Knowledge Acquisition for Configuration Tasks : The EXPECT Approach

Surya Ramachandran Yolanda Gil
USC Information Sciences Institute
4676 Admiralty Way, Suite 1001.
Marina del Rey, California 90292
{rama, gil}@isi.edu

Abstract

Configuration systems often use large and complex knowledge bases that need to be maintained and extended over time. The explicit representation of problem-solving knowledge and factual knowledge can greatly enhance the role of a knowledge acquisition tool by deriving from the current knowledge base, the knowledge gaps that must be resolved. This paper details EXPECT's approach to knowledge acquisition in the configuration domain using the propose-and-revise strategy as an example. EXPECT supports users in a variety of KA tasks like filling knowledge roles, making modifications to the knowledge base including entering new components, classes and even adapting problem-solving strategies for new tasks. EXPECT's guidance changes as the knowledge base changes, providing a more flexible approach to knowledge acquisition. The paper also examines the possible use of EXPECT as a KA tool in the complex and real world domain of computer configuration.

Introduction

Knowledge Acquisition is an integral part of any configuration system. Changes and modifications need to be continuously made with respect to changes in markets. With the emergence of new product-lines and the discontinuation of old ones there is a need not only for good configuration systems, but also for knowledge acquisition tools that will help to keep knowledge bases current. Further, with the changes in the business needs of customers more sophisticated tools that help change configuration constraints and parameters are needed. These would be a very useful capabilities, since product knowledge changes at a high rate (40-50%/year) is reported for configuration systems such as R1 (McDermott 82) and PROSE (Wright et al. 93).

EXPECT (Swartout and Gil 95; Gil 94; Gil and Paris 94) is a flexible KA tool that has been used for a variety of tasks and domains including configuration. The problem-solving strategy is represented explicitly, and the knowledge acquisition tool reasons about it and dynamically derives the knowledge roles that must be filled out, as well as any other information needed for problem solving. Because the problem-solving strategy is explicitly represented, it can be modified, and as a result, the KA tool changes its interaction with the user to acquire knowledge for the new strategy. EXPECT provides greater flexibility in adapting problem-solving strategies because their representations can be changed as much as needed.

The paper begins by describing propose-and-revise and its use in a role-limiting tool for knowledge acquisition. Then we summarize the work described in (Gil and Melz 96) to illustrates how EXPECT's knowledge acquisition tool works when the system is using a specific problem-solving strategy. EXPECT not only supports users in filling out knowledge roles, but extends the support to acquire additional knowledge needed for problem-solving. We use the propose-and-revise paradigm in small domain for U-Haul rentals (Gennari et al. 93). We then look at a possible implementation and usefulness of a KA tool for the computer configuration domain, a domain that is considerably more complex and real world. Though not implemented, it serves as an example of the power of EXPECT as a KA tool. We describe the types of knowledge that need to be acquired for configuration tasks and show how EXPECT could support users in implementing them.

Solving Configuration Design Tasks with Propose-and-Revise

Propose-and-revise is a problem-solving strategy for configuration design tasks. A configuration problem is described as a set of input and output *parameters* (or variables), a set of *constraints*, and a set of *fixes* to resolve constraint violations. A solution consists of a value assignment to the output parameters that does not violate any constraint.

Propose-and-revise constructs a solution by iteratively extending and revising partial solutions. The *extension* phase consists of assigning values to parameters. In the *revision* phase, constraints are checked to verify whether they are violated by the current solution and if so, the solution is revised to resolve the violation. Violated constraints are resolved by applying fixes to the solution. A fix produces a revision of the solution by changing the value of one of the parameters that are causing the constraint violation.

Propose-and-revise was first defined as a problem-solving method for configuration in VT (Marcus 88) for designing elevator systems. Input parameters for VT included features of the building where the elevator was to be installed. Output parameters included the equipment selected and its layout in the hoistway. An example of a constraint is that a model 18 machine can only be used with a 15, 20, or 25 horsepower motor. An example of a fix for a violation of this constraint is to upgrade the motor if the current configuration was using one without enough horsepower.

SALT (Marcus and McDermott 89) which was used to build VT, is a knowledge acquisition tool for propose-and-revise using a role-limiting approach. In this problem-solving strategy, there are three types of knowledge roles; procedures to assign a value to a parameter which would result in a design extension, constraints that could be violated in a design extension and fixes for a constraint violation. Consequently, the user could enter one of the three types of knowledge. For each type of knowledge, a fixed menu is presented to the user to be filled out. SALT does not provide support in updating or maintaining the knowledge about elevator components.

Explicit Representations in EXPECT

In EXPECT, both factual knowledge and problem-solving knowledge are represented explicitly. This means that the system can access and reason about the representations of factual and problem-solving knowledge and about their interactions. Factual knowledge is represented in LOOM (MacGregor 91), a knowledge representation system based on description logic. Factual knowledge includes concepts, instances, and the relations among them. Problem-solving knowledge is represented in a procedural-style language that is tightly integrated with the LOOM representations. Subgoals that arise during problem solving are solved by methods. Each method description specifies: 1) the goal that the method can achieve, 2) the type of result that the method returns, and 3) the method body that contains the procedure that must be followed in order to achieve the method's goal. A method body can contain nested expressions, including subgoal expressions that need to be resolved by other methods; control expressions such as conditional statements and some forms of iteration; and relational expressions to retrieve the fillers of a relation over a concept. Some method bodies are calls to Lisp functions that are executed without further subgoaling.

We first look at an example of EXPECT's representations using propose-and-revise as a strategy for solving the following type of problems in the U-Haul domain: Given the total volume that the client needs to move, the system recommends which piece of equipment (e.g., a truck, a trailer, etc.) the client should rent. Figure 1 graphically shows parts of the factual domain model for propose-and-revise and for the U-Haul domain. [1]The upper part of the picture shows factual knowledge that is domain independent and can be reused for any domain. In the configuration process, there is an explicit representation of state variables (which denote a configuration) and constraints. The state variables can be associated with components the make up the configuration. Constraints are associated with valid sets of instantiations for the state variables. The lower part of the picture shows factual knowledge that is relevant to the U-Haul domain. Here state variables denote things like pieces of equipment and constraints contain information about capacity restrictions, etc.

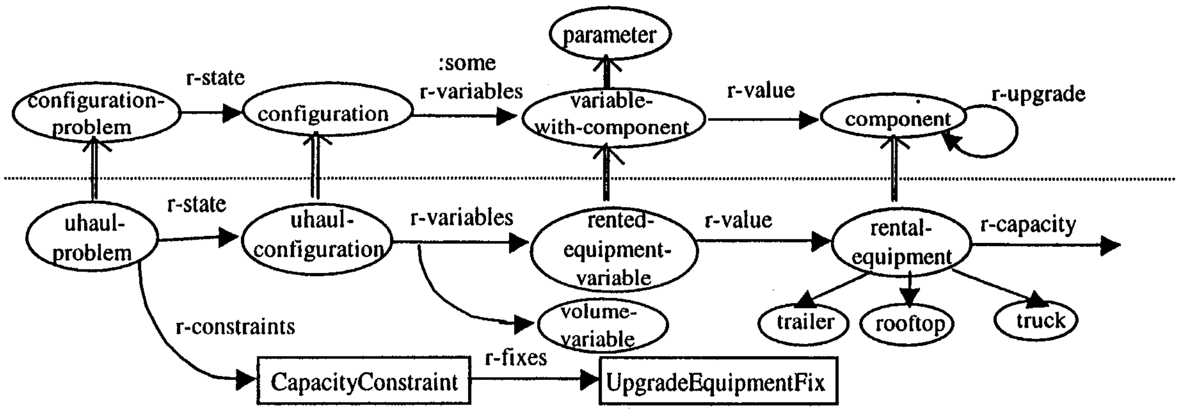

Figure1: EXPECT's representation of some of the factual knowledge needed for propose and revise problems, for configuration problems and the U-Haul domain.

There is a continuum between the representation of domain-dependent and domain-independent factual knowledge in EXPECT. They are represented in the same language, yet they can be defined and maintained separately. Once a U-Haul problem is specified as a kind of configuration problem, it inherits the fact that it has constraints and fixes. Trucks are not defined as having upgrades, since having upgrades is a way to look at components from the point of view of configuration problems. Instead, they are defined as configuration components, which have upgrades.

```
(defmethod REVISE-CS-STATE
  "To revise a CS state, apply the fixes found for the
  constraints violated in the state."
  :goal (revise (obj (?state is (inst-of cs-state))))
```

1 By convention, we denote relations with the prefix r-.

```
  :result (inst-of cs-state)
  :body (apply (obj (find (obj (set-of (spec-of fix)))
                (for (find
                  (obj (set-of (spec-of violated-constraint)))
                  (in ?state)))))
              (to ?state)))

(defmethod CHECK-CAPACITY-CONSTRAINT
  "To check the Capacity Constraint of a U-Haul
  configuration, check if the capacity of the rented
  equipment is smaller than the volume to move."
  :goal (check (obj CapacityConstraint)
              (in (?c is (inst-of uhaul-configuration))))
  :result (inst-of boolean)
  :body (is-greater-or-equal
          (obj (r-capacity (r-rented-equipment ?c)))
          (than (r-volume-to-move ?c))))
```

Figure 2: Problem-solving knowledge in EXPECT.

Figure 2 shows two different problem-solving methods. REVISE-CS-STATE is one of the methods that specifies how propose-and-revise works. The CHECK-CAPACITY-CONSTRAINT specifies that the capacity of the equipment rented must at least be equal to the volume of the client's needs.

Knowledge Acquisition in EXPECT

EXPECT's problem-solver is designed to detect errors and to report them to the KA tool (see table 1) together with detailed information about how they were detected. The KA tool uses this information to support the user in fixing them. Other modules that can detect and report errors are the parser (which detects syntax errors and undefined terms), the method analyzer (which detects errors within a problem-solving method), and the instance analyzer (which detects missing information about instances).

EXPECT's problem-solver can analyze how the different pieces of knowledge in the knowledge-based system interact. For this analysis, it takes a generic top-level goal representing the kinds of goals that the system will be given for execution. In the U-Haul example, the top-level generic goal would be (solve (obj (inst-of uhaul-problem))), and a specific goal for execution would be (solve (obj jones-uhaul-problem)).

EXPECT analyzes how to achieve this goal with the available knowledge. EXPECT expands the given top-level goal by matching it with a method and then expanding the subgoals in the method body. This process is iterated for each of the subgoals and is recorded as a search tree. Throughout this process, EXPECT propagates the types of the arguments of the top-level goal, performing an elaborate form of partial evaluation supported by LOOM's reasoning capabilities. During this process, EXPECT derives the interdependencies between the different components of its knowledge bases. This analysis is done every time the knowledge base changes, so that EXPECT can re-derive these interdependencies.

The EXPECT's problem solver is designed to detect goals that do not match any methods, and to detect relations that try to retrieve information about a type of instance that is not defined in the knowledge base. In addition to detecting an error, each module is able to recover from the error if possible, and to report the error's type and the context in which it occurred. It would also report this error to the knowledge acquisition module, together with some context information and a pointer to the part of the problem-solving trace where the subgoal was left unsolved.

Once the errors are detected, EXPECT can help users to fix them as follows. EXPECT has an explicit representation of types of errors, together with the kinds of corrections to the knowledge base that users can make in order to solve them. This representation is based on typical error situations that we identified by hand. Table 1 shows some of the errors that can currently be detected by two of the modules: the problem solver (e1 through e3) and the instance analyzer (e5).

Knowledge Acquisition for Propose-and-Revise in EXPECT

Previously, we pointed out some of SALT's limitations in terms of its lack of flexibility as a knowledge acquisition tool. In this section, we illustrate how EXPECT's explicit representations support a more flexible approach to knowledge acquisition.

Code	Error/Potential Problem	Suggested Corrections
e1	no method found to achieve goal G in method body M	modify method body modify another method's goal add a new method modify instance, concept, relation
e2	role R undefined for type C in method M	modify method M add relation R to C
e3	expression E in method M has invalid arguments	modify method M modify another method's goal modify instance, concept, relation
e5	missing filler of role R of instance I needed in method M	add information about instance modify method body delete instance

Table1: Some of the potential problems in the knowledge bases detected by EXPECT.

Acquiring Domain-Specific Knowledge

Suppose that U-Haul decided to begin renting a new kind of truck called MightyMover. The user would add a new subclass of truck, and EXPECT would immediately request the following:

E1---I need to know the capacity of a MightyMover.

The reason for this request is that EXPECT has detected that the capacity of rental equipment is a role that is used during the course of problem solving, specifically while achieving the goal of checking the CapacityConstraint with the method shown in Figure 2.

This corresponds to errors of type e5 in Table 1. EXPECT will only request the information that is needed by the problem-solving methods.

Acquiring New Constraints and Fixes

Instead of needing the definitions of schemas to enter constraints and fixes, EXPECT requests them as constraints and fixes that are to defined by the user. Suppose for example that the user wants to add a new constraint that restricts the rental of trailers to clients with cars made after 1990 only. The user would add a new instance of constraint: TrailersForNewCarsOnly. EXPECT would analyze the implications of this change in its knowledge base and signal the following problem:

E2---I do not know how to achieve the goal
(check (obj TrailersForNewCarsOnly) (in (inst-of uhaul-configuration)))

This is because during problem solving EXPECT calls a method that tries to find the violated constraints of a configuration by checking each of the instances of constraint of U-Haul problems. This is a case of an error of type e1. Before defining this new instance of constraint, the only subgoal posted was (check (obj CapacityConstraint) (in (inst-of uhaul-configuration))) and now it also posts the subgoal (check (obj TrailersFor NewCarsOnly) (in (inst-of uhaul-configuration))). There is a method to achieve the former subgoal (shown in Figure 2), but there is no method to achieve the latter.

To resolve E2, the user chooses the third suggestion for errors of type e1 and defines the following method to check the constraint: Once this method is defined, E2 is no longer a problem and disappears from the agenda. EXPECT's error detection mechanism also notices possible problems in the formula to check the constraint. For example, if r-year had not been defined EXPECT would signal the following problem (of type e2):

E3---I do not know what is the year of a car.

When the user defines the role r-year for the concept car this error will go away. EXPECT can also detect other types of errors in the formulas to check constraints. For example, if r-year was defined to have a string as a range, then EXPECT would detect a problem. It would notice that there is no method to check if a string is greater than a number, because the parameters of the method for calculating is-greater must be numbers. EXPECT would then tell the user:

E4---I do not know how to achieve the goal
(is-greater (obj (inst-of string)) (than 1990))

Like E2, E4 is an error of type e1. But in this case the user chooses a different way of resolving the error, namely to modify the definition of the relation r-year. If the user defined a fix for the new constraint, then EXPECT would follow a similar reasoning and signal the need to define a method to apply the new fix.

EXPECT changes its requests for factual information according to changes in the problem-solving methods. This can be illustrated in this example of adding a new constraint. An effect of the fact that the user defined the new method to check the constraint is that new factual knowledge about the domain is needed. In particular, EXPECT detects that it is now important to know the year of the car that the client is using (and that is part of the configuration), because it is used in this new method. The following request will be generated for any client that, like in this case Mr. Jones, needs to rent U-Haul equipment:

E5---I need to know the year of the car of Jones.

This is really requiring that the information that is input to the system is complete in the sense that configuration problems can be solved. In EXPECT, the requirements for inputs change as the knowledge base is modified.

Changing the Propose-and-Revise Strategy

Suppose that the user wants to change the revision process of propose-and-revise to introduce priorities on what constraint violations should be resolved first. The priorities will be based on which variable is associated with each constraint.

The user would need to identify which of the problem-solving methods that express propose-and-revise in EXPECT needs to be modified. The change involves adding a new step in the method to the revise state in the propose-and-revise methodology. The new step is a subgoal to select a constraint from the set of violated constraints. EXPECT would signal the following request:

E6---I do not know how to achieve the goal
(select (obj (spec-of constraint)) (from (set-of (inst-of violated-constraint))))

This is an error of type e5, and it indicates that the user has not completed the modification. The user needs to create a new method to achieve this goal. The user may also need to define a new method for the take subgoal.

With these modifications to the knowledge base, the propose-and-revise strategy that EXPECT will follow has changed. Because the representation of the new strategy is explicit, EXPECT can reason about it and detect new knowledge gaps in its knowledge base. As a result of the modification just made, there is additional factual information needed including new information about an existing knowledge role and a new kind of knowledge role. EXPECT would then signal the following requests (both of type e5):

E7---I need to know the constrained variable of TrailersForNewCarsOnly.

E8---I need to know the preference of equipment-variable.

E7 and E8 illustrate that EXPECT has noticed that the change in the problem-solving strategy requires the user to provide new kinds of information about the factual knowledge used by the strategy. This shows that in EXPECT the acquisition of problem-solving knowledge affects the acquisition of factual knowledge. Recall that E2 illustrated the converse.

Knowledge Acquisition for Computer Configuration

In this section we shall describe how the explicit representation of knowledge approach used in the EXPECT architecture can aid knowledge acquisition in the computer configuration domain. This is not an implemented domain but serves as an example to show that the approach taken in the U-Haul domain can be applied to more complex real world domains. In EXPECT the separation of different pieces of knowledge can help the user (or developer) in acquiring different forms of knowledge. In the computer configuration domain we shall look at problem solving methods (PSMs), constraints, class hierarchies and the actual instances of data that populate these classes as examples of knowledge.

For lucidity in the explanation of the benefits of a KA tool in the computer configuration domain, let us look at a possible way of representing constraints and component specifications. As the reader is aware the knowledge representation framework used by EXPECT is LOOM a description logic based KR system (detailed in the section on explicit representations in EXPECT). Frame based semantics that provide for the definition of Concepts (or frames), Instances of these concepts and Roles which provide a way to relate instances. LOOM allows for class hierarchy descriptions of components. A hierarchy of constraints can then be defined that closely relates to the class hierarchy.

The advantages of using a description logics based system as described in the PROSE system (Wright, et al 93) which uses C-Classic (Weixelbaum 91) are applicable.

- Classification. The ability to find all descriptions applicable to an object; finding all descriptions that are more general or more specific than it (subsumption architecture).
- Completion or propagation of logical consequences. including but not limited to inheritance.
- Contradiction detection where a particular instantiation of features does not represent a legal combination.
- Dependency maintenance. A truth (or falsity) preservation over the entire set of assertions.

Figure3 depicts, on the left, a possible representation of the class hierarchy for the general class of storage mediums that can be further decomposed into *hdd* (hard disk drives), *fdd* (floppy disk drives), *CDROMs* and so on. On the right, a similar hierarchy is shown describing a common data bus architectures found in the computer domain today.

An example of what kinds of attributes an actual instance would have are also depicted. Instances here are actual components that are manufactured and that make up the final configuration.

Generally, constraints can be defined as the rules or heuristics that govern the binding of values to a set of variables for a given problem specification. A constraint limits the possible values that can be assigned to these variables. The constraint satisfaction problem (CSP) can thus be defined as a consistent set of variable assignments such that the resultant solution does not violate any of the given constraints. Configuration can be classified as a type CSP where the final solution is a list of instances (or variable instantiations) from the domain of products that do not violate any constraints (which is represented in the top half of figure 1). Constraints in the computer configuration domain can be represented either as roles (or relations) defined on the component class hierarchy at the LOOM level or as explicit problem solving methods in EXPECT.

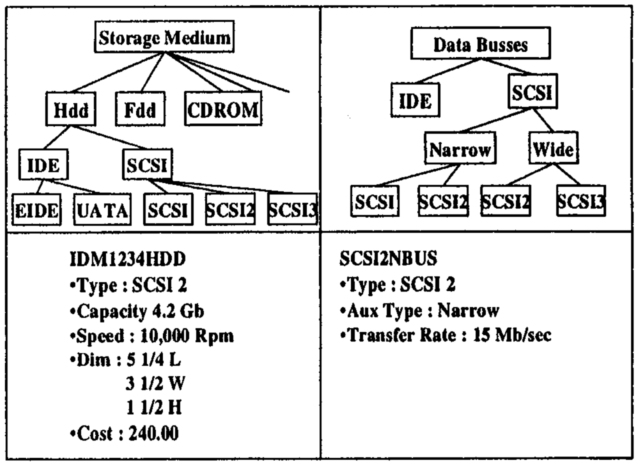

Figure3 : Hierarchical representation of classes and instances.

Let us now look at some examples of constraints. A typical constraint on the class of storage mediums may try to assign a data bus to connect the device to. This constraint may in turn be composed of two disjunctive sub-constraints. One trying to assign an existing data bus checking for bandwidth availability. Failing which (OR), the assignment of some component to be included in the configuration that will provide the correct bus type and enough bandwidth. But the focus of this section is not on the formulation of constraints or the actual configuration process but rather on enumerating the clear benefits of using an architecture like that of EXPECT for the ease of knowledge acquisition. We shall now describe the benefits that a KA tool provides at various levels of the hierarchy and aspects of the configuration process. We have identified four main levels that a KA tool can provide support and aid the user/developer.

The addition of a new component.

By representing explicitly the information used in the constraints and problem solving methods, the KA tool can identify which attributes of a class are needed in the configuration task. Thus the KA tool can be used to guide the users (or data entry operators) to enter only relevant information for configuration. The benefits here are that not only do data entry operators spend less time entering information on new components but the automation of this process will lead to the configuration system co-existing with other applications in the enterprise by sharing information from different databases. An example of the extraction of component information from databases may include; a database that maintains product sheets for user information, a database that has stock and warehouse information and another that has current pricing information. This is similar to E1 for knowledge acquisition in the U-Haul domain.

The addition of a new component class.

By exploiting the inter-dependencies between the various constraints and classes (i.e. the attributes used in a constraint must be defined in at least one class in the constraint hierarchy) the KA tool can suggest the possible list of attributes that would be needed for that class by comparing the list of attributes that are mentioned in the constraints and the inherited attributes at that level of the component hierarchy. The KA tools can further help in defining the value, range, and domain if such information is exploited in the constraints. Thus the configuration system can be modified with relative ease and without the domain knowledge of either the component manufacturer or the component hierarchy knowledge. This resembles E1 for KA in the U-Haul domain.

The addition of new constraints.

By identifying the level of applicability of a constraint with respect to it's position in the constraint hierarchy and the class of components it affects, the KA tool can guide the user in effectively using all parts of the KB that are available at that level (often times with complex class hierarchies and inheritance patterns, the user may not be completely aware of what attributes are available to him while writing a constraint). Another area where a KA tool can help the user write constraints is by looking for similar constraints that may already exist, and the modification of which would not only lead to greater uniformity and less mistakes in the knowledge bases but would also be a conservation of time and effort. This is similar to E2 through E5 for KA in the U-Haul domain.

The modification/extension of PSMs.

By explicitly representing the problem solving methods used by the configuration system and having access to a library of PSMs, the KA tool can help the user make the appropriate change to the PSMs for a new application in a domain requiring different inferential capabilities. For example, the choice of which part of a disjunctive constraint to evaluate first may be based on some criteria like the amount of computation needed to explicitly make a choice at to which branch to explore first. The primary benefit of having a KA tool actually guiding the user throughout this process is the fact that its usage will lead to a more robust and efficient configuration system. This is similar to E6 through E8 for KA in the U-Haul domain.

Discussion and Conclusion

Throughout the paper, we have referred to a generic user wanting to make changes to the knowledge base. This is not necessarily one user, and not necessarily the end user or domain expert. For example, the end user may only enter knowledge about clients and new parts. A more technical user would be able to modify propose-and-revise. A domain expert who does not want to change the problem-solving methods can still use EXPECT to fill up knowledge roles and populate the domain-dependent factual knowledge base. Supporting a range of users would require adding a mechanism that associates with each type of user the kinds of changes that they can make to the knowledge base and limiting the users to make only those changes. The important point is that all the changes, no matter who ends up making them, are supported by the same core knowledge acquisition tool. EXPECT's explicit representations of problem-solving strategies can be used to support flexible approaches to knowledge acquisition. Our goal is to apply this approach to support users to maintain and extend configuration systems.

Acknowledgments

The authors are grateful to all the members of the EXPECT group, past and present. Especially to Eric Melz for his work on the U-Haul Domain. We are grateful for the support of DARPA with contract DABT63-95-C-0059 as part of the DARPA/Rome Laboratory Planning Initiative, and with grant F30602-97-1-0195 as part of the DARPA High Performance Knowledge Bases Program.

References

Gennari, J. H., Tu, S. W., Rothenfluh, T. E., and Musen, M. A. Mapping methods in support of reuse. In Proc. of the 8th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, 1994.

Gil, Y. Knowledge refinement in a reflective architecture. In Proc. of the 12th National Conference on AI, Seattle, WA, 1994.

Gil, Y., and Paris, C. Towards method-independent knowledge acquisition. Knowledge Acquisition, 6(2):163--178, 1994.

Gil, Y. and Melz, E. Explicit Representations of Problem-Solving Strategies to Support Knowledge Acquisition. In Proc. of 13th National Conference on AI, AAAI 96. 469-476, 1996.

McDermott, J. R1: A rule-based configurer of computer systems. Artificial Intelligence 19:39--88, 1982.

MacGregor, R. The evolving technology of classification-based knowledge representation systems. In J. Sowa, editor, Principles of Semantic Networks: Explorations in the Representation of Knowledge. Morgan Kaufmann, San Mateo, CA, 1991.

Marcus, S., and McDermott, J. SALT: A knowledge acquisition language for propose-and-revise systems. Artificial Intelligence, 39(1):1--37, May 1989.

Marcus, S., Stout, J., and McDermott, J. VT: An expert elevator designer that uses knowledge-based backtracking. AI Magazine 9(1):95--112, 1988.

Swartout, W. R., and Gil, Y. EXPECT: Explicit Representations for Flexible Acquisition. In Proc. of the 9th Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Alberta, 1995.

Weixelbaum, E. C-CLASSIC Reference Manual, Release 1.0, Technical Memorandum 59620-910731-07M, AT&T Bell Laboratories, Murray Hill, NJ. 1991.

Wright, J. R., Thompson, E. S., Vesonder, G. T., Brown, K. E., Palmer, S. R., Berman, J., and Moore, H. A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T Network Systems. AI Magazine 14(3):69--80, 1993.

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

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

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

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

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

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 $\land$ and $\lor$ 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 $D \subset AB$, $\Delta = D \cup \{\neg ab(c) | ab(c) \in AB \setminus D\}$ 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 corresponds 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.