeck. A Diagnostic Approach to Repairing Constraint Violations in Databases. In Proceedings DX'95 Workshop, Goslar, October 1995.

[GSW89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. Artificial Intelligence, 41(1):79–88, 1989.

[MW98] Deborah L. McGuinness and John R. Wright. Conceptual Modelling for Configuration: A Description Logic-based Approach. AI EDAM Special Issue: Configuration Design, 12(4):333–344, 1998.

[RBB94] J. T. Runkel, A. Balkany, W. P. Birmingham. Generating non-brittle Configuration-design Tools. In Proceedings Artificial Intelligence in Design '94, pages 183–200, Lausanne, August 1994. Kluwer Academic Publisher.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. Artificial Intelligence, 32(1):57–95, 1987.

[SH+98] Markus Stumptner, Alois Haselböck, and Gerhard E. Friedrich. Generative Constraint-Based Configuration of Large Technical Systems. Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM), Special Issue: Configuration Design, 12(4):307–320, 1998.

[Sha83] Ehud Shapiro. Algorithmic Program Debugging. MIT Press, Cambridge, Massachusetts, 1983.

[SW98] Markus Stumptner and Franz Wotawa. VHDLDIAG+: Value-Level Diagnosis of VHDL Programs. In Proceedings DX'98 Workshop, Cape Cod, 1998.