Consistency-Based Configuration

Gerhard Friedrich and Markus Stumptner[1]

Abstract. Research in automated configuration is a traditional core application area for knowledge-based systems that is increasing in commercial relevance, but there exists no generally agreed basic formal representation scheme. Based on a small simplified example taken out of the knowledge base of an deployed configuration application, we describe a configuration knowledge based on an analogy to the classic consistency based diagnosis approach. A configuration problem is formally described in terms of a logic theory that depicts a part library and set of requirements, and a configuration is a set of components and connections that satisfy the library and requirements. We propose a "reference algorithm" in analogy to the standard HS-Dag diagnosis algorithm and show how CSP-based configuration representations can be expressed in terms of the consistency-based representation. The various parts of that representation possess direct counterparts in the representation language used by the commercial COCOS configuration tool.

1 Introduction

Since the beginnings of industrial use of expert systems, the automated configuration of technical systems has been an important application area for knowledge-based approaches, and has remained one of the staple applications for AI systems, with use in areas from production planning (PPS) systems over integrated business packages, to material resource planning systems as well as sales support, order processing, and sales force automation systems.

The main advantages of the knowledge-based approach can be considered to lie in the reduced development cost of configurators, the reduced maintenance costs after the configurator has been deployed, higher throughput (because of less effort spent on individual cases), and improved configuration quality (due to the ability to check the consistency of the knowledge). These advantages are the result of a long development and research history. The XCON/R1 system, a configurator for VAX computers developed in 1981, was one of the first classical rule-based expert system applications, but exhibited the long-term maintenance problems and brittleness [11] that were recognized as the main weaknesses of rule-based systems. A number of schemes were developed in the past decade to provide a high level representation for configuration problems.

A conceptual model for technical configuration was developed by Mittal in the shape of the *key component* approach [9], which lists key components that satisfy a certain functionality (e.g., a loudspeaker), and require connections to certain other components (e.g., an amplifier) to function. This was later mapped to a constraint satisfaction problem (CSP), but a pure CSP representation was found to be deficient in expressive power, as it could not express the fact that the existence of certain components could not be specified in advance. This led to the development of Dynamic CSPs [8, 4] and finally to generative CSPs [13], to represent the fact that multiple components of a given type can exist and be generated from the catalog during the configuration process. Other representations developed include the hybrid "structure-based" approach [1], representations based on Description Logics [14], and the Constructive Problem Solving approach [6]. In addition, there are a number of systems with highly specific inference methods appropriate for specific domains, e.g., [7]. So far though, there is no model that is accepted as showing the basic representational and computational assumptions that underlie these different, sometimes highly complex and specialized representations.

At the same time, there are deep similarities between the configuration and diagnosis domains. In both cases, systems are represented in terms of connected components, with declarative knowledge describing the "behavior" of the components (I/O behavior in the diagnosis case, connection behavior in the configuration case). The goal of this paper is to show the similarities and dualities between the two domains by expressing configuration along the lines of the quite concise and elegant consistency-based diagnosis paradigm [10]. We first define a simple example of a configuration problem, then provide a mapping to the consistency-based approach, and show that this mapping can be carried through even to the definition of a basic consistency-based configuration algorithm and its properties. Finally, we show the relationship from the consistency-based configuration representation to the language employed by the commercial COCOS configuration tool. The main contribution of this paper therefore is to show that it is possible, with a basic, simple and elegant suite of mechanisms, that has been successfully used in another domain for a long time, to provide a clear representational background for configuration.

2 An example configuration domain

In order to introduce our concepts we use a simplified configuration problem from the area of telecommunications, but the basic properties of the example occur in a variety of related domains, from computer systems to digital signal systems for railways. The example we use is a small part of the domain of the EWSD telephone switching system. We use a small part of the EWSD domain for presentation. There are only two basic types of components, modules and frames. Modules offer analog or digital transmission, and for digital switching modules we need controller modules. Components can possess *ports*, which are employed to model the connections between components. In our example, a frame possesses 8 ports (a real-world frame has 32) whereas modules just have one port (by which they need to be connected to a frame). In addition to ports, *attributes* are used to model properties of components, e.g., each controller module has an individual address. Each attribute is further characterized by its domain. In the exposition we limit ourselves to discrete, enumerable domains.

We introduce the foundations of our approach in first order logic, in order to facilitate a clear and precise presentation. Various reasoning methods can then be applied to implement configuration systems. By this approach the reasoning methods can be adapted according to the special properties of the problem domain on the one hand. On the other hand the foundations are independent of certain applications and general enough to hold in a wide range of configuration areas.

$types(frame, analog\_module, digital\_module, control\_module)$.

Ports of these types are listed using the function $ports$:

$ports(analog\_module) = \{mounted\_on\}$.
$ports(digital\_module) = \{mounted\_on\}$.
$ports(control\_module) = \{mounted\_on\}$.
$ports(frame) = \{slot1, \ldots, slot8, contr1, contr2\}$.

[1] Authors' address: G. Friedrich: Universität Klagenfurt, Institute for Information Technology, Universitätsstraße 65-67, A-9020 Klagenfurt, Austria, and Siemens Austria, Electronics Development Center, A-1030 Vienna. M. Stumptner: Technische Universität Wien, Institut für Informationssysteme, Paniglgasse 16, A-1140 Vienna, Austria, Email: gerhard.friedrich@ifi.uni-klu.ac.at, gerhard.e.friedrich@siemens.at, mst@dbai.tuwien.ac.at

$attrs(digital\_module) = \{sw\_v\}$.
$dom(digital\_module, sw\_v) = \{1, 2, 3\}$
$attrs(control\_module) = \{address, sw\_v\}$.
$dom(control\_module, sw\_v) = \{2, 3\}$.
$dom(control\_module, address) = \{1, \ldots, 64\}$.

We use three predicates for associating types, connections, and attributes with individual components. A type t is associated with a component c through 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 $val(c, a, v)$.

Attributes and connections between components must obey the following application-specific constraints (some only given verbally for space reasons):

C1 Digital modules must be bundled with software of version 1 or 2, controller modules with version 2 and 3.

$\forall M : type(M, digital\_module) \rightarrow val(M, sw\_v, 1) \vee val(M, sw\_v, 2)$.
$\forall M : type(M, control\_module) \rightarrow val(M, sw\_v, 2) \vee val(M, sw\_v, 3)$.

C2 A $mounted\_on$-port of a module must be connected to a slot of a frame:

$\forall M : type(M, analog\_module) \vee type(M, digital\_module) \vee$
$type(M, control\_module)$
$\rightarrow \exists F, P : type(F, frame) \wedge conn(M, mounted\_on, F, P)$.

C3 The mixing of analog and digital modules within a frame is not allowed:
$\forall F, P1, P2, M1, M2 : type(F, frame) \wedge P1 \in ports(F) \wedge P2 \in ports(F)$
$\wedge conn(F, P1, M1, mounted\_on) \wedge conn(F, P2, M2, mounted\_on)$
$\wedge type(M1, analog\_module) \wedge type(M2, digital\_module)$
$\rightarrow false$.

C4 Connections are symmetric.

C5 A port can only be connected to one other port:

C6 If there exists a slot in a frame which is connected to a digital module, then at least one of the slots contr1 and contr2 must also be connected to a control_module and the control module must be set to the appropriate address.

C7 Control modules and digital modules in a frame must have the same software version.

A simple configuration task in this domain could be: configure a system which includes the following modules: four $digital\_module$ and three $analog\_module$. This task can be easily represented with the following facts

$type(dm1, digital\_module)$. $type(dm2, digital\_module)$.
$type(dm3, digital\_module)$. $type(dm4, digital\_module)$.
$type(am1, analog\_module)$. $type(am2, analog\_module)$.
$type(am3, analog\_module)$.

which give requirements for valid configurations in one particular problem instance.

Based on this domain and problem description there are numerous valid configurations, where a valid configuration is one that satisfies the set of logic sentences. A configurations with minimal number of components is depicted below.

$type(dm1, digital\_module)$. $type(dm2, digital\_module)$.
$type(dm3, digital\_module)$. $type(dm4, digital\_module)$.
$type(am1, analog\_module)$. $type(am2, analog\_module)$.
$type(am3, analog\_module)$.
$type(f1, frame)$. $type(cm1, control\_module)$.
$conn(f1, slot1, dm1, mounted\_on)$. $conn(f1, slot2, dm2, mounted\_on)$.
$conn(f1, slot3, dm3, mounted\_on)$. $conn(f1, slot4, dm4, mounted\_on)$.
$conn(f1, contr1, cm1, mounted\_on)$.
$type(f2, frame)$.
$conn(f2, slot1, am1, mounted\_on)$. $conn(f2, slot2, am2, mounted\_on)$.
$conn(f2, slot3, am3, mounted\_on)$.
$val(dm1, sw\_v, 2)$. $val(dm2, sw\_v, 2)$.
$val(dm3, sw\_v, 2)$. $val(dm4, sw\_v, 2)$.
$val(cm1, sw\_v, 2)$. $val(cm1, address, 1)$.

Figure 1. A Configuration

All other cost optimal configurations use the same set of components and are only permutations of the connections of the depicted configuration.

In the following section we will provide a precise definition for the notion of valid configuration.

3 Definition of Configuration

We first define the concept of a configuration problem, i.e., the specification for a particular system that is to be configured. The description consists of a generic part and a problem specific part.

Definition 3.1 (Configuration Problem) *A Configuration problem is defined as a pair of sets of logical sentences (DD, SRS), where DD is the domain description and SRS is the specific requirements which are application dependent.*

In practice, configurations are built from a catalog of component types that is is fixed for a given domain, e.g., a particular company's product line of telephone exchanges or computers. This catalog specifies the basic properties and the set of logical or physical connections that can be established between different components. Therefore the domain description DD must contain the definition of a set $types$.

To define the properties and connections, DD must define functions $ports$ and $attributes$. $ports$ 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. $attributes$ 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 describes valid value assignments to ports and other conditions.

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

Definition 3.2 (Configuration) *Let (DD, SRS) be a configuration problem. A configuration is a triple $(COMPS, CONNS, ATTRS)$:*

- *$COMPS$ is a set of ground literals $type(c, t)$ where $t \in types$ and c is a Skolem constant. The type assignment is unique for a given component (we refer to this requirement as $AX1$).*
- *$CONNS$ is a set of ground literals $conn(c1, p1, c2, p2)$ where $c1, c2$ are components and $p1, p2$ are ports. The types of these components and their ports must correspond to DD, and each port can be part of a connection which must then be unique ($AX2$)*
- *$ATTRS$ is a set of ground literals of the form $val(c1, a1, v1)$ where $c1$ is a component, $a1$ is an attribute name, and $v1$ is the value of the attribute. Attribute values must be unique ($AX3$)*

The set of axioms $AX = \{AX1, AX2, AX3\}$ is assumed to be included in DD. Note that there is no innate requirement that ports *have* to be connected to some component. This information is part of individual domain descriptions.

Definition 3.3 (Consistent Configuration) *Let (DD, SRS) be a configuration problem. 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, taking only one frame (e.g., $f2$) and all its connections from Figure 1 would also be consistent configuration. As we see, for practical purposes, it is necessary that a configuration explicitly includes all needed components (as well as their connections and attributes), in order to manufacture and assemble a correctly function system. We need to introduce an explicit formula to guarantee this completeness property.

For ease of notation we write $Comps(t, COMPS)$ for the set of all components of type t mentioned in $COMPS$, i.e., $Comps(t, COMPS) = \{c | type(c, t) \in COMPS\}$.

We describe the fact that a configuration uses not more than a given set of components $Comps(t, COMPS)$ of a type $t \in types$ by the literal $complete(t, Comps(t, COMPS))$ and the formula

$complete(t, Comps(t, COMPS)) \leftrightarrow$
$(\forall C : type(C, t) \rightarrow C \in Comps(t, COMPS))$.

We denote the fact that $COMPS$ describes all allowed $type$ facts by

$$(CL1)\, UCOMPS = \{complete(t, Comps(t, COMPS)) | t \in types\}$$

We will write $\widehat{COMPS}$ to denote $COMPS \cup UCOMPS$.

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

$$(CL2)\quad UCONNS = \{\forall X, Y : \neg conn(c, p, X, Y) | \\ type(c, t) \in COMPS \wedge p \in ports(t) \wedge \\ conn(c, p, \_, \_) \notin CONNS \wedge conn(\_, \_, c, p) \notin CONNS\}$$

and write $\widehat{CONNS}$ to denote $CONNS \cup UCONNS$. Finally, the completion of attribute values is specified by

$$(CL3)\, UATTRS = \{\forall V : \neg val(c, a, V) | type(c, t) \in COMPS \wedge \\ a \in attrs(t) \wedge val(c, a, \_) \notin ATTRS\}$$

We write $\widehat{ATTRS}$ to denote $ATTRS \cup UATTRS$, and define $CL = \{CL1, CL2, CL3\}$.

Given a particular configuration according to the above definition, the most important requirement is that is satisfies the domain description and does not contain any components or components which are not required by the domain description (note that this does not imply that a valid configuration is unique or has to have minimum cost).

We will define $CONF$ to refer to a configuration consisting of components $COMPS$, connections $CONNS$, and attributes $ATTRS$, and

$$CONF = COMPS \cup CONNS \cup ATTRS$$
$$\widehat{CONF} = \widehat{COMPS} \cup \widehat{CONNS} \cup \widehat{ATTRS}$$

Definition 3.4 (Valid and Irreducible Configuration) *Let (DD, SRS) be a configuration problem. A configuration $CONF$ is valid iff $DD \cup SRS \cup \widehat{CONF}$ is satisfiable. If $CONF$ is valid, we call it* irreducible *if there exists no other valid configuration $CONF'$ such that $CONF' \subset CONF$.*

Because we use Skolem constants as component identifiers (which do not appear in the underlying theory) we have decoupled the set $COMPS$, i.e., the individual component instances, from the problem description. Therefore the validity and irreducibility of configurations is independent of a bijective renaming of these constants.

With the above definition we have defined a class of configurations which are interesting from a practical point of view, since we are interested in parsimonious configurations. In some tasks where no definite costs function for configurations exists, it is necessary to consider the generation of (all up to equality) valid configurations which are irreducible, although they may not be cost optimal. In real world settings such situations do sometimes occur, e.g., in some cases all racks should look the same as much as possible even if this means that this is not a cost optimal solution. However, in many applications it is exactly the minimal cost configurations which are of interest.

Note that for a configuration problem (DD, SRS), a valid configuration $CONF$ exists iff $DD \cup SRS \cup CL$ is satisfiable. Note that up to now valid configurations would include those which use infinitely many components. In practice, DD will of course be specified in such a way that if a valid configuration exists, it is finite. E.g., using an upper bound on the term-depth for decidability reasons ensures also the property that finite models exist (and therefore finite configurations as well) if any model exists at all. In addition, the configuration process can be aborted if a certain bound on the number of components is exceeded.

4 A simple configuration language

Based on the problem definition above the important task in order to achieve successful applications is to find a compromise between expressive power and efficiency. In particular, purely rule-oriented knowledge bases, whether based on OPS-style production rules or horn clauses, tend to be brittle. First, even simple constraints have to be expressed by multiple rules, second, reasoning strategies are encoded in the rules, thus leading to maintenance problems.

Based on our experiences in various configuration domains it is necessary to allow general clauses. (See, e.g., constraint C6 in our introductory example.) Our example clauses are presented by using the implication form as notation, made more compact by allowing both $\wedge$ and $\vee$ in the consequent and antecedent of the implication (which can be can be translated to the usual implication form). In addition, we demand that clauses are range restricted, i.e., every variable in the consequence part of a clause occurs in the antecedent as well

In the clauses we allow the predicates $type$, val, and $conn$ as well as interpreted predicates (e.g., tests for port and attribute names, equality, inequality, and use use of the functions $ports$ and $attributes$), and interpreted functions. These are defined over the specified ports, types, attributes, and attribute domains. Therefore, no new symbols (e.g., types) can be introduced as a side effect and no function symbols are allowed except for interpreted functions.

In practical configuration problems the number of components that will make up the finished configuration is rarely known a priori. What is needed is a mechanism that allows expressing information about components that are not initially specified (e.g., in SRS), but are added later as required by the domain description. We achieve this by a local weakening of the range restrictedness condition through allowing sorted existential quantification on components in the consequence part of the clauses. All other existential quantifications in the consequent (e.g., of ports) are only allowed if they can be translated into a disjunction by substituting individuals (e.g., individual port names) for the variable, e.g., see constraint **C2** above.

By introducing existential quantification the question regarding decidability becomes an important issue. As we noted, existential quantification is only allowed for components and therefore decidability could be simply enforced by limiting the number of components. The generation of valid configurations depends not only on the content of the knowledge base but also on the search strategies. In practical applications we have made the experience that it was quite easy to ensure that for each possible customer requirement, valid configurations are generated.

Another extension is the inclusion of aggregate operators. These are used to express global conditions that require the testing of properties of a larger set of components, since, the enumeration of all components on which such a global constraint must be evaluated would be tedious, error-prone, or (in cases which refer to the majority of the components in the final configuration, such as a global cost boundary), not possible beforehand, e.g., a constraint that states that the maximum traffic load of all modules in a frame (which may be up to 32 modules in a full configuration) must not exceed a certain boundary. We use a macro operator to apply a condition over sets of components and aggregate operators (such as Σ, max or $average$) to evaluate the properties of these components in combination. For space reasons we do not discuss these operators in more detail; they can be found in the full version of this paper [12].

5 Consistency-based configuration vs. diagnosis

The definitions presented so far consider configuration as finding a consistent theory that specifies a set of components, their types and attribute values, and the connections between those components. Not surprisingly, this task is related to other consistency-based reasoning tasks like consistency-based (model-based) diagnosis [10, 2]. In model-based diagnosis we seek a consistent mode assignment where each component of a fixed set of components is assumed to behave correctly or abnormally according to a fault mode. This section examines the relationship between diagnosis and configuration from the common viewpoint of consistency-based reasoning, to provide a platform for analyzing similarities and differences between the two problem area, and to examine what results and methods from diagnosis can be carried over to configuration.

We use the following definitions for model-based diagnosis [10]:

- A *system* is a triple $(SD, COMPONENTS, OBS)$ where:
  - SD, the system description, is a set of first-order sentences;
  - $COMPONENTS$, the system components, is a finite set of constants;
  - OBS, a set of observations, is a set of first-order sentences.
- Furthermore, we define an *ab-literal* to be $ab(c)$ or $\neg ab(c)$ for some $c \in COMPONENTS$, and $AB = \{ab(c) | c \in COMPONENTS\}$.

- For $D \subseteq AB$, $\Delta = D \cup \{\neg ab(c) | ab(c) \in AB \setminus D\}$ is a *diagnosis* for $(SD, COMPONENTS, OBS)$ *iff* $SD \cup OBS \cup \Delta$ is consistent.
- Finally, a *conflict CONFL* of $(SD, COMPONENTS, OBS)$ is defined as a disjunction of ab-literals containing no complementary pair of ab-literals s.t. $SD \cup OBS \models CONFL$. A minimal conflict is a conflict such that no proper subclause is a conflict.

Table 1 shows the correspondence between consistency-based diagnosis and consistency-based configuration.

Diagnosis	Configuration
SD	$DD \cup CL$
OBS	SRS
AB	$UNIV$
D	$CONF$

Table 1.

The system description SD and the domain description DD describe the general properties of an application domain. Since the general behavior of the diagnosis problem is completely defined through the system description, when viewed from the diagnosis point of view, the system description would consist of the domain description *and* the closure axiom set CL.

In contrast to these static items, the observations OBS and the system requirements SRS depend on a specific problem instance. Since the number of needed components in configuration is not known a priori, the number of facts needed for the description of a configuration is not known in advance (thus the introduction of CL). A diagnosis solution is one-dimensional (one ab literal per component). In configuration, there are three types of literals ($type$, $conn$, and val), and there may be several $conn$ and val literals per component.

In addition, since the number of components is theoretically unbounded, so is the number of potential literals. This set of potential literals is what we call the *universe* of the configuration problem ($UNIV$). Note that $UNIV$, like AB, only contains positive literals, and in searching for an irreducible, valid, finite configuration (if one exists), only finite subsets should ever be generated, i.e., those (exclusively positive) literals which actually occur in partial solutions. Therefore, the diagnosis Δ (specified extensionally) is *equivalent* to $\widehat{CONF}$ (which consists of an extensionally specified part, $CONF$, and an intensional one, CL.

Despite these differences, the following theorems have a natural correspondence to results in model-based diagnosis.

6 Conflicts and Configurations

For characterizing configurations and pruning the search space we employ the concept of conflicts. If we find that a given configuration is not valid, we have to conclude that at least one literal exists in the configuration (or in CL) that must be negated to arrive at a state that can be extended to a valid configuration.

Definition 6.1 (Conflict) *A conflict C for a configuration problem (DD, SRS) and a set of components $COMPS$ of a configuration $CONF$ is a disjunction (not necessarily ground) of literals:*

- $type(c, t)$ *where* $type(c, t) \in COMPS$ *and* $t \in types$
- $conn(c1, p1, c2, p2)$ *where* $type(c1, t1) \in COMPS$, $type(c2, t2) \in COMPS$, $p1 \in ports(t1)$, *and* $p2 \in ports(t2)$
- $val(c1, a1, v1)$ *where* $type(c1, t1) \in COMPS$, $a1 \in attrs(t1)$,
- $complete(t, Comps(t, COMPS))$ *where* $t \in types$
- $uconn(c, p)$ *where* $type(c, t) \in COMPS$, $p \in ports(t)$.
- $noval(c, a)$ *where* $type(c, t) \in COMPS$, $a \in attrs(t)$.

such that $DD \cup SRS \cup CL \models C$.

Stated conversely, the negation of a conflict is a conjunction of assumptions about the type of components, their connections and attributes, and the completeness of components s.t. this conjunction is inconsistent with $DD \cup SRS \cup CL$.

The predicate $uconn(c, p)$ holds if port p of component c is not connected to any other component, and $noval(c, a)$ likewise states that attribute a is not assigned a value. (Basically, they are shorthand for expressing the absence of such assignments.)

A valid configuration can be described by the following theorem.

Theorem 6.1 *Let (DD, SRS) be a configuration problem, $CONF$ a configuration, and CS the set of all conflicts of this configuration problem and configuration. $CONF$ is a valid configuration iff $CS \cup \widehat{CONF}$ is satisfiable.*

The concept of conflicts was introduced by [10] in order to detect those assumptions which are inconsistent with a given theory. Given an irreducible set of conflicting assumptions, at least one assumption has to be negated in order to restore consistency.

Solving a configuration problem can be seen as assuming sets of components for each component type as well as connections between components such that these assumptions are consistent with the requirements defined by DD and SRS. Conflicts provide the information which sets of component types and connections are invalid. They are used to prune the generation of assumptions and are re-used during the search for valid configurations. Therefore, we are interested in most general conflicts in order to maximize pruning and reusability. For a configuration to be valid it is sufficient if it is consistent with the most general conflicts. A conflict $C1$ is *most general* iff there exists no other conflict $C2$ such that $C2 \models C1$. This characterization helps avoid useless search and assumption testing.

Example (continued) Consider our example in Section 2 and the set of components $\{type(dm1, digital\_module), type(dm2, digital\_module), type(am1, analog\_module), type(fr1, frame)\}$.

$$\neg conn(dm1, mounted\_on, fr1, slot1) \vee \neg conn(am1, mounted\_on, fr1, slot2) \vee \neg type(fr1, frame).$$

is a conflict since digital and analog modules may not be mixed in one frame. It is not most general, since it is entailed by the conflict

$$\forall F, S1, S2 : \neg conn(dm1, mounted\_on, F, S1) \vee \neg conn(am1, mounted\_on, F, S2) \vee \neg type(F, frame).$$

7 Computing Configurations

For computing irreducible configurations we employ a search strategy that is based on constructing an interpretation, i.e., we seek a model for all conflicts. This approach is closely related to the construction of prime implicants as described in [2, 5].

Since the set of irreducible configurations is often prohibitively large, we employ a best first search algorithm for the construction of the best irreducible valid configurations, e.g., valid configurations within a certain distance from the cost optimal valid configuration.

Definition 7.1 (Configuration-Tree/C-Tree) *Let CS be a set of conflicts, for a given configuration problem (DD, SRS). All nodes are labeled by a ground conflict L or by* sat. *For each node n, we write $c(n)$ for the label of n.*

- *The edges are labeled by positive ground $conn$, $type$, or val literals.*
- *For each node n that is not the root, we define $PC(n)$ as the union of edge labels that occur in the path from the root node to n. If n is the root of the tree, then we define $PC(n) = \emptyset$.*
- *Let $CONNS$, $COMPS$, and $ATTRS$ be the set of (positive) $conn$, $type$, and val literals in $PC(n)$, respectively, defining a configuration (with $CONF = COMPS \cup CONNS \cup ATTRS$ as usual).*
- *A node n is labeled by* sat *iff $DD \cup SRS \cup \widehat{CONF}$ is satisfiable, i.e., the node represents a valid configuration described by $PC(n)$.*
- *A node n is labeled by* unsat *iff $DD \cup SRS \cup CONF$ is unsatisfiable, i.e., $PC(n)$ cannot be extended to a valid configuration. Such a node has no successors, i.e., there are no edges leading away from it.*
- *If $c(n) \notin$ {sat, unsat}, then each edge leading away from n is labeled by $conn(c1, p1, c2, p2)$, $val(c1, a1, v1)$, or $type(c, t)$ literal l such that l implies the conflict L and l is consistent with $\bigwedge_{p \in PC(n)} p$.*

We now define an algorithm that computes configurations based on the definition of the C-Tree. Let $CONF$ be the configuration defined by a node n in the tree. For the algorithm, we assume the existence of a theorem prover $TP(CONF)$ which outputs

- sat if $CONF$ is a valid configuration.
- $unsat$ if $DD \cup SRS \cup CONF$ is unsatisfiable. In this case $CONF$ cannot be extended to a valid configuration.
- c, a most general conflict of the conflict $\neg c'$ where c' is $\widehat{CONF}$. Since $CONF$ is not a valid configuration (otherwise the label would be sat instead of c) and therefore $DD \cup SRS \cup \widehat{CONF}$ is unsatisfiable, $\neg c'$ is a conflict.

Various sophisticated techniques for implementing TP exist, e.g., [2]. Note that previously found conflicts are re-used.

The following algorithm generates all or the best irreducible configurations based on a cost function. We assume an admissible heuristic function which assigns a cost value $costs(PC(n))$ to each path $PC(n)$. Typically, costs will be associated with each $type$ and $connection$ literal. The costs of a node $PC(n)$ are $\sum_{l \in PC(n)} costs(l)$. The costs of $PC(root)$ are 0.

Algorithm 1

1. Initialize:
   open_nodes= $\{root\}$
   minimal_costs= $+\infty$
2. Choose the node n with minimum costs $costs(PC(n))$ from *open_nodes*
3. Mark node n by $TP(PC(n))$
   Case $c(n)$ is
   - $unsat$: delete n from open_nodes
   - sat: delete n from open_nodes
     If $costs(PC(n)) <$ *minimal_costs* then *minimal_costs*:= $costs(PC(n))$
   - c: Generate all possible edges leading away from n.
     Insert the remaining successor nodes in *open_nodes*
4. If *open_nodes*$\neq \emptyset$ then go to 2.

Generating all possible edges: A node n is labeled by a generalized ground conflict $c \models c'$ where $\neg c'$ is a subset of $\widehat{CONF}$. Therefore, the conflicts c and c' contain negative $type$, $conn$, $uconn$, val, and $complete$ literals. The edges leading away from node n have to be labeled with positive ground $conn$ or $type$ literals so they are consistent with $AX \cup PC(n)$ and AX together with edge the label $l(n)$ imply c, i.e., at least one literal of the conflict has to be implied.

There are several different types of literals in a conflict c returned by TP for a node n:

- $\neg type(c,t) \in c$: Since $type(c,t)$ is contained in $PC(n)$ there is no label $l(n)$ such that $l(n) \cup AX \models type(c,t)$ and $l(n) \cup AX \cup PC(n)$ is satisfiable.
- $\neg conn(c1,p1,c2,p2)$: as in the previous case $conn(c1,p1,c2,p2)$ is contained in $PC(n)$. Therefore there is no edge labeling for this case.
- $\neg val(c1,a1,v1)$: as in the previous case $val(c1,a1,v1)$ is contained in $PC(n)$.
- $\neg complete(t, Comps(t, COMPS))$: we have to extend the components of type t. We generate an edge labeled with $type(c,t)$ where c is a Skolem constant.
- $\neg uconn(c,p)$: the port p of component c has to be connected to some other port. For each port of a component c' mentioned in $COMPS$ and some port p' of c' not mentioned in $CONNS$, i.e. not used, we generate an edge labeled by $conn(c,p,c',p')$.
  In addition there may exist a component c' not mentioned in $COMPS$ with port p' to which port p is connected. We generate for each type $t \in types$ a component $type(c',t)$. Port p can be connected to each port p' of these components. For each possible connection we generate an edge with label $\{type(c',t), conn(c,p,c',p')\}$.
- $\neg noval(c,a)$: the attribute a of component c of type t has to be assigned some attribute value. For each attribute of a component c' mentioned in $COMPS$ that does not occur in $ATTRS$, we generate an edge labeled by $val(c,a,v)$, where $v \in dom(c,t)$.

Note the role of the closure predicates in the labeling: whenever one of the closure-related predicates $uconn$ and $complete$ occurs in the conflict, this means the partical configuration is unsatisfiable unless extended (by a component or a connection, respectively). As a relaxing condition on TP, [5] describes pruning techniques in the case the conflicts returned be TP are not most general. The main reason for presenting the technique was pointing out the way in which the basic assumptions made in the representation interact during reasoning: Closing connections, finding attribute values, and adding components.

8 From consistency-based to constraint satisfaction

So far, we have used first order logic for the description of configuration problems. This provides a concise representation and solid basis for examining the properties of the representation. However, for implementation purposes, the formulas presented can be regarded as instantiation schemes for a transformation to other representations, e.g., propositional logic or constraint networks. In particular, the content of the previous chapters presents a high-level view of the languages developed and used in the COCOS configuration project [13], which used as representation a constraint satisfaction scheme that can be defined by a direct mapping from the consistency based semantics of the previous sections.

Formally, a constraint satisfaction problem (CSP) is defined by a set of variables, and a set of constraints. Each variable can be assigned values from an associated domain. Each constraint is an expression or relations that expresses legal combinations of variable assignments. The fundamental reasoning task in CSPs is finding an assignment to all variables in the CSP so that all constraints are satisfied. It is clear that if a mapping can be constructed from the logical representation of a configuration problem (DD, SRS) to a CSP, finding a solution to the CSP will mean that a solution to (DD, SRS) exists and that the assignment is also a model for the configuration problem defined by (DD, SRS). Since many effective algorithms and heuristics for solving CSPs have been presented in the literature, this provides an approach to efficient implementation of a configuration problem solver, while retaining the formal properties of the first order representation.

Representing configuration as a CSP was first mentioned in [3]. Variables in the CSP correspond either to parameters in the configuration or to "locations" where missing components can be placed. Values either correspond directly to parameter values, or to the components that are part of the solution. Initially, no distinction was made between the individual component and its type (e.g., in [8], the variable battery corresponds to the one place for a battery that exists in the car to be configured). Parts of the problem irrelevant to the user could be "masked out" for efficiency in Dynamic Constraint-Satisfaction Problem (DCSP) [8], which use a set of meta constraints to constrain whether variables in the constraint network are *active* or not. This corresponds directly to the property that, for example, connection or attribute literals are only introduced when they are needed in the configuration.

A DCSP still assumes that the set of components is specified in advance. As already discussed in this paper, this is not generally the case in real-world configuration domains, where configurations may comprise thousands of components and multiple components of a given type may exist, which can be assigned individual attribute values and individually connected to others. Therefore, components must be represented as individuals, and the existence of these individuals must be determined during the generation of valid configurations, leading to the Generative CSP (GCSP) approach [13]. A GCSP uses three meta-level extensions to operate with a variable number of components. In the first-order logic formalism, we can express them directly through predicates. First, in GCSPs, components play a double role as variables (for type assignments) and values (for connections). In consistency-based configuration, type assignments can be made explicit via the $type$ predicate, and the Skolem constants which are used as component identifiers simply occur as arguments in the $type$, $conn$, and val literals. Second, attribute values, which are defined in a GCSP by use of activation constraints of the form "if component variable is active and assigned a type, then the correct port and attribute values for that type are active". Finally, the creation of new components in GCSPs is either implicit (if a new component must be generated so that a variable can be assigned a value) or explicit through the use of resource constraints. In both cases, this cor-

responds to the existence quantifiers which occur in our constraints, e.g., in constraints C2 or C6.

In summary, the consistency-based configuration view provides a convenient formal capstone and reference architecture to a representation based on extensions to conventional CSPs, while the implementation in terms of a GCSP also allows the use of efficient CSP algorithms. Configuration strategies can be expressed in terms of value and variable orderings, as is the case in the COCOS system.

9 Inheritance

Configuration knowledge is naturally suited to being represented in an object-oriented manner. Components are described in terms of their attributes and connections, and in the domains discussed in this paper, also have individual existence and are created as needed. It is therefore natural to think about arranging component types in an inheritance hierarchy. In fact, Configuration knowledge is well suited to an efficient use of inheritance.

First, the task of finding taxonomies in the problem domain is often trivial for at least part of a domain, as technical knowledge about the parts catalog is typically already partitioned into subareas, at the topmost layer based on the function and physical characteristics of the parts represented, and at lower levels based on finer functional distinctions, different attribute domains, and the availability of specific versions and subtypes of components.

Second, one common trait of the domain taxonomies is that they are monotonic due to their technical origin. The components in a catalog are not the result of a natural creation process but typically arose as part of a design task that aimed at clearly structuring systems, so they could be effectively handled and understood by sales,assembly, and engineering personnel. The ability to use monotonic forms of inheritance removes one of the more conceptually and computationally complex aspects of inheritance hierarchies from consideration without greatly reducing the applicability of a representation.

Third, as mentioned earlier, a parts catalog for configuration specifies the set of available parts *completely*. This means that individual physical components will always correspond exactly to one of the types in the inheritance hierarchy. As an illustration, consider that where actual physical components are being configured, only components that are actually physically manufactured will be available to be inserted into a configuration. In certain cases, there can be further simplifications, for example that only the leaves of the inheritance hierarchy correspond to actual components, but these do not concern us here. Therefore, in general, it is not necessary to provide a general subsumption algorithm in a configuration reasoner, since particular component merely have to be matched exactly to a given type.

A further conclusion that can be drawn from the monotonicity and specificity properties is that in implementation terms, it is easy to incorporate such a hierarchy in configuration systems that are implemented in object-oriented programming languages, since the inheritance hierarchies of OOPL's, while generally more restrictive than those of AI reasoning systems, will be largely able to represent configuration type hierarchies directly in terms of class hierarchies of the implementation language of the inference engine. This reduces implementation effort and increases efficiency.

10 Conclusion

Based on our experience in developing knowledge-based configuration tools, the goal of using the consistency-based approach for describing configuration problems was to gain a simple, straightforward but reasonably expressive formal basis for discussing the properties of configuration representations. For example, the creation of new components through Skolem constants corresponds to the creation of new constraint variables in Generative CSP's and incorporates the Dynamic CSP capability.

This paper has presented a formal view of configuration as a close relative of the the consistency-based diagnosis process. The system description of model-based diagnosis corresponds to the domain description and type library of the configuration problem, and the observations of diagnosis correspond to the specification in configuration. The correspondence extends to the possibility of directly adapting Reiter's Hitting Set diagnosis algorithm, with conflict labeling providing for the creation of the correct components, their types and connections. This provides a straightforward and clear formal basis for more research into the nature of configuration problems. In particular, it is a direct first order logic counterpart to the representation of the COCOS/LAVA configuration tool.

REFERENCES

[1] R. Cunis, A. Günter, I. Syska, H. Bode, and H. Peters. PLAKON – an approach to domain-independent construction. In *Proc. IEA/AIE Conf.*, Tennessee, 1989. UTSI.

[2] J. de Kleer. Focusing on probable diagnoses. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 842–848, Anaheim, July 1991.

[3] F. Frayman and S. Mittal. COSSACK: A constraint-based expert system for configuration tasks. In D. Sriram and R. Adey, editors, *Knowledge-Based Expert Systems in Engineering, Planning, and Design*, July 1987.

[4] E. Gelle and R. Weigel. Interactive configuration with constraint satisfaction. In *Proceedings of the 2nd International Conference on Practical Applications of Constraint Technology (PACT)*, Aug. 1996.

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

[6] R. Klein, M. Buchheit, and W. Nutt. Configuration as model construction: The constructive problem solving approach. In *Proceedings Artificial Intelligence in Design '94*, pages 201–218. Kluwer, Aug. 1994.

[7] S. Marcus, J. Stout, and J. McDermott. VT: An expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, 9(2):95–111, 1988.

[8] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings AAAI Conference*, pages 25–32, Aug. 1990.

[9] S. Mittal and F. Frayman. Making partial choices in constraint reasoning problems. In *Proceedings AAAI*, pages 631–636, 1987.

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

[11] E. Soloway, J. Bachant, and K. Jensen. Assessing the maintainability of XCON-IN-RIME: Coping with the problems of a VERY large rule-base. In *Proceedings AAAI*, pages 824–829, 1987.

[12] M. Stumptner and G. Friedrich. Consistency-based configuration. Technical Report DBAI-CSP-TR 99/01, Technical University of Vienna, Feb. 1999.

[13] M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12(4), Dec. 1998.

[14] J. R. Wright, E. S. Weixelbaum, K. Brown, G. T. Vesonder, S. R. Palmer, J. I. Berman, and H. G. Moore. A Knowledge-Based Configurator that Supports Sales, Engineering, and Manufacturing at AT&T Network Systems. In *Proceedings of the 5th Conference on Innovative Applications of AI*. AAAI Press, 1993.