Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 113–114.

doi:10.1017/S0890060410000569

GUEST EDITORIAL

Special Issue: Configuration

ALEXANDER FELFERNIG,[1] MARKUS STUMPTNER,[2] AND JUHA TIIHONEN[3]

[1]Institute for Software Technology, Graz University of Technology, Graz, Austria
[2]Advanced Computing Research Centre, University of South Australia, Adelaide, Australia
[3]Department of Computer Science and Engineering, Aalto University, Helsinki, Finland

Configuration can be defined as the composition of a complex product from instances of a set of component types, taking into account restrictions on the compatibility of those component types. For supporting product configuration, different artificial intelligence (AI) approaches are well established as central technologies in industrial configuration systems. However, the wide industrial use of configuration technologies and the increasing size and complexity of configuration problems make the field more challenging than ever. Today the mass customization paradigm has been extended from traditional physical products to the fields of software and service configuration. Configuration systems have evolved into interactive Web-based applications that need to support highly sophisticated knowledge representation and reasoning methods. A wide range of AI techniques are applied in this context: just to mention a few, constraint satisfaction, intelligent user interfaces, preference handling, and explanations.

As a successful AI application area, configuration has attracted lasting industrial interest and renewed research, as demonstrated by a series of workshops on configuration that have been arranged in conjunction with leading AI conferences such as IJCAI, ECAI, and AAAI.

The goal of this Special Issue on configuration is to demonstrate novel and innovative configuration research as well as new industrial applications of configuration technologies. The contributions of this Special Issue on configuration are a continuation of high-quality papers in previous special issues on configuration published in such journals as *IEEE Intelligent Systems* (1998), *AI EDAM* (1998 and 2003), and *International Journal of Mass Customization* (2010). The seven papers (five full-length papers and two short papers) were selected from 17 submissions, which corresponds to a full-length paper acceptance rate of 29%. Each paper underwent two to four double-blind reviews by experts in the configuration domain. Papers with a positive reviewer feedback after the first review round were reviewed again to assure that all of the reviewer comments of the first round had been taken into account. The reviews of papers that included a coeditor as an author were managed in a screened manner by uninvolved coeditors or members of the Special Issue program committee.

The major topics of the current Special Issue include personalization techniques and algorithms in knowledge-based configuration, different issues of configuration knowledge representation, industrial configuration environments and new application domains, and business-oriented aspects of the application of configuration technologies.

"Modeling and Solving Technical Product Configuration Problems" by Andreas Falkner, Alois Haselboeck, Gottfried Schenner, and Herwig Schreiner contains an introduction to the "partner units" problem and provides a discussion of possible alternative knowledge representation approaches (e.g., Unified Modeling Language/Object Constraint Language and Alloy). In addition, the paper contains a discussion of possible approaches to solve the "partner units" problem (from basic backtracking to local search approaches such as "simulated annealing"). The paper is concluded with an in-depth analysis of the applied search algorithms.

In their short paper on "Product Configuration as Decision Support: The Declarative Paradigm in Practice" Albert Haag and Steffen Riemann discuss knowledge representation issues in the SAP configuration environment. As an application domain for configuration technologies they introduce the customization of SAP systems. Besides the discussion of the advantages and trade-offs of procedural and declarative knowledge representations, the authors provide an in-depth discussion of the application of assumption-based truth maintenance approaches in their configuration environment.

"A Declarative Framework for Work Process Configuration," written by Wolfgang Mayer, Markus Stumptner, Peter Killisperger, and Georg Grossmann, extends established constraint-based configuration approaches with a constraint representation language for representing specific properties of execution paths in work processes. In this context, a framework for semiautomated process customization is introduced. It integrates the extended constraint approach with a metamodel of work processes. Valid process configurations are then semiautomatically built on the basis of heuristic search.

In their short paper on "Reasoning about Conditional Constraint Specification Problems and Feature Models" Raphael

Reprint requests to: Alexander Felfernig, Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b, Graz A-8010, Austria. E-mail: alexander.felfernig@ist.tugraz.at

Finkel and Barry O'Sullivan show how techniques from formal methods and answer set programming can be applied to represent conditional constraint satisfaction problems. Besides the intuitive handling of variable existence, their knowledge representations allow for "model reflection" in that several kinds of model flaws can be automatically detected, for example, a variable declared as optional is actually required in all solutions.

"Personalized Diagnoses for Inconsistent User Requirements" by Alexander Felfernig and Monika Schubert provides a discussion of the advantages of applying different types of personalization techniques (e.g., utility-based and content-based recommendation) to identify preferred diagnoses in interactive configuration settings. A diagnosis denotes a minimal set of user requirements that has to be adapted or relaxed to identify a solution. Such functionalities are especially useful in "open configuration" scenarios where the user is free to select options and the configurator provides explanations in the case of inconsistencies.

In their paper on "Adaptive Attribute Selection for Configurator Design Via Shapley Value" Yue Wang and Mitchell Tseng introduce concepts that support the personalized ranking of questions posed to users within the scope of interactive configuration sessions. The overall goal is to keep the number of needed interaction steps with a configurator as low as possible, that is, to ask only those questions that are relevant for the user in a certain configuration context. The method that they introduce iteratively selects the attributes (questions) that best contribute in terms of information content from the pool of remaining unanswered questions.

Finally, "The Impact of Product Configurators on Lead Times in Engineering Oriented Companies," by Anders Haug, Lars Hvam, and Niels Henrik Mortensen, summarizes the results of a study on the impact of configuration technologies in commercial environments. Fourteen companies applying configuration technologies were analyzed regarding the impact of configuration technologies on processes related to the creation of quotes and product specifications. The study includes impressive outcomes, for example, the quotation lead time was reduced on an average by about 85%.

In sum, the papers of this Special Issue exhibit configuration as a continuously active field of research with new and challenging research questions and application domains attracting lasting industrial interest.

Alexander Felfernig is a Professor of applied software engineering at Graz University of Technology. Alexander is also Cofounder and Director of ConfigWorks, a company focused on the development of knowledge-based recommendation technologies. Prof. Felfernig's research focuses on intelligent methods and algorithms supporting the development and maintenance of complex knowledge bases. Furthermore, he is interested in the application of AI techniques in the software engineering context, for example, the application of decision and recommendation technologies to make software requirements engineering processes more effective. In 2009, Dr. Felfernig received the Heinz-Zemanek Award from the Austrian Computer Society for his research.

Markus Stumptner is a Professor of computer science at the University of South Australia, where he directs the Advanced Computing Research Centre. He received MS and PhD degrees in computer science from the Vienna University of Technology. Dr. Stumptner's research interests include object-oriented modeling, knowledge representation, and model-based reasoning in areas such as configuration and diagnosis.

Juha Tiihonen is a Researcher in the Department of Computer Science and Engineering at Aalto University School of Science (previously Helsinki University of Technology). He received his MS and LicSc degrees in computer science from Helsinki University of Technology. His main interest is product and service configuration in its various forms, including modeling, configurators, operations management aspects of business processes based on product and service configuration, and design for configuration. Mr. Tiihonen's most recent research includes recommendation support for configurable offerings and installed base management of complex products.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 115–129.

doi:10.1017/S0890060410000570

Modeling and solving technical product configuration problems

ANDREAS FALKNER, ALOIS HASELBÖCK, GOTTFRIED SCHENNER, AND HERWIG SCHREINER
Siemens AG Österreich, Corporate Technology Central and Eastern Europe, Research and Technologies, Wien, Austria

(RECEIVED April 4, 2010; ACCEPTED October 29, 2010)

Abstract

This paper describes and evaluates approaches to model and solve technical product configuration problems using different artificial intelligence methodologies. By means of a typical example, the benefits and limitations of different artificial intelligence methods are discussed and a flexible software architecture for integrating different solvers in a product configurator is proposed.

Keywords: Artificial Intelligence; Constraint Logic Programming; Constraint Satisfaction; Product Configuration; SAT

1. INTRODUCTION

Product configurators have a long history in artificial intelligence (AI; Sabin & Weigel, 1998; Felfernig, 2007), the first and most famous example being the rule-based configurator R1/XCON system (McDermott, 1982) for DEC-Computer. Today there are several established vendors of commercial configurators based on AI methods (SAP, Oracle, ILOG, Tacton, ConfigIt, etc.). Nevertheless, many products especially in technical domains are configured by engineers using in-house software without AI technology. A reason for this may be that there is little literature available on how to use AI methods specifically for product configuration. Therefore, this paper aims at readers with only limited AI background, who are interested in how to model and solve product configuration problems using AI methodologies. For the AI experts it provides insight into how to map a problem between the different paradigms (logic programming, object oriented, constraint based) and proposes a flexible software architecture for product configurators.

Product configurators for technical artifacts pose other challenges than product configurators for customer products. Customer products are typically designed for easy configurability and can be configured by the average customer. Configuring technical systems often requires an engineer with high domain knowledge. Depending on the business domain, various structural, physical, chemical constraints, and so forth, on the assembly of the system or product may arise. With their dependencies between several system components, those constraints can get quite complicated. Although algorithms for general-purpose solvers have been significantly improved over the last years (e.g., Cooper et al., 2008), they turn out to be too inefficient in many cases (Mayer et al., 2009). Then, problem-specific implementations seem necessary. Unfortunately, they also have drawbacks: their maintenance and adaptations to changing requirements are more difficult; often they require deep insight into the nature of the problem that an average knowledge engineer does not necessarily have.

Fortunately, over the last years many (often free) solvers suitable for real-world applications have been developed. In contrast to the monolithic AI systems of the past there is a trend to integrate relatively small specialized AI tools within conventional software systems. A typical example is SAT4J, a satisfiability library, which ships with every instance of the Eclipse integrated development environment (IDE) and is deployed on millions of computers. Most of the users of the Eclipse IDE are even unaware of the AI technology inside of Eclipse.

In the rest of this paper we show a typical product configuration problem as an example for such kinds of real-world problems as well as corresponding solution approaches using different AI methodologies. Section 2 describes a technical product configuration problem. Although seemingly simple, it poses hard efficiency demands on the solving process. In Section 3 an object-oriented model of the problem is derived and some of its properties are analyzed using Unified Modeling Language (UML)/Object Constraint Language (OCL), Alloy, and generative constraint satisfaction problem CSP.

Reprint requests to: Gottfried Schenner, Siemens AG Österreich, Corporate Technology Central and Eastern Europe, Research and Technologies, Siemensstraße 90, A-1210 Wien, Austria. E-mail: gottfried.schenner@siemens.com

In Section 4 we present and evaluate different approaches for solving configuration problems. Section 5 gives a summary of the results and arrives at the conclusion that the challenge for the knowledge engineer consists not only in choosing the right solver(s) for the problem, but also in how to integrate the different solvers into one coherent system. Finally, Section 6 answers that question and proposes an architecture for integrating different solvers into a product configurator framework.

2. PROBLEM DESCRIPTION

The first step for developing a product configurator is the specification of the customer requirements, that is, the properties of the configurable product. As an example we use a fictitious people-counting system for museums. The structure of the problem is similar to problems we encountered in different real-world domains of our product configurators.

A museum has lots of rooms, and there are doors between some of them. In order to prevent damage to the objects in exhibition, the number of visitors shall be restricted. This is done by a people-counting system that consists of the following components: door sensors, counting zones, and communication units.

A door sensor detects everybody who moves through its door (directed movement detection). There can be doors without a sensor.

Any number of rooms may be grouped to a counting zone. Each zone knows how many persons are in it (counting the information from the sensors at doors leading outside of the zone—doors between rooms of the zone are ignored). Correct function requires that all doors leading outside a zone have a sensor (the corresponding constraint is not part of this problem). Zones may overlap or include other zones, that is, a room may be part of several zones.

A communication unit can control at most two door sensors and at most two zones. If a unit controls a sensor that contributes to a zone on another unit, then the two units need a direct connection: one is a partner unit of the other and vice versa. Each unit can have at most N partner units. For the sake of simplicity, we use $N = 2$ throughout this paper, whereas higher values for N are more common in real-life problems of this kind. Of course, the problem diminishes or even vanishes when N is chosen sufficiently high or unbounded, but we assume technical reasons inhibiting high values.

PartnerUnits problem: Given a consistent configuration of door sensors and zones, find a valid assignment of units (i.e., a maximum of two partners) striving for a minimal number of units.

Example 1: Rooms 1 to 8 with doors, eight of the doors having a door sensor, for example, there is a sensor between rooms 1 and 2, or 3 and 4, but not between 2 and 3 (Fig. 1).

Given zones: 1 (white), 2378 (light gray), 45 (dark gray), 6 (medium gray), 456, 2367, 2345678. They are consistent to the door sensors because all doors without sensors are only inside zones. The sensor between 7 and 8 is ignored for zones 2378 and 2345678, but necessary for zone 2367.

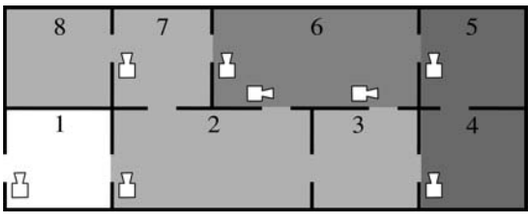

Fig. 1. The room layout of example 1.

The door sensors are named D01, D12, D26, D34, and so forth for the rooms that they connect (and 0 for the outside, respectively). The zones get their names from the rooms that they contain: Z1, Z2345678, Z2367, and so forth.

The relation between zones and door sensors, which is represented as a bipartite graph, is shown in Figure 2. A minimal solution using only four units is, for example, solution 1 in Table 1.

For small examples (i.e., less than six sensors and six zones), it is easy to find a solution. If the number of sensors for each zone is less than three and vice versa, a trivial but far from minimal solution would be to put each zone and each sensor onto a separate unit.

For bigger configurations, the constraint of maximal two partner units makes the problem hard. For example, adding zone Z23 (and new sensor D27) has no solution at all. However, adding Z18 to the free position on unit 4 is a solution. Even adding Z4 (and new sensor D45) has a solution with the minimally achievable number of five units, for example, solution 2 in Table 2.

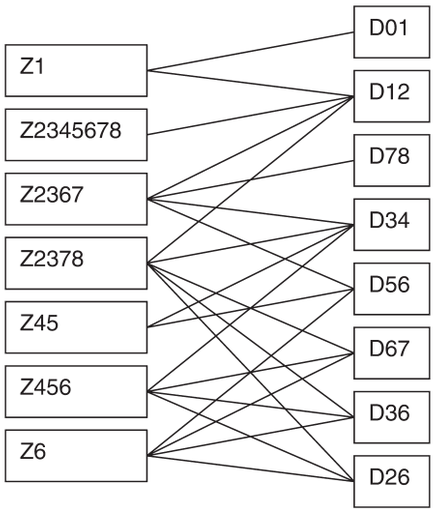

Fig. 2. The relation between zones and door sensors in example 1.

Table 1. *A minimal solution of example 1*

Unit	Zone1	Zone2	Sensor1	Sensor2	Partner1	Partner2
U1	Z1	Z2345678	D01	D56	U3	U4
U2	Z2378	Z456	D34	D67	U3	U4
U3	Z45	Z6	D26	D36	U1	U2
U4	Z2367	—	D12	D78	U1	U2

In addition to finding a consistent solution to the PartnerUnits problem, the following questions may be asked:

- What is the minimal number of units needed? Clearly, the absolute minimum is the smallest integer greater or equal to the half of the maximum of the number of zones and the number of door sensors. However, we do not know whether there is always a solution with such few units.
- Given a partial assignment (i.e., not all sensors or zones have a unit yet), is there a valid extension?
- Reconfiguration: what is the minimal set of changes to already assigned units, so that a valid extension is possible? For example, given solution 1, add Z4 (and new sensor D45) and find a solution with minimal differences to solution 1, that is, change as few assignments to units as possible.

This paper concentrates on finding (preferably minimal) solutions. The other topics are subject to further research.

3. MODELING

After collecting the customer requirements of the product to be configured, the configurator designer must choose an appropriate language and tool for modeling the problem. A model consists of the representation of the configuration components as well as constraints and rules defining valid solutions. For many technical domains, the models get complex and large, so that a high-level modeling language is required. It provides for an easy, natural, and elegant problem description, supporting readability, validation, and maintainability of the model.

We demonstrate the modeling of the PartnerUnits problem prototypically by three languages: UML/OCL (http://www.omg.org/spec/UML/2.3, http://www.omg.org/spec/OCL/2.2), widely used as analysis and design specification language, Alloy (Jackson, 2002), a first-order logic language well suited for associations, and generative CSP (Fleischanderl et al., 1998; Gottlob et al., 2007), which allows the formulation of dynamic problems like configuration as CSPs. For brevity reasons, we do not cover description logics in this article. Description logics is prominent for the formal representation and reasoning on the concepts of complex knowledge networks and is a wide research field in AI (see, e.g., Baader et al., 2003; Felfernig et al., 2003).

3.1. UML/OCL

UML class diagrams (Fig. 3) are a common way to describe the structure of a system in object-oriented modeling. The primary use of UML diagrams is to communicate the model visually inside a software project. In combination with OCL it is also expressive enough to describe product configuration (Felfernig et al., 2002).

The UML diagram shows a class diagram derived from the description. It contains the cardinality constraints, but there is no way to express the fact that the *partner units* association is derived from the path over the *zone2sensor* relation inside the class diagram. It must be expressed using an OCL constraint:

```
context ComUnit inv:
myPartnerUnitsSensor = sensor.zone.unit
  ->excluding(self)->asSet() and
myPartnerUnitsZone = zone.sensor.unit
  ->excluding(self)->asSet() and
myPartnerUnitsSensor->union
  (myPartnerUnitsZone)->size() <= 2
```

Let *myPartnerUnitsSensor* be the set of units reachable from a unit by navigating from its sensors to the zones and then

Table 2. *A minimal solution of extended example 1*

Unit	Zone1	Zone2	Sensor1	Sensor2	Partner1	Partner2
U1	Z1	Z2345678	D01	D78	U2	—
U2	Z2367	Z45	D12	D56	U1	U3
U3	Z2378	Z6	D34	D67	U2	U4
U4	Z456	Z4	D26	D36	U3	U5
U5	—	—	D45	—	U4	—

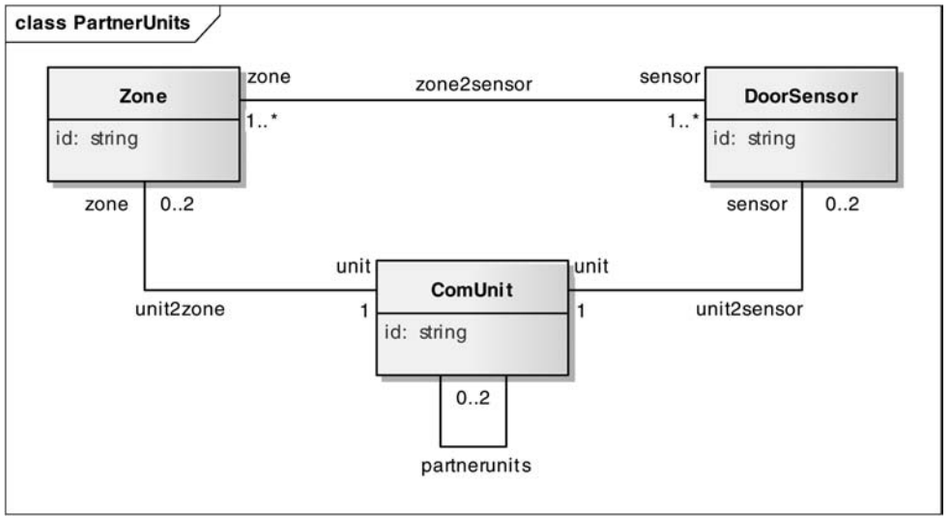

Fig. 3. UML diagram of class PartnerUnits.

their units (*sensor.zone.unit*), and let *myPartnerUnitsZone* be the set of units reachable by navigating from the zones to the sensors and then to the units (*zone.sensor.unit*). Then the cardinality of the union of *myPartnerUnitsSensor* and *myPartnerUnitsZone* (not counting the unit itself) must not be greater than 2.

Although UML/OCL is widely used in software engineering projects especially for the model driven architecture approach, there are few tools available that actually support reasoning with UML/OCL.

One example is the UML-based specification environment USE (Gogolla et al., 2008). It allows the creation of example configurations (called snapshots in USE terminology) and checks the validity of the examples in relation to the UML/OCL specification.

3.2. Alloy

Alloy is a lightweight specification language and tool (Jackson, 2002). The language of Alloy, which is a combination of first-order logic and relational calculus, is relatively easy to learn and use (compared to other specification languages). Using the Alloy Analyzer tool, instances satisfying the specification can be found and assertions about the specification can be checked within a given scope. An Alloy specification of the PartnerUnits problem looks like this:

```
module PartnerUnits
sig Zone{
  zone2sensor: set DoorSensor
}
sig DoorSensor{}

fact cardinalities_zone2sensor{
  all z:Zone | #z.zone2sensor > 0
        // at least 1 sensor for zone
  all d:DoorSensor | #d.~zone2sensor > 0
        // at least 1 zone for sensor
}
sig ComUnit{
  unit2sensor: set DoorSensor,
  unit2zone: set Zone,
  partnerunits: set ComUnit
}
fact cardinalities_unit2sensor{
  all u:ComUnit | #u.unit2sensor <=2
        // at most 2 sensors for a unit
  all d:DoorSensor | #d.~unit2sensor = 1
        // 1 unit for a sensor
}
fact cardinalities_unit2zone{
  all u:ComUnit | #u.unit2zone <=2
        // at most 2 zones for a unit
  all z:Zone | #z.~unit2zone = 1
        // 1 unit for a zone
}
fact derivedassoc_partnerunits{
  partnerunits =
       unit2zone.zone2sensor.~unit2sensor +
       ~(unit2zone.zone2sensor.~unit2sensor)
       -iden
}
fact cardinalities_partnerunits{
  all u: ComUnit | #u.partnerunits <= 2
        // at most 2 partner units
  }
```

The *sig* definitions of the Alloy specification correspond to the UML class definitions. An expression like *zone2sensor* denotes the binary relation between *Zone* and *DoorSensor*. The symbol "$\sim$" denotes the inverse of a relation, that is, $\sim$*zone2sensor* is the relation from *DoorSensor* to *Zone*. The dot operator denotes the relational join: for example, a navigational expression like *unit2zone.zone2sensor* evaluates to a binary relation that relates the units to all sensors that belong to a zone of the unit.

Using Alloy as an instance (model) finder, we can analyze the specification. Suppose we want to verify whether the specification allows the existence of partner units at all. By executing

```
run{some u:ComUnit | #u.partnerunit = 1} for 4
```

Alloy finds all instances of the specification (within the scope, in this case up to four instances for every class) that contain at least a unit with exactly one partner unit. If no instance is found, we know that we made an error, for example, by overconstraining the specification. If unexpected instances are found then there are still constraints missing. This is very useful to detect inconsistencies in the knowledge base at an early stage.

Furthermore, we can check assertions about the specification that may lead to additional constraints. For instance, it is easy to conclude that any configuration containing a zone with more than six door sensors is inconsistent. We can prove this assumption (within the given scope) by checking the following assertion:

```
check{ all z: Zone | #z.zone2sensor <= 6}
       for 10 but 5 int
```

Alloy tries to find a counterexample, but because the assertion is valid, it does not succeed. Because the problem is symmetric for zones and sensors, there cannot be more than six zones for a sensor as well. Therefore, the cardinalities of both sides of the *zone2sensor* association can be restricted to 1..6 (from 1..*). Deriving cardinality restrictions from an UML model is an area of active research (Falkner et al., in press). Such restrictions are very valuable for ruling out inconsistent requirements (such as in the classical pigeonhole problem) at an early stage of the configuration process.

3.3. Generative constraint satisfaction

Constraint satisfaction is widely used to represent and solve configuration problems. A CSP in the classical sense consists of a fixed set of variables and their domains, as well as constraints that restrict the assignment of the variables. A valid solution is an assignment of all variables with values from their domains where all constraints are satisfied. A formulation of the PartnerUnits problem as a standard CSP using the open source constraint library Choco can be found in Section 4.9.

In our problem, zones and door sensors are input values and therefore fixed, but the number of communication units is not. Thus, the formulation as a generative CSP instead of a classical, static CSP is appropriate (Fleischanderl et al., 1998; Gottlob et al., 2007).

The modeling of our problem in a generative configurator framework looks like this: zones, door sensors, and communication units are the component classes.

```
class Zone
class DoorSensor
class ComUnit
```

Each class represents a theoretically infinite set of instances (i.e., components). A class can have attributes, associations, and constraints. Whenever a new instance of a class is created, instances of its attributes, associations, and constraints are created also. This is the object-oriented view of the modeling.

From the constraint-oriented point of view, attributes and associations represent the variables. The domain of an attribute variable is its type, for example, Boolean, an integer interval, and so on. Associations are bidirectional and induce two association variables, one for each side. The domain of such an association variable is the set of all instances of the class specified on the other side of the association. For example, the association definition

```
assoc Zone.unit(1) - ComUnit.zones(0..2)
```

represents the connection of zones to units. The two association variables induced are *Zone.unit* (the link from a zone to its unit) and *ComUnit.zones* (the link from a unit to all its associated zones). Allowed cardinalities are given in brackets. Implicit constraints check that all instances associated to an association variable are of the specified type (e.g., *ComUnit* for *Zone.unit*) and that the given cardinalities are not violated (e.g., *Zone.unit* must contain exactly one instance).

The set of all associations in our problem are the following:

```
assoc Zone.sensors(1..*)
      - DoorSensor.zones(1..*)
assoc Zone.unit(1) - ComUnit.zones(0..2)
assoc DoorSensor.unit(1)
      - ComUnit.sensors(0..2)
assoc ComUnit.partnerunits(0..2) - self
```

A constraint in the context of a *ComUnit* instance specifies which elements are to be in the *partnerunits* association of that unit. These are all units reachable via its zones and its sensors, where the unit itself is not member of the association.

```
constraint ComUnit.derivedPartners :
    Partnerunits = zones.sensors.unit
    + sensors.zones.unit - self
```

Typical tasks in a CSP are to decide whether a given problem has a solution, to find a valid solution (i.e., a consistent assignments to the variables *Zone.unit* and *DoorSensor.unit*), and to find a good/optimal solution (i.e., one with few or a minimal number of units).

Generative CSP is well suited for the modeling of configuration problems because of its object-oriented touch (natural and maintainable formulation of the problem structure), its constraint-orientedness (declarative formulation of the problem logics), and its dynamicity. Suitable solvers (e.g., backtracking, heuristic repair, SAT) can easily be integrated.

4. SOLVING

In this section we investigate different solving strategies for the PartnerUnits problem. As there is a huge number of solving and

search algorithms available as tools and in the literature, we selected a variety of typical proponents, trying to cover a wide spectrum of distinct approaches. Of course, this selection cannot claim to be exhaustive. Our aim was to apply existing techniques/tools and analyze their practicability to our problem.

The main questions are the following:

- How easy is it to apply a particular solver to our problem?
- How easy is the mapping of a high-level problem description to a particular solver?
- How powerful and efficient is the solver on our problem?

We used the techniques/tools in a straightforward way without trying to invent new or improved solving algorithms, taking over the role of an average knowledge engineer who wants to use existing AI technology and adjust it to her/his needs.

4.1. Backtracking search

Backtracking is a well-established technique for generating one or all solutions of a CSP by incrementally finding assignments to the variables, ruling out branches where constraints are violated. In case of a dead end, the chronologically last choice is retracted and another option is investigated.

There are two spots to control which branches are visited in which order: choose the variable that is to be assigned next; choose the value for the current variable from its domain. For good performance, it is important to find a statical or dynamic order that recognizes and prunes inconsistent branches as soon as possible. There are several established heuristics that are known to perform quite well, for example, for variable ordering: dynamic search rearrangement, preferring variables with a minimum number of consistent values; for example, maximum cardinality ordering for values (see Dechter & Meiri, 1989).

Domain-specific heuristics are promising as well. For instance, sort the variables so that zones and door sensors that belong together are handled consecutively, increasing the chance that zones and their sensors are placed on the same communication unit.

Beside the simplicity of the backtracking algorithm, the main advantage is its completeness: if there is a solution to a problem, backtracking will find it. It can also find all solutions, if necessary. However, the price is high computational costs. Big problems with complex dependencies usually cannot be solved with backtracking.

There are improved backtracking algorithms (such as backjumping or backmarking), but they are not as easy to implement as basic backtracking, and they normally do not improve the applicability of backtracking by orders of magnitude.

Symmetry breaking is another way of improving the performance. It tries to avoid choices that are symmetrical to already made choices that have been proven to be invalid (see, e.g., Gent et al., 2006). To find and represent all symmetries in a configuration problem is usually a complex task. However, often the main symmetries are easy to find and avoid.

In our problem, the following symmetries are already excluded by the choice of representing the problem: it does not matter if a zone is connected to the first or second zone-port of a communication unit. We do no represent communication ports, but only the connection of the zone to the unit. The same is valid for the connection of a door sensor to the communication unit.

However, another symmetry can easily be identified. If a zone or door sensor is to be connected to a unit, all units that are not yet connected to another zone/door sensor, are symmetrical. If we can prove that one of them does not lead to a solution, we know it for all the others. We exploit this symmetry by removing all units from the domain of the current variable that are not used yet, keeping just one of them. This is a considerable improvement and allows for tackling larger problems.

4.2. Generative backtracking search

The classical form of backtracking is able to solve static problems, where all variables and domains are known beforehand. For solving the dynamic PartnerUnits problem with backtracking, an iterative widening approach can be used.

Create a minimal number of communication units and try to solve this now static problem with classical backtracking. If backtracking does not find a solution, add a new unit and try again to search with backtracking. Do this until a solution is found or the maximum number of units is reached.

The minimum amount of units is obviously

$$\min = \frac{\max(|zones|, |sensors|)}{2}$$

because each communication unit can take up to two zones/door sensors. A generous upper bound is

$$\min = |zones| + |sensors|.$$

This simple iterative widening backtracking approach guarantees to find a solution, if one exists, and furthermore, it finds the solution with the minimum number of units. However, keeping in mind that backtracking often performs quite badly on problems with no solution (because the whole search tree is traversed), we cannot expect high efficiency on hard problems, where the minimum number of units is not sufficient.

We could use a lower value for max (e.g., min + 2) in order to iterate less, but would lose completeness of search (unless there is a proof that whenever a solution exists, there is also a solution for that lower value, e.g., min in the optimal case).

A variant of this approach is to create the maximum number of communication units and guide backtracking search so that the assignment of a so far empty unit to the current variable is delayed until all other domain values of that variable lead to a conflict. When a solution is found, remove all unused units. This algorithm is easy to implement and complete, but it does not guarantee that the first solution found is a minimal one. Another disadvantage is that the implementation of

sophisticated domain-ordering heuristics is difficult because of the fact that unused units are to be delayed.

Another approach is generating components during search. It modifies the static backtrack search so that in certain situations new components are generated. In our problem, we add a special wildcard domain value *new-unit* to all domains of our unit variables of the zone and door sensor components. If this value is selected, a new unit is generated and used. This *new-unit* value is added at the end of each domain, which means that new units are generated only if the current set of units is not sufficient. If a new unit is generated but a dead end is reached, the new unit is destroyed.

The usage of wildcard components is very similar to the semifinite sets described in Albert et al. (2008), where possibly infinite domains are made quasifinite inventing such wildcards.

This generative backtrack search with wildcard components needs no initial units generated, because the units are created during search. It is complete, but it is not guaranteed that the first solution found is a minimal one. It depends on which branches are investigated first. It could be that the only way out of a dead end situation is the generation of an additional unit, which would not be necessary, if a better constellation has been chosen in previous steps.

The performance of generative backtracking is comparable with the classical version of backtracking, but with superior suitability and elegance in solving dynamic problems. All these three methods can easily be advanced by symmetry breaking as described above.

4.3. Heuristic search methods

In heuristic search methods, a heuristic function is used to locally guide the variable assignment during search. These methods are normally fast but not complete, which means that it is not guaranteed that a solution is found even if one exists.

The crucial part of these methods is the definition of the heuristic function. For the PartnerUnits problem, we use the following terms as part of a multiobjective heuristic function:

- *vc(sol):* The number of violated constraints in the (partial) solution *sol*. This value is to be minimized. A solution is valid, if $vc(sol) = 0$.
- *mp(sol):* The sum of all *partnerunits* connections in the (partial) solution *sol*. Minimizing this value results in compact solutions, where zones and door sensors that belong together are preferably situated on the same communication unit. Although, this does not contribute directly to the goal of a minimal number of units, it directs search earlier to better results.
- *mu(sol):* The number of units that have at least one zone or door sensor assigned. Minimizing this value results in minimizing the number of units used in the solution.

It is interesting that we heuristically guide search not only to find a good solution but also, and of more importance, to find a valid solution: by the term *vc(sol)*. Therefore, when combining the three terms to a single value, the *vc(sol)* has the highest weight, favoring a valid solution over a smaller invalid solution.

The general metaheuristic of heuristic algorithms is to first make an initial assignment of the problem variables, and then iteratively improve that assignment using problem-specific heuristic functions (like fitness functions) until a valid and acceptable solution is found.

Sections 4.4 to 4.8 contain different heuristic algorithms.

4.4. Domain-specific heuristics

For comparison to the general algorithms, we implemented a simple problem-specific heuristic: place zones having sensors in common on to the same units, starting with those having higher cardinality. If it does not find a solution, try simple repair steps that swap unit allocations one-by-one, striving to reduce the number of violated constraints.

Of course, this algorithm will not always find a valid solution, but is expected to perform well on weakly connected configurations.

4.5. Iterative repair

For making an initial assignment, a greedy technique shows potential: for each variable, the locally best choice is made, hoping that this leads to a good solution candidate with no or only few violated constraints. Because our PartnerUnits problem does not have an optimal substructure (optimal substructure means that it is guaranteed that each best local choice leads to a solution), greedy assignment will normally return an invalid solution candidate.

To correct this initial assignment to a valid solution, iterative repair can be used. Iterative repair continually tries to improve the constellation, hoping to end up at a valid solution. To avoid a local optimum, choices during search have a probabilistic aspect, possibly leading to temporary solution candidates that are worse than the best one already found. A maximum number of cycles or a timeout avoid endless loops, especially in cases where no solution exists.

```
// iterative repair pseudo-code
// sol = (initial) assignment of all variables
// h(sol) = heuristic function as combination
            of vc and mp

iterative_repair(sol)
  if (sol is consistent)
    return sol // solution found
  if (timeout)
    return sol // timeout,
                  no valid solution found
  A :={}
  vars := all variables which are involved
          in violated constraints
  for each var in vars
     for each val in domain(vars)
        A = A + <var, val>
  <var, val> = choose(A, h, p)
```

```
new-sol = sol/var:=val // exchange
                  var assignment in sol (repair)
iterative-repair(new-sol)
```

The goal in each cycle of the algorithm is to change the value of a variable so that as much conflicts as possible are repaired. The function `choose` selects a repair assignment from all possible repair assignments. Using the probabilistic function `p`, which is not always the best candidate with regard to the heuristic function `h`, is chosen, but of course, preferring those with low `h` values. This may lead out from a local optimum. Another way to break out from a local optimum is to restart search with a different initial constellation.

Dynamic configuration problems can only be tackled with iterative repair by providing a fixed set of components and try to solve this problem that is a static one now. The method of iteratively increasing the components (see iterative widening in Section 2.2) can be applied. Creating components during search is not possible because search does not investigate the search space in a determined manner. It would be hard to decide if the creation of a new component or the deletion of an existing one leads to a solution.

Although iterative repair is not complete, for several problem classes it performs very well and has a high probability to converge fast at a solution if one exists.

4.6. Simulated annealing

A slight change in the iterative repair algorithm leads to an algorithm in the style of simulated annealing (Kirkpatrick et al., 1983). The basic idea of simulated annealing is to change the probabilistic function p, which chooses the next repair assignment, during search. In analogy to metallurgy, the probabilistic function reflects the temperature of the system. High temperature means that variables and values to be repaired are chosen almost randomly, ignoring the scheme of preferring repair steps that improve the current constellation best. Over time, the temperature is gradually cooled down, that is, the probability of choosing the best candidate increases.

The idea behind this approach is to move the system out of local optima in the start phase of search, and with the proceeding of time, to improve the system by taking more and more attention to the heuristic function leading, hopefully, to a valid and good solution.

4.7. Genetic algorithm (GA)

A quite different approach to solve the PartnerUnits problem is to use an evolutionary technique like a GA (see Goldberg, 1989). GAs (Fig. 4) are built on the metaphor of Darwin's evolution theory. In a population the fittest individuals survive and evolve to the next generation. Variations are induced by mutation and recombination.

To use a GA for solving our configuration problem, we have to define a mapping from the configuration world to the GA world, and we have to provide a fitness function.

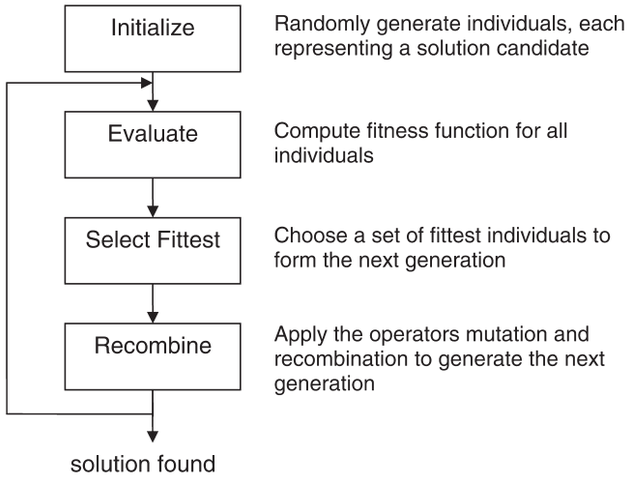

Fig. 4. The genetic algorithm schema.

As fitness function we use a heuristic function as described in Section 4.3. It is a multiobjective function combining the sum of violated constraints and the sum of partnerunits. We just have to multiply this heuristic function by -1 to inverse its meaning: low values reflect bad fitness, high values (with maximum 0) good fitness.

In addition, the mapping from CSP to GA is straightforward. CSP variables represent the connections of zones and door sensors to communication units. Each such variable is mapped to a gene in the GA chromosome. These genes are not binary, but are a number representing the index of the unit in the list of all units.

In the simple example in Figure 5 we have two zones z1 and z2, four door sensors d1 to d4. Zone z1 is associated with d1, d2, and d3. Zone z2 is associated with d3 and d4. Of course, to use GA for our dynamic problem, we have to provide a set of units using the iterative widening method described in Section 2.2. Thus, we provide two units u1 and u2.

Now we map our variables *z1.unit*, *z2.unit*, *d1.unit*, . . . , *d4.unit* to six genes, each is having the possible values 1 or 2, representing units u1 and u2. Each chromosome configuration (Fig. 6) uniquely represents a variable assignment.

The recombination operator (Fig. 7) performs a crossover of two chromosomes. Normally, a crossover point is chosen randomly, and the new chromosome is built from the head of the first chromosome and the tail of the second one.

The mutation operator (Fig. 8) changes the value of one or more genes: which genes to change and which new values are completely random choices.

During GA search, each new individual chromosome is mapped back to our CSP model, which provides the follow-

Fig. 5. A simple example with two zones and four door sensors.

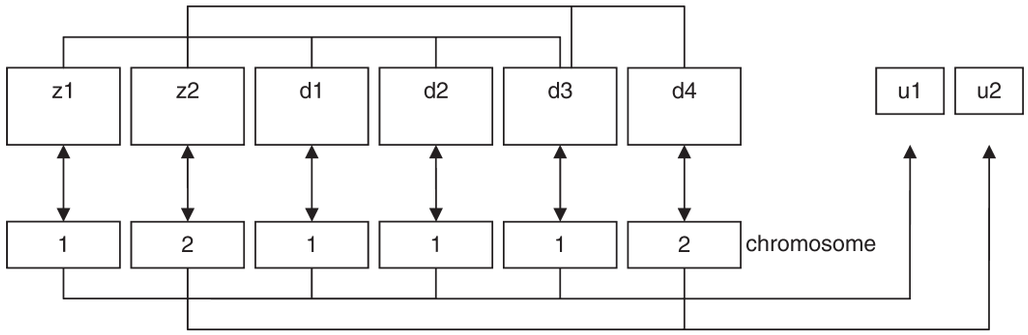

Fig. 6. The chromosome configuration.

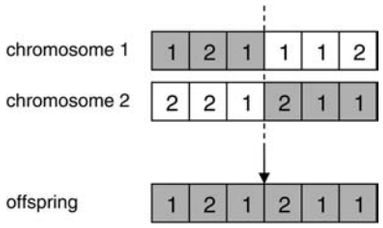

Fig. 7. The recombination operator.

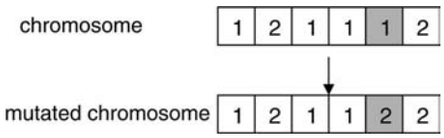

Fig. 8. The mutation operator.

```
// ACO pseudo-code
aco()
  do until solution found or timeout
    for each ant in the colony
      create solution, preferring choices
            with high pheromone
    update pheromone map with best solution
          in this iteration
```

ing information: is this individual a valid solution? What is the fitness of this individual?

For implementation we use the JGAP package. Following the idea of clearly separating the model from the solver (see Section 4), we let the model provide the fitness function and kept domain-specific heuristics out of the GA solver. Integrating domain-specific heuristics in the gene combination steps certainly would improve GA performance, but with the loss of simple and straightforward integration into a complex configuration environment.

4.8. Ant colony optimization (ACO)

ACO is a probabilistic technique based on how a colony of ants finds paths to food sources. Each individual ant is laying down a pheromone trail. Other ants follow such trails. The more pheromone is on the trail, the more likely other ants follow that trail. In that way, high-pheromone trails develop over time on short paths.

ACO is typically used for graph search problems, like the traveling salesman problem. However, ACO can also be applied to other problem fields, like solving CSPs (Schoofs & Naudts, 2000; Khichane et al., 2008) and configuration problems (Albert et al., 2008).

We apply ACO to the PartnerUnits problem in a straightforward way. Each ant of a colony assigns a value to each variable iteratively, preferring choices with high pheromone values. Pheromones are stored for each variable-value pair in a pheromone map. The first iterations are fully random choices. However, preferred paths emerge over time.

The best solution candidate in an iteration, which is the one with best fitness function, is rewarded in the pheromone map by the following formula:

$$\tau_{ij} \leftarrow (1 - \rho)\,\tau_{ij} + \Delta_{ij},$$

where τ_{ij} is the pheromone value of variable assignment $\text{var}_i = \text{val}_j$; ρ is the evaporation rate, and pheromones evaporate over time to forget bad choices; and Δ_{ij} is the amount of pheromone. It is zero, if variable assignment $\text{var}_i = \text{val}_j$ is not in the best solution candidate of the colony. Otherwise it is (total number of constraints/number of violated constraints + 1); that is, the better the solution, the higher the pheromone drop.

Like GAs, ACO is very general in the sense that it requires only a minimum amount of problem-specific knowledge: only a fitness function for rating an individual solution is needed. Unfortunately, ACO in its plain version does not perform very well on the PartnerUnits problem because of its complicated and highly connected inner structure. The usage of higher sophisticated variants of ACO along with domain-specific heuristics and local search could possibly be more successful in dealing with the PartnerUnits problem. However, the goal of this study was to plainly use ACO without highly specialized expertise and without packing any domain knowledge into the solver.

4.9. Choco

Choco is an open source constraint library written in Java. To encode the assignment of sensors and zones to the units, we use the two-dimensional IntegerVariable arrays *sensor2unit* and *zone2unit*. Each array represents one of the relations of our problem, that is, sensor *i* is associated with unit *j*, if and only if *sensor2unit[i][j]* = 1. To ensure that a sensor/zone *i* is only assigned to one unit, a constraint is added that the sum of integer variables in each row must be 1.

```
Choco.eq(Choco.sum(sensor2unit[i]), 1))
Choco.eq(Choco.sum(zone2unit[i]), 1))
```

In addition, there must not be more than two sensors or zones for every unit. This is ensured by the constraint

```
Choco.leq(Choco.sum(column), 2);
```

for every column of the arrays *zone2unit* and *sensor2unit*.

To restrict the number of connections between units another two-dimensional array *partnerunits* is needed. *partnerunits[i][j]* = 1, if there is a connection between unit *i* and unit *j*. The following constraints are posted:

```
Choco.leq(Choco.sum(partnerunits[i]),2)
              // There must not be more than
                2 Partners for every unit
Choco.eq(partnerunits[i][j],
         partnerunits[j][i]) for i != j
              // The relation is symmetric
```

Whenever there is a zone assigned to unit *i* and one of its sensors assigned to a different unit *j*, there must be a connection between the units:

```
Choco.implies(Choco.and(
Choco.eq(zone2unit[zoneindex][i],1),
Choco.eq(sensor2unit[sensorindex][j],1)),
Choco.eq(partnerunits[i][j],1))
```

Given this encoding Choco can solve the basic example without the need of additional heuristics. If the solver finds a solution, mapping the result back to an object-oriented model is straightforward. For every *IntegerVariable vij = 1* of the arrays *zone2unit* and *sensor2unit*, associate zone/sensor *i* with unit *j*.

4.10. KodKod

KodKod is a SAT-based constraint solver for relational logic (Torlak, 2009). Alloy 4, which is based on KodKod, can convert Alloy specifications to KodKod-Java source files. We used this option to translate our Alloy specification from Section 1.1 to KodKod.

The KodKod Solver works by translating the problem to a SAT-problem. The SAT-problem is then solved by an external SAT-Solver (such as SAT4J). Therefore, the use of KodKod (like all SAT-based approaches) is limited by the number of the created clauses for encoding the problem as a SAT-problem. Although KodKod may not be suitable for solving problems with many instances (>30), its ability to enumerate models is for instance convenient for generating test cases.

Mapping the results of the solving process back to the source model is particular easy because KodKod allows using the Java objects of the source model directly as atoms in the relations. Thus, for instance, translating the relation between units and sensors back to our object-oriented model looks like this:

```
unit2sensor = Relation.nary
              ("this/ComUnit.unit2sensor", 2);
Iterator<Tuple> unit2sensorItor =
            solution.instance().
            tuples(unit2sensor).iterator();
   while(unit2sensorItor.hasNext()){
     Tuple t = unit2sensorItor.next();
     ComUnit u = (Unit)t.atom(0);
     DoorSensor d = (DoorSensor)t.atom(1);
     d.setUnit(u);
}
```

4.11. DLV (Datalog)

DLV Complex is an Answer Set Programming System extending DLV, a system for disjunctive datalog with constraints, true negation, and queries (Eiter et al., 1997). It offers a very concise representation of the problem.

The relation between zones and door sensors and the maximally usable amount of communication units are given as positive facts. The implicit unique name assumption ensures that the listed zones (z1, z2) and door sensors (d1, d2, d3, d4) are considered different, for example, for the example in Section 4.7:

```
zd(z1,d1).
zd(z1,d2).
zd(z1,d3).
zd(z2,d3).
zd(z2,d4).
unit(1..2).
```

We formulate the possible assignment of units to zones and sensors as a disjunction of positive and negative facts. Constraints restrict their cardinalities: not more than two zones per unit, exactly one unit per zone (analogous for sensors):

```
zu(Z,U) v -zu(Z,U) :- zd(Z,_), unit(U).
:- unit(U), not #count{ Z: zu(Z,U)} <= 2.
:- zd(Z,_), not #count{ U: zu(Z,U)} = 1.
du(D,U) v -du(D,U) :- zd(_,D), unit(U).
:- unit(U), not #count{ D: du(D,U)} <= 2.
:- zd(_,D), not #count{ U: du(D,U)} = 1.
```

Similarly, we calculate and restrict the partner units:

```
pu(U,P) :- zu(Z,U), zd(Z,D), du(D,P), P!= U.
pu(U,P) :- pu(P,U).
:- unit(U), not #count{ P: pu(U,P)} <= 2.
```

Running the program and filtering the relevant facts `zu`, `du`, and `pu` results in

```
zu(z1,1). zu(z2,1).
du(d1,1). du(d2,1). du(d3,2). du(d4,2).
pu(1,2). pu(2,1).
```

In addition, the complete program has a final step that pretty-prints the result (omitted for brevity).

It performs well for finding solutions for medium-sized problems like the example at the beginning of this paper. However, it takes very long to realize if there is no solution at all. Furthermore, it will not find minimal solutions if there are too many units available. One can avoid this by setting the number of available units to the minimum (e.g., *unit(1..2)* in the example above).

The use of weak constraints for optimizing the solution (in case the number of available units is higher than the number of necessary ones) increases the runtime considerably so that it is not recommended:

```
used(U) :- zu(_,U).
used(U) :- du(_,U).
:~ used(U).
```

4.12. Constraint handling rules (CHR)

CHR is a declarative concurrent committed-choice constraint logic programming language consisting of guarded rules that transform constraints (represented as multisets of relations) until no more change occurs (Frühwirth, 2008). With its built-in reasoning mechanism for simplification and propagation rules it is well suited for optimizing constraint satisfaction.

We use CHR with host language SWI-Prolog. Therefore, we can take advantage of Prolog's inference machine and variable unification: the relation between zones and door sensors as an input is represented by facts `zdu/2`, which relate variables that later will be instantiated to the unit for that zone or sensor (exploring an idea of Frühwirth, 2009, personal communication), for example, for the simple example in Section 4.7:

```
?- zdu(Z1,D1), zdu(Z1,D2), zdu(Z1,D3),
   zdu(Z2,D3), zdu(Z2,D4),
   label([z1-Z1,z2-Z2],[Z1,Z2],
         [d1-D1,d2-D2,d3-D3,d4-D4],
         [D1,D2,D3,D4],[1,2]).
```

They are simplified to relations for the partner units (pu/2): when both arguments are bound (i.e., zone and sensor are placed on a unit), then the two units are related as partners.

```
zdu(ZU,DU) <=> nonvar(ZU), nonvar(DU)
             | pu(ZU,DU), pu(DU,ZU).
```

Partner units are optimized and restricted (to two): remove reflexive and duplicate relation instances with simplification rules that have a "true" body. Raise a failure when there are too many partner units for a given unit (all anonymous variables "_" in the third rule are considered different).

```
% same unit is never a partner
pu(U,U) <=> true.
% remove duplicates
pu(U,PU) \ pu(U,PU) <=> true.
% not more than 2 partner units
pu(U,_), pu(U,_) \ pu(U,_) <=> fail.
```

The placement of doors and sensors to units is done by a naïve labeling of zones and sensors with a unit, which at first tries to place two zones and two sensors onto one unit, and only if it fails, places fewer ones. By using variable binding for that, it realizes a natural way of symmetry breaking. If a variable is bound then it triggers generation of the partner unit in the simplification rule of *zdu/2*.

```
label(Zs,ZVs,Ds,DVs,[U|Us]) <=>
            % symmetry breaking
            (ZVs=[U|ZVs1]; ZVs=[],ZVs=ZVs1),
            (select(U,ZVs1,ZVs2); ZVs1=ZVs2),
            (select(U,DVs,DVs1); DVs=DVs1),
            (select(U,DVs1,DVs2); DVs1=DVs2),
            label(Zs,ZVs2,Ds,DVs2,Us).
label(_,_,_,[_|_],[]) <=> fail.
label(_,[_|_],_,_,[]) <=> fail.
label(Zs,[],Ds,[],_) <=> label(Zs,Ds).
```

The results are the facts for the partner units and the final labeling, for example,

```
label([z1-1,z2-1],[d1-1,d2-1,d3-2,d4-2]),
      pu(1,2), pu(2,1).
```

The complete program has in addition a preparation step that creates the initial query and a final step that pretty-prints the result (omitted for brevity).

The program performs similar as the backtracking approach. It prunes dead ends early and finds good solutions for smaller problems quite fast. However, sometimes it does not find a solution within a reasonable time period, and it always takes a long time to detect that there is no solution at all.

5. EVALUATION OF THE RESULTS

In the preceding section we presented several approaches to solve the PartnerUnits problem: various general-purpose solvers parameterized to the problem (Choco, KodKod, DLV), different AI methods adapted to the problem (generative backtracking, iterative repair, simulated annealing, GA, ant colonies), and problem-specific algorithms (domain-specific heuristic and repair, CHR).

The problem could be mapped to all of them with only little effort. Some of them are easier to understand (e.g., the object-oriented approaches and DLV with their close relation to real-world concepts) than others (e.g., Choco because of the mapping of object connections to integer arrays).

Clearly, analyzing the properties of the problem (like complexity) and exploiting them in the algorithms would help to improve their performance. However, it takes time and mathematical expertise to get deep insight in the problem, which may not be available for the average knowledge engineer in real-world projects. Furthermore, tuning algorithms or implementing special solutions for better performance tends to be expensive and difficult to adapt when requirements change. We are sure that experts for the used tools can do better than us. However, we wanted to evaluate the results for average knowledge engineers.

We tested all algorithms described in the previous sections on the following example configurations:

- small: examples from Section 2; the first with seven zones (example 1), the second with eight (solution 2), the third (with eight zones) having no solution (small-no); see Table 3.
- single: a highly packed configuration with 11 zones, 6 sensors, and 22 connections between them; see Table 3.
- double (see Fig. 9): a double row of connected rooms, each room being a zone (number of zones given as parameter); a variant (dv in Table 4) has additional zones for each two connected rooms vertical to the row; to be solved with maximum of partner units raised to three or four for the variant, respectively (as no solver found a solution with smaller bound for partners within the given time frame)
- triple (see Fig. 10): a weakly connected group of rooms, each room being a zone (their number given as parameter); in some cases with additional two or four zones consisting of 2 to 3 rooms; in Table 5 we used one to four blocks of "width" 10 (i.e., of 30 rooms); to be solved with max partners raised to 4

The input data for the evaluation as well as some of the used algorithms are available by e-mail from the authors.

Tables 3–5 summarize the results as the time for finding a valid solution on a 2-GHz PC; or in the case of small-no, for finding a proof that the problem has no solution. The time is given in seconds (i.e., the numbers in the table). A "—" means time out, that is, no solution was found within 30 min. We introduced this time out because the users expect a result within a few minutes. For some examples, the domain-specific algorithms got stuck in a local optimum and gave up before time out, which is represented by an "x." Furthermore, "m" means that an algorithm ran out of memory before time out (memory was limited to 600 MB).

Table 3. *Evaluation results*

Examples	Small-7	Small-8	Small-no	Single-11
4.2 Generative BT	1	1	12	6
4.4 Domainspec. heuristics	x	x	x	x
4.4 Domainspec. repair	x	x	x	x
4.5 Iterative repair	1	—	—	—
4.6 Simulated annealing	1	1490	—	—
4.7 Genetic algorithm	6	—	—	499
4.8 Ants	1002	—	—	—
4.9 Choco (classical CSP)	1	251	—	155
4.10 KodKod (SAT)	1	1	79	1
4.11 DLV	3	8	—	2
4.12 CHR	1	349	—	123

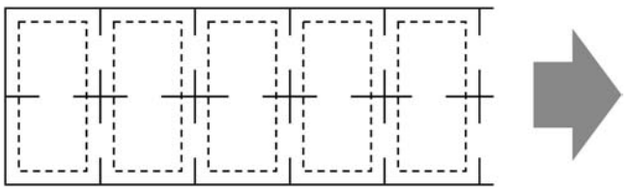

Fig. 9. Example configuration "double."

It is interesting that the general heuristic methods (like simulated annealing or GA) do not perform very well on large PartnerUnits problems, because of the complicated inner structure of the resulting configurations. These methods are better be used for problems where the focus lies on optimization, and not on consistency. The main advantage of these heuristic methods is the possibility to give a time limit. Although the most complete methods, like backtracking, have no solution at all if stopped after a time out, heuristic methods most often return a solution candidate that is close to a consistent solution.

As expected, the domain-specific algorithms perform very well for most of the large but simple (i.e., weakly connected) examples. Unfortunately, they do not find solutions to harder problems even when they are quite small (e.g., see Table 3).

We conclude that depending on the problem and even on the problem instance, different solving strategies are necessary to arrive at a valid solution. Therefore, a configuration system needs an architecture that allows selecting suitable solvers, dependent on properties, structure, and size of the problem.

6. A FLEXIBLE CONFIGURATION ARCHITECTURE

A configurator roughly consists of three main components: the modeling framework, a solving engine, and interfaces to the user, file data, a database, Web services, and so forth. A clean separation of these components allows for using best-fitting technologies and frameworks for each part. Especially the separation of and relationship between modeling and reasoning is crucial. The surrounding interfaces are not in the focus of this paper.

For complex engineering domains, it is not appropriate to model the configuration problem using a general-purpose solving framework, like a standard CSP or a GA framework. To achieve efficient, natural, and easy to maintain knowledge modeling, an object-oriented type hierarchy, augmented by powerful constraint and rule concepts is state of the art. Such a knowledge base supports design and implementation of the surrounding interface components in a straightforward way.

In contrast, specialized reasoning capabilities are required, ranging from domain filtering, satisfiability checks, finding a

Table 4. *Evaluation results*

Double Examples	d-20	dv-30	d-40	dv-60	d-60	d-80	d-100
4.2 Generative BT	1	—	1	—	2	4	6
4.4 Domainspec. heuristics	x	x	x	x	x	x	x
4.4 Domainspec. repair	3	15	37	16	135	x	x
4.5 Iterative repair	1	55	2	—	6	153	134
4.6 Simulated annealing	35	—	149	—	138	84	350
4.7 Genetic algorithm	—	—	—	—	—	—	—
4.8 Ants	—	—	—	—	—	—	—
4.9 Choco (classical CSP)	1	—	9	—	m	m	m
4.10 KodKod (SAT)	25	130	m	m	—	—	m
4.11 DLV	—	—	—	—	—	—	—
4.12 CHR	—	—	—	—	—	—	—

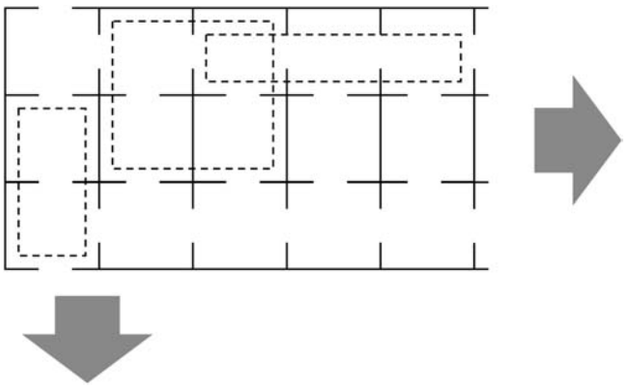

Fig. 10. Example configuration "triple."

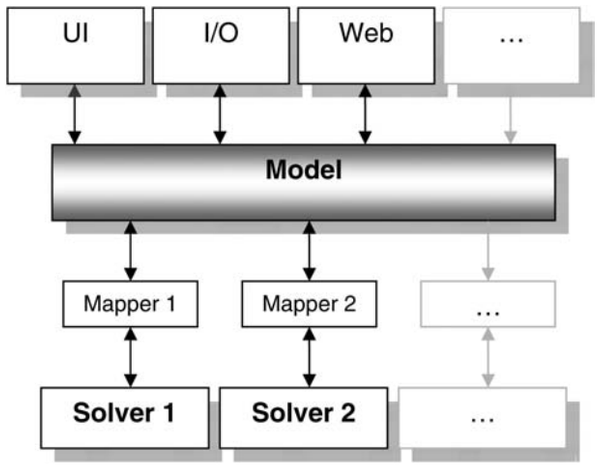

Fig. 11. A flexible configuration architecture.

valid solution, to finding the *n* best solutions. Unfortunately, there is no silver bullet capable of handling all these tasks. Thus, the architecture should be open to plug in different solvers for the different purposes. Figure 11 sketches such an architecture.

The problem is modeled in a high-level language. Specialized mappers transform the problem or parts of the problem to appropriate solvers. In turn, the solver results are mapped back to the high-level model, ready to be presented to the user or exported to other processes. Typically, the solver needs information from the model about the model state during search (e.g., the state of the constraints or the value of a fitness function).

For example, the PartnerUnits problem is modeled in a high-level language and solved using a GA solver (see Section 4.7). First, the problem is mapped to genes in the GA language. In addition, solving parameters are provided, like the maximum number of generations, a time out, or the population size.

Table 5. *Evaluation results*

Triple Examples	t-30	t-32	t-34	t-60	t-64	t-90	t-120
4.2 Generative BT	3	3	—	11	—	29	65
4.4 Domainspec. heuristics	1	x	x	1	x	x	x
4.4 Domainspec. repair	1	5	1	1	1	1	x
4.5 Iterative repair	5	6	76	35	—	93	1473
4.6 Simulated annealing	5	5	150	113	—	1209	—
4.7 Genetic algorithm	—	—	—	—	—	—	—
4.8 Ants	—	—	—	—	—	—	—
4.9 Choco (classical CSP)	3	—	—	150	—	m	m
4.10 KodKod (SAT)	602	m	m	—	—	—	m
4.11 DLV	—	—	—	—	—	—	—
4.12 CHR	—	—	—	—	—	—	—

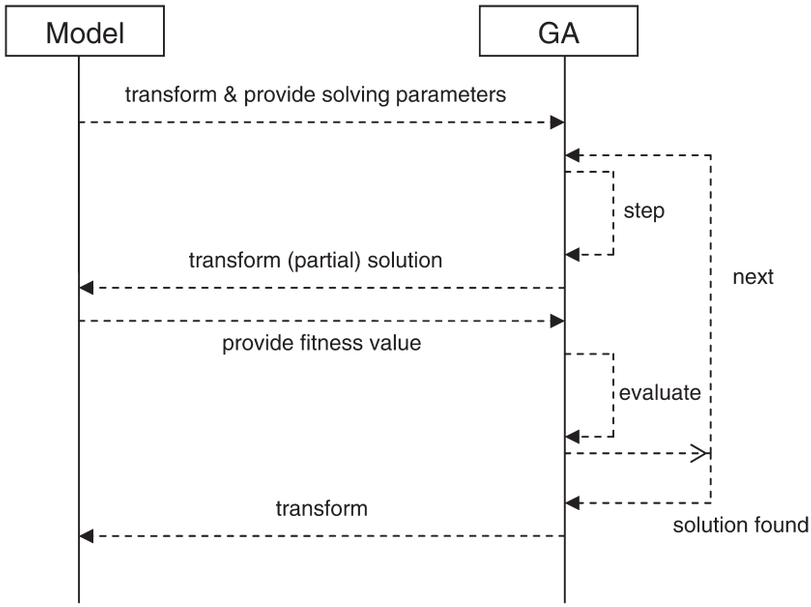

Fig. 12. The flow of control between the model and solver.

Then the GA algorithm generates an initial solution or, in the following steps, an offspring of an existing individual solution. To rate the quality of a solution, a fitness function value is required. We want to keep the solvers, in this case the GA, as independent of the domain as possible, avoiding duplicated representations of a large part of the model at the solver. Therefore, the fitness function is provided by the high-level model, because the object network and constraints are available there, with the possibility of complex, problem-specific computations.

When the GA solver has finished, eventually the best GA solution candidate is mapped back to the high-level model (Fig. 12).

The benefits of this architecture are a clean separation of modeling and reasoning and, hence, the possibility of using best-suited techniques and tools for those tasks. Complex systems may have different corners with different needs for reasoning. Although usually one high-level modeling system is used, which all other components are based upon, it is sometimes useful to work with more than one reasoning tool. This architecture is open for this.

The trade-off is the mapping functions from and to modeling and solver. Design and implementation of these mapping functions must be added to the modeling costs, the runtime overhead must be added to the solving time and should not be underestimated.

7. CONCLUSION

When we started writing this paper, we did not anticipate how hard solving the PartnerUnits problem would turn out to be. This is a typical scenario for a knowledge engineer when faced with building a configurator for a new product. Therefore, we advocate the use of formal tools (such as Alloy) at an early stage of knowledge engineering to analyze the complexity of the problem before choosing a suitable solving technology. Still, it is undeniable that most approaches to product configuration have a problem with large-scale configurations (i.e., containing a lot of instances).

Often the knowledge engineer must be able to find a special heuristic for the problem at hand or map the problem to algorithms from other fields (graph theory, OR, etc.). As stressed in Michalewicz and Fogel (2004), we cannot expect one general method to solve all the problems of all domains. Knowledge engineering especially for product configurators is an interdisciplinary approach.

What makes the PartnerUnits problem hard to solve is the restriction to N partnerunit connections (e.g., $N = 2$). Aside from the intellectual fun to tackle such a problem, it is worth asking the design engineers of the product whether this restriction is really necessary or whether there is another technical solution without this restriction. Experiences have shown that many hard configuration problems could be avoided just by an early integration of configuration architects into the product design process to make the product easier to configure (see Falkner & Haselböck, 2009).

At present, there is no well-established modeling language for product configuration. All the transformations from the modeling language to the solving language (e.g., UML/OCL → CSP) had to be implemented especially for this problem. Automatic translation between the different formalisms would be a great benefit.

7.1. Tools

To make it easier for the reader to evaluate our results and experiment with the described problem, we have chosen only freely available tools in this paper.

Alloy: http://alloy.mit.edu/alloy4/

Choco: http://www.emn.fr/z-info/choco-solver/index.html

DLV: http://www.dlvsystem.com/

DLV-Complex: http://www.mat.unical.it/dlv-complex

Eclipse IDE: http://www.eclipse.org

JGAP: http://jgap.sourceforge.net/

KodKod: http://alloy.mit.edu/kodkod/

SAT4J: http://www.sat4j.org/

SWI-Prolog (incl. CHR): http://www.swi-prolog.org/

USE: http://www.db.informatik.uni-bremen.de/projects/USE/

REFERENCES

Albert, P., Henocque, L., & Kleiner, M. (2008). Ant colony optimization for configuration. *Proc. 20th IEEE Int. Conf. Tools With Artificial Intelligence*, pp. 247–254.

Baader, F., McGuinness, D.L., Nardi, D., & Patel-Schneider, P.F. (2003). *The Description Logic Handbook*. Cambridge: Cambridge University Press.

Cooper, M., Jeavons, P., & Salamon, A. (2008). Hybrid tractable CSP's which generalize tree structure. *Proc. ECAI*, pp. 530–534.

Dechter, R., & Meiri, I. (1989). Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *Proc. 11th IJCAI*, pp. 271–277.

Eiter, T., Gottlob, G., & Mannila, H. (1997). Disjunctive datalog. *ACM Transactions on Database Systems 22/3*, 315–363.

Falkner, A., Feinerer, I., Salzer, G., & Schenner, G. (in press). Computing product configurations via UML and integer linear programming. *International Journal on Mass Customization*.

Falkner, A., & Haselböck, A. (2009). A simple evaluation process for configurability. *Proc. IJCAI-09 Workshop on Configuration*, pp. 17–22.

Felfernig, A. (2007). Standardized configuration knowledge representations as technological foundation for mass customization. *IEEE Transactions on Engineering Management 54(1)*, 41–56.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., & Zanker, M. (2003). Configuration knowledge representations for Semantic Web applications. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 17(1)*, 31–50.

Felfernig, A., Friedrich, G., Jannach, D., & Zanker, M. (2002). Configuration knowledge representation using UML/OCL. *Proc. 5th Int. Conf. Unified Modeling Language*, pp. 49–62. Berlin: Springer–Verlag.

Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H., & Stumptner, M. (1998). Configuring large-scale systems with generative constraint satisfaction. *IEEE Intelligent Systems 13(4)*, 59–68.

Frühwirth, T. (2008). Welcome to constraint handling rules. In *Constraint Handling Rules—Current Research Topics* (Schrijvers, T., & Frühwirth, T., Eds.), L Vol. 5388. New York: Springer–Verlag.

Gent, I.P., Petrie, K.E., & Puget, J. (2006). Symmetry in constraint programming. In *Handbook of Constraint Programming* (Rossi, F., van Beek, D., & Walsh, T., Eds.), pp. 329–376. Amsterdam: Elsevier.

Goldberg, D.E. (1989). Genetic *Algorithms in Search, Optimization & Machine Learning*. Reading, MA: Addison–Wesley Professional.

Gottlob, G., Greco, G., & Mancini, T. (2007). Conditional constraint satisfaction: logical foundations and complexity. *Proc. IJCAI 2007*, pp. 88–93.

Jackson, D. (2002). Alloy: a lightweight object modeling notation. *ACM Transactions on Software Engineering Methodologies 112*, 256–290.

Khichane, M., Albert, P., & Solnon, C. (2008). Integration of ACO in a constraint programming language. *Proc. ANTS*, pp. 84–95.

Kirkpatrick, S., Gelatt, C.D., Jr., & Vecchi, M.P. (1983). Optimization by simulated annealing. *Science 220(4598)*, 671–680.

Mayer, W., Bettex, M., Stumptner, M., Falkner, A., & Faltings, B. (2009). On solving complex rack configuration problems using CSP methods. *Proc. IJCAI-09 Workshop on Configuration*, pp. 53–60.

McDermott, J. (1982). R1: a rule-based configurer of computer systems. *Artificial Intelligence 19*, 39–88.

Michalewicz, Z., & Fogel, D.B. (2004). *How to Solve It: Modern Heuristics*. Berlin: Springer.

Sabin, D., & Weigel, R. (1998). Product configuration frameworks—a survey. *IEEE Intelligent Systems 13(4)*, 42–49.

Schoofs, L., & Naudts, B. (2000). Solving CSPs with ant colonies. *Proc. ANTS, 2000*.

Torlak, E. (2009). *A constraint solver for software engineering: finding models and cores of large relational specifications.* PhD Thesis. MIT.

Andreas Falkner is Program Manager and Senior Research Scientist in the global technology field entitled Constraint-Based Configurators at Siemens' Corporate Research & Technologies division. He received MS and PhD degrees in computer science from the Vienna University of Technology. Since 1992 he has been working for Siemens AG Austria, where he develops product configurators for complex technical systems in various domains, for example, for railway interlocking systems. For that purpose, his team has created a domain-independent configuration framework based on generative constraint satisfaction and is continuously enhancing it for further real-world requirements.

Alois Haselböck is a member of the research and development staff at Siemens AG Austria and is a Senior Research Scientist in the global technology field entitled Constraint-Based Configurators at Siemens' Corporate Research & Technologies division. He received MS and PhD degrees in computer science from the Vienna University of Technology. His research interest comprises knowledge representation and solving techniques for constraint-satisfaction systems where he has contributed fundamental findings in the field of generative constraint satisfaction.

Gottfried Schenner is a Senior Research Scientist in the global technology field entitled Constraint-Based Configurators at Siemens' Corporate Research & Technologies division. He joined Siemens AG Austria as a software developer in 1997. Mr. Schenner received an MS in computer science (main subject AI) from the Vienna University of Technology. Since 1997 he has been working on projects developing product configurators, including a domain-independent constraint-based configurator framework. His research interests comprise constraint-based technology and software architecture.

Herwig Schreiner heads the global technology field entitled Constraint-Based Configurators at Siemens' Corporate Research & Technologies division. He holds a Senior Project Manager (zSPM) degree from the International Project Management Association and received his MS in computer science from the Vienna University of Technology. His research interests include semantic technologies and knowledge representation for model-based diagnosis and configurators.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 131–142.

doi:10.1017/S0890060410000582

Product configuration as decision support: The declarative paradigm in practice

ALBERT HAAG AND STEFFEN RIEMANN
SAP AG, Walldorf, Germany

(RECEIVED May 17, 2010; ACCEPTED October 29, 2010)

Abstract

Product configuration is a key technology, which enables businesses to deliver and deploy individualized products. In many cases, finding the optimal configuration solution for the user is a creative process that requires them to decide trade-offs between conflicting goals (multicriteria optimization problem). These problems are best supported by an interactive dialog that is managed by a dedicated software program (the configurator) that provides decision support. We illustrate this using a real example (configuration of a business software system). This productively used application makes the user aware of which choices are available in a given situation, provides assistance in resolving inconsistent choices and defaults, and generates explanations if desired. One of the key configurator components used to manage this is a truth maintenance system. We describe how this component is used and two novel extensions to it: methods for declarative handling of defaults (of varying strength) and the declarative handling of incompleteness. Finally, we summarize our experiences made during the implementation of this application and the pros and cons of declarative versus procedural approaches.

Keywords: Business Configuration; Constraints; Decision Support; Interactive Configuration; Truth Maintenance Systems

1. INTRODUCTION

1.1. SAP® Business ByDesign™ Scope Selection: Our case study

Configuring a large and powerful software system is typically a complex task that often requires highly trained experts and involves considerable effort. A major aspect of this task is to ensure a consistent configuration that reflects the given constraints on the system functions and avoids conflicting user choices.

Our case study, which is the Scope Selection application in the SAP Business ByDesign solution (a product of SAP AG, Walldorf, Germany), offers substantial support for configuring a ByDesign system. It is based on the Business Adaptation Catalog (BAC) that represents the available functions of the solution. Customers select the desired functions to define an individual solution scope and answer questions to determine the desired behavior of the selected functions. The result is a "business blueprint" that is the basis for loading a set of predefined configuration settings into the system.

Reprint requests to: Albert Haag, SAP AG, Dietmar-Hopp-Allee 16, Walldorf 69190, Germany. E-mail: albert.haag@sap.com

To simplify the process of scope definition, it starts with a number of typically required choices that are recommended by SAP based on existing scenarios dependent on high-level decisions such as countries of operation and type of business (e.g., service provider in the United States). These recommended defaults are loaded as so-called preselections. This helps to minimize the necessary user interaction and customers can focus on their individual requirements.

The system ensures that customer choices are consistent and complete and informs why a given choice is in conflict with other choices. After all choices have been made and their consistency and completeness have been verified by the system, the defined solution scope is deployed and the customizing settings of the solution are changed appropriately.

The BAC (see Fig. 1) is a hierarchical arrangement of scoping elements. The first level of the hierarchy is defined by "Business Areas" that are purely structural elements. On the second level, "Business Packages" define the set of business functions. On the third level, "Business Topics" are used to model the key functions of business packages.

The behavior of the business functions depends on which business options are selected. The selection of business packages and business topics corresponds to the question "what functions should be used," but business options are used to

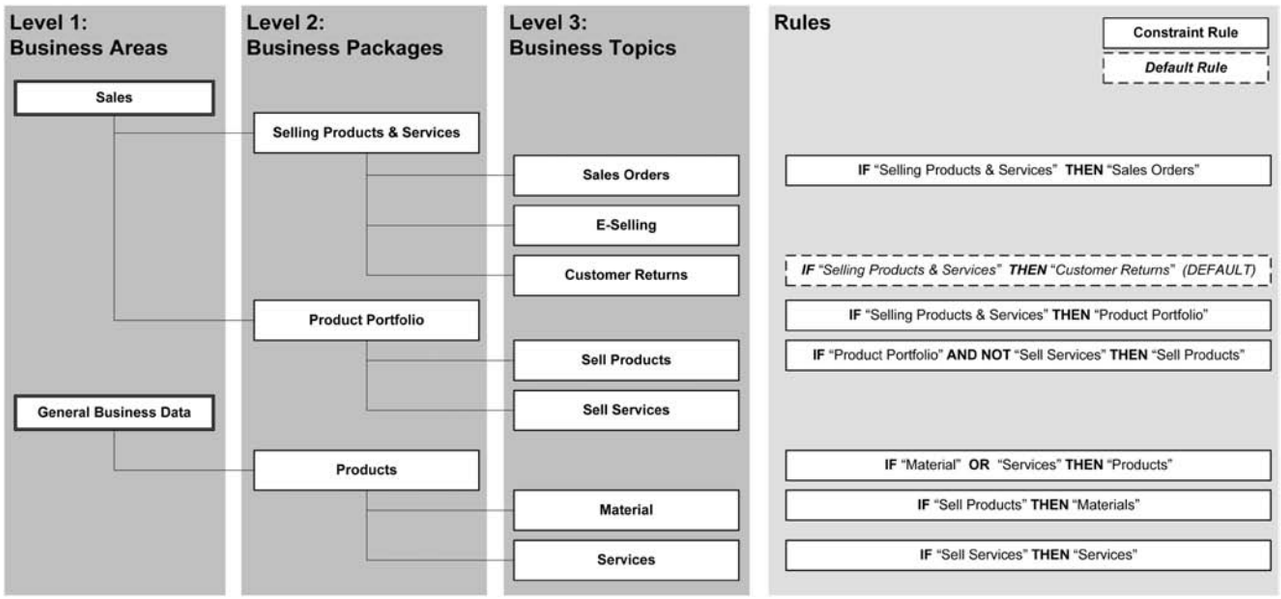

Fig. 1. The Business Adaptation Catalog (BAC) with business rules.

determine "how should the selected functions behave." For example, the customer can choose "Sales Orders" by selecting the appropriate business topic. Within Sales Orders, the customer can select the applicable business option to define whether or not an availability check is to be used.

Business packages, business topics, and business options can have one attached constraint rule that states the conditions when the item is automatically selected or deselected. These rules are termed self-centric. This format was originally chosen in the belief that it makes rule maintenance more manageable in that the relevant dependencies of an item are expressed in a single rule. In addition to the self-centric rule it is possible to add one default rule per element. This rule states a condition that will cause the element to be selected as a default. (The user can later choose to override defaults.)

Figure 2 shows an example of the Scope Selection user interface (UI). In this example, the user selected the business package "Selling Products & Services," which caused a selection of further items. In particular, this user choice led to a selection of the business package "Product and Service Portfolio" by constraint and a default selection of the business topic "Sell Products." The user choice and the default selection both deduce a selection of the business topic "Material." The explanation is based on the label of fact *BT_MAT* in Table 1. The meaning and calculation of labels is explained in Section 3.3.

2. SCOPE SELECTION AS A SPECIAL CASE OF PRODUCT CONFIGURATION

SAP provides product configuration built in to its standard enterprise resource planning and customer relationship management offerings. The current SAP product configuration engines are the SAP variant configurator and SAP Internet Pricing and Configuration (the IPC). These tools enable customers with configurable products to define product models and to manage these products in the context of their business, for example, sales, invoicing, and fulfillment. In particular, they provide interactive high-level configuration capability. A monograph on SAP Variant Configuration can be found in Blumöhr et al. (2010), which also references the IPC.

Scope Selection serves a different business process and is an SAP in-house application with a predetermined model (the BAC with its self-centric rules). Configuration results (the "Business Blueprints") are stored in a dedicated repository (called workspaces). Nevertheless, Scope Selection can be implemented using standard SAP product configuration by transforming both the BAC and the workspaces into the corresponding standard representations. This was proven in an earlier feasibility study using the IPC.

However, the required transformations are a source of additional complexity, and some unwanted and unneeded dependency on other software components is incurred. This creates additional cost and risk for the application. To avoid these drawbacks, an implementation of Scope Selection using a custom business rule approach (production rules) was first done without the IPC.

To facilitate reusing the IPC functionality the SAP Core Constraint Engine (CCE) is now provided, which encapsulates the constraint-based configuration logic of the IPC and is independent from other software components. When Scope Selection needed to adopt new standards for a following release, the CCE was available, met the new requirements, and it was decided to replace the custom rule engine with the CCE. The resulting current architecture (our case study) will be provided later in the article. This implementation is not a prototype, but it is used productively in Scope Selection.

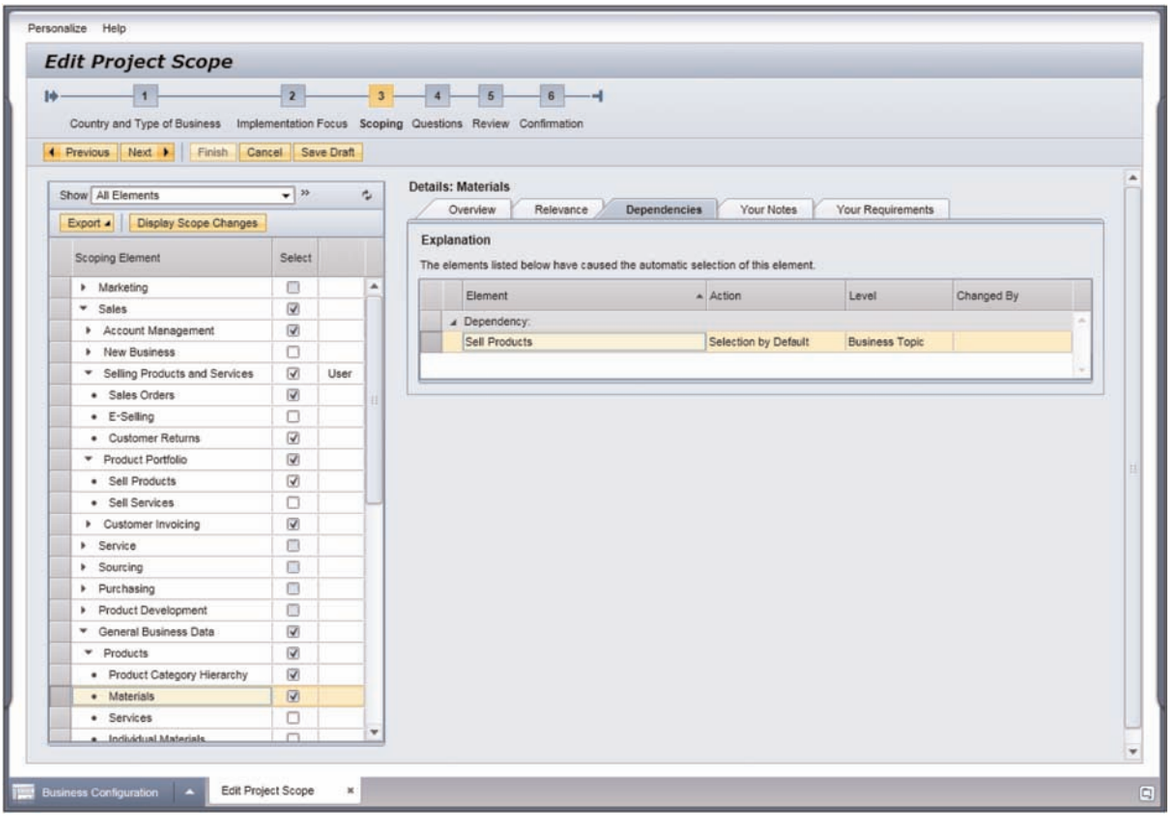

Fig. 2. The Scope Selection user experience. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

The switch to using the CCE was also a shift from a procedural paradigm to a declarative paradigm. Roughly speaking, we call an approach procedural if attention must be paid to the order in which the product model (e.g., the catalog items, the rules and/or constraints) and the user selections are processed. The production rules used in the initial implementation of Scope Selection are procedural because they perform side effects like overwriting defaults (and sometimes user selections) and act on unspecified information. Catalog items the user has not explicitly selected or deselected may be treated as deselected by default and acted on accordingly. A classical account of a production rule application with its advantages and drawbacks is given in McDermott (1982). We present experiences with this paradigm shift in Scope Selection in Section 4.4.

Table 1. *Example facts corresponding to the BAC excerpt in Figure 1*

Fact ID	BAC Item (Problem Variable)	Value	(A)TMS Status	(A)TMS Label
false			s_unfounded	
BP_PP	Product portfolio	Selected	s_in	BP_PP ∨ BP_SPS
BT_SP	Sell products	Selected	s_in	BT_SP
BT_SS	Sell services	Selected	s_in	
¬BT_SP	Sell products	Deselected	s_unfounded/ s_nil	
¬BT_SS	Sell services	Deselected	s_unfounded/ s_nil	
BP_SPS	Selling products & services	Selected	s_in	BP_SPS
BT_SO	Sales order	Selected	s_in	BP_SPS
BT_ES	E-selling	Selected	s_unfounded	
BT_CR	Customer returns	Selected	s_in	BT_CR
BP_PROD	Products	Selected	s_in	BT_SP
BT_MAT	Material	Selected	s_in	BT_SP
BT_SER	Services	Selected	s_unfounded	

Note: BAC, Business Adaptation Catalog; (A)TMS, SAP Core Constraint Engine component that facilitates decision support.

In contrast, we call an approach declarative if it is based on logical evaluations that do not depend on any specific order. Scope Selection can be seen as a classical constraint satisfaction problem (CSP). This is a declarative problem formulation that allows declarative processing. Whereas many product configuration problems are not formally CSPs, a constraint-based paradigm is the accepted approach to product configuration, mainly because it is declarative (Desisto, 2004). A treatment of constraint programming can be found in Rossi et al. (2006). Junker (2006) deals specifically with the topic of constraints and configuration. The SAP product configurators support both declarative and procedural elements. The feasibility study performed with the IPC was declarative and based on the SAP dependency type constraint (for SAP dependency types, see Blumöhr et al., 2010).

We illustrate the kind of behavior we expect in an interactive configuration with an example dialog based on Figure 1. The user performs the following six steps in order:

1. The user selects the BAC business topic "Customer Returns." The BAC rules will then select a slew of dependent items: business package "Selling Products & Services," which in turn will select business package "Product Portfolio," business topic "Sell Products" as a default, and subsequently business topic "Material" and its parent business package "Products."
2. The user takes back the above selection. All these items should disappear from the configuration.
3. The user directly selects "Selling Products & Services." All of the above items are again selected ("Customer Returns" this time as a default).
4. The user overrules (retracts) the default "Customer Returns." Only this item should disappear. All other items are still supported by the user choice for "Selling Products & Services."
5. The user selects "Customer Returns" again.
6. The user retracts their selection of "Selling Products & Services." Nothing disappears as everything is still supported by the choice of "Customer Returns."

As a general principle, the user could always expect the same result independent of the order in which the selections/retractions are voiced. The procedural rule-based approach was not able to consistently ensure this. This was not seen as a problem, because the UI did not allow the user to make selections in arbitrary order. Nevertheless, it was not possible to completely rule out situations in which items remained selected, although they were not really supported by user selections anymore.

3. THE (A)TMS: THE TRUTH MAINTENANCE SYSTEM (TMS) IN SAP CCE

The CCE component that facilitates decision support is its TMS that we call the (A)TMS. This is actually the only CCE component used in Scope Selection. The role of the (A)TMS is fivefold:

1. Record all dependencies between properties of the configuration. This is the basis for all of the other functions.
2. Distinguish the influence of logical constraints, user selections, and defaults. Defaults are used to suggest a property the user might want.
3. Determine whether a configuration state is consistent (and complete). In particular, identify properties that cannot be consistently added to the current configuration (and might be grayed-out in the UI).
4. Support consistent retraction and assertion of properties by the user as illustrated in the example in Section 2. This includes identifying user input and defaults the user may want to forego.
5. Provide explanations of a property. This includes identifying which choices must be retracted in order to resolve an inconsistency or otherwise remove an unwanted property from the configuration. It also includes identifying the actual business rules (constraints) used to establish the presence or exclusion of a property (or inconsistency) in the configuration. The role of a TMS in explanation is also dealt with in (Haag et al., 2007).

We describe the (A)TMS in more detail below. However, we limit ourselves to functionality used in Scope Selection. In particular, the concept of specialization relations (Haag, 1991, 1998) used in the TMS of the variant configurator and the IPC to handle dynamic domain restrictions is not dealt with here.

At heart the (A)TMS is a monotonic justification-based TMS (JTMS; Doyle, 1979); there are no "out-lists." The following extensions used in Scope Selection are new and are not yet available in the IPC:

- the concept of "choices" (see Section 3.2),
- a declarative mechanism for automatically dropping inconsistent defaults (see Section 3.5), and
- a mechanism for recording logical completeness directly in the (A)TMS (see Section 3.6).

3.1. Facts and justifications

The (A)TMS treats each property in the current configuration state as an atomic proposition that we call a "fact." Table 1 lists some facts that occur in Scope Selection. For example, the property that "Selling Products & Services" is selected is represented by the fact *BP_SPS*. A fact is only a valid part of the configuration if it is justified. To this end the (A)TMS maintains a set of "justifications." Each justification encodes a logical implication: *?justificand* → *?fact* where *?justificand* is a conjunction of other facts that allows inferring *?fact*. If *?justificand* is the empty conjunction then *?fact* will hold unconditionally and the justification is termed an "independent justification." Figure 3 depicts some justifi-

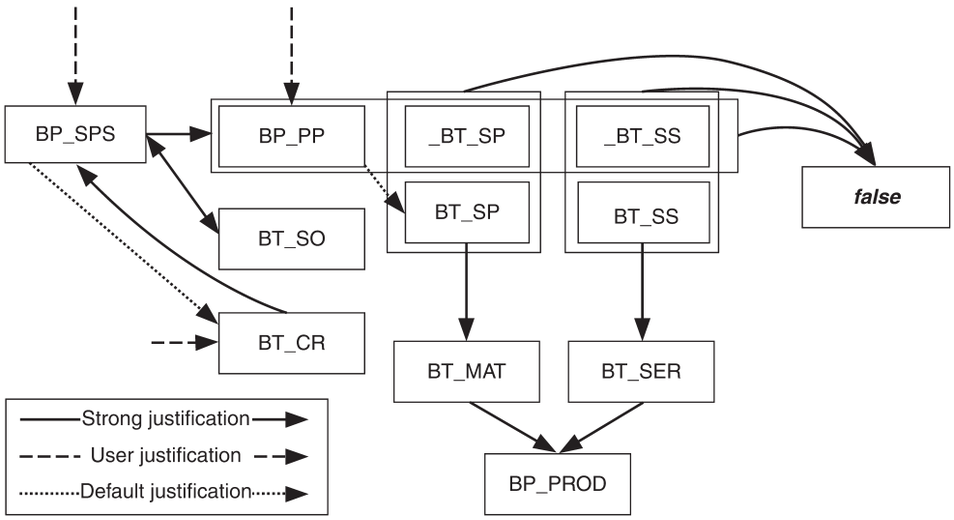

Fig. 3. Example justifications corresponding to the Business Adaptation Catalog (BAC) rules in Figure 1.

cations used in Scope Selection. For example, it contains an independent justification for *BT_SPS*.

Justifications are annotated with additional information about their source (the so-called "owner" of the justification). For example, the justification $BT\_SPS \rightarrow BP\_PP$ is linked to the BAC rule that spawned it. This is a feature used in explanations but we do not elaborate on it further here. More important, justifications are categorized by an attribute, "strength." The strength of a justification s is a positive number that is used to categorize the justification based on the two defined constants *c_strong* and *c_user* (where $c\_strong > c\_user$). There are three categories as follows:

1. Strong justifications ($s \geq c\_strong$): These are due to constraints.
2. User justifications ($c\_strong > s \geq c\_user$): These are stated on behalf the user.
3. Default justifications ($c\_user > s > 0$): These encode defaults.

There is a unique special fact *false* that encodes inconsistency. A constraint violation (inconsistency or "conflict") is recorded as a strong justification for *false*. $BT\_SP \wedge \neg BT\_SP \rightarrow false$ is an example of a conflict depicted in Figure 3.

In Scope Selection all facts and strong justifications are entered into the (A)TMS at the time of initialization of the configuration. In order to keep things simple we limit our exposition to this scenario in the sequel, although facts and justifications can be introduced to and removed from the (A)TMS at any time.

The (A)TMS determines which facts are parts of the current configuration state. These facts are assigned a logical status *s_in*. Facts with independent justifications have status *s_in*. In addition, a fact has status *s_in* if it has at least one justification with overall status *s_in*, that is, where all facts in the justificand have status *s_in*. All facts that are not assigned status *s_in* are considered to be "unfounded" (have status *s_unfounded*). We revise this slightly in the next section in connection with the concept of "dropping a choice."

The configuration state is inconsistent if *false* has status *s_in*. Table 1 also gives the logical status of the facts that results from the given set of justifications depicted in Figure 3.

3.2. Choices

The user can directly express that they support a particular property (fact) by means of a user justification. They can later change their mind about previous decisions or choose to forego defaults and user selections in the process of juggling alternatives in search for an acceptable consistent overall configuration solution. The (A)TMS must both support the user in juggling alternatives and provide a means for a consistent "undo" functionality, that is, allow the user to revert back to previous steps without any visible side effects.

We call facts the user can directly influence (including the defaults) "choices." More formally, we define the term "soft justification" to cover both user justifications and default justifications. A fact justified by at least one soft justification is termed a "choice." A choice with an independent soft justification is an "independent choice."

Note that a fact with an independent strong justification is a "premise." A premise cannot be a choice because the user has no influence on its validity. In addition, a choice without a soft justification whose justificand has overall status *s_in* is considered to be inactive as a choice.

The (A)TMS supports the following operations:

- The user or an agent on their behalf can communicate a new user justification. This may turn a fact into a choice, or it may add a soft (user) justification to an already existing choice. Such user justifications are usually (but must not be) independent justifications.

- If a user justification from the (A)TMS has been asserted "in error" it may be retracted. This would happen in the context of an "undo" operation. It may remove a choice if the fact has no other soft justifications.
- Normally, if the user is willing to forego a choice they will "drop" it as a whole. Formally, this results in disabling all soft justifications of the choice. (In particular, the user can drop a choice that is only due to default justifications, thus overruling the default.) The disabled justifications are not physically removed from the (A)TMS but they are ignored when determining the facts to be assigned status *s_in*. A dependent choice can only be dropped if it is active as a choice.
- The user can "pick" a dropped choice. This is the inverse function to dropping a choice. It means reenabling all soft justifications. A dependent choice can only be picked if it is active as a choice. [The application may decide to ensure that a choice picked by the user has/gets a user justification, but this is a decision outside the (A)TMS.]

In conjunction with choices we introduce the additional logical status *s_out*. A fact has status *s_out* if it does not have status *s_in*, but could be switched to status *s_in* merely by picking choices (i.e., without adding new justifications). A fact that has neither status *s_in* nor *s_out* has status *s_unfounded*.

When deciding which choice to automatically drop in case of a conflict it is important to take the strength of its soft justifications into account. We define the strength of a choice to be the highest strength of its *founded* soft justifications (enabled or not). (A founded justification is one whose *justificand* contains no unfounded facts). Note that dropping or picking a choice will not affect the strength of any choices.

3.3. Assumption-based TMS (ATMS)-like label calculation

In order to automatically decide which choices to drop in case of a conflict, to provide guidance to the user in resolving conflicts and removing unwanted facts, and to facilitate exploring alternatives in general, the (A)TMS can calculate the minimal logical support in terms of choices for selected facts. This calculation is performed on demand for selected facts only and then incrementally maintained for these facts. The idea of calculating minimal logical supports on demand is not new: an approach is presented by Petrie (1992, 1993) in which the minimal set of "decisions" that support a constraint violation is produced upon request. Retraction of such decisions, similar at least in intent to our "choices," removes the violations. This is done with a model above the TMS, which in this implementation is a JTMS. However, the dependency-directed backtracking of the underlying JTMS in Petrie (1992) versus the (A)TMS labeling in this paper is a fundamental utility for determining this minimal set.

Our calculation of minimal supports derives from the ATMS labeling defined by De Kleer (1986*a*) for the ATMS, and we refer to the minimal support of a fact as its "label." Because this is based on the concept of "choices" rather than "assumptions" we might call our component a choice-based TMS, but we chose to stick with established terms. We place the "A" in parentheses to indicate that there are some deviations from the original definitions of De Kleer (1986*a*).

We first introduce the terminology. An "environment" is a logical conjunction of choices (implemented as a set). An environment is "active" if all of its choices are active and picked. Otherwise it is "inactive." A "label" is a disjunction of environments (implemented as a set). A fact that has status *s_in* will have at least one active environment in its label (when calculated). An environment in the label of a fact means that the fact can be derived from the set of choices in the environment but not from any proper subset of these choices. Table 1 also shows the labels for selected facts. An environment that allows deriving *false* is called a "nogood."

Labels can be characterized as follows [Fig. 4 summarizes the two basic operations on labels: addition ($\oplus$) and multiplication ($\otimes$)]:

- Labels are kept minimal: logically redundant environments (supersets) are removed. A nogood can be removed from any label other than that of *false*. Hence, the nogoods are always computed as a prerequisite to computing other

f is a fact; σ and τ are conjunctions of facts;
ξ, ψ, and ζ are environments;
$\Lambda(\cdot)$ is the labeling

Label multiplication:
$\Lambda(f \wedge \tau) := \Lambda(f) \otimes \Lambda(\tau) := \min\{\zeta = \xi \wedge \psi \mid \xi \in \Lambda(f) \wedge \psi \in \Lambda(\tau)\}$

Label addition:
$\Lambda(\sigma \wedge \tau) := \Lambda(\sigma) \oplus \Lambda(\tau) := \min\{\xi \mid \xi \in \Lambda(\sigma) \vee \xi \in \Lambda(\tau)\}$

Fig. 4. The addition and multiplication of labels.

labels if and only if the configuration state is inconsistent as determined by the JTMS functionality.
- The singleton environment of a choice *c* is in the label of *c* regardless of whether *c* is an independent choice or not (this differs from de Kleer, 1986*a*).
- The label of a conjunction of facts is the product ($\otimes$) of the labels of all of the facts in the conjunction. The label of the empty conjunction (left-hand side of an independent justification) is the *empty environment*, designating unconditional truth.
- The labels of the justificands of all strong justifications for a fact are added ($\oplus$) to its label.
- If a choice is conditional (i.e., it is not independent), a conditional label is maintained separately. This is the sum ($\oplus$) of the labels of the justificands of all its *soft* justifications. The conditional label plays a role in determining the activation status of a choice and can be used for more detailed explanations. In Table 1 the choices *BT_SP* and *BT_CR* are conditional choices with conditional labels *BP_PP* $\vee$ *BP_SPS* and *BP_SPS* (not shown in Table 1), respectively.

We give an example of why it is opportune to have the singleton environment of a dependent choice in its label (as opposed to the $\otimes$ product of its conditional label with the singleton environment). In Figure 3 we see that *BT_MAT* is selected when *BT_SP* has status *s_in* because of a strong justification. In the setting of Table 1 *BT_SP* is a dependent choice (a default that only holds if *BP_PP* holds as well). Now the label of *BT_MAT* gives its minimal support. The "immediate" support is the default justification. Of course, *BT_MAT* will also be switched to *s_out* if *BT_PP* is removed from the configuration by dropping both (user) choices in its label, but there is no need to consider this if the user simply wants to remove *BT_MAT*. Hence, the total information can be reconstructed using the conditional label if needed, but only the immediate support is expressed in the label (and propagated).

Because rapid dropping and repicking is to be supported, it is not opportune to remove inactive environments from labels. If a choice is dropped, all environments that contain the choice are simply deactivated. If the choice is repicked, the corresponding environments must be checked to see if they should be reactivated. In addition, any environments in a label that had not been propagated in the label calculation must be propagated on reactivation.

If a label cannot be calculated within allocated time or space resources then the union of all environments in the label is calculated as a fallback. This is a much easier calculation, but still useful. Assume we have an inconsistency and the union of all nogoods. Then it is clear that at least one of the choices in this union must be dropped. After a decision to drop one is made, the nogoods (or their union) must be recalculated and the process continues until the inconsistency is resolved. Thus far, this fallback calculation has not been invoked in conjunction with Scope Selection.

3.4. Retracting justifications

If strong justifications are retracted, many internal computations must be redone, particularly if the set of nogoods is potentially affected. Retracting a soft justification is easier. The effect retracting a soft justification can have is similar to that of dropping a choice. It differs in that it can affect the strength of a choice or remove a choice entirely. If only the strength is affected this may cause a revision of automatically dropped choices (see Section 3.5). If the choice is removed this will cause removal of environments as well.

Although a configuration session ideally consists of just adding justifications and dropping and picking choices, retraction of soft justifications is also needed to be able to provide an "undo" function without any side effects.

3.5. Declarative handling of defaults

To deactivate a nogood it suffices to drop one choice in the nogood. This can be done automatically in a declarative fashion if the decision to do so is unambiguous and independent of any processing order. This is done in the (A)TMS based on the strength of choices.

Given a set of founded justifications *J* and a set of (manually) dropped choices there is a unique set of choices that is dropped automatically by the (A)TMS. This set can be constructed iteratively as follows: let *D* be the overall set of dropped choices (initially only the manually dropped choices). Let *N* be the set of active nogoods that have a unique minimal (strength) choice given *J* and *D*. Order *N* by descending strength of minimal choices. Drop the minimal choice in the first nogood in *N*. Remove any now inactive nogoods from *N* (at least the first element). *N* is still ordered by descending strength of the least preferred choices. Repeat the above process until *N* is empty. Note, that when dropping a minimal choice for some nogood in *N* this choice cannot be stronger than any minimal choice of a previous element and will thus not make it necessary to revise dropping previously dropped choices.

Although the set of automatically dropped choices is defined as above, the (A)TMS provides an incremental mechanism to determine which choices must be automatically dropped or repicked when changes to either justifications *J* or the manually dropped choices occur.

As a matter of principle, choices because of user input should never be dropped automatically. This is the origin of the basic distinction between user and default justifications. Hence, in practice, we restrict *N* to nogoods with minimal default choices. If user choices are also dropped automatically, then more elaborate explanations or messages are required.

3.6. Declarative handling of incompleteness in the ATMS

The configuration must be not only consistent but also complete. In Scope Selection there are requirements that at least one of a set of BAC items is selected. The added difficulty in

an interactive setting is that the case that some items in such a set have not yet been specified (incompleteness) must be distinguished from the case that they are all intentionally deselected (constraint violation).

The BAC items correspond to CSP variables with possible values *selected* or *deselected*. However, the (A)TMS as a propositional system has no notion of a problem variable. Consequently, in the IPC completeness is handled procedurally outside the CCE (mainly by checking if a required variable assignment is indeed present). There is, however, a special representation used there to represent an intentionally unspecified variable and to allow constraints to act on this. This is a fact stating that a variable is "nil" (unassigned). We call such a fact a "nil facet fact" and represent it as $nil(v)$, where v denotes a problem variable [known outside the (A)TMS]. Thus, the justification $nil(v_1) \wedge \cdots \wedge nil(v_n) \rightarrow^{strong} false$ can be used to express that leaving all variables $v_1, \ldots, v_n$ (intentionally) unspecified is inconsistent (a constraint violation).

If none of these nil facet facts can be disproved and some of them are unfounded this is an incompleteness that cannot be detected by the (A)TMS without some further mechanism. The seemingly easy way to handle this is to initialize the $nil(v)$ fact of each variable v with a default justification (of suitably weak strength). In Scope Selection there was a requirement to have the fact $v = deselected$ represent $nil(v)$ as well for a problem variable v variable representing a BAC item.

However, initially deselecting all BAC items as a default is not a good idea as it clutters up the labels (and explanations) in the (A)TMS and causes much irrelevant calculation. For this reason we provide a separate mechanism as follows: The (A)TMS can be told that a particular fact represents $nil(v)$ for a problem variable v. If any other fact pertaining to the variable v implies $\neg nil(v)$ this must be communicated to the (A)TMS as an opposing fact to $nil(v)$. This will be any value assignment to v [except one that happens to coincides with the fact representing $nil(v)$]. In the case of a variable v representing a BAC item in Scope Selection $v = selected$ is the only opposing fact. In case the variable v represents the observable *color*, $nil(v)$ would not relate to any value assignment. All value assignments for *color* would be opposing facts. In case the variable v is numeric the fact $v = 0$ might be considered as representing $nil(v)$ or not (as the case may be for the particular product model).

The basic idea is that the (A)TMS treats an unfounded fact $nil(v)$ in a special way. The status of such a fact is set to *s_out* if any opposing fact has status *s_in*. Otherwise, it is set to status *s_nil* (instead of *s_unfounded*). The status *s_nil* is thus a flavor of the status *s_unfounded* used for nil facet facts. A conjunction of facts has an overall status *s_nil* if it contains at least one fact with status *s_nil* and no fact with status s_*out* or *s_unfounded* (as distinct from status *s_nil*).

The (A)TMS uses the status *s_nil* solely to detect incompleteness. This means that a justification for *false* will signal incompleteness if its justificand has status *s_nil*. If the justificand has status *s_in* it will signal inconsistency. For all other purposes status *s_nil* is treated as *s_unfounded*. Any justification for *false* that contains nil facet facts is thus "a potential source of incompleteness." Figure 3 contains one example,

$$BP\_PP \wedge \neg BT\_SP \wedge \neg BT\_SS \rightarrow^{strong} false,$$

where the last two facts in the justificand are nil facet facts.

Often the purpose of a default is to prevent incompleteness from occurring in the first place. A justification for such a default may be entered into the (A)TMS by normal means, but it will be present regardless of whether incompleteness is signaled or not. For this reason the (A)TMS provides an interface function *tms_choose* that allows defining a potential source of incompleteness and associating it with a default to handle it in such a way that the default is only activated in case this source signals incompleteness. *tms_choose* has the following four arguments and causes justifications (1) and (2) to be entered into the (A)TMS with some special handling for (2):

1. *?condition:* A set (conjunction) of nonnil facet facts that must hold as a precondition (*BP_PP* in the above example from Fig. 3)
2. *?nil_facet_facts:* A set (conjunction) of nil facet facts defining the incompleteness ($\neg BT\_SP$ and $\neg BT\_SS$ in the above example from Fig. 3)
3. *?default:* An opposing fact to one of the given nil facet facts that is to be set as default (as discussed in Section 4.3 this will be the fact *BT_SP*)
4. *?default_strength:* A (fitting) strength for the ensuing default justification (the normal default strength is used here in Scope Selection)

$$?condition \wedge ?nil\_facet\_facts \rightarrow^{strong} false \quad (1)$$

$$?condition \rightarrow^{?default\_strength} ?default \quad (2)$$

Justification (2) is automatically disabled in circumstances when it would generate an additional unneeded conflict, for example, when (1) signals a conflict or when the nil facet fact that *?default* opposes attains logical status *s_in*. It is reenabled when these circumstances no longer apply.

We consider this approach to be simpler and more practical to compute than logically more complete approaches such as the extended ATMS (de Kleer, 1986*b*).

4. THE DECLARATIVE APPROACH IN SCOPE SELECTION

4.1. Architecture

An overview of the architecture of the Scope Selection application (our case study) is given in Figure 5. It makes use of the declarative configuration logic of the (A)TMS component of the CCE. The other components of the CCE, in particular its variable management, are not used. The application implements its own variable management in the scoping adaptor. The scoping adaptor also manages the access to the scoping

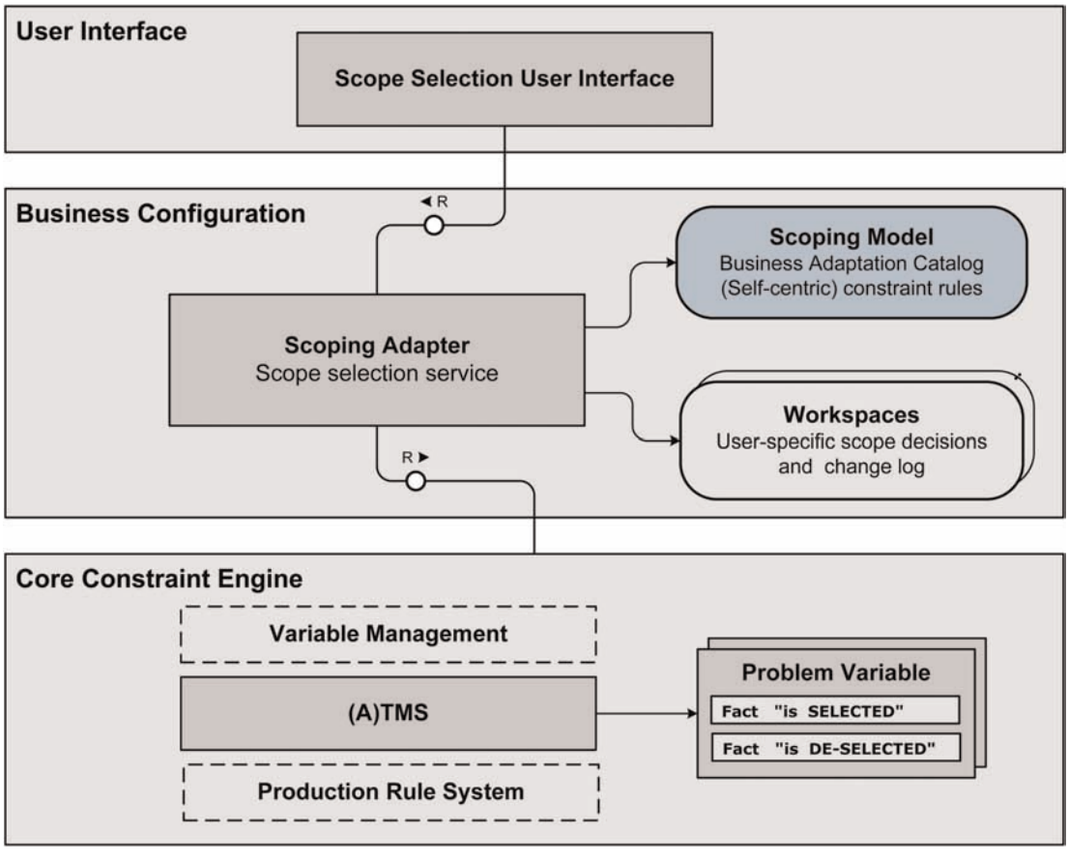

Fig. 5. The architecture of scope selection application (case study). [A color version of this figure can be viewed online at journals.cambridge.org/aie]

model (the BAC) and the workspaces. One workspace contains all the configuration decisions of one customer.

The services of the scoping adaptor are designed to meet specific requirements of the application. For example, the Scoping Adapter performs the following:

- Manages a problem variable for each item of the BAC.
- Translates the business rules in the BAC into (A)TMS justifications. We look at this in more detail in Section 4.3.
- Splits the Scope Selection process into several substeps: high-level scope definition, detailed Scope Selection, and the definition of business options. The (A)TMS is initialized with the relevant variables and business rules for each individual step.
- Includes a change history to enable the user to track changes made to the configuration.

Use was made of the new features of the (A)TMS: the declarative handling of defaults and incompleteness. Defaults occur with two different strengths: normal defaults formulated in the business rules and stronger "preselection" defaults formulated in conjunction with predefined scenarios the user can select.

4.2. Structure of scoping elements

For each problem variable v (item in the BAC) we create two facts in the (A)TMS:

- One fact represents "$v = selected$."
- The second fact represents "$v = deselected$."

If exactly one of the two facts is active (i.e., have status *s_in*), the assignment (selection status) of the variable is well defined and consistent. However, if both facts are active this is regarded as a conflict (inconsistency). To this end a strong justification is entered into the (A)TMS for each variable v as follows:

$$v = selected \wedge v = deselected \rightarrow^{strong} false$$

Furthermore, if neither fact is active, then this variable is "unspecified." In the Scope Selection application, any variable that remains unspecified at the end of the Scope Selection process is considered as "out of scope" and the corresponding item is then treated as deselected.

All facts of the form $v = deselected$ are declared as nil facet facts in the (A)TMS; $v = selected$ is the opposing fact to $v = deselected$ (see Section 3.6).

4.3. Transition to the declarative approach

The existing product model was reused without any adaptation of the preexisting rules. This was actually a precondition for the successful transition from the previous implementation and increased the acceptance of the new Scope Selection application by the responsible managers. The existing rules, however, had originally been defined in the context of a procedural approach. With the transition to the declarative approach, these rules are now interpreted as logical inferences (constraints).

The rules of the product model are defined in form of "self-centric" IF–THEN–ELSE rules of the following form:

IF «condition» **THEN** «status 1» [**ELSE** «status 2»] (3)

where «condition» refers to an expression comprising one or more variable assignments connected with logical operators **AND** and **OR** and «status 1» defines a selection status of the BAC item that the rule is attached to. The rule specifies that this will be set if «condition» is fulfilled. Here, «status 2» is the opposite of «status 1». It will be set if «condition» is not fulfilled and **ELSE** is specified.

In the following we describe how these **IF–THEN–ELSE** rules are translated into logical inferences.

First, «condition» is transformed into disjunctive normal form (DNF), and references to the variable assignments are replaced by corresponding references to facts. An element of this DNF will be a conjunction of facts (e.g., $f_1 \wedge \cdots \wedge f_n$). The system derives an implication (4) for each such element of the DNF:

$$f_1 \wedge \cdots \wedge f_n \Rightarrow f_0 \tag{4}$$

where f_0 is the fact that assigns «status 1» to the variable that the rule is assigned to. In the event of an optional **ELSE** statement, «status 2» is handled similarly based on the DNF of **NOT** «condition». We do not elaborate on this explicitly in the sequel.

Each implication (4) can be treated as a logical clause (5) (recall that $p \Rightarrow q$ is equivalent to $\neg p \vee q$).

$$\neg f_1 \vee \cdots \vee \neg f_n \vee f_0 \tag{5}$$

where $\neg f_i$ represents the opposing assignment to f_i, that is, that $\neg(v = \mathit{deselected})$ is $v = \mathit{selected}$ and vice versa.

For logical completeness it would seem advisable to generate a complete set of $n + 1$ justifications, one for each f_i (e.g., $\neg f_0 \wedge f_2 \cdots \wedge f_n \rightarrow^{strong} \neg f_1$ for f_1). However, logical completeness is not necessarily considered a boon by all. In Scope Selection the business experts that formulated an implication $p \Rightarrow q$ through a rule did not believe that $\neg q \Rightarrow \neg p$ made sense in an explanation (e.g., because it violates perceived causality) and considered the latter inference to be "unwanted." (Managers required that even if $\neg q$ is chosen, p is not to be excluded in the UI.)

Therefore, the system creates only two justifications based on clause (5). One justification directly implements the implication (4) [which is closely linked to the business intent of the rule (3)]:

$$f_1 \wedge \cdots \wedge f_n \rightarrow^{strong} f_0$$

A second justification is needed to check the consistency and completeness of the configuration in the absence of a complete set of justifications:

$$f_1 \wedge \cdots \wedge f_n \wedge \neg f_0 \rightarrow^{strong} \mathit{false}$$

An inconsistency is detected if all involved facts in the justificand are founded. In the case that at least one such fact is a nil facet fact (representing $\neg v$ for some problem variable v) which is in *s_nil* status (logically unfounded), incompleteness is detected (refer to Section 3.6).

Some rules that are formulated to act on the deselection of an item require special attention. The simplest form of such a rule is

IF $A = \mathit{deselected}$ **THEN** $B = \mathit{selected}$

where A and B are items in the BAC. Let $\neg a$ be the fact $A = \mathit{deselected}$ and b be the fact $B = \mathit{selected}$. If the rule were simply translated into the justifications

$$\neg a \rightarrow^{strong} b \text{ and } \neg a \wedge \neg b \rightarrow^{strong} \mathit{false}$$

this will not infer b if $\neg a$ is unfounded (or *nil*). As discussed in Section 3.6, stating an initializing default justification for the deselection of each variable does not work well. A different translation is instead applied to these rules based on the actual business intent of the rule that states a (directed) fulfillment of a completeness requirement in a procedural form (b is needed in the absence of a). Here we use the *tms_choose* function to generate the following two justifications (see Section 3.6):

- $\neg a \wedge \neg b \rightarrow^{strong} \mathit{false}$ [as above, this corresponds to (1)]
- *empty justificand* (*true*) $\rightarrow^{default\_strength} b$ [this corresponds to (2) and is only active in case of an incompleteness]

The justification $\neg a \rightarrow^{strong} b$ is generated in addition against the event that $\neg a$ attains status *s_in* (is deselected on purpose).

We close with an example using two rules from the BAC. We abbreviate by using fact substitutions directly in the formulation of the rules (the example refers to facts and variables given in Table 1):

IF *BP_SPS* **THEN** *BP_PP* (6)

IF *BP_PP* **AND** ¬*BT_SS* **THEN** *BT_SP* (7)

Rule (6) states, the function "Sell Product & Services" requires "Product Portfolio." The second rule (7) will ensure

that "Product Portfolio" cannot be used without the functions "Sell Services" and "Sell Products." In fact, this rule is formulated in a procedural manner to select the item "Sell Products" in the case that "Product Portfolio" is selected but "Sell Services" is not selected. The business intent is to associate a completeness requirement with a default to handle it.

Rule (6) is transformed into the clause $\neg BP\_SPS \vee BP\_PP$ and leads to the following justifications:

$$BP\_SPS \rightarrow^{strong} BP\_PP$$

$$BP\_SPS \wedge \neg BP\_PP \rightarrow^{strong} false$$

Rule (7) includes the pattern to infer the selection of *BT_SP* based on the deselection of *BT_SS* given the fact that *BP_PP* is selected. It corresponds to the clause $\neg BP\_PP \vee BT\_SS \vee BT\_SP$. That this formulates a completeness requirement can be seen directly in that the clause contains a disjunction of selections (i.e., is not a Horn clause). The original rule gives information that this incompleteness is to be handled by selecting *BT_SP* as a default if required. Consequently, *tms_choose* is used with the following arguments: $?condition = \{BP\_PP\}$, $?nil\_facet\_facts = \{\neg BT\_SS, \neg BT\_SP\}$, $?default = BT\_SP$, and $?default\_strength = default$.

Altogether the following justifications are generated:

$BP\_PP \wedge \neg BT\_SS \rightarrow^{strong} BT\_SP$ (direct transformation of rule)

$BP\_PP \wedge \neg BT\_SS \wedge \neg BT\_SP \rightarrow^{strong} false$ (generated *by tms_choose*)

$BP\_PP \rightarrow^{default\_strength} BT\_SP$ (generated *by tms_ choose*)

4.4. Benefits and practical compromises

The new Scope Selection application based on the (A)TMS in the CCE proved to be successful in fulfilling the functional requirements and also met the required SAP product standards for runtime performance. In one scoping step the system deals with up to 4000 problem variables that are connected via 14.000 logical inferences (justifications).

The existing product model could be reused without adaptation, but the rules are now interpreted in a declarative manner (as logical clauses), which provides some concrete benefits:

- There is no need for any rule execution logic on top of the (A)TMS. This simplified the implementation of the new Scope Selection application significantly.
- The development process of new constraint rules will become easier, because the developer does not have to worry about the rules defined by other developers and the order in which rules are applied.
- Unwanted behavior caused by the procedural execution of business rules was eliminated. (An example of such unwanted behavior is that sometimes default selections were triggered that were meant to be set in a different situation. In addition, sometimes properties set by rules were not completely retracted when the user retracted the choices that triggered these rules. Previously, retracting a user choice did not always consistently remove all dependent items selected by rules along with it. The user was alerted to the need to verify the scope after removing an item and perhaps manually removing further items as well even if the user never directly selected these.)
- More consistent explanations can be constructed based on the (A)TMS labels. Table 1 lists the labels for some facts used in our examples. In particular, Figure 2 shows the explanation of one fact as it is presented in the UI. Previously the production rules generated explanations by constructing chains of applied rules. There were, however, cases when this led to explanations more complex than needed that were confusing for the user.
- The ability to retract user choices consistently was the basis for multistep undo/redo functionality in Scope Selection.

These benefits proved to be instrumental in convincing the managers to gradually accept a more purely declarative processing. However, the new Scope Selection application has also kept some of the previous procedural control flow. This has been done as a practical compromise to preserve the look and feel of the existing UI as much as possible and to avoid a major change in the user interaction of the Scope Selection process. There are two main points here:

1. The UI provides strict guidance concerning the order in which some selections may be made. Mainly, the user has to follow the hierarchy of the BAC and first select a business package before they are able to select one of its business topics. To ensure this it is also unwanted that a business topic is inferred from a without inferring the parent business package by the same rule. For this reason the generic logical inference to deduce the selection of an item given the selection of a child item is not present in the current implementation (although it is logically correct).
2. Although the (A)TMS provides support for declarative resolution of conflicts (declarative default handling), the managers requested that the system in addition resolve conflicts because of some heuristics, even when this is not justified declaratively. These heuristics are implemented by procedural means in the adapter layer (not in the CCE). In some cases, the system even retracts user decisions. However, the user is informed if this happens and can use the undo feature to revert to a previous state.

5. CONCLUSION AND SUMMARY

In the new Scope Selection implementation the declarative approach had definite advantages over the previous procedural one. This is substantiated by the fact that over the course of the reimplementation the managers became more convinced of the approach and its potential for future benefits re-

garding easier maintenance of rules. Interpreting a rule as a set of logical clauses opens the future possibility of letting it affect the selection of other items besides the one it is attached to. Whereas this circumvents the self-centric principle, it actually allows a more modular approach to modeling, because the self-centric rule of an item does not need to be modified anytime an additional criterion for the selection of an item is formulated.

Some open issues remain. One main reason managers adhere to a procedural specification of parts of the application is because they believe they can achieve a simpler handling for the user, who may not understand the logic behind the declarative approach. Whereas it is our feeling that, on the contrary, people do better with an approach that exhibits consistent logical behavior than with the unavoidable occasional idiosyncrasies inherent to a procedural approach, this perception remains to be substantiated. At the same time, further work must be done to simplify the user experience as much as possible without abandoning the declarative paradigm. Creating explanations that are not only correct but focus on the actual business reasoning behind the rules and hide unwanted complexity is probably instrumental here. The current state of explanations, although being a definite step up from the previous state, is not yet ideal in this respect. This is one area for future improvements.

REFERENCES

Blumöhr, U., Münch, M., & Ukalovic, M. (2009). *Variant Configuration with SAP®*. Dedham, MA: SAP Press.

De Kleer, J. (1986*a*). An assumption–based truth maintenance system. *Artificial Intelligence 28(1)*, 127–162.

De Kleer, J. (1986*b*). Extending the ATMS. *Artificial Intelligence 28(1)*, 163–196.

Desisto, R.P. (2004). *Constraints Still Key for Product Configurator Deployments*. Gartner Report T-22-9419, June 1, 2004. Stamford, CT: Gartner.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence 12(3)*, 231–272.

Haag, A. (1991). Konzepte zur praktischen Handhabbarkeit einer ATMS basierten Problemlösung. In *Das Plakon Buch* (Cunis, R., Günter, A., & Strecker, H., Eds.), pp. 212–237. Heidelberg: Springer–Verlag.

Haag, A. (1998). Sales configuration in business processes. *IEEE Intelligent Systems 13(4)*, 78–85.

Haag, A., Junker, U., & O'Sullivan, B. (2007). Explanation in product configuration. *IEEE Intelligent Systems 22(1)*, 78–90.

Junker, U. (2006). Configuration. In *Handbook of Constraint Programming* (Rossi, F., van Beek, P., & Walsh, T., Eds.), pp. 837–874. Amsterdam: Elsevier.

McDermott, J. (1982). R1: a rule-based configurer of computer systems. *Artificial Intelligence Journal 19*, 39–88.

Petrie, C.J. (1992). Constrained decision revision. *Proc. AAAI 1992*, pp. 393–400.

Petrie, C.J. (1993). The redux server. *Proc. CoopIS 1993*, pp. 134–143.

Rossi, F., van Beek, P., & Walsh, T. (Eds.). (2006). *Handbook of Constraint Programming*. Amsterdam: Elsevier.

Albert Haag is a Development Architect at SAP AG. He has been involved in designing and implementing the SAP product configurators since 1992 in various roles including project lead and group manager. He received a PhD in computer science from the University of Kaiserslautern and a Dipl. in mathematics from the University of Hamburg. Dr. Haag's research interests include product configuration applications and truth maintenance systems.

Steffen Riemann is a Development Architect at SAP AG. He designs and develops tools to aid customers in configuring their SAP solutions. He graduated from the University of Mannheim with a degree in business and computer science. Mr. Riemann was recently involved in the development of the Scope Selection application, a customizing tool that is based on a new implementation methodology and utilizes a product configuration engine.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 143–162.

doi:10.1017/S0890060410000594

A declarative framework for work process configuration

WOLFGANG MAYER, MARKUS STUMPTNER, PETER KILLISPERGER, AND GEORG GROSSMANN
University of South Australia, Adelaide, Australia

(RECEIVED May 24, 2010; ACCEPTED October 29, 2010)

Abstract

This article describes the technical principles and representation of a constraint-based configuration method for work processes. Methods developed for the configuration of modular systems comprising components have traditionally adopted a representation where the properties and compatibility requirements are expressed as constraints associated with individual components. However, this representation does not accurately capture constraints on paths and subprocesses and is therefore unsuitable for process configuration. This article extends established constraint-based configuration methods with a constraint language for specifying properties of execution paths in work processes. A framework for semiautomated process customization is presented. It integrates the extended constraint language with a metamodel of the work processes in an organization and allows to adapt generic work processes to fit the requirements of specific development projects. Heuristic search methods are applied to build valid process configurations by incrementally resolving constraint violations. The declarative framework facilitates the adaptation of abstract work processes as well as the validation and repair of existing processes. The approach was developed in the context of a real-world system of complex design and development processes where it was shown that significant process improvements and reduction in effort required to edit process models can be achieved.

Keywords: Configuration; Constraints; Process Design; Workflow Modeling

1. INTRODUCTION

Work processes in large organizations often constitute generic frameworks that encompass all possible development activities rather than specific development processes that are well suited to the needs of a project. Organization-wide guidelines are often specified as comprehensive work processes describing the sequence of activities to be followed in development. In order to provide commonality, accountability, and to communicate acknowledged good practice across the enterprise, such processes are often established as reference processes that cover a wide range of development projects. As a result, these processes exhibit large size, high complexity, and require a high degree of individualization for specific projects. Companies such as Motorola (Fitzgerald et al., 2003) and Siemens AG (Schmelzer & Sesselmann, 2004) are committed to process-driven systems development and have developed formalized work processes that each development project must comply with.

The nature of products and environments among business units exerts significant pressure toward diversity and create the desire to engage in adaptation of generic reference processes to suit specific development projects and business units, while retaining as much commonality as possible with the reference process. The size of the processes and discrepancies between the reference processes and the needs of particular projects render this task difficult. In addition to the structure of the reference process, organization-specific constraints associated with particular tasks must be respected when altering the reference template (Bajec et al., 2007). Current practice for this adaptation is to manually apply changes to the reference process template, which is a time-consuming and error-prone approach.

Commercial process modeling tools often only provide support for basic editing tasks. The inability to express and validate organization-specific requirements has led to poor process management, where errors in processes are often rectified once compliance violations with respect to the reference process have been detected. Automated support for process design, adaptation, and validation has the potential to yield significant improvements in the quality of customized processes and a reduction in the effort required to perform adaptation. Although considerable results have been achieved by formal modeling and analysis of processes (van der Aalst, 2000), analysis techniques predominantly focus on verifying that undesirable states, such as deadlocks, cannot be reached

Reprint requests to: Wolfgang Mayer, University of South Australia, School of Computer and Information Science Mawson Lakes Boulevard, Mawson Lakes, SA 5095, Adelaide, Australia. E-mail: wolfgang.mayer@cs.unisa.edu.au

in any execution. Organization-specific requirements and constraints have largely been disregarded.

In contrast, methods developed for automated configuration and customization of complex systems have traditionally incorporated vast quantities of domain-specific knowledge. In this context, *configuration* means assembling a larger system from a set of available components. Modeling principles to guide the formulation of large domain-specific knowledge bases have been established (Soininen et al., 1998; Stumptner et al., 1998; Felfernig et al., 2001; Asikainen & Mannistö, 2009) in conjunction with efficient inference procedures (Mailharro, 1998; Stumptner et al., 1998; Magro, 2010). Declarative modeling principles advocate maintainability and scalability of the approach. However, most approaches have been designed for the configuration of physical systems and product lines, whereas work processes have received comparatively little attention. Albert et al. (2005) present a workflow composition methodology that relies on domain specific and generic models, but requires a library of given workflow templates. Conceptual (Heiskala et al., 2005) and constraint-based models (Mayer et al., 2009) have been developed for the synthesis of executable software processes from individual components. Customization methods for service agreements have also been developed (Dausch & Hsu, 2006).

However, existing configuration methods cannot directly be applied to work process adaptation: the absence of detailed specifications of the predominantly manual activities in development processes precludes the application of process synthesis from individual work steps, while the need to adapt the process structure based on specific project requirements prohibits the creation of a comprehensive library of specialized process templates. Instead, generic reference processes in conjunction with lightweight domain-specific models can be exploited to guide the configuration process in a semiautomatic manner.

This article presents extensions of established configuration mechanisms to support process adaptation and validation in the presence of domain-specific entities and constraints. A process manipulation framework is presented that allows the semiautomatic adaptation of generic processes to specific requirements of a given project. The framework extends constraint-based configuration techniques to processes by introducing a process-oriented constraint language and solving strategies. This article shows that existing constraint-based formalisms lack the ability to adequately represent path constraints, such as precedence and dominance, in generic work process models. It contributes a constraint language suited to formalize path constraints within a generic process metamodel for work processes and demonstrates how this formalism can be integrated into a generative constraint satisfaction framework for reference process configuration. A version of the approach was used to provide support in instantiating real-world processes in the software development process framework employed by Siemens AG.

First, process models and related adaptation tasks are discussed. Second, a model-based framework for process customization is introduced and a constraint language for process configuration is presented. Third, the results obtained from using a version of the framework for providing process editing support at Siemens AG are discussed.

2. WORK PROCESS MODELS

A variety of models for documenting work processes in large organizations have been developed. Process models describe the type of activities and the order in which they are performed. Some models include explicit time constraints and resources consumed and produced by activities. Process models generally differ in the level of detail, the representation language, and the degree to which the meaning of each model element is captured in a formal language. Aalst and Hee (2004) provide a summary of the major business process and workflow modeling approaches.

In this article, we use the notation of the event-driven process chain (EPC) modeling framework (Keller et al., 1992) for illustration purposes. The EPC framework has been widely adopted to describe business processes related to system development and software project management, and its use by Siemens AG meant that EPC processes were available for testing and project outcomes could be directly applied. However, the presented approach is largely independent of any concrete process modeling language and can be applied to any framework where activities, control flow, and constraints are captured explicitly. In particular, it must be noted that EPCs are a semiformal notation, and our framework is consistent with the use of the notation in the ARIS toolkit that Siemens uses for working with EPCs.

The EPC approach relies on a flow-chart-like graphical language for presentation, where individual activities are linked together to form process "chains" describing the possible sequences of executions (the "control flow" in process modeling terminology). Figure 1 shows an example of an EPC process model. A process is represented as a graph consisting of nodes and edges (flow connectors). Nodes of different shapes denote functions and flow connectors specify the possible paths through the process. Nodes represent "regular" functions (activities), control functions, or events. Regular functions are represented as rectangular shapes and represent activities that are performed. The label of a function informally describes the activity it represents. Control functions are drawn as circles and govern the execution and synchronization of different branches in the process. The EPC language provides control functions to select one of several successor branches ($\vee$ split function) and to execute several branches in parallel ($\wedge$ split function). The same notation is used to merge alternative branches ($\vee$ join function) and to synchronize the execution in different branches ($\wedge$ join function). Events are represented by hexagons and indicate that a certain state in the process has been reached.

The semantics of the EPC language (Keller et al., 1992; Kindler, 2006) are inspired by the operational semantics of Petrinets (Aalst & Hee, 2004) and specify the possible sequences of functions that can be executed using a simple activation model based on the transfer of activation tokens along the

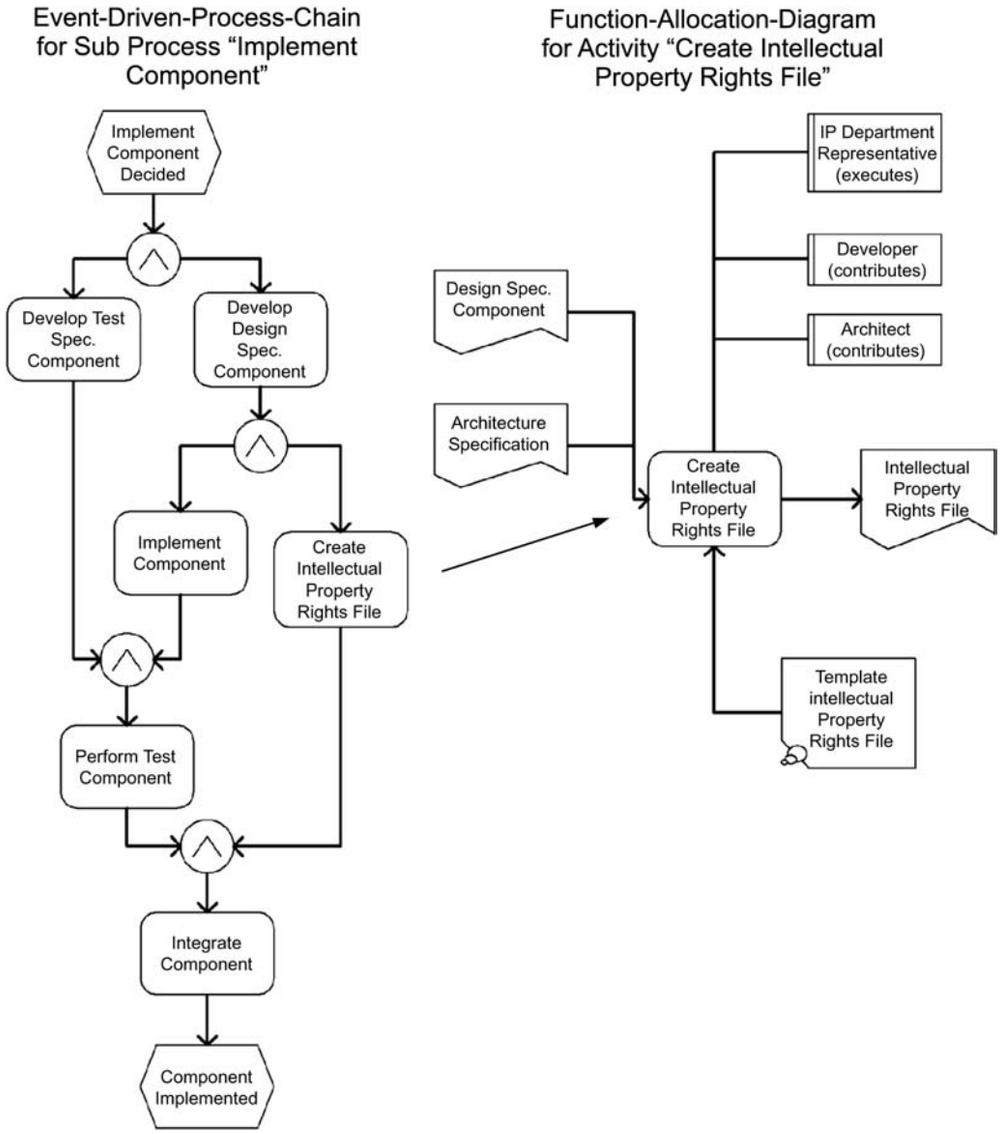

Fig. 1. The event-driven process chain and corresponding function allocation diagram.

control flow connectors. Informally, a function is ready to be executed if all incoming control flow connectors provide a control token. Once activated, a function consumes one or more control tokens from its incoming flows, and generates one or more control tokens on its outgoing flows once the function has finished executing. The tokens traveling along outgoing flows subsequently activate their successor function(s).

The example process in Figure 1 specifies that the test specification can be developed in parallel with the design specification and the implementation, but both must be completed before testing can be performed. Similarly, the intellectual property declaration can be created while the implementation and testing activities are in progress, provided that all are completed before the software component is integrated with other components.

Although the EPC framework prescribes the structural rules and activation rules for activities every model must adhere to, the meaning of the user-defined activities is described by textual labels and is not otherwise captured formally. The EPC model can be complemented with function allocation diagrams (FAD; Scheer, 2000) that model the artifacts, data, and resources that are used and produced by a function. Figure 1 shows an example of an FAD. The diagram shows the documents that are required and produced by the activity (polygons with zig-zag bottom line), resources allocated to the activity (rectangular nodes with double border), and files related to its execution.

The EPC and the FAD diagrams provide orthogonal views of the same process. Although the EPC notation facilitates visualization and manipulation of the control flow of the process,

the FAD's main concern is resource allocation and artifact tracking.

To effectively manipulate a process, both views must be considered in unison. However, the size of typical development processes prohibits detailed visualization of an entire process and its dependencies. Because changes made in a model of a subprocess or FAD easily propagate to other parts in the process, it is difficult to assess implications of local changes on the entire process. As a result, manual process adaptation may result in processes that no longer conform to the prescribed reference processes, and may contain activities that are unnecessary in the context of a specific project. Similar problems can arise with resource allocation, where resources are allocated that may be unavailable at the required time.

2.1. Process adaptation

Process adaptation approaches can be distinguished by the adaptation principles that are applied. We distinguish reductive adaptation and generative adaptation.

Reductive adaptation is predominantly concerned with making minor modifications to a given reference process. Reductive adaptation deals with minor changes to subprocesses and the implied effects that propagate throughout the entire development process. Typically, some subprocesses would be deleted, alternative subprocesses for an abstract activity would be selected from a given library, and resources would be defined and allocated to individual activities.

Generative adaptation strives to synthesize entire processes and subprocesses in order to satisfy a given goal. Different from reductive adaptation, no reference process is given, and the goal and properties of individual activities must be specified in a formal language. This approach has gained prominence particularly in the context of service oriented software architecture, where complex software systems are synthesized using models of the individual software components. The main focus of this article is on reductive adaptation that will support the specialization of generic work processes to suit a specific project context. Reductive adaptation fits more naturally in this context, because the activities in the generic process are not usually specified in a formal language. However, the formal framework presented in this article can be extended to process synthesis (Mayer et al., 2009).

Typical adaptation operations include adding and removing process elements, such as activities, milestones, artifacts and resources, duplicating a subprocess, and associating activities with specific artifacts and resources.

Although reference processes are often created with dedicated process modeling tools, such as ARIS (IDS Scheer, 2006), and executed with the help of project management tools and workflow engines, little support is available for semiautomated tailoring and compliance checking of resulting processes with the reference processes. Niknafs and Ramsin (2008) investigated available tools and came to the conclusion that only limited prototypes exist that support a few steps of the method engineering process but do not provide a complete solution. For example, the authors identified only a single tool that is process oriented. However, it does not provide a semantic data model that is required for automated verification. Killisperger (2010) also concluded that the same limitations apply in the areas of information system development, human-centered processes, and system-centered processes.

Recognizing and managing the implications of local changes on distant activities and finding suitable repairs for violations that occur during adaptation all remain predominantly manual tasks. Consequently, it has been observed that adaptation frequently results in invalid or inefficient processes, where activities are performed in a way that violates the reference process, or where activities are carried out that are not actually required to achieve the goal of the development project.

Project-specific adaptation of development processes has attracted significant attention, such as Brinkkemper's method engineering proposal (Brinkkemper, 1996) for the creation of situational methods, and the extension of the "product lines" concept to software processes (Armbrust et al., 2008) in order to manage variation. Simple forms of process adaptation have also been introduced in business process and workflow modeling frameworks, such as configurable EPCs (Rosemann & van der Aalst, 2007). However, their focus is usually on structural properties of the process and methods are restricted to simple adaptations, such as enabling and disabling individual activities and pattern-based rewriting of local subprocesses. Validation of the resulting process focuses on the structural integrity of the process and its operational behavior, such as verifying the absence of deadlocks. Domain-specific constraints and constraints governing the assignment of resources throughout the process are usually not respected in the generic adaptation and validation procedures.

Transforming a generic reference process into a project-specific development process generally involves three tasks: tailoring, resource allocation, and instantiation of artifacts (Killisperger, 2010).

2.1.1. Tailoring

Tailoring is "the act of adjusting the definitions and/or of particularizing the terms of a reference process description to derive a new process applicable to an alternative (and probably less general) environment" (Ginsberg & Quinn, 1995). Tailoring is not restricted to cutting away unneeded parts but also includes the addition, change, and duplication of parts of the process. Reference processes describe the development of components in an exemplary way for all components of a system to be developed. This general description must be instantiated for each component. Tailoring may include particular sections or constructs in the process as well as specific elements such as activities, artifacts, milestones, and resources.

2.1.2. Resource Allocation

The assignment of resources to activities is called *resource allocation* (Aalst & Hee, 2004). In software development, this predominantly involves the assignment of human resources to activities. However, reference processes are defined in terms

of abstract roles rather than functions, which must be translated into specific project resources. This translation may be complicated by availability and scheduling constraints, some roles may be filled by multiple resources, and project participants may act in several roles. For example, the role *Developer* in a reference process would normally be filled by multiple engineers.

2.1.3. Instantiation of artifacts

Because a reference process is independent of any particular project, artifacts are described by generic templates and placeholders. Similar to processes, such generic artifacts must be individualized for a specific project, and an appropriate number of copies must be associated with relevant activities in the tailored process.

2.2. Process adaptation requirements

Large-scale processes are notoriously difficult to tailor, as it can be difficult to keep track of the structure of the overall process hierarchy, interdependencies between tasks in different subprocesses, and the allocation of resources to tasks at different times. Tools like the ARIS process modeling system (IDS Scheer, 2006) have been used in many organizations to document their work processes. The availability of convenient graphical user interface tools to create and visualize complex processes make these tools an attractive choice for process modelers. However, complex adaptation and restructuring of large processes remains difficult, as little support is provided to ensure that the result adheres to the structural and domain-specific constraints that user may have imposed on the processes.

Several aspects must be considered simultaneously in order to provide effective support for creating and editing process models.

2.2.1. Custom Metamodel

Activity types, their properties, and relations to other process elements shall reflect organization-specific processes. A flexible approach is desired, where the individual classes of activities, relations, and other process entities can be changed without altering the software implementation. For example, most existing frameworks provide either a fixed model that is usually accompanied with rigid mathematical analysis tools (van der Aalst, 2000) or flexible process editors with few analysis capabilities. The approach presented in this article aims to cover the middle ground between the two extremes, providing a flexible metamodel that is enriched with constraints that govern structural and organization-specific requirements on process models.

2.2.2. Syntactic correctness

The syntax of a modeling language is generally defined by rules governing the structural composition of process elements and their connections. These rules are typically defined in a metamodel that specifies the valid process models that will be accepted in a modeling language. Figure 2 depicts a metamodel for the EPC language introduced earlier in this article. For example, the EPC metamodel requires that control flow connectors link only functions and events. Two subsequent flow connectors without an intermediate function or event violate the metamodel and should be rejected by any decent modeling environment. Furthermore, it is required that each flow connector originates and ends in a function or event; no "dangling" connectors are allowed.

Although syntactic correctness is an essential requirement for any process model, this criterion is insufficient to guarantee that a model with well-defined operational behavior is obtained in general.

2.2.3. Semantic correctness

The execution of a process model is governed by rules outlining the operational behavior of the model elements. For EPC models, the execution of activities is defined by firing rules derived from Petri nets as outlined in van der Aalst (1999). Each node in an EPC model with its connectors is mapped to a Petri net fragment. The simple and precise execution semantics of Petri nets specify the execution sequence of the activities in the EPC model. The behavior exhibited by a process model is defined by applying the firing rules to each activity in the process model.

Although the propagation of tokens by a single activity can be understood easily, the global behavior of complex processes may be difficult to assess. In particular, syntactic correctness does not imply that the overall process model functions correctly, that is, each possible execution admitted by the model will eventually reach a final activity and complete the process. For example, a connector that flows into the same activity it originates from will prohibit this activity from ever being activated. Consequently, the execution will halt at this point and prohibit the process from completing successfully. This condition is generally known as "deadlock" (van der Aalst, 2000).

Although some undesirable patterns can be detected locally, in general, the entire process must be analyzed in order to validate its behavior. Simulators and other process analysis tools have been developed to aid in this task (van der Aalst,

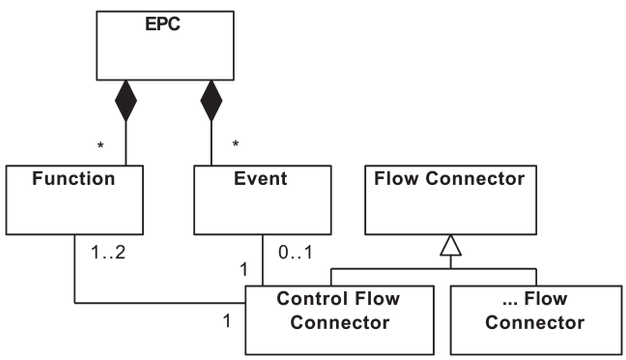

Fig. 2. The partial event-driven process chain metamodel in Unified Modeling Language notation (List & Korherr, 2005).

2000); however, most process editing tools provide only limited support for such validation. As a result, the implications of local changes in large processes may be difficult to assess.

Assessing semantic correctness for EPC models is further complicated by the fact that the meaning and effects of individual user-defined activities is only captured informally. Although the flow of tokens can be simulated, the EPC model lacks the information that is necessary to verify that domain-specific assumptions and constraints have not been violated. Typically, this verification must be performed manually once the model has been created (Killisperger et al., 2008). Because this task is time consuming and may require detailed knowledge of the entire process, modified processes often do not undergo appropriate validation when changes are applied. Serious errors in several large business processes have been identified through systematic analysis (Mendling, 2009), and previously unknown errors in the customized Siemens processes were uncovered by our analysis described in Section 4 in this article.

2.2.4. Data flow and resource usage

In addition to correct control flow, work processes must also adhere to the correct use of artifacts and resources generated and consumed by the process. For software development processes, typical examples include artifacts, such as documents and source code that is produced, and resources such as manpower, software licenses, and computing systems. Pure process models generally focus on the control flow aspects of processes and leave the implications for resource allocation and availability hidden.

If resource allocation is done separately, critical resources may be unavailable at the time an activity would require them. In addition, the shared use of artifacts may induce dependencies between tasks that may not be explicit in the process model. If processes are restructured without consideration of hidden dependencies, seemingly correct process models may exhibit deadlocks because of unavailable input artifacts and resources. Furthermore, resource-induced dependencies may link seemingly unrelated subprocesses, making it difficult to assess the impact of local changes on the overall process.

Consider the example process in Figure 1. Both activities *Implement Component* and *Create Intellectual Property Rights File* require input from the software developer. The shared resource induces a potential dependency between the activities. If resources are tight, activities may have to be scheduled sequentially rather than in parallel (as would be permitted by the process model). Furthermore, the execution of the entire subprocess *Implement Component* also depends on the remote IP Department's schedule. This dependency is not evident from the control flow and the activity labels alone, but it may severely constrain the possible process schedules, something that must be taken into consideration when altering the *Implement Component* subprocess.

Large work process models are typically defined as a (hierarchical) process where the possible sequences of (abstract) activities are specified, but the precise meaning of the individual activities is not usually formally expressed. It follows that any attempt to provide automated support to the process designer will have to rely on semi-interactive mechanisms and seek the designer's input to resolve ambiguities and choose between different alternative adaptations.

3. PROCESS CONFIGURATION FRAMEWORK

Adapting a common reference development process to the needs of a particular project can be seen as a configuration task, where the general reference process must be tailored to suit the requirements of a given project. The configuration goal in this scenario is a process that is consistent with the reference process template, has appropriate resources allocated, and no extraneous activities. Although project managers will usually be able to decide which changes must be made to the reference process, applying these changes consistently remains challenging for the reasons outlined in the previous sections. It is here where semiautomated support to check the validity of the customized process and suggest alternative changes is most valuable.

Approaching this problem as a configuration task, the reference process constitutes the initial configuration, and the configuration goal is a valid process that conforms to both the generic organizational and the project-specific requirements. However, different from traditional applications of automated configuration tools, where systems are predominantly built up from an empty initial state, configuration of processes works on an initial process that is subsequently modified to suit a goal that may not have been completely formalized. Rather, configuration in this context is largely a task that follows the propose-and-revise principle (Marcus & McDermott, 1989), where manual changes applied by the project managers are validated, and remedies for problems are proposed. The ability of configuration frameworks to incorporate both the concrete initial process state and the generic requirements of the reference process makes this approach particularly attractive. The search for repairs of inconsistencies between the given modified process and the reference process replaces the synthesis of complete configurations from a sparsely instantiated initial configuration.

Although the overall knowledge-based configuration principle is suitable for process adaptation, the underlying knowledge base describing the valid process variants is quite different to that of classical configuration. In the process adaptation scenario, the exact properties of activities are rarely formalized in detail, and constraints on processes will usually involve a number of activities along a path.

The following sections introduce the conceptual framework that supports the representation of processes and the constraints required in the system development process domain.

The framework addresses the major expectations set out for a semi-interactive process configuration tool: organization-specific metamodels are incorporated in an extensible process metamodel, which also forms the basis for constraints that enforce syntactic and semantic correctness. Artifact and resource usage are also governed by constraints. Process entities that

should not be altered (because they correspond to activities that have already executed or that have been introduced manually) are excluded by constraints that prohibit the application of certain modifications to these process entities. Temporary inconsistencies are tolerated while manual changes are made, as explanations for constraint violations are provided by highlighting the process entities involved in a violated constraint. Remaining inconsistencies are resolved later through heuristic search. Heuristics attempt to confine changes to local subprocesses and derive processes with no unnecessary entities.

3.1. Conceptual metamodel for process configuration

Knowledge-based configuration requires a specification of the potential constituent components and constraints that specify the set of possible configurations. For work processes, the available component types, their properties and associated value domains, and possible relationships between components must be given. Constraints attached to each entity restrict the valid values and relationships between entities comprising a configuration.

Our process configuration framework relies on a generic process metamodel that captures process entities such as elements, relations between elements, and process scopes. Figure 3 shows the major components of the conceptual process metamodel that has been developed for the Siemens AG application. The model consists of a generic framework that remains the same for all processes, and organization-specific entities that represent the concepts and properties relevant for a particular organization or application domain. Although some model parts are specific to the Siemens Reference Process, the modeling approach is generic and can be applied to a variety of different process models, languages, and domains. The metamodel is also consistent with the EPC metamodel in Figure 2. The embedding can be extended to the full EPC metamodel described in (List & Korherr, 2005) by introducing additional subclasses and constraints. Some classes have been renamed to reflect the organization-specific terms that are used in the definition of the Siemens reference process. For example, the *FlowConnector* in the EPC metamodel is named *Flow* in our framework. Intermediate subclasses such as *ExecutableElement* have also been introduced to capture constraints common to all subclasses that exist in the reference process but not in the generic EPC metamodel.

Conceptual class *Entity* forms the central component of our metamodel that acts as an abstraction of all elements of a process. This concept introduces a unique identifier for each entity instance and establishes the relationship and constraint between the model entities.

A process is represented as a graph structure, where the nodes represent executable actions, resources, and artifacts. Class Element forms the basis of a hierarchy that models specific types of nodes in each category.

Class *ExecutableElement* models an executable node. Most immediate specializations are determined by the EPC process modeling language. Class *Split* represents EPC nodes that introduce multiple branches into the process, and class Join models nodes where multiple branches rejoin.

The subclasses representing $\wedge$- and $\vee$-split and join activities have been omitted for space reasons. Class *Event* represents

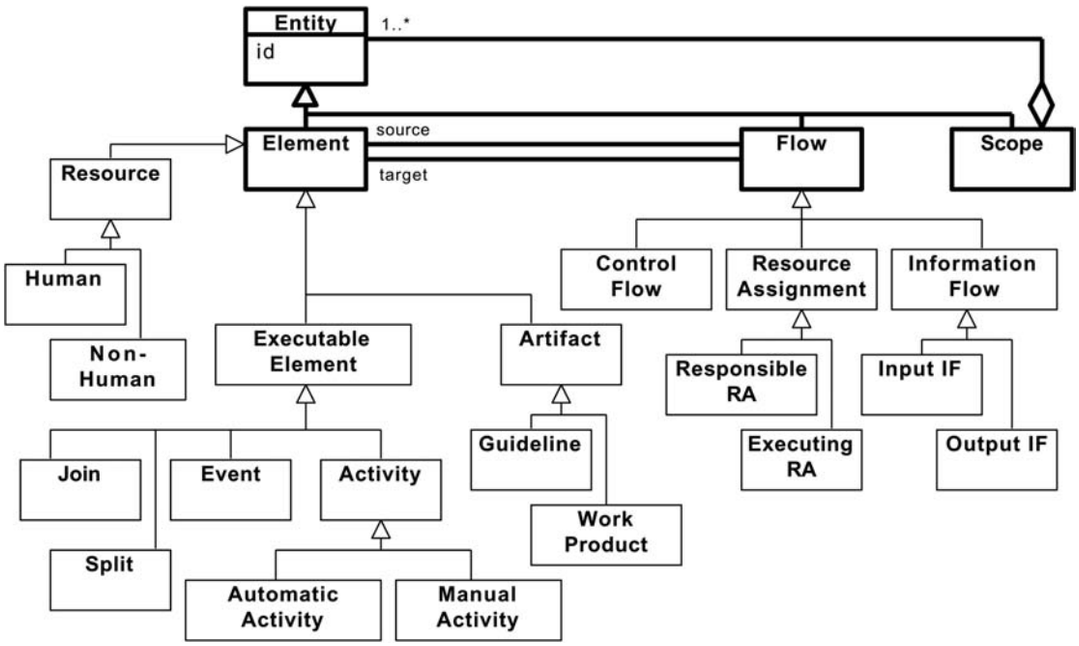

Fig. 3. The main entities of the Siemens Reference Process metamodel in Unified Modeling Language notation. The model part highlighted in bold represents the generic process framework, and the remaining parts are specific to the Siemens Reference Process. Most organization-specific attributes have been omitted for brevity.

signals that trigger subsequent executable elements in the process. Class *Activity* is used to model activities that will be carried out in the process (represented as Functions in the EPC language). The subclasses *AutomaticActivity* and *HumanActivity* distinguish between manual activities and those that are carried out automatically by software systems. These classes are modeled separately, because different constraints apply to each kind.

Executable elements are complemented with models of resources and artifacts involved in the execution of activities. Class *Resource* models the common properties of all resources available in the reference process, and its specializations *HumanResource* and *NonhumanResource* cover personnel and software and hardware tools.

Class *Artifact* and its specializations *WorkProduct* and *Guideline* express the data that are associated with activities. Class *WorkProduct* covers documents, source code, and other data that are used and created in the development process. Class *Guideline* represents external information that is required to perform an activity, where the information does not change throughout the development process. Constraints specific to each subclass reflect the different usage patterns.

In addition to the element hierarchy, control flow between executable elements and associations between activities, resources, and artifacts must be captured in the metamodel. Different to many other configuration frameworks, relations are represented explicitly as conceptual classes. This approach is preferred over models where connections are represented implicitly as properties, because conceptually different relationships can be distinguished by specialization, and constraints can be attached to each class. An equivalent model where all constraints are attached to *Elements* can only be obtained at the expense of duplicated of constraints in multiple classes.

Class *Flow* represents a relation between elements. Subclass *ControlFlow* models the propagation of activation between activities in a process. The subtree rooted in class *ResourceAssignment* models the assignment of resources to activities. Classes *ResponsibleResourceAssignment* and *ExecutingResourceAssignment* are specific to the Siemens Reference Process and distinguish the different roles that resources can play. Class *InformationFlow* links an artifact with the activities where it is created and used. The information flow is partitioned into input and output flows, because different constraints apply to each class.

Class *Scope* comprises the entities in a process or subprocess. Its main purpose is to capture constraints that apply to an entire subprocess rather than a single element or flow entity. This model element is essential for representing constraints that express properties of paths limited to certain subprocesses.

To avoid potential confusion, it is important to make the distinction that this model is a metamodel of the process models built by process designers. It is not a metamodel of the configuration process. The metamodel is a conceptual description of the problem domain, and can hence be categorized as a *configuration model* of Soininen et al. (1998). The metamodel does not describe configuration model concepts, which could be seen as a metamodel of the configuration task. Because the term metamodel is not actually used by Soininen et al. (1998), we retain the conventional use of the term in software engineering as "a collection of concepts in a domain."

The conceptual metamodel provides the basis for specifying the activities and flows comprising the reference process and its constraints that all variants must satisfy. The next section introduces the formal framework that will be used to encode the conceptual metamodel as constraints that can be used to automatically validate process variants.

3.2. Constraint-based configuration

As a problem-solving technique in technical systems, the term *configuration* refers to the assembly of systems from smaller components. To automate this process, the available components and possible interconnections must be formalized. Typically, a taxonomy of principal component types, their attributes, their interfaces through which components can be connected to other components, and constraints that govern the valid compositions must be specified. It is assumed that the available components types are known, but the size and structure of a particular solution is not. It has been shown that this knowledge about a problem domain can be elegantly expressed as a constraint satisfaction problem (CSP; Mittal, 1990; Stumptner et al., 1998).

A CSP consists of a finite set of variables $\mathbf{V}$ and a finite set of constraints $\mathbf{C}$ in which bold is used to denote set-valued variables. A constraint is a relation over a set of variables $\mathbf{V}' \subseteq \mathbf{V}$ that specifies the valid combinations of value assignments to these variables. The set of values that may be assigned to a variable is called the domain of that variable. Solving a CSP means finding an assignment to all the variables such that all constraints are satisfied.

The main obstacle in using CSPs for configuration is that the set of variables and constraints is static and thus cannot be used to describe problems where the structure or size of the solution is unknown. Dynamic constraint satisfaction formalisms have been developed where the set of variables is extended on demand, and conditional constraint satisfaction techniques have been devised in order to flexibly enable and disable constraints (Rossi et al., 2006).

*Generative constraint satisfaction problem*s (GCSPs; Stumptner et al., 1998) provide a combination of both techniques by lifting constraints and variables to a metalevel, where *generative constraints* describe the valid CSPs instances. Generative constraints can be seen as constraint schema that drives the solving process by introducing fresh CSP variables and constraints as new components are added to the configuration. Therefore, GCSPs can adapt the problem size and structure during the solving process based on the information contained in the partial solution.

A GCSP is characterized by the set of available component types, their attributes, and a set of generative constraints: let $\mathbf{T}$ be the set of available component types and let $\mathbf{A}$ be the set of attribute names.

DEFINITION 1 (GCSP). A GCSP is a tuple (**X**, **Γ**, **C, P**), where **X** is a set of meta variables, **Γ** is a set of generative constraints over **X**; **C** is an infinite set of constraint variables representing component instances, and $\mathbf{P} = \{C.a | C \in \mathbf{C}, a \in \mathbf{A}\}$ is an infinite set of constraint variables representing the components' attributes. Each component variable $C \in \mathbf{C}$ may be assigned a value from **T**, whereas attribute variables $P \in \mathbf{P}$ may choose their value from **C** (if P represents an attribute connecting to another component), or may take on a numeric or a string value (if P represents an attribute). ■

Each variable in $C \in \mathbf{C}$ may be *active* or *inactive*; it is active if C corresponds to a component that is part of the solution of a configuration problem, and it is inactive otherwise. Similarly for variables in **P**.

A generative constraint is a logical implication of the form $\Phi \Rightarrow \Psi$ over variables in **X**, where Φ represents the precondition of the constraint and Ψ specifies the variables and constraints that are introduced into the CSP if Φ holds for some instantiation of **X** in **C**. Typically, Φ is used to express that constraint Ψ is applicable only for particular component types, whereas Ψ enforces the requirements on attribute values and connections. Free variables in generative constraints are implicitly universally quantified.

A configuration problem confines a GCSP to those CSPs that satisfy the configuration goal, which is expressed by an initial set of components and constraints that must be met.

DEFINITION 2 (configuration problem). A configuration problem is a CSP $\langle \mathbf{V}, \Delta \rangle$ where $\mathbf{V} \subseteq \mathbf{C} \cup \mathbf{P}$ represents the initially active variables, and Δ contains a set of (generative) constraints over **V** that express the initial configuration and the desired configuration goal. ■

A solution to a configuration problem R is given by an assignment of values to all CSP variables in R such that all constraints in R are satisfied and all instances of generative constraints over variables in R are satisfied.

To find a solution, all generative constraints $\phi \Rightarrow \psi \in \Gamma$ are instantiated with variables in **V** to check if ϕ is satisfied. If an instantiation of a generative constraint is created, the CSP is extended with the variables and constraints in ψ. This procedure repeats until no more constraints and variables can be instantiated. A solution to the configuration problem has been found if all possible instantiations of generative constraints have been applied and a consistent complete assignment of values to variables in **V** has been found. Otherwise, the procedure backtracks and selects alternative instantiations of generative constraints.

Consider the example where a software component is to be implemented as part of a work process. Among others, this activity requires the software design specification, a developer, and suitable computing resources. Assume that the component types are $\mathbf{T} = \{$*ImplementationActivity*, *SourceCode*, *SWDeveloper*, *DesignSpec*, *Workstation*$\}$, and let $\mathbf{A} = \{C, D, S, W\}$. Here the attributes model connections between components: C refers to a source file, D models the software developer, S the specification, and W the assigned workstation to be used for development. The GCSP representing this domain contains an infinite number of component variables c_i, each of which may be assigned a component type from **T**. Similarly, the set of property variables is given by $\mathbf{P} = \{c_1.C, c_1.D, \ldots, c_2.C, \ldots\}$, each with domain $\{c_1, c_2, \ldots\}$.

Generative constraints describe the relationships and restrictions between component and attribute variables. For example, to assert that each *ImplementationActivity* component must be connected with a design specification (a component of type *DesignSpec*) via its S attribute, a generative constraint

$$X = \mathit{ImplementationActivity} \wedge X.S = Y \Rightarrow Y = \mathit{DesignSpec} \quad (1)$$

is asserted in Γ. Here, X and Y denote metavariables that will be instantiated with variables in **C**. Note that the domain of $X.S$ is the set of component variables; therefore, the generative constraint establishes the structure of the instance-level CSP dynamically. Once instantiated, the generative constraint will manifest as an ordinary CSP constraint. The constraints will assert that, for some CSP variable c_i, the property variable $c_i.S$ must refer to another component variable c_j whose value is *DesignSpec*. Assuming that similar generative constraints are defined for *DesignSpec* and the remaining component types, further constraints will be created that restrict the values of c_j's property variables and neighbors. A complete configuration is then given by a set of component variables together with instantiated generative constraints such that all instantiations of generative constraints have been formed and are consistent.

3.3. Constraints for process configuration

In an attempt to extend the configuration approach from physical systems to processes, constraints associated with the entities in the process metamodel must be identified. The constraints will be used by the configuration engine to assess the validity of a (possibly incomplete) candidate process and to guide the search for admissible process variants.

In order to obtain a formal model suitable for automated configuration, the process metamodel and its associated constraints must be rephrased in terms of components and connections. The process entities in Figure 3 form the component types, and their attributes translate into attributes of the corresponding component type. Associations in the metamodel are also represented as attributes. We adopt the model introduced by Mailharro (1998) to represent set-valued attributes and relations: associations with cardinality greater than 1 appear as multiple instances of the same attribute. For example, a *Scope* entity may be associated with multiple subentities via its entities association. An instance of the *Scope* entity may have multiple *entities* attribute instances, each referring to a different *Entity* instance. Cardinality constraints restricting the number of associations are also an integral part of process modeling. In the following sections, Mailharro's work on direct associations between components is extended to generative constraints formalizing the reachability of components within a process.

Table 1 lists the predicates and operators permitted in generative constraints. Although other connectives may be introduced, this set was found to be sufficient to express all constraints used when applying the approach to the large-scale company-wide Siemens reference process described in this article. Following an established constraint notation (Stumptner et al., 1998), the term $T(C)$ expresses that metavariable C must refer to an entity of type T, or a subtype thereof. Expression $C.A$ refers to an attribute A of a component represented by variable C.

The configuration of processes poses additional challenges related to expressing properties of paths and subprocesses that may not be known precisely at the time the constraint is written. Different types of constraints can be distinguished based on the locality of affected process entities.

3.3.1. Entity and attribute constraints

Entity constraints are concerned with constraints that affect only a single entity or local scope in the entire process. Such a constraint affects only the values of attributes of entities that can be reached from a single entity by navigating a fixed path comprising the relations between entities in the process model. This type of constraint is most often used to specify value range restrictions of attributes.

In the simplest case, only the attributes of a single process entity are affected by a constraint. For example, assume that entity *ExecutableElement* has an attribute duration that holds the length of the time span during which the element will be executed, and that the duration must always be a nonnegative number. This constraint can be expressed as generative constraint:

$$ExecutableElement(E) \Rightarrow E.duration \geq 0 \quad (2)$$

Similar to the constraints in the work process configuration example given earlier, *ExecutableElement(E)* acts as a type test that determines when the consequent of the constraint will apply. In this example, the constraint applies to all constraint variables that model an instance of entity *ExecutableElement*. Type hierarchies and inheritance relationships in the metamodel are taken into account implicitly; thus, the constraint also applies to all instances of subtypes.

Constraints reflecting project-specific restrictions on the entity types that may appear in a process also belong to this category. Certain activities and flows can be excluded from a process by type constraints and constraints on attribute values. For example, the following constraint prohibits all activities with a label containing the word "Mechanical."

$$Activity(C) \Rightarrow C.label \neq \text{``*Mechanical*''} \quad (3)$$

A comprehensive domain model for reasoning about the properties of activities and artifacts can be incorporated in the same way if available. Although the given example constraint may seem overly simplistic and error prone, the controlled language used in the definition of the reference process resulted in sufficient precision when applying the approach solve the process adaptation problem in the Siemens Process Framework (SPF) as described later in this article.

3.3.2. Association constraints

Constraints that govern the valid connections between entities belong in this category. Different from the previous example, the constraint may refer to the properties of neighboring components by navigating via relations to connecting entities. For example, the requirement that only a *HumanResource* element can be responsible for the execution of an activity can be expressed as follows:

$$Activity(A) \wedge ResponsibleRA(RA) \wedge RA.source = A \wedge RA.target = R \Rightarrow HumanResource(R) \quad (4)$$

Here, the dot notation is first used to navigate from the *ResponsibleRA* to the connected *Activity* entity, and is used again to constrain the possible values of the target attribute

Table 1. *Predicates and operators for generative constraints*

Predicate	Meaning	Connective	Meaning
$T(C)$	Entity C is of type T	$C.A$	Attribute A of entity C
$\{C_1, \ldots\} \lhd C_2$	$\{C_1, \ldots\}$ s-dominates C_2	$\neg E$	Logic negation
$C_1 \rhd \{C_2, \ldots\}$	$\{C_2, \ldots\}$ s-postdominates C_1	$E_1 \wedge E_2$	Logic conjunction
$=, \neq$	Equality and disequality	$E_1 \wedge E_2$	Logic disjunction
$<, \leq, \ldots$	Numeric inequalities	$E_1 \Rightarrow E_2$	Logic implication
$\subset, \subseteq$	Subset relation	$\exists V : E$	Existential quantification
		$\forall V : E$	Univarsal quantification
		$\bigcirc V : \boldsymbol{E}$	Set reification
		$\|\|V\|\|$	Set cardinality
		$N_1 \oplus N_2$	Numeric expression $\oplus$ can be one of $+, -, \times, \div$, *max*, *min*

Note: C, C_1, and C_2 denote metavariables in generative constraints; E, E_1, and E_2 represent logic expressions; N_1 and N_2 represent numeric expressions; T is an entity type name; and V and A denote a variable and an attribute identifier, respectively.

of the connecting entity. In this case, the target is restricted to be of type *HumanResource* (or a subtype thereof).

3.3.3. Cardinality constraints

The same mechanism can be used to enforce cardinality constraints for relations between entities. For example, the following generative constraints express that every *ExecutableElement* must be related to a resource via exactly one instance of the *ResponsibleRA* relation:

$$ExecutableElement(E) \Rightarrow \exists R : ResponsibleRA(R) \wedge R.source = E \tag{5}$$

$$ExecutableElement(E) \wedge ResponsibleRA(R1) \wedge ResponsibleRA(R2) \wedge R1.source = E \wedge R2.source = E \Rightarrow R1 = R2 \tag{6}$$

However, cardinality constraints can be expressed more succinctly by using set reification, denoted as $\bigcirc V : E$, in conjunction with the set cardinality operator $\| \cdot \|$. Set reification collects all values bound to variable V in a model of a given logic expression E into a multiset. Variable V must appear free in E and is implicitly universally quantified. This operation is commonly known as *collect* in other object-oriented constraint languages (Object Management Group, 2006). This connective is convenient for the application of predicates to sets of selected entities. Using the reified notation, the constraint can be expressed more naturally:

$$ExecutableElement(E) \Rightarrow \|\bigcirc R : (ResponsibleRA(R) \wedge R.source = E)\| = 1 \tag{7}$$

3.3.4. Path constraints

Restrictions that relate to the properties of the entities in a *path* of a process require a different modeling approach. Path constraints arise when relationships between entities that are distant from each other must be enforced. For example, a reference process constraint that every software component must have undergone testing before it can be integrated into larger units can be informally expressed as a path constraint:

> *In every feasible process execution from an* Implement Component *activity to an* Integrate Component *activity, there must be a* Perform Test Component *activity.* (CONS)

Although it is easy to see that this constraint is satisfied in the example process shown in Figure 1, the constraint may affect different entities in other processes. For example, if there were additional entities between "Implement Component" and "Perform Test Component," the constraint would have to propagate over those components. Such scenarios arise easily in process adaptation as processes are merged and restructured to accommodate resource constraints and scheduling conflicts. In general, the entities involved in a path constraint may not be known at the metamodel level, and the exact path connecting affected entities may not be known at the time the constraint is formulated. Instead, a (partial) process instance is required to determine the concrete set of entity instances that form the connecting paths. In the configuration of physical systems, one can introduce an additional attribute to express aggregate properties, such as total voltage with a battery compartment instead of dealing with individual batteries. However, the same principle cannot easily applied in the process domain, where activities of unrelated subprocesses may be interleaved freely. Many attributes and propagation constraints would be required, which would yield a difficult to maintain knowledge base. Therefore, this type of constraint cannot be expressed in a GCSP that relies on fixed paths between components. The constraint language must be extended to incorporate path constraints with dynamic scope that is determined on a concrete process instance. Once a concrete process is available, the execution paths and their components can be determined easily, and path constraints can be instantiated and evaluated.

In order to evaluate a path constraint the affected entities must be determined. This computation is essentially a reachability problem defined on the process graph, where constraints specify properties that must be true in some execution state in the future (or past) relative to another state. This problem has been studied thoroughly in the field of modal logic and its application to automated verification of software and hardware systems (Clarke et al., 2003).

Computing all reachable states induced by a given process model is computationally difficult (Jensen, 1997). It is of more importance that this can only be done once a concrete process model is available for analysis, which is not usually the case when the configuration knowledge base is built, because generative constraints should be written to be independent of any particular example configuration. The differences between the structural process model and its reachable state space may make it difficult for users who are familiar with EPCs but may not be experts in formal logic to write appropriate constraints. Process algebras (Milner, 1990) may align more closely with parallel processes but may be difficult to use for nonexperts, as algebraic specification languages, the EPC framework and constraint-based techniques all follow different modeling paradigms.

The discrepancy between representations of structure and transitions between reachable states has profound implications for writing path constraints. Writing temporal logic formulas to verify particular reachability properties requires us to know the branching behavior of the concrete process, where the execution is split into multiple parallel branches, and where the branches rejoin. For example, in an attempt to formalize constraint (CONS) the formula

$$Activity(I) \wedge I.label = \text{``}Integrate\ Component\text{''} \Rightarrow EO\begin{bmatrix} Activity(T) \wedge T.label = \text{``}Perform\ Test\ Component\text{''} \wedge \\ EO[Activity(M) \wedge M.label = \text{``}Implement\ Component\text{''}] \end{bmatrix} \tag{8}$$

could be devised. Here, the modal operator EO (Latvala et al., 2005) is used to denote that a control-flow path exists where,

in the past, its argument proposition is true at least once. Note that the constraint is adequate for the process example in Figure 1, but may not be strong enough for other processes with different structure and other instances of the same activity types. The fundamental problem is that the transition relation underlying the modal operators can only be determined once the branching nodes in the process are known. A formula specific to that structure can then be devised. Unfortunately, this constraint may also apply in different process variants but may have unintended effects. For example, if the parallel split and join nodes in Figure 1 were replaced with decision and merge nodes (this alternative may arise because of erroneous manual change), the constraint would still apply but would no longer enforce that *Integrate Component* is always preceded by implementation and testing.

We circumvent this problem by adopting a restricted language to express path constraints that is based directly on the process structure and not on its corresponding state representation. Our language is based on the observation that many interesting path properties can be formalized based on the concept of *dominance* (Cytron et al., 1991).

The binary *dominance* relation describes the relation between process entities with respect to the set of *all* possible process executions. It is usually defined based on the Control Flow Graph (Muchnick, 1997), a representation of the possible sequential execution behaviors of a program. Informally, an entity A *dominates* an entity B if and only if, in all process executions from the start element to the end element, an execution of A precedes every execution of B. For example, activity *Develop Design Spec Component* dominates activity *Implement Component* in Figure 1. The dominance relation defined on the inverse control flow graph is called *postdominance*. An entity A *postdominates* entity B if and only if all process executions from B to the end of the process subsequently pass through A.

Dominance and postdominance can be generalized to a set of elements:

DEFINITION 3 (domination by a set). Let $\mathbf{A} = \{A_1, \ldots, A_n\}$ be a set of entities, and let B be an entity in a sequential process P. Set **A** dominates B if and only if in all executions from the start element to the end element, an execution of an entity in **A** precedes every execution of B. ■

Postdominance can be generalized analogously. For notational convenience, the set braces will be omitted for singleton sets in the remainder of this article.

The classic definitions of control flow graph, dominance, postdominance, and their extensions to sets assume a sequential process. We extended the definition to concurrent processes as follows:

DEFINITION 4 (sequentially implied dominance). Let P be a process model and let P' be the projection of P onto subtypes of entities *Executable Element* and *Control Flow*. That is, P' is the subprocess of P where all other elements have been removed. Let further **A** be a set of entities and B be an entity in P'. Set **A** *s-dominates* B if and only if **A** dominates B in all sequential executions admitted by P'. ■

The projection of P onto its executable elements is the smallest process that admits the same set of executions as the original process. The s-postdominance relation can be defined analogously.

For example, the process definition in Figure 1 admits 11 possible sequential executions (split and join nodes are ignored for simplicity). In all executions, activity *Implement Component* is executed before *Perform Test*, hence the former s-dominates the latter. Activity *Create IP Rights File* does not s-dominate *Perform Test*, as there is a possible execution sequence where testing is performed before any intellectual property is documented. In fact, there are 3 such sequences.

The use of *s-dominance* for specifying process constraints allows the process modelers to specify constraints in terms of the process structure, whereas the evaluation engine will be responsible for checking that the relation is satisfied in all possible sequential executions. Although this operation is potentially more expensive than evaluating simple component constraints, dominance can be inferred from the process structure (Cytron et al., 1991). No exhaustive simulation of the possible executions is necessary.

The constraint language for specifying generative constraints has been extended to include s-dominance and s-postdominance relations and set reification. s-dominance is denoted by $A \lhd B$, s-postdominance by $A \rhd B$, and set reification by $\bigcirc V : E$. Table 1 lists the predicates and operators permitted in generative constraints. Other connectives may be introduced; however. these operations were sufficient to express all constraints used in our case study.

The constraint that implementation activities must precede testing, which must precede integration, can now be expressed in the extended constraint language:

$$\begin{aligned} &\mathit{Activity}(M) \wedge M.\mathit{label} = \text{“Implement Component”} \wedge \mathit{Activity}(T) \\ &\quad \wedge T.\mathit{label} = \text{“Test Component”} \wedge \mathit{Activity}(I) \wedge I.\mathit{label} \\ &\quad = \text{“Integrate Component”} \Rightarrow M \lhd T \wedge T \lhd I \end{aligned} \quad (9)$$

The use of the s-dominance relation simplifies the constraint such that no modal operator and nesting of temporal expressions is necessary.

3.3.5. Scope constraints

Dominance constraints can also be used to restrict the scope of generative constraints to subprocesses. For example, certain quality assurance activities and milestones related to component development must not be bypassed by any path throughout the development process. By using dominance constraints, the scope of a generative constraint can be reduced to the subregion or the subprocess that is relevant to such a path constraint. A conjunction of s-dominance and s-postdominance can be used to isolate a region that is delimited by unique entry and exit elements. For example, to express that in each subpro-

cess delimited by entities *Implement Component Decided* and *Component Implemented*, activity *Create IP Rights File* must be executed, can be formalized as follows:

$$\begin{aligned} &\mathit{Activity}\,(S) \wedge S.\mathit{label} = \text{“Implement Component Decided”} \\ &\quad \wedge \mathit{Activity}\,(E) \wedge E.\mathit{label} = \text{“Component Implemented”} \\ &\Rightarrow \exists P : \begin{pmatrix} S \lhd P \wedge P \rhd E \wedge P \lhd E \wedge \\ \mathit{Activity}(P) \wedge P.\mathit{label} = \text{“Create IP Rights File”} \end{pmatrix} \end{aligned} \quad (10)$$

The dominance constraints $S \lhd P \wedge P \rhd E$ confine P to match only elements that are inside a region that can only be entered via S and only be exited via E. The third constraint $P \lhd E$ states that in all executions where E is executed, P must be executed before E. Hence, P cannot be bypassed.

In the previous discussion we have implicitly assumed that there is only one instance of each activity type in a process. If multiple instances may be present, additional constraints must be introduced to ensure that the entry and exit elements of each subprocess match. We resort to constraints relating the entities of a subprocess to its unique *Scope* entity. The same principle can also be used to enforce that all activities in a subprocess share a common property. For example, all development activities in a subprocess should relate to the same work product:

$$\begin{aligned} &\mathit{Scope}(S) \wedge \mathit{Activity}\,(A1) \wedge A1.\mathit{parent} = S \wedge \mathit{OutputIF}\,(F1) \\ &\quad \wedge F1.\mathit{source} = A1 \wedge \mathit{Activity}\,(A2) \wedge A2.\mathit{parent} \\ &\quad = S \wedge \mathit{OutputIF}\,(F2) \wedge F2.\mathit{source} = A2 \Rightarrow \exists W : \mathit{WorkProduct}\,(W) \\ &\qquad\qquad \wedge F1.\mathit{target} = W \wedge F2.\mathit{target} = W \end{aligned} \quad (11)$$

3.3.6. Data flow constraints

Dominance and postdominance provide natural means to relate the data flow between entities that create certain artifacts with those entities that use the artifacts. Constraints are associated directly with the artifacts and flows rather than with the activities that create or use an artifact. Generative constraints associated with an artifact entity can navigate its associated flow entities to reach the activities that create and use the artifact. For example, the invariant that an artifact must be created before it can be used as input to an activity is modeled as follows:

$$\begin{aligned} &\mathit{Artifact}(A) \wedge \mathit{InputIF}(IF) \wedge IF.\mathit{source} = A \wedge IF.\mathit{target} \\ &\quad = U \wedge \mathit{Activity}(U) \Rightarrow \exists C : \exists OF : (\mathit{Activity}(C) \wedge \mathit{OutputIF}(OF) \\ &\qquad\qquad \wedge OF.\mathit{source} = C \wedge OF.\mathit{target} = A \wedge C \lhd U) \end{aligned} \quad (12)$$

The constraint expresses that for each activity that requires an artifact, there must be an activity that creates the artifact and dominates the activity using the artifact. If an artifact can be created by one of multiple activities, the constraint can be revised to

$$\begin{aligned} &\mathit{Artifact}(A) \wedge \mathit{InputIF}\,(IF) \wedge IF.\mathit{source} = A \wedge IF.\mathit{target} \\ &\quad = U \wedge \mathit{Activity}(U) \Rightarrow \exists Cs : (Cs \subseteq \bigcirc C : (\mathit{Activity}(C) \wedge \mathit{OutputIF}(IF') \\ &\qquad\qquad \wedge IF'.\mathit{source} = C \wedge IF'.\mathit{target} = A) \wedge Cs \lhd U) \end{aligned} \quad (13)$$

The constraint states that for any activity U using an artifact A there must be a set of activities Cs that supply the artifact and that together dominate its use. Therefore, no execution may exist where U is reached without A being created first. However, it may not be known precisely which activity will create A when the process is executed.

3.4. Solution search

Configuration systems in some domains can operate largely without human intervention; however, fully automatic process configuration is difficult to achieve. Although constraints on process structure and some domain-specific aspects can be exploited, many possible configurations remain for large processes. Albeit additional formalization of process entities would eliminate some candidates, the effort required to build the detailed specifications of activities precludes this approach in practice. For example, EPC models specify possible control and data flow between process elements, but do not usually include specifications about detailed behavior of functions or structure and meaning of artifacts. Although more formal models may exclude invalid process candidates that would be admitted by a simpler model, formalizing the missing knowledge is time consuming and often requires considerable expertise in formal languages. For these reasons, process configuration as presented in this article abandons the goal of obtaining all optimal configurations and adopts semi-interactive heuristic search procedures to compute solutions.

The overall search for processes proceeds as outlined in Figure 4: starting with a copy of the reference process as initial configuration, the requirements and constraints are specified by the user. In addition, manual changes to the initial process can be made in an *ad hoc* manner (e.g., through a graphical editor). When changes are made, the configurator evaluates the generative constraints on the candidate process and highlights violated constraints and their associated process entities. Subsequently, heuristic search is applied to resolve constraint violations. The resulting candidate process is presented to the user for validation, who may request further changes and recommence the search. This process repeats until a valid process that is consistent with all constraints is obtained.

For clarify of presentation we omit much of the specific technical implementation details and optimizations and focus on the overall conceptual mechanism. For example, change operations can be applied incrementally by maintaining the differences to a base model rather than copying the entire model each time the configuration is modified.

The initial candidate process is obtained from a copy of the generic reference process and its constraints, amended with the constraints describing project-specific requirements. For example, the process fragment in Figure 5 can be obtained from the reference process. Assuming the process will be tailored to a pure software development project, a generative constraint can be added that excludes all development activities not related to software. Resource and scheduling constraints can be added to direct the assignment of resources to activities. Furthermore, facts specifying the available personnel and their

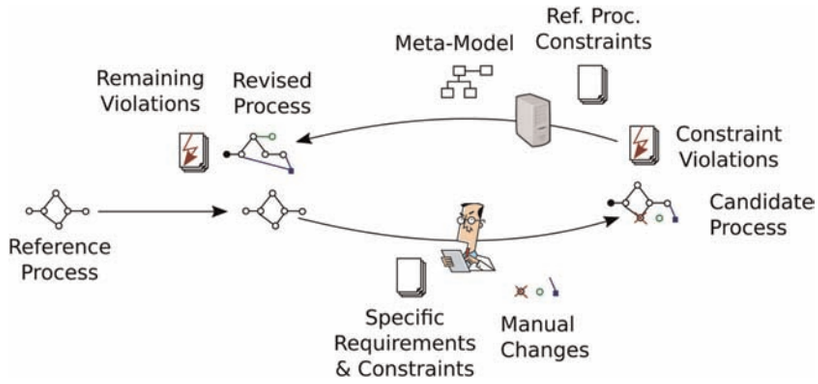

Fig. 4. The interactive process configuration workflow. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

roles can be added to the process. Manual operations can be applied to create, modify, and delete process entities. Changes are recorded and constraints may be added to ensure that the subsequent heuristic search respects the manual modifications.

Once the initial configuration and constraints have been established, automated search procedures are applied to resolve constraint violations. Because exhaustive search is prohibitively expensive, heuristics are applied to find good solutions. When applying the framework to the Siemens reference process, the quality of a configuration is indicated by the inverse of the number of constraint violations and the number of process entities. Other organization-specific measures, such as makespan or robustness with respect to delays or resource availability, can also be applied to guide the search. In our case study, minimizing the number of constraint violations while minimizing the process size as a secondary measure was sufficient to obtain "satisfying" configurations that improved upon the best processes that had been created manually. Optimizing makespan and other process measures was not a primary concern in our study.

Change operators are applied to the process in order to resolve constraint violations. Our framework incorporated the basic operations to add and remove process elements in a path, add and remove flows between elements, duplicate subprocesses, and assign resources and artifacts to elements. These operations closely align with the operations that had been carried out manually by project managers. The selection of operators is guided by heuristics and restricted by constraints reflecting the modifications made by project managers. Numeric and other constraints are solved by standard CSP techniques.

For semi-interactive configuration, heuristics that apply only minimal alterations to the existing configuration are preferred to extensive but "optimal" changes, because large modifications tend to be difficult to trace by users. Currently, the framework adopts the following generic principles to direct search; organization-specific heuristics can be implemented if desired. Heuristics are listed in order of preference.

1. *Promoted pruning:* The predominant operation in process tailoring is the removal of unnecessary entities from the generic reference process. The removal of one entity may propagate throughout the process and yield smaller processes with fewer (violated) constraints. For this reason, deletion of entities is considered the preferred resolution for constraint violations, unless the entity was introduced manually by the user.
2. *Minimum change:* Dually, the creation of entities is only considered if no other resolution yields a valid configuration. This strategy, in conjunction with constraints on the number of created entities, favors small processes and prohibits infinite regress of the search procedure in subprocesses where no consistent solution exists (Mayer et al., 2009).
3. *Minimal violation:* If multiple repair candidates apply, select one with the least number of remaining violated constraints.
4. *Locality of changes:* A large number of possible alternatives may exist for routing data and control flow in a partially specified process. Although most alternatives may be admissible given the generative constraints, flows between distant entities make it hard to comprehend processes. Furthermore, artifacts tend to be manipulated in closely related activities. To account for this observation, flows between entities with small distance are preferred. Distance is measured as the minimum number of control flow steps between two entities.
5. *Deferred synchronization:* Frequent use of synchronization elements yields processes with little concurrent execution. Because the absence of parallel activities may impact on future resource assignment, the introduction of synchronization elements is delayed as long as possible (but not beyond the scope of the current subprocess). This heuristics favors parallel branches but respects locality of control flow.

Using these heuristics, the neighborhood of the candidate configuration is explored in order to find a revised, valid process. Because the heuristics may not be able to resolve all constraint violations, further user interaction may be necessary.

If the candidate configuration obtained from the heuristic configuration step satisfies all constraints, the adaptation process terminates. Otherwise, the user is given the option to re-

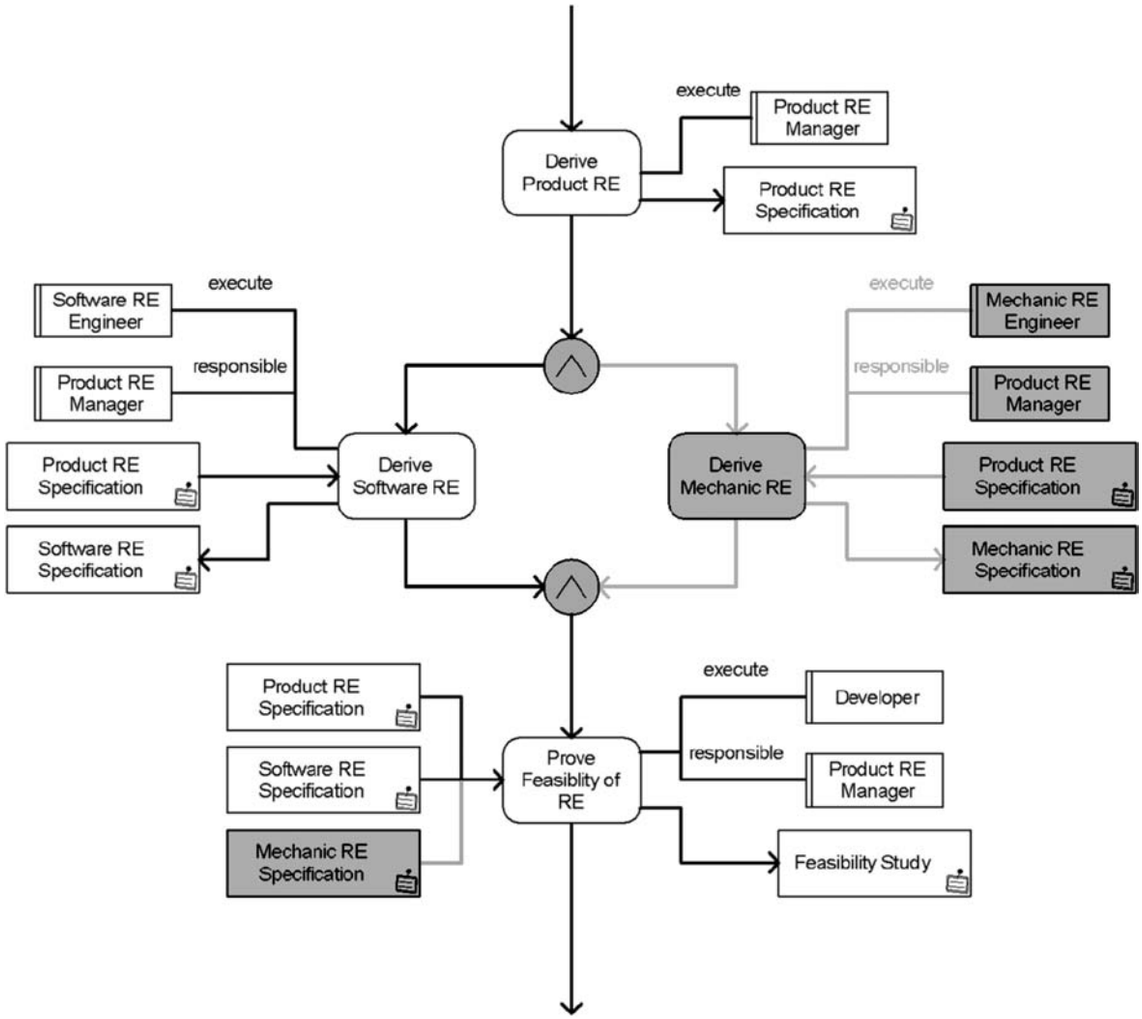

Fig. 5. An example process fragment. Activities and resource assignments highlighted in gray are deleted when the process is tailored to a pure software development project.

solve some of the remaining inconsistencies manually and repeat the heuristic search.

From an organizational point of view, custom adaptation of reference processes may also induce a maintenance problem when changes in the reference process must be applied to a large number of variants. Although reference process maintenance is beyond the scope of this article, techniques from version control systems and configuration can also be applied to resolve this problem. Graph-based differencing algorithms, such as the approach used by Zeller (2002), can be applied to compare different versions of the reference process and merge differences into the modified processes. Once changes have been applied, constraints can be verified and configuration recommenced if necessary. In this approach, the search for valid configuration that repair violated constraints corresponds the conflict resolution step in manual version management. Once tailored processes have been revised, running instances of previous process versions may need to be adapted to conform to the new model (Weber et al., 2008).

Figure 5 shows a generic reference process fragment that includes activities related to both software and hardware development. For a software project, the hardware development-related aspects are irrelevant and should be removed. Although deleting the relevant entities is trivial in this example, it can be a tedious task in a large process. The constraint-based approach allows the project manager to exclude all hardware-related development activities by asserting a constraint like Eq. (3). The adaptation can then proceed largely automatically, which will result in the removal of the shaded entities in Figure 5. Control flow connectors will be rerouted accordingly. Data flow constraints ensure that artifacts related to the development of mechanical parts will be removed even if they are only indirectly related to the deleted activities. For example, the use of *the Mechanic RE Specification* document in activity *Prove Feasibility of RE* will be automatically removed. Constraints prohibiting unnecessary process entities, such as the empty parallel control flows arising after deletion, trigger the removal of the redundant split and join functions.

3.4. Discussion

Constraint formalisms on top of object-oriented models are not new. The Object Constraint Language (OCL; Object Management Group, 2006), an extension of the Unified Modeling Language (UML), also follows this approach. Although arbitrary constraints over object graphs can be expressed in OCL, the constraint language cannot easily capture path constraints where the structure of the specific process configuration is not known. For example, expressing a dominance constraint requires quantification over all paths using the *forall* operator. However, the knowledge engineer would have to anticipate the expressions to reach the set of relevant branching points in the process for all possible process configurations. Alternative implementations where dominance is simulated using recursion, custom properties and OCL iterators are too restrictive or require complex expressions (Beckert & Trentelman, 2005). Related languages like XPath (Berglund et al., 2007) can express reachability and quantification but suffer from the same problems as the approach using OCL's *forall* operator.

Temporal constraint satisfaction problems have also been applied to reason about the order of execution of activities. Allen's (1983) temporal operators, for example, are sufficient to express partial orderings of tasks along a path, but quantification over execution paths is difficult in the standard constraint formalism if the process structure is not known at the time the constraints are written. Therefore, precedence relationships between artifacts on alternative branches, such as constraint [Eq. (13)], cannot easily be enforced in a generative way on the process metamodel.

Thomas and Fellmann (2007) combined EPCs with domain-specific models such that activities and related metadata in a process can be formalized. In this approach the entities in an EPC process model are annotated with concepts defined in an EPC metamodel similar to the one presented in this article. Entities in the EPC metamodel are linked with concepts and instances described in a domain-specific ontology. From the annotations the domain- or organization-specific meaning of each process element can be inferred in terms of the underlying ontologies, and ambiguities in the textual labels in the plain EPC model can be resolved. This approach is orthogonal to the configuration and repair aspects addressed in this article, but could be used to introduce additional problem-specific metadata and constraints in the process model.

Egyed et al. (2008) used language-specific constraints to highlight errors in software design models. Software design diagrams in the UML are converted into a CSP where constraints verify the consistency of multiple diagrams representing different aspects of the same software component. From violated constraints the conflicting diagram elements and potential repairs are inferred and ranked using heuristics. The approach shares the semi-interactive repair principle and constraints expressed in OCL, but relies on a metamodel specific to the UML.

Ly et al. (2008) validated concrete workflow executions using workflow models annotated with semantic constraints. In this work workflow models are enriched with dependency and exclusion constraints between activities. The constraints are used to verify workflows after *ad hoc* changes have been made and if running instances of the old workflow can be migrated without conflict to the modified model. This work focuses mainly on verification of manual changes and not on computing tailored work processes. Although domain-specific constraints are used to detect errors in executions, their constraint language lacks expressiveness and configuration and repair aspects are left aside.

Existing workflow adaptation methods predominantly focus on the avoidance of deadlocks and on the transition of running instances after workflows have been changed (Weber et al., 2008). In particular, workflow patterns and structural adaptation of process models and their instances in response to manual changes have been studied widely. However, domain-specific models, resources, and constraints are not usually considered. Methods based on pattern recognition and rewriting rely on a predetermined catalog of possible patterns and remedies, which is typically unavailable in *ad hoc* process configuration scenarios.

Specific executable processes can also be generated by automated planning approaches (Sirin et al., 2004). Although planners use powerful reasoning techniques, detailed models of actions and goals are required which prohibits the direct application of planning algorithms to the work process configuration problem addressed in this article. If available, our approach could also benefit from detailed semantic constraints to improve conflict detection and search heuristics.

4. SIEMENS AG PROCESS INSTANTIATION

Information system engineering and design processes at Siemens AG are nested within a general framework, the SPF (Fig. 6). Its purpose is to standardize the management and structure of processes. The main components of the SPF are the *Reference Process House (RPH)*, the *Roles* of individuals and committees in process management, and *Process Modeling Methods* (Schmelzer & Sesselmann, 2004). The RPH describes the hierarchical structure of Siemens processes. It includes all business processes that cover the relationship to customers as well as internal management processes, such as strategic planning and control, and support processes, for example, assigning of human resources to roles in a project.

The development processes are nested in several levels of abstraction. The RPH provides the structure of the top four levels (levels 0–3) with the RPH as detailed in Figure 6 corresponding to level 0. Processes at level 1 typically specify an abstract sequence of activities, such as *Plan*, *Product management*, *Define*, *Implement*, *Operate*, and *Phase out*. The structure and appearance of lower levels is determined by business units according to their individual needs. Detailed processes are specified as EPCs and FADs (see Fig. 1).

Overall, the product life cycle management part comprises nearly 3800 elements. Although a wide variety of tasks are

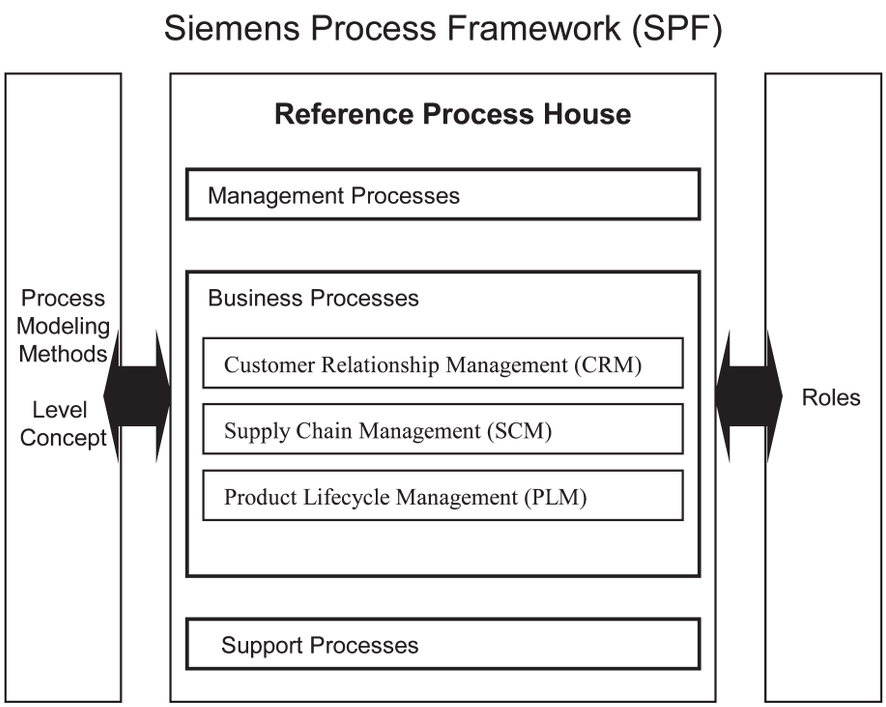

Fig. 6. The structure of the Siemens Process Framework.

covered, not all activities apply to all projects, and the generic process framework can usually be reduced significantly.

Adaptation is performed in two steps. First, processes relevant to a project are identified in the reference processes at levels 0 and 1, and irrelevant activities are removed. Second, detailed tailoring is performed in order to create extra activities required by a project, fill roles, and assign resources. This process is performed iteratively over the duration of the project, because the complexity of the reference process and unplanned changes prohibit project managers from forming a comprehensive plan at the start of a project.

To support adaptation, a process editor has been built that integrates with the overall process management systems at Siemens AG. The architecture of the system is shown in Figure 7. The editor manipulates the processes stored in the commercial ARIS system via a custom-built interface. The XML-based XPDL process definition language (WFMC, 2008) is used as data exchange format. The interface allows the project manager to edit and visualize the processes and resources associated with a project. Figure 8 shows snapshots of the user interface for manipulating processes and associated resources. This architecture ensures that the ARIS system remains the central authority for process information and existing project management tools need not be altered, a factor that we considered to be crucial for the adoption of any advanced process adaptation tool. Generative constraints were expressed in a language derived from the OCL (Object Management Group, 2006). This has the benefit that the metamodel and the constraints can be developed and versioned together in a UML tool.

We constructed the metamodel for the case study to fit the structure of the reference software development processes at Siemens AG, where the processes are stored, retrieved, and displayed using the EPC modeling language. After testing various smaller scenarios using the metamodel with interactive users, a major case study applied to the model to create a tailored variant of the reference process for a particular business unit for real world use. Manual adaptation had been tried and abandoned several times as even in teams with elaborate review processes, consistency of the adapted process could not be guaranteed.

The metamodel of the reference process was inspired by the EPC process modeling language, but includes additional entities and constraints that are specific to the development practices at Siemens AG. The model was developed in collaboration with the project managers and developers at Siemens AG. The process comprises 33 entity types and phases (composite activities modeled as subprocesses), milestones, and control flows. Figure 3 shows the structure of the main entities in the model. In addition, 139 generative constraints describe the valid composition of entities and flows into processes. Restrictions derived from the EPC language and also rules specific to the business unit's development process have been incorporated. The model was developed iteratively, intertwining constraint formulation and evaluation on selected subprocesses, to ensure that the constraints are specific enough to rule out invalid processes while tolerating the variation necessary for effective process customization. Overall, the resulting reference process includes about 3800 process elements and about 14,000 instantiations of generative constraints.

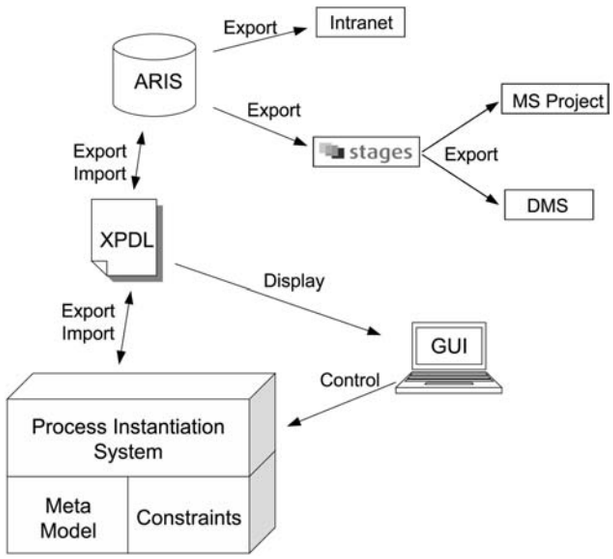

Fig. 7. The process management systems architecture at Siemens AG.

As a significant fraction of Siemens development processes involves embedded software in electronic and mechatronic systems, the process actually is of much wider scope than pure software development, which results in significant variation for different business units. However, the size and complexity of the process meant that even seemingly trivial adaptions could result in insurmountable effort. As a result, process management had so far been restricted to merely using the reference process as a rough guideline without adapting to business unit or project specific needs as originally envisioned.

Once the model had been built, the reference process could be tailored to suit selected development scenarios. For the initial full-scale application of the framework, the adaptation of the reference process for a software business unit was chosen. Experts at Siemens AG identified 108 activities and 41 work products that had to be removed. These changes triggered additional adaptations that were applied automatically by our tool. The resulting process included 1959 elements, down from 3790 before the adaptation (a reduction of 48%). The execution of this task completed in 221 s of CPU time on a standard PC (excluding the time required to import and export the process from ARIS). Manual creation of this particular process (using project teams and review meetings) had been tried several times and abandoned because of the impossibility of tracing dependencies of process elements and verify conformance with constraints.

Note that because of the declarative modeling approach, the framework and tool can also be used to validate any given process. Applied to the overall 3800 element Siemens reference process, the tool identified 177 previously unknown constraint violations.

The application to the real world Siemens problem has demonstrated that automated process configuration has the potential to achieve tremendous improvements in process quality while reducing the cost of process maintenance. We anticipate that automated process configuration will yield significant savings that is due to a reduction of the time required to tailor processes and the absence of activities that do not contribute to the project's outcomes.

5. CONCLUSION AND FUTURE DIRECTIONS

The adaptation of large-scale generic processes to the requirements of specific projects is a challenging task where automated tool support has been desired. The process adaptation

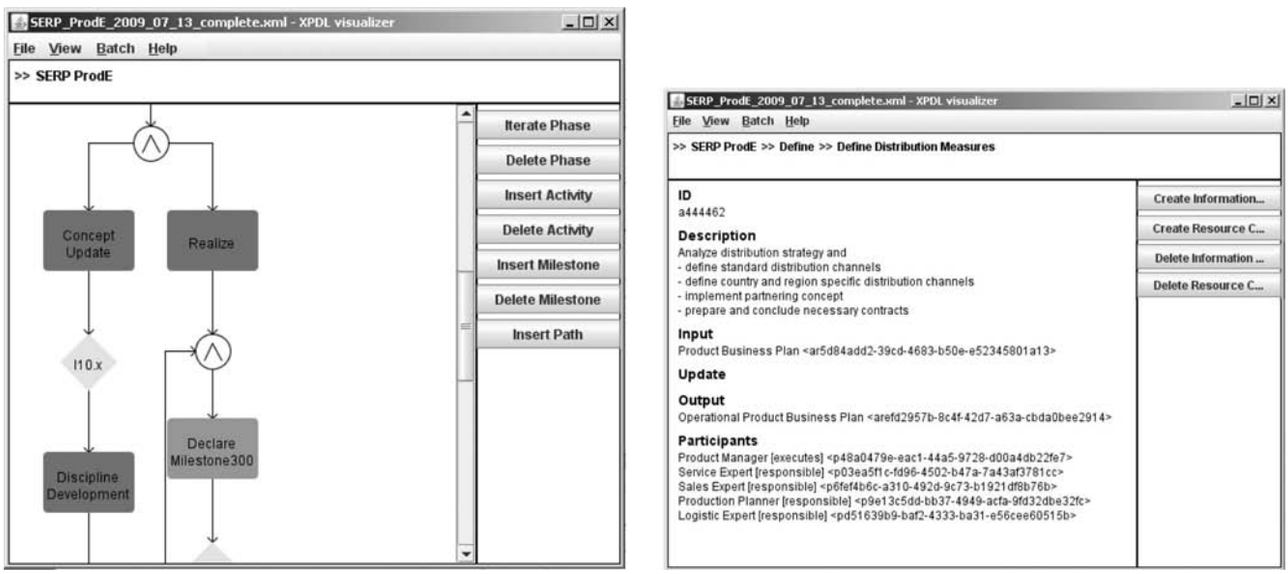

Fig. 8. The graphical user interface of the process adaptation tool. The left image shows the process adaptation interface, and the right image shows the interface to browse and manipulate activities and resources.

framework presented in this article extends well-known constraint-based configuration principles to the process domain. The approach generalizes traditional component-oriented configuration techniques into a uniform formal framework for the configuration of entire processes. The framework comprises an extensible metamodel of organization-specific process entities and relations and a declarative constraint language that incorporates constraints on process structure and execution paths. The metamodel in conjunction with the declarative constraint language allows organizations to minimize the effort required to adopt the approach by gradually amending the level of detail that is reflected formally. Generic heuristics are applied to select changes that resolve inconsistencies such that the overall process structure is maintained. Methods or algorithms that implement the approach can be generic or chosen for the specific context.

For the Siemens case, a semi-interactive tool was built that fit within the Siemens process environment, allowing project managers to tailor and validate work processes with respect to a set of project-specific constraints and a given reference process. The approach has helped to identify a large number of errors in development processes that had been manually tailored at Siemens AG. Furthermore, substantially smaller processes have been obtained from the monolithic reference process with very little effort. The efforts involved in creating the initial metamodel, constraints, and tool implementation have been offset by a significant reduction in time required to tailor and validate specific processes.

Given that considerable improvements in the management of development processes at Siemens AG have been demonstrated, we intend to conduct further studies and evaluate our configuration approach on additional processes sourced from different organizations and work process domains. Our primary objective will be to refine the heuristic adaptation mechanisms for common work process patterns and to further improve the quality of tailored processes for common process patterns and change operations.

The investigations in this project have focused largely on structural and flow aspects within a process. Further extensions can be incorporated that reflect additional properties of the domain, such as an explicit representation of time and detailed resource allocation and scheduling. Expanding the declarative representation to incorporate approaches that strive to capture semantics of larger building blocks of the design also opens further research directions. Developing declarative means to specify context-dependent search strategies in order to synthesize efficient implementations would also be desired to yield even more efficient yet flexible tools.

ACKNOWLEDGMENTS

This research was supported by the Australian Research Council (Grant DP0988961) and by Siemens A.G.

REFERENCES

Albert, P., Henocque, L., & Kleiner, M. (2005). Configuration-based workflow composition. *Proc. IEEE Int. Conf. Web Services (ICWS)*, pp. 285–292, Orlando, FL.

Allen, J.F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM 26(11)*, 823–843.

Armbrust, O., Katahira, M., Miyamoto, Y., Münch, J., Nakao, H., & Ocampo, A. (2008). Scoping software process models—initial concepts and experience from defining space standards. *Proc. Int. Conf. Software Process (ICSP'08)*, LNCS, Vol. 5007, pp. 160–172. Berlin: Springer.

Asikainen, T., & Männistö, T. (2009). A metamodelling approach to configuration knowledge representation. *IJCAI'09 Workshop on Configuration*, pp. 9–16, Pasadena, CA.

Bajec, M., Vavpotič, D., & Krisper, M. (2007). Practice-driven approach for creating project- specific software development methods. *Information & Software Technology 49(4)*, 345–365.

Beckert, B., & Trentelman, K. (2005). Second-order principles in specification languages for object-oriented programs. *Proc. LPAR*, LNCS, Vol. 3835, pp. 154–168. Berlin: Springer.

Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., and Siméon, J. (2007). *XML Path Language (XPath) 2.0*. World Wide Web Consortium, Recommendation REC-xpath20-20070123.

Brinkkemper, S. (1996). Method engineering: engineering of information systems development methods and tools. *Information & Software Technology 38(4)*, 275–280.

Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM 50(5)*, 752–794.

Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., & Zadeck, F.K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13(4)*, 451–490.

Dausch, M., & Hsu, C. (2006). Engineering service products: the case of mass-customising service agreements for heavy equipment industry. *International Journal of Services Technology and Management 7(1)*, 32–51.

Egyed, A., Letier, E., & Finkelstein, A. (2008). Generating and evaluating choices for fixing inconsistencies in UML design models. *Proc. IEEE Conf. Automated Software Engineering (ASE'08)*, pp. 99–108, L'Aquila, Italy.

Felfernig, A., Friedrich, G., & Jannach, D. (2001). Conceptual modeling for configuration of mass-customizable products. *Artificial Intelligence in Engineering 15(2)*, 165–176.

Fitzgerald, B., Russo, N., & O'Kane, T. (2003). Software development method tailoring at Motorola. *Communications of the ACM 46(4)*, 65–70.

Ginsberg, M., & Quinn, L. (1995). *Process tailoring and the software capability maturity model*. Technical report. Pittsburgh, PA: Software Engineering Institute (SEI).

Heiskala, M., Tiihonen, J., & Soininen, T. (2005). A conceptual model for configurable services. *Proc. IJCAI'05 Workshop on Configuration*, Edinburgh, Scotland.

IDS Scheer. (2006). *ARIS design platform*. Accessed September 8, 2010, at http://www.ids-scheer.com/us/en/ARIS/ARIS_Platform/ARIS_Design_Platform/32390.html

Jensen, K. (1997). *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Vol. 1, 2nd ed. Berlin: Springer–Verlag.

Keller, G., Nüttgens, M., & Scheer, A. (1992). *Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK)*. Technical Report 89, Universität des Saarlandes.

Killisperger, P. (2010). *Instantiation of information systems development processes*. PhD Thesis. University of South Australia, School of Computer and Information Science.

Killisperger, P., Stumptner, M., Peters, G., & Stückl, T. (2008). Challenges in software design in large corporations—a case study at Siemens AG. *Proc. Int. Conf. Enterprise Information Systems (ICEIS) (3-2)*, pp. 123–128, Barcelona, Spain.

Kindler, E. (2006). On the semantics of EPCs: resolving the vicious circle. *Data & Knowledge Engineering 56(1)*, 23–40.

Latvala, T., Biere, A., Heljanko, K., & Junttila, T.A. (2005). Simple is better: efficient bounded model checking for past LTL. *Proc. VMCAI*, pp. 380–395, LNCS, Vol. 3385. Berlin: Springer.

List, B., & Korherr, B. (2005). A UML 2 profile for business process modeling. In *Proc. ER (Workshops)*, LNCS, Vol. 3770, pp. 85–96. Berlin: Springer.

Ly, L.T., Rinderle, S., & Dadam, P. (2008). Integration and verification of semantic constraints in adaptive process management systems. *Data & Knowledge Engineering 64(1)*, 3–23.
Magro, D. (2010). F: conceptual language-based configuration. *AI Communications 23(1)*, 1–46.
Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing 12(4)*, 383–397.
Marcus, S., & McDermott, J. (1989). SALT: a knowledge-acquisition language for propose-and-revise systems. *Artificial Intelligence 39(1)*, 1–37.
Mayer, W., Thiagarajan, R., & Stumptner, M. (2009). Service composition as generative constraint satisfaction. *Proc. IEEE Int. Conf. Web Services (ICWS)*, pp. 888–895, Los Angeles.
Mendling, J. (2009). Empirical studies in process model verification. *T. Petri Nets and Other Models of Concurrency 2*, 208–224.
Milner, R. (1990). Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics (B)*, pp. 1201–1242. Cambridge, MA: MIT Press.
Mittal, S. (1990). *Reasoning about resource constraints in configuration tasks*. Technical report, Xerox PARC.
Muchnick, S.S. (1997). *Advanced Compiler Design and Implementation*. Pasadena, CA: Morgan Kaufmann.
Niknafs, A., & Ramsin, R. (2008). Computer-aided method engineering: an analysis of existing environments. *Proc. 20th Int. Conf. Advanced Information Systems Engineering (CAiSE'08)*, pp. 525–540. Berlin: Springer.
Object Management Group. (2006). Object constraint language: OCL specification v2.0 [Computer software]. http://www.omg.org/spec/OCL/2.0/PDF
Rosemann, M., & van der Aalst, W.M.P. (2007). A configurable reference modelling language. *Information Systems 32(1)*, 1–23.
Rossi, F., Beek, P.v., & Walsh, T. (2006). *Handbook of Constraint Programming*. New York: Elsevier Science.
Scheer, A.-W. (2000). *ARIS—Business Process Modeling*, 3rd ed. New York: Springer–Verlag.
Schmelzer, H., & Sesselmann, W. (2004). *Geschäftsprozessmanagement in der Praxis: Produktivität steigern—Wert erhöhen—Kunden zufriedenstellen*, 4th ed. Munich: Hanser Verlag.
Sirin, E., Parsia, B., Wu, D., Hendler, J.A., & Nau, D.S. (2004). HTN planning for Web Service composition using SHOP2. *Journal of Web Semantics 1(4)*, 377–396.
Soininen, T., Tiihonen, J., Männistö, T., & Sulonen, R. (1998). Towards a general ontology of configuration. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing 12(4)*, 357–372.
Stumptner, M., Friedrich, G., & Haselböck, A. (1998). Generative constraint-based configuration of large technical systems. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing 12(4)*, 307–320.
Thomas, O., & Fellmann, M. (2007). Semantic EPC: enhancing process modeling using ontology languages. *Proc. SBPM, CEUR Workshop,* Vol. 251. Accessed at http://www.CEUR-WS.org
Van der Aalst, W.M.P. (1999). Formalization and verification of event-driven process chains. *Information and Software Technology 41(10)*, 639–650.
Van der Aalst, W.M.P. (2000). Workflow verification: finding control-flow errors using petri-net-based techniques. In *Business Process Management, Models, Techniques, and Empirical Studies* (van der Aalst, W.M.P., Desel, J., & Overweis, A., Eds.), pp. 161–183. Berlin: Springer–Verlag.
Van der Aalst, W.M.P., & van Hee, K.M. (2004). *Workflow Management—Models, Methods, and Systems*. Cambridge, MA: MIT Press.
Weber, B., Reichert, M., & Rinderle-Ma, S. (2008). Change patterns and change support features enhancing flexibility in process-aware information systems. *Data & Knowledge Engineering 66(3)*, 438–466.
WFMC. (2008). *WFMC-TC-1025-Oct-10-08A (final XPDL 2.1 specification*). Technical report, WFMC. Accessed April 28, 2009, at http://www.wfmc.org
Zeller, A. (2002). Isolating cause-effect chains from computer programs. *Proc. Foundations of Software Engineering (SIGSOFT FSE)*, pp. 1–10, Charleston, SC.

Wolfgang Mayer is a Lecturer at the University of South Australia. He received his PhD from the University of South Australia and his MS in computer science from the Vienna University of Technology. His research interests include analysis, composition, and diagnosis of process models and software systems. Dr. Mayer's work focuses on model-based reasoning and knowledge representation techniques with applications to fault localization, automated configuration, and data- and workflow integration.

Markus Stumptner is a Professor of computer science at the University of South Australia, where he directs the Advanced Computing Research Centre. He received MS and PhD degrees in computer science from the Vienna University of Technology. Dr. Stumptner's research interests include object-oriented modeling, knowledge representation, and model-based reasoning in areas such as configuration and diagnosis.

Peter Killisperger is a Researcher at Siemens Corporate Research and Technologies in Munich. He received a PhD in information technology from the University of South Australia, an MS in distributed and multimedia information systems from the Heriot–Watt University in Edinburgh, Scotland, and a diploma degree in business informatics from the University of Applied Sciences in Augsburg, Germany. Dr. Killisperger's work focuses on improving information systems development processes.

Georg Grossmann is a Research Fellow at the University of South Australia. His PhD thesis on behavior-based integration of object-oriented information systems was awarded the Ian Davey Research Thesis Prize for the most outstanding research thesis at the University of South Australia. He also received an MS in economics and computer science (jointly from University of Vienna and Vienna University of Technology). Dr. Grossmann's current research interests include business process integration, behavior-based integration of Web services, ontology-driven data integration, and distributed event-based systems.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 163–174.

doi:10.1017/S0890060410000600

Reasoning about conditional constraint specification problems and feature models

RAPHAEL FINKEL[1] AND BARRY O'SULLIVAN[2]

[1]Department of Computer Science, University of Kentucky, Lexington, Kentucky, USA
[2]Cork Constraint Computation Centre, University College Cork, Cork, Ireland

(RECEIVED March 19, 2010; ACCEPTED October 29, 2010)

Abstract

Product configuration is a major industrial application domain for constraint satisfaction techniques. Conditional constraint satisfaction problems (CCSPs) and feature models (FMs) have been developed to represent configuration problems in a natural way. CCSPs are like constraint satisfaction problems (CSPs), but they also include potential variables, which might or might not exist in any given solution, as well as classical variables, which are required to take a value in every solution. CCSPs model, for example, options on a car, for which the style of sunroof (a variable) only makes sense if the car has a sunroof at all. FMs are directed acyclic graphs of features with constraints on edges. FMs model, for example, cell phone features, where utility functions are required, but the particular utility function "games" is optional, but requires Java support. We show that existing techniques from formal methods and answer set programming can be used to naturally model CCSPs and FMs. We demonstrate configurators in both approaches. An advantage of these approaches is that the model builder does not have to reformulate the CCSP or FM into a classic CSP, converting potential variables into classical variables by adding a "does not exist" value and modifying the problem constraints. Our configurators automatically reason about the model itself, enumerating all solutions and discovering several kinds of model flaws.

Keywords: Alloy; Answer-Set Programming; Configuration; Constraint Satisfaction; Flaw Detection

1. INTRODUCTION

Product configuration has provided constraint programming with one of its most successful application domains (Sabin & Weigel, 1998; Junker, 2006). Model-based, particularly constraint-based, approaches to configuration are the most successful in practice (http://www.gartner.com), because constraint-based product configurators are specified in a highly declarative formalism.

Configuration presents several modeling and reasoning challenges. First, it is challenging to maintain consistent integration between product catalogs and constraint-based configurator models. Second, constraint-based approaches need to be able to handle taxonomic inheritance among components and subsystems. Third, the space of possible configurable products is often unbounded but might be subject to resource restrictions. Fourth, users have preferences, and full customization must be possible.

Reprint requests to: Raphael Finkel, Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA. E-mail: raphael@cs.uky.edu

Most configurator engines restrict the configuration process to some degree. In particular, a configurator will typically configure systems before subsystems. Also, isomorphic configurations provide challenges for the configurator; isomorphic configurations can be regarded as being structurally symmetric. For this reason, many configurators represent the product being configured as a set of systems rather than associating each system with a variable, which can introduce unnecessary symmetries into the configuration space.

Although constraint satisfaction techniques have supported configuration for many years, they have required extensions to the basic constraint satisfaction problem. For example, *composite* constraint satisfaction (Sabin & Freuder, 1996) has been introduced to handle hierarchical system configuration. Mittal and Falkenhainer (1990) introduced *dynamic* constraint satisfaction to cover problems in which the existence of features depended on the existence or values of other features; this scheme was subsequently given a formal logical semantics (Bowen & Bahler, 1991). This work was subsequently extended by other researchers (Mailharro, 1998; Stumptner et al., 1998). Dynamic constraint satisfaction has more recently been referred to as *conditional* constraint satisfaction (Gelle & Faltings, 2003) to

distinguish dynamism due to conditional relevance of some variables and constraints from dynamism due to uncertainty and environmental change.

Many authors (including Mittal and Falkenhainer) reformulate conditional constraint satisfaction problems (CCSPs) into classic CSPs by introducing redundant domain values and augmenting the problem constraints so that some problem variables take a "not defined" value (Sabin & Gelle, 2006). Although feature models (FMs) appear quite different from CCSPs, they can also be mapped to CSPs (Benavides et al., 2005) and other data structures. These reformulations are problematic. First, they seem unnatural as a modeling technique, especially for large real-world configuration problems. Second, they become impractical and difficult to maintain, especially when the configuration space is extremely large or unbounded.

Our motivation arises from sophisticated tools that the formal methods community has developed for modelling and reasoning about complex engineered artifacts that can be regarded as configuration problems (Hinchey et al., 2008). Our objective is to study the utility of formal methods for modeling and reasoning about configuration models. The two main contributions of this paper are the following:

1. Using well-known examples, we show how to model constraint-based configuration problems naturally and concisely in the formal methods package Alloy (Jackson, 2002), which is usually used for modeling software systems, and in the answer-set programming (ASP) language *lparse* (http://www.tcs.hut.fi/Software/smodels/)
2. In addition to providing a natural modeling paradigm, these approaches are capable of providing reasoning capabilities that are very appropriate for configuration, in particular, *verifying the specification* of the configuration problem to ensure that specific flaws are absent, a problem identified and studied in earlier work (Sabin & Freuder, 1998). We argue that formalisms such as Alloy and *lparse* provide modeling tools that can be easily used by nonexperts to model and reason about configuration problems directly and naturally.

In Section 2 we informally present conditional constraint satisfaction, motivated by a well-known configuration problem, which we use as a running example. We also list some flaws that can occur in the specification of conditional configuration problems. We present both a formal methods approach (Section 3) and an answer set programming approach (Section 4) to reasoning about CCSPs. In Section 5 we show how we can easily identify flaws in CCSPs and demonstrate that Mittal and Falkenhainer's benchmark problem exhibits such flaws. We briefly show how ASP can easily find solutions involving the minimum or maximum number of options in Section 6. We turn our attention to FMs in Section 7, showing how an ASP approach can reason about these configuration models. In Section 8 we present XML representations for both CCSPs and FMs and report on our software to apply ASP methods to the data stored in such XML representations. Finally, in Section 9 we draw several conclusions and summarize our plans for future study.

2. CONDITIONAL CONSTRAINT SATISFACTION

Mittal and Falkenhainer (1990) introduced CCSPs. A CCSP differs from a classical CSP in that some variables are marked as **potential**, which means that they need not take a value in all solutions. CCSPs allow activity constraints that deal with the existence of **potential variables**, including the following:

- **require variable (RV)**, which stipulates that under certain value assignments to other variables, a potential variable must exist;
- **require not variable (RN)**, which stipulates that under certain value assignments to other variables, a potential variable must not exist;
- **always require variable (ARV)**, which stipulates that the existence of some other variable implies the existence of a potential variable; and
- **always require not variable (ARN)**, which stipulates that the existence of some other variable precludes the existence of a potential variable.

Mittal and Falkenhainer demonstrate these concepts by presenting two examples. In the first, the task is to generate valid configurations of options for a car. Because we plan to encode this example for our own purposes, we present it essentially as Mittal and Falkenhainer do in Figure 1. This small model captures, among other constraints, that luxury vehicles must have some sort of sunroof (constraint 1), that any sort of sunroof requires an option for glass (constraint 6), that an `sr1` sunroof has no opener (constraint 10), and that a luxury car may not have an `ac1` air conditioner (constraint 14).

Given such a CCSP, one can pose several queries:

1. Find/count/enumerate solutions to the CCSP. To **find** is to compute a single solution; to **count** is to discover the number of unique solutions, and to **enumerate** is to list all those solutions.
2. Enumerate all variable flaws in the CCSP. A **variable flaw** is a potential variable that is present in all solutions, so it is really a classical, not a potential, variable, or a potential variable that is never present in any solution.
3. Enumerate all value flaws in the CCSP. A **value flaw** is a value for a variable (actual or potential) that is never achieved by any solution.
4. Find/count/enumerate minimum/maximum solutions to the CCSP. A **minimum (maximum) solution** is one with the fewest (most) potential variables.
5. Find/count/enumerate minimal/maximal solutions to the CCSP. A **minimal (maximal) solution** is one in which removing (adding) any potential variable leads to a nonsolution.

classical variables

```
Package          {luxury, deluxe, standard}
Frame            {convertible, sedan, hatchback}
Engine           {small, med, large}
```

potential variables

```
Battery          {small, med, large}
Sunroof          {sr1, sr2}
AirConditioner   {ac1, ac2}
Glass            {tinted, notTinted}
Opener           {auto, manual}
```

activity constraints

1. Package = luxury $\overset{RV}{\Rightarrow}$ Sunroof
2. Package = luxury $\overset{RV}{\Rightarrow}$ AirConditioner
3. Package = deluxe $\overset{RV}{\Rightarrow}$ Sunroof
4. Sunroof = sr2 $\overset{RV}{\Rightarrow}$ Opener
5. Sunroof = sr1 $\overset{RV}{\Rightarrow}$ AirConditioner
6. Sunroof $\overset{ARV}{\Rightarrow}$ Glass
7. Engine $\overset{ARV}{\Rightarrow}$ Battery
8. Opener $\overset{ARV}{\Rightarrow}$ Sunroof
9. Glass $\overset{ARV}{\Rightarrow}$ Sunroof
10. Sunroof = sr1 $\overset{RN}{\Rightarrow}$ Opener
11. Frame = convertible $\overset{RN}{\Rightarrow}$ Sunroof
12. Battery = small & Engine = small $\overset{RN}{\Rightarrow}$ AirConditioner

classical constraints

13. Package = standard $\Rightarrow$ AirConditioner $\neq$ ac2
14. Package = luxury $\Rightarrow$ AirConditioner $\neq$ ac1
15. Package = standard $\Rightarrow$ Frame $\neq$ convertible
16. Opener = auto & AirConditioner = ac1 $\Rightarrow$ Battery = med
17. Opener = auto & AirConditioner = ac2 $\Rightarrow$ Battery = large
18. Sunroof = sr1 & AirConditioner = ac2 $\Rightarrow$ Glass $\neq$ tinted

Fig. 1. A car-configuration problem based on Mittal and Falkenhainer (1990).

3. REASONING ABOUT CCSPs IN ALLOY

Alloy Analyzer 4.0 is a language originally intended to model design of data structures. Jackson presents the formal semantics of Alloy in a comprehensive manner (Jackson, 2002). Alloy has been widely used for modeling large complex engineering systems (http://alloy.mit.edu/community/models). It provides a way to specify types and constrain their instances. It can convert those types and constraints into SAT problems that it then solves, displaying the solutions via a graphical interface. If it fails to find a solution, the specification is most likely inconsistent, although the solver might not have searched a large enough population of instances; the specification indicates how many instances of each type to generate for testing purposes. In this sense, Alloy is not a complete solver.

If the graphical representation of the solution seems erroneous to the user, new constraints that the user adds to the specification can prevent the erroneous interpretation.

We find that Alloy is well suited to represent CCSPs. Figure 2 presents our Alloy representation of part of the car example from Figure 1. In Alloy, a **`sig`** introduces a type. These types, something like classes in object-oriented programming languages, may be defined to contain members. To model the car-configuration problem, we introduce a **`sig`** called `Car` with a member for each variable. During configuration we define instances of `Car`.

Each car has required attributes, including a package. The fact that every instance of a car comprises one of each of these attributes is specified with the keyword **`one`**. These attributes represent the *classical* CSP variables of the problem.

A car has additional optional attributes, including a battery and an air conditioner. These attributes correspond to the *potential* variables of the problem. We specify them with the keyword **`lone`** to state that each instance of a car may have *at most one* instance of each of these attributes. Alloy can now generate one instance of every classical variable and

```
sig Car {
 package:one Package,
 battery:lone Battery,
 ac:lone AirConditioner,
 glass:lone Glass,
 sunroof:lone Sunroof
}
abstract sig Package {}
lone sig Luxury, Deluxe, Standard extends Package {}
abstract sig Battery {}
lone sig BSmall, BMedium, BLarge extends Battery {}
abstract sig AirConditioner {}
lone sig AC1, AC2 extends AirConditioner {}
abstract sig Opener {}
lone sig Auto, Manual extends Opener {}
fact {
 all c:Car | c.package in Luxury => one c.ac // RV 2
 all c:Car | one c.sunroof => one c.glass // ARV 6
 all c:Car | one c.sunroof and c.sunroof in SR1 => no c.opener // RN 10
 all c:Car | c.package in Standard => no (c.ac & AC2) // Classical 13
 all c:Car | c.package in Luxury => no (c.ac & AC1) // Classical 14
}
run {} for 1
```

Fig. 2. An Alloy model encoding the car-configuration problem (excerpt).

an optional instance of every potential variable. The particular instance that Alloy generates captures the CSP idea of a variable's value.

We introduce each CCSP variable with an **abstract sig**, introducing a type (such as `Package`) that has no direct instances. Then we introduce subtypes (such as `Luxury`). These subtypes may have at most one instance each.

Constraints are represented inside a **fact**. RV and ARV constraints differ in the form of their left-hand side, referring either to values (like `Luxury` in constraint 2) or variables (like `sunroof` in constraint 6). RN and ARN constraints (like constraints 13 and 14) differ from RV and ARV constraints only in that they have **no** on the right-hand side.

This representation would lead to a fifth and sixth sort of constraint not contemplated by Mittal zFalkenhainer, in which the nonexistence of a potential variable leads to the existence or nonexistence of another potential variable. We would model such constraints in Alloy by facts with **no** on the left-hand side and either **one** or **no** on the right-hand side.

The Alloy program is executable. It generates the solution in Figure 3, among others. Unfortunately, Alloy gives us no way to directly count or enumerate the solutions, short of interacting multiple times with the Alloy Analyzer to request the next solution.

4. REASONING ABOUT CCSPs IN ASP

To represent CCSPs using ASP, we use the syntax that *lparse* recognizes and converts to a form acceptable to solvers such as *smodels* (Niemela & Simons 1997), *clasp* (Gebser et al., 2007), and *Cmodels* (which converts *lparse* into SAT and invokes a SAT solver; Giunchiglia et al., 2004). ASP programs deal with **predicates**, which are either true or false. We introduce a predicate for each value of each actual and potential variable. For instance, the predicate `package(luxury)` represents the value `luxury` for the variable `package`. In any given model, this predicate is either true or false.

Many ASP solvers allow **cardinality-constrained predicate**s, in which the number of true predicates in a given list is bounded above, below, or both. We say

```
0 { battery(bsmall), battery(bmed),
    battery(blarge) } 1
```

to represent a cardinality-constrained predicate stating that at least 0 and at most 1 of the three predicates in the list is true. The car may have no battery at all, but if it has one, the battery must be one of small, medium, or large. *lparse* allows a shorthand for lists of predicates that share the same functor; we can equivalently write

```
0{ battery(bsmall; bmed; blarge) } 1
```

We use ASP **implications** to represent CCSP constraints:

```
1 { battery(bsmall; bmed; blarge) }
  :- package(luxury).
```

This implication says that if a car has the luxury package, it must have at least one of the battery sizes.

Finally, ASP programs have **failures**,[1] indicated by an empty left-hand side of an implication. The conjunction of

[1] The ASP community calls them *constraints*, but we avoid that term here because it conflicts with CSP terminology.

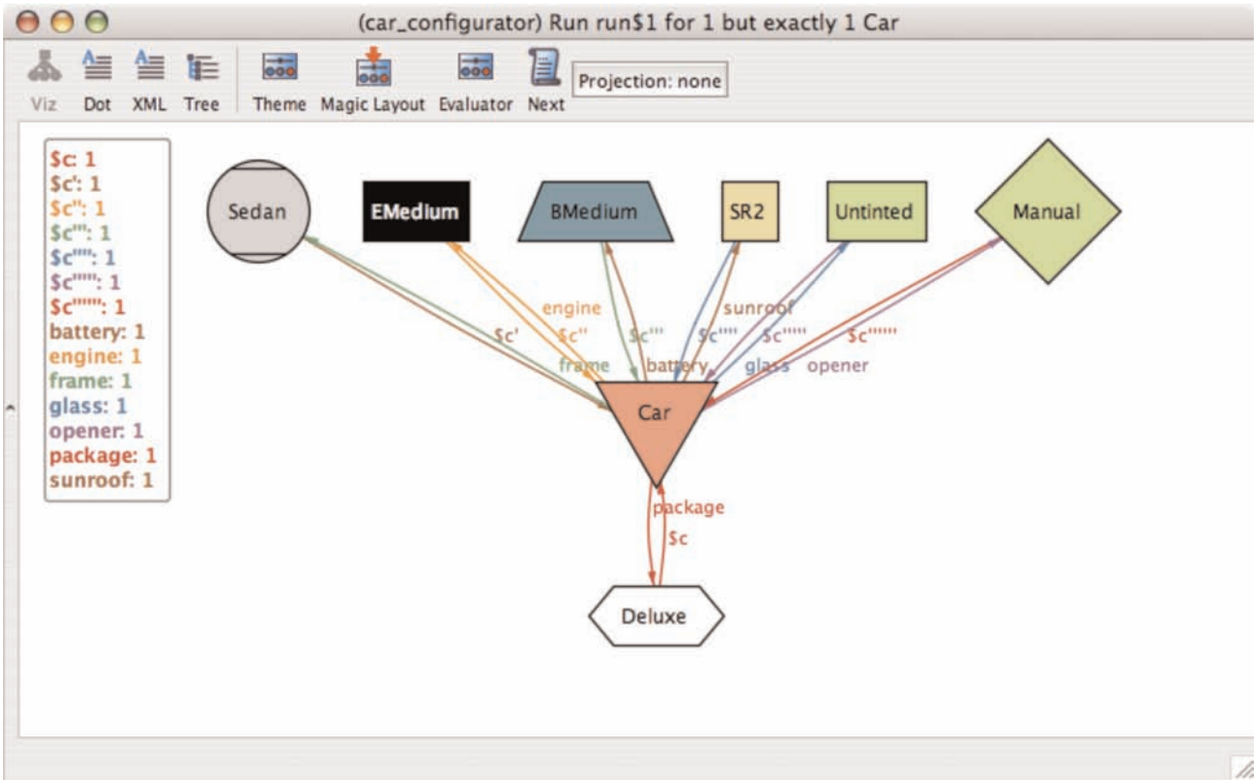

Fig. 3. A car configuration comprising a deluxe package, a sedan frame, a medium engine, a medium battery, sunroof `SR2` with untinted glass, and a manual opener. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

predicates (some of which may be negated) on the right-hand side must not be true in any satisfying model. For example,

```
:- 1 { opener(auto; manual)}, sunroof(sr1) .
```

specifies that no model may have an `sr1` sunroof and have either an automatic or a manual opener.

An excerpt of the cars specification in *lparse* syntax is presented in Figure 4. Although the *lparse* representation is not as elegant as the Alloy one, it is not difficult to read. Instead of **one**, we bound above and below by 1 (as in the rule for the classical variable `package`). Instead of **lone**, we bound below by 0 and above by 1 (as in the rule for the potential variable `battery`). Instead of **no**, we bound a failure below by 1, as in rule 10. We represent constraints that preclude particular values by failures (rule 13). We represent classical constraints that imply particular values by implications (rule 16).

ASP solvers, unlike Alloy, can easily enumerate all solutions. Each solution is an answer set, that is, a set of predicates that satisfies all the rules in the specification. A few of the 450 solutions to the car configuration specification are presented in Table 1. One can look through the list of solutions to search for variable and value flaws. However, one can also generate them automatically, as we show in the next section.

5. AUTOMATICALLY CHECKING FOR SPECIFICATION FLAWS

Specifications of CCSPs can contain a variety of flaws that can be difficult to detect manually (Sabin & Freuder, 1998).

```
1 {package(luxury; deluxe; standard)} 1. % classical variable
0 {battery(bsmall; bmed; blarge)} 1.     % potential variable
0 {airConditioner(ac1; ac2)}      1.     % potential variable
1 {airConditioner(ac1; ac2)} :- package(luxury). % RV #2
1 { glass(tinted; untinted)} :- 1 { sunroof(sr1; sr2))}. % ARV #6
:- 1 { opener(auto; manual)}, sunroof(sr1). % RN #10
:- package(standard), airConditioner(ac2). % classical #13
battery(bmedium) :- opener(auto), aircon(ac1). % classical #16
```

Fig. 4. The car-configuration problem implemented in *lparse* (excerpt).

Table 1. *A sample of solutions from the ASP model*

Pack	Frame	Engine	Battery	Sunroof	AC	Glass	Opener
Standard	Sedan	esmall	blarge	sr2	—	—	Auto
Standard	Hatch	esmall	bsmall	—	—	—	—
Deluxe	Hatch	esmall	bmed	sr1	ac1	Tinted	—
Deluxe	Hatch	esmall	bsmall	sr2	—	Not	Manual

Note: ASP, answer-set programming.

We can apply both the Alloy and ASP approaches to identify flaws in the constraint specification. In particular, we can discover that the car-configuration problem exhibits both a variable flaw and a value flaw. As we mentioned earlier, a **variable flaw** occurs when a potential variable is required in all models, that is, an option is not really optional. A **value flaw** occurs when a value cannot exist in any valid configuration, so it does not represent an option.

5.1. Checking for specification flaws in alloy

We extend the model we presented in Figure 2 to introduce an abstract type `Flaw`, with **`lone`** subtypes for each category of flaw for we would like to test. Here are some of the **`sig`** definitions for the possible flaws in our model.

```
abstract sig Flaw {}
lone sig
  noLuxury, noDeluxe, noStandard,
  noBSmall, noBMedium, noBLarge, batteryFlaw,
  noAC1, noAC2, ACFlaw
  } extends Flaw{}
```

We then introduce constraints that force the existence of flaw instances, such as one for `BatteriesFlaw`.

```
fact { one noBSmall iff no BSmall }
fact { one batteryFlaw
       iff (no c: Car| no c.battery) }
```

Given similar definitions for each flaw, we can run the Alloy Analyzer requiring no flaws for a large number of cars as follows:

```
run {} for 4 but exactly 4 Car, 0 Flaw
```

This run fails to find an instance; by experiment, we need to raise the number of flaws to 2 before the analyzer finds a solution. This solution includes an instance of `batteryFlaw` (all cars have batteries: a variable flaw) and `noConvertible` (there are no convertibles: a value flaw).

Figure 5 shows the Alloy Analyzer visualization of the flaws in Mittal and Falkenhainer's specification from Figure 1. We clearly see four instances of `Car`, with links to their associated components. However, on the far right of the figure we see an instance of `batteryFlaw` and of `noConvertible`. The fact that we see instances of these flaws demonstrates that they exist in the specification. The instance of `batteryFlaw` indicates the presence of the variable flaw highlighted above, namely, that batteries are not optional, despite the fact that the specification suggests the opposite. The presence of the value flaw that no convertible cars are possible is indicated by the instance of `noConvertible`.

We can explain these flaws, once we find them, by referring to the original specification of Figure 1. Rule 7 forces a battery in every car that has an engine, and `engine` is a classical variable. We might as well call `battery` a classical variable as well.

The other, `noConvertible`, is a value flaw: it is impossible to generate a convertible. This flaw is hidden in the implications of the activity and classical constraints. By constraint 11, convertibles do not have sunroofs. By constraints 1 and 3, cars with the luxury and deluxe packages do have sunroofs, so by elimination, convertibles must have the standard package. But by rule 15, cars with the standard package are not convertibles.

5.2. Checking for specification flaws in ASP

We expand the *lparse* representation for the car CCSP by adding a second, numeric, argument to every predicate. The new argument represents car number. For example, `package(luxury,4)` is a predicate indicating that the fourth car has a luxury package. Now rules like

```
1 {opener(auto,N), opener(manual,N)}
  :- sunroof(sr2,N) .
```

are shorthands that *lparse* expands (in a process called **grounding**) to a new rule for each valid value of `N`. We can limit any solution to four car designs.

```
number(1..4).
#domain number(N).
```

The number 4 is arbitrary; we will use it for the examples to follow. Next, we introduce new nullary predicates for each value of each variable (both classical and potential) to indicate the fact that no car at all uses a particular value, such as in this rule:

```
noLuxury :- {package(luxury,M):number(M)} 0.
```

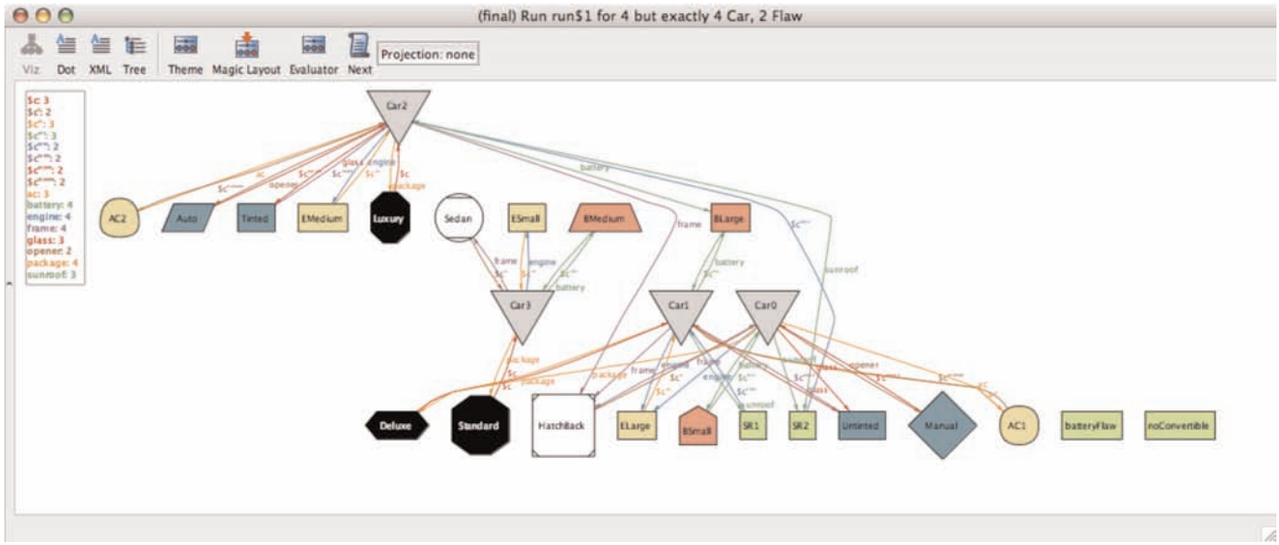

Fig. 5. The Alloy Analyzer visualization of the flaws in Mittal and Falkenhainer's (1990) specification from Figure 1. We clearly see four instances of `Car`, with links to their associated components. However, on the far right of the figure we see instances of `batteryFlaw` and `noConvertible`. That we see instances of these flaws demonstrates that they exist in the specification. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

The grounder converts this shorthand into a rule containing a list of predicates `package(luxury,1)` to `package(luxury,4)`. If not a single one of these `package` predicates is true, which happens if none of the N cars has the luxury package, then `noLuxury` is true, indicating a possible value flaw.

For each potential variable, we introduce two rules, like these:

```
okSunroof :- {sunroof(sr1,N), sunroof(sr2,N)}0.
sunroofFlaw :- not okSunroof.
```

The grounder expands the first rule to four rules, one for each car. If for any car, no sunroof at all is specified, then `okSunroof` is asserted. If no car at all asserts `okSunroof`, then we have a variable flaw, as evidenced by asserting `sunroofFlaw`.

A solution to this expanded program may contain one of the predicates indicating a flaw for several reasons. One is that the particular solutions chosen for the *N* cars may simply not be the ones that demonstrate the use of each value and the absence of each potential variable. We deal with this possibility by asking the solver to minimize the number of such predicates:

```
minimize { % (excerpt)
  noLuxury, noDeluxe, noStandard,
  noBSmall, noBMedium, noBLarge, batteryFlaw,
  noAC1, noAC2, acFlaw } .
```

The solver now searches for solutions containing *N* cars that have the fewest flaws. If we limit *N* to 3, for instance, we find at least four flaws: `noLuxury`, `noConvertible`, `batteryFlaw`, `noManual`. Setting $N = 4$ produces the apparent flaws `batteryFlaw` and `noConvertible`. No matter how high we set *N*, these flaws remain.

When we try the same technique on the second example that Mittal and Falkenhainer present (we omit the second example in the interest of space), we also find both a variable flaw (can capacity) and a value flaw (the particle-physics value of the ontology variable).

It is instructive to note that only four cars are needed to cover the reachable parts of the variable domains; we might have expected that far more are needed. We can inspect these cars to verify that all reachable values are covered and that potential variables can be omitted, as in Table 2.

6. OPTIMAL CARDINALITY CONFIGURATIONS

We might often be interested in finding solutions to a set of conditional constraints that involve the fewest number of options or the largest number of options. We briefly demonstrate how such queries can be answered using our ASP model. We can use the **`minimize`** construct of *lparse* with our original formulation (before we add the numeric argument) to find a minimum solution, that is, a solution with the fewest potential variables. The requirement we add is simply as follows:

```
minimize {
  battery(bsmall; bmed; blarge),
  sunroof(sr1; sr2),
  airConditioner(ac1; ac2),
  glass(tinted; notTinted),
  opener(auto; manual) } .
```

Table 2. *A set of configurations covering all reachable values in the domains of each variable*

Package	Frame	Engine	Battery	Sunroof	AC	Glass	Opener
Standard	Sedan	elarge	blarge	sr2	ac1	Tinted	Manual
Standard	Sedan	esmall	bsmall	—	—	—	—
Deluxe	Sedan	emed	blarge	sr2	ac2	Not	Auto
Luxury	Hatch	esmall	bmed	sr1	ac2	Not	—

Using the *clasp* solver for our *lparse* model we obtain 18 optimal (minimum) solutions, including those presented in Table 3. Similarly, by using **`maximize`**, we can enumerate all 176 maximum solutions, such as those also presented in the table.

7. FMs

We now consider FMs, another form of specification that is encountered in domains such as software configuration, in which the architecture of an artifact is represented graphically. Although FMs appear quite different from CCSPs, they have very similar purposes and yield to very similar analysis. FMs are directed acyclic graphs, where nodes are called **features** and edges imply various kinds of constraints (Czarnecki & Eisenecker 2000). A **solution** is a subset of the features that satisfies all of the constraints. If a feature is present in a solution, then all the features on the path from it to the root of the tree must also be present. A feature in the tree may be marked as **mandatory**, meaning that it must be present in any solution if its parent is present; otherwise, it is **optional**. A feature may indicate that its set of children constitutes an **OR set**, meaning that if the feature is present, at least one of the children must be present. Similarly, a feature may indicate that is set of children constitutes an **XOR set**, meaning that if the feature is present, exactly one of its children must be present. Additional nontree edges indicate that if a feature is present, its successor along the edge must also be present or must not be present.

Figure 6 is based on Segura's (2008) FM for mobile telephones. Each node in the tree is a feature that might or might not be included in any model. Filled circles above features indicate that the feature is mandatory if the parent feature is included in a model. Open circles indicate optional features. Filled semicircles under a node indicate an OR set of children; if the parent is included in the model, at least one of the children must be included. Open semicircles under a node indicate an XOR set of children; if the parent is included in the model, exactly one of the children must be included. Therefore, the `Media` feature is optional, but if it is present, the `MP3` subfeature is mandatory. The `OS` feature requires that exactly one of its subfeatures, `Symbian` or `WinCE`, must be present. The `Messaging` feature requires that at least one of its subfeatures, `SMS` and `MMS`, must be present. The `Games` feature requires the presence of the `Java` feature elsewhere in the tree.

FMs are in most ways like CCSPs. Features in FMs are like variables in CCSPs. These variables have only one possible value, which we can depict as `yes`. The mandatory, optional, and edge constraints are like activity constraints. The OR and XOR constraints do not map directly to CCSPs, however.

Given these similarities, it is not surprising that representing FMs in Alloy or ASP is very much like representing CCSPs. In *lparse*, for instance, we can indicate the mandatory nature of `Settings` and the optional nature of `Media` this way, where `N` refers to the serial number distinguishing phones:

```
1 {settings(N)} 1 :- mobilePhone(N) .
0 {media(N)} 1 :- mobilePhone(N) .
```

We specify the constraint that `Settings` is implied by any child:

```
settings(N) :- 1 {os(N), java(N)} .
```

Table 3. *Sample optimum cardinality configurations*

Package	Frame	Engine	Battery	Sunroof	AC	Glass	Opener
Sample Minimum Cardinality Configurations							
Standard	Hatch	Medium	Medium	—	—	—	—
Standard	Sedan	Large	Medium	—	—	—	—
Standard	Hatch	Large	Large	—	—	—	—
Sample Maximum Cardinality Configurations							
Standard	Hatch	esmall	bmed	sr2	ac1	Tinted	Auto
Deluxe	Hatch	emed	blarge	sr2	ac2	Tinted	Manual
Standard	Hatch	elarge	blarge	sr2	ac1	Not	Manual

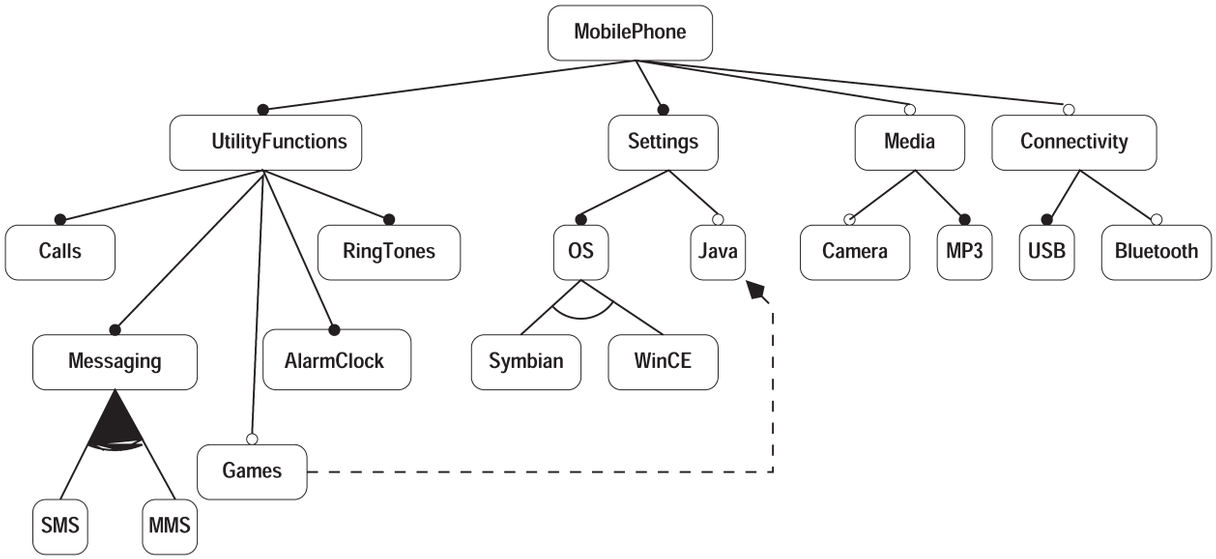

Fig. 6. A feature model for mobile phones based on Segura (2008).

We represent the OR and XOR set constraints for the children of `Messaging` and `OS`:

```
1 {sms(N), mms(N)} :- messaging(N) .
1 {symbian(N), winCE(N)} 1 :- os(N) .
```

Finally, the extra edge constraint from `Games` to `Java`:

```
java(N) :- games(N).
```

FMs are also subject to flaws. If a feature is marked as optional but exists in all solutions, the FM has a **optional-feature flaw**. If a feature is absent in all models, the FM has a **missing-feature flaw**. These flaws can result from non-tree edges. For instance, an edge from `AlarmClock` to `Symbian` in Figure 6 (Segura, 2008) would create a missing-feature flaw for `WinCE`. An edge from `AlarmClock` to `Java` would create an optional-feature flaw for `Java`.

Methods very similar to those we use in CCSPs can discover these flaws in FMs.

8. XML REPRESENTATIONS

In order to standardize how we represent CCSPs and FMs, we have designed XML Document Type Definitions (DTDs) for both, based roughly on XCSP 2.1, the DTD for CSPs (http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf). We have also written Perl scripts that accept instances of CCSPs and FMs obeying these DTDs, generate *lparse* renditions of the constraints, and then apply *clasp* to count or enumerate solutions, find minimum and maximum solutions, and detect flaws. In this way we can automatically generate a formal model of a configuration problem from a very natural specification.

Figure 7 shows our XML representation of the cars CCSP. The constraints typically name a variable or value as a condition and as a result. Either may be negated (as in constraint 10). The XML representation may include the logical connector and in the condition (constraint 12).

Figure 8 shows our XML representation of the phones FM, which nests `feature` nodes to mirror the picture of Figure 6.

9. DISCUSSION

Product configuration is a major industrial application domain for constraint satisfaction techniques. CCSPs and FMs have been developed to represent configuration problems in a direct and natural way. In this paper we have presented two alternative approaches to reasoning about specifications of conditional constraint sets: one approach based on well-established formal methods techniques for reasoning about software specifications, and another based on ASP. The models of the constraint specification are natural in both cases and do not require any reformulation of the original CCSP or FM. We have also shown how we could automate the testing for variable and value flaws (for CCSPs), and missing-feature and optional-feature flaws (for FMs), and that it is possible to find optimal cardinality specifications.

The DTD and Perl script are available from the authors under the GNU General Public License (http://www.gnu.org/copyleft/gpl.html). We have used this software on the fairly large "bikes" configuration (http://www.itu.dk/research/cla/externals/clib/Bike.pm), with 27 variables, some them with domains of size 14, 16, and 36. Our analyzer sets N to twice the largest domain size and tries for 10 s to minimize flaws. It then uses divide and conquer to verify each of the discovered flaws, which might be false positives due to insufficiently large N or incomplete minimization within the time limit. Each verification, however, is very fast and not subject to false positives. In the "bikes" specification, our analyzer finds

```
<instance>
 <presentation
  name="cars"
  description="Cars, from S. Mittal and B. Falkenhainer, Dynamic Constraint
   Satisfaction Problems, AAAI-90, p.  25."
 />
 <domains>
  <domain name="package" nbValues="3" values="luxury deluxe standard"/>
  <domain name="frame" nbValues="3" values="hatchback convertible sedan"/>
  <domain name="engine" nbValues="3" values="esmall emedium elarge"/>
  <domain name="battery" nbValues="3" values="bsmall bmedium blarge"
   potential="true"/>
  <domain name="sunroof" nbValues="2" values="sr1 sr2" potential="true"/>
  <domain name="aircon" nbValues="2" values="ac1 ac2" potential="true"/>
  <domain name="glass" nbValues="2" values="tinted untinted"
   potential="true"/>
  <domain name="opener" nbValues="2" values="auto manual" potential="true"/>
 </domains>
 <constraints>
  <constraint name="1" condition="luxury" result="sunroof"/>
  <constraint name="2" condition="luxury" result="aircon"/>
  <constraint name="3" condition="deluxe" result="sunroof"/>
  <constraint name="4" condition="sr2" result="opener"/>
  <constraint name="5" condition="sr1" result="aircon"/>
  <constraint name="6" condition="sunroof" result="glass"/>
  <constraint name="7" condition="engine" result="battery"/>
  <constraint name="8" condition="opener" result="sunroof"/>
  <constraint name="9" condition="glass" result="sunroof"/>
  <constraint name="10" condition="sr1" result="not opener"/>
  <constraint name="11" condition="convertible" result="not sunroof"/>
  <constraint name="12" condition="bsmall and esmall" result="not aircon"/>
  <constraint name="13" condition="standard" result="not ac2"/>
  <constraint name="14" condition="luxury" result="not ac1"/>
  <constraint name="15" condition="standard" result="not convertible"/>
  <constraint name="16" condition="auto and ac1" result="bmedium"/>
  <constraint name="17" condition="auto and ac2" result="blarge"/>
  <constraint name="18" condition="sr1 and ac2" result="not tinted"/>
 </constraints>
</instance>
```

Fig. 7. An XML representation of Mittal and Falkenhainer's (1990) car-configuration problem.

100 potential flaws in 10 s of minimization and then in another 9 s verifies that 5 are actual value flaws. Finding a solution with a given variable set to a specific value is quite fast (about 0.04 s) even in this relatively large specification; verifying a flaw takes about 0.09 s. We therefore think that the ASP approach scales well. Alloy also scales well; it is used routinely for reasoning about large complex industrial specifications (http://alloy.mit.edu/community/).

10. CONCLUSION

Our future work will study three problems.

1. We will generalize constraint-based explanation techniques so we can give advice on resolving flaws in problem specifications, thus contributing to the emerging literature on conflict detection in formal specifications (Torlak et al., 2008).
2. We will apply fault detection to configuration, so fixing the value of a variable will eliminate all newly unreachable values of other variables.
3. We will investigate how to handle nondiscrete variables, such as real ranges.

ACKNOWLEDGMENTS

Raphael Finkel's work was partially supported by the US National Science Foundation (Grant IIS-0325063). Barry O'Sullivan is funded by the Science Foundation Ireland (Grant 05/IN/I886). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect

```
<instance>
  <presentation
    name="atomic sets"
    description="Phone example of Atomic Sets, from ??" />
  <feature name="mobilePhone">
    <feature name="UtilityFunctions" style="mandatory">
      <feature name="Calls" style="mandatory"> </feature>
      <feature name="Messaging" style="mandatory" childStyle="some">
        <feature name="SMS"> </feature>
        <feature name="MMS"> </feature>
      </feature>
      <feature name="Games" style="optional"> </feature>
      <feature name="AlarmClock" style="mandatory"> </feature>
      <feature name="RingingTones" style="mandatory"> </feature>
    </feature>
    <feature name="Settings" style="mandatory">
      <feature name="OS" style="mandatory" childStyle="one">
        <feature name="Symbian"> </feature>
        <feature name="WinCE"> </feature>
      </feature>
      <feature name="JavaSupport" style="optional"> </feature>
    </feature>
    <feature name="Media" style="optional">
      <feature name="Camera" style="optional"> </feature>
      <feature name="MP3" style="mandatory"> </feature>
    </feature>
    <feature name="Connectivity" style="optional">
      <feature name="USB" style="mandatory"> </feature>
      <feature name="Bluetooth" style="optional"> </feature>
    </feature>
  </feature>
  <constraint condition="Games" result="JavaSupport" />
  <!-- the following two constraints introduce model flaws -->
  <constraint condition="AlarmClock" result="Symbian" />
  <constraint condition="AlarmClock" result="JavaSupport" />
</instance>
```

Fig. 8. An XML representation of Segura's (2008) phone-configuration instance.

the views of the funding agencies. This work is an extension of a conference paper (Finkel & O'Sullivan, 2009).

REFERENCES

Benavides, D., Ruiz-Cortés, A., & Trinidad, P. (2005). Automated reasoning on feature models. *Proc. 17th Int. Conf. Advanced Information Systems Engineering, CAiSE 2005* (Pastor, O., & Cunha, J.F., Eds.), L, Vol. 3520, pp. 491–503. New York: Springer.

Bowen, J., & Bahler, D. (1991). Conditional existence of variables in generalised constraint networks. *Proc. AAAI*, pp. 215–220.

Czarnecki, K., & Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Reading, MA: Addison–Wesley Professional.

Finkel, R.A., & O'Sullivan, B. (2009). Reasoning about conditional constraint specifications. *Proc. ICTAI, IEEE Computer Society*, pp. 349–353.

Gebser, M., Kaufmann, B., Neumann, A., & Schaub, T. (2007). *Clasp*: a conflict-driven answer set solver. *Proc. LPNMR*, pp. 260–265.

Gelle, E., & Faltings, B. (2003). Solving mixed and conditional constraint satisfaction problems. *Constraints 8(2)*, 107–141.

Giunchiglia, E., Yu, L., & Maratea, M. (2004). Cmodels-2: SAT-based answer set programming. *Proc. AAAI*.

Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., & Margaria, T. (2008). Software engineering and formal methods. *Communications of the ACM 51(9)*, 54–59.

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology 11(2)*, 256–290.

Junker, U. (2006). Configuration. In *Handbook of Constraint Programming, Foundations of Artificial Intelligence* (Rossi, F., van Beek, P., Walsh, T., Eds.), pp. 837–873. New York: Elsevier.

Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing 12*, 383–397.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. AAAI-90*, pp. 25–32.

Niemelä, I., & Simons, P. (1997). Smodels—an implementation of the stable model and well-founded semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning* (Dix, J., Furbach, U., & Nerode, A., Eds.), LNCS, Vol. 1265, pp. 420–429. New York: Springer.

Sabin, D., & Freuder, E.C. (1996). Configuration as composite constraint satisfaction. *Proc. Artificial Intelligence and Manufacturing. Research Planning Workshop*, (Luger, G.F., Ed.), pp. 153–161. Menlo Park, CA: AAAI Press.

Sabin, D., & Weigel, R. (1998). Product configuration frameworks—a survey. *IEEE Intelligent Systems 13(4)*, 42–49.

Sabin, M., & Freuder, E.C. (1998). Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. *Proc. CP98 Workshop on Constraint Problem Reformulation*.

Sabin, M., & Gelle, E. (2006). Evaluation of solving models for conditional constraint satisfaction problems. *Proc. AAAI*. New York: AAAI Press.

Segura, S. (2008). Automated analysis of feature models using atomic sets. *Proc. 1st Workshop on Analyses of Software Product Lines (ASPL 2008), SPLC'08*, pp. 201–207, Limerick, Ireland.

Stumptner, M., Friedrich, G.E., & Haselböck, A. (1998). Generative constraint-based configuration of large technical systems. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing 12*, 307–320.

Torlak, E., Chang, F.S.-H., & Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In *FM* (Cuellar, J., Maibaum, T.S.E., & Sere, K., Eds.), LNCS, Vol. 5014, pp. 326–341. New York: Springer.

Raphael Finkel has been a Professor of computer science at the University of Kentucky in Lexington since 1987. He attained his PhD from Stanford University in 1976. He was associated with the first work on quad trees; k-d trees; quotient networks; and the Roscoe/Arachne, Charlotte, Yackos, and Unify operating systems. Dr. Finkel was involved in developing DIB, a package for dynamically distributing tree-structured computations on an arbitrary number of computers. His research includes tools for Unix system administration, databases, operating systems, distributed algorithms, computational morphology, Web-based homework, and ASP applications. He has published over 50 articles in refereed journals and has written two textbooks.

Barry O'Sullivan is the Associate Director of the Cork Constraint Computation Centre and a Senior Lecturer in the Department of Computer Science at University College Cork. He attained his PhD from University College Cork in 1999. His main areas of research interest are constraint programming, artificial intelligence, and optimization, with a focus on application domains such as cancer care, health, environmental sustainability, computer/network security, configuration, design, telecommunications, combinatorial auctions, and electronic commerce. Dr. Sullivan is also interested in theoretical computer science, particularly parameterized complexity and its applications.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 175–183.

doi:10.1017/S0890060410000612

Personalized diagnoses for inconsistent user requirements

ALEXANDER FELFERNIG AND MONIKA SCHUBERT
Institute for Software Technology, Graz University of Technology, Graz, Austria

(RECEIVED May 25, 2010; ACCEPTED October 29, 2010)

Abstract

Knowledge-based configurators are supporting configuration tasks for complex products such as telecommunication systems, computers, or financial services. Product configurations have to fulfill the requirements articulated by the user and the constraints contained in the configuration knowledge base. If the user requirements are inconsistent with the constraints in the configuration knowledge base, users have to be supported in finding out a way from the *no solution could be found* dilemma. In this paper we introduce a new algorithm (PERSDIAG) that allows the determination of personalized diagnoses for inconsistent user requirements in knowledge-based configuration scenarios. We present the results of an empirical study that show the advantages of our approach in terms of prediction quality and efficiency.

Keywords: Configuration; Model-Based Diagnosis; Personalization

1. INTRODUCTION

On an informal level, *configuration* can be defined as a *special case of design activity, where the artifact being configured is assembled from instances of a fixed set of well-defined component types which can be composed conforming to a set of constraints* (Sabin & Weigel, 1998). Configuration systems typically exploit two different types of knowledge sources: the explicit knowledge about the *user requirements* and deep *configuration knowledge* about the underlying product. Configuration knowledge is represented in the form of a product structure and different types of constraints (Felfernig et al., 2003) such as *compatibility constraints* (which component types can or cannot be combined with each other), *requirements constraints* (how user requirements are related to the underlying product properties), or *resource constraints* (how many and which components have to be provided such that needed and provided resources are balanced).

Interacting with a knowledge-based configurator typically means to *specify* a set of requirements, to *adapt* inconsistent requirements, and to *evaluate* alternative configurations (solutions). In this paper we focus on a situation where the configurator is *not able to find a solution*. In such a situation it is very difficult for users to find a set of changes to the specified set of requirements such that a configuration can be found (Felfernig et al., 2004). In order to better support users, we introduce PERSDIAG, which is an algorithm for the *personalized diagnosis of inconsistent user requirements*. PERSDIAG improves the performance of diagnosis calculation and the precision of diagnosis predictions.

State-of-the-art approaches to the determination of minimal diagnoses for inconsistent user requirements are focusing on minimal-cardinality diagnoses (Felfernig et al., 2004) or on the precalculation of all possible diagnoses (McSherry, 2004). In the context of recommender systems (Burke, 2000; Felfernig et al., 2007), the complement of such a diagnosis is often denoted as *maximally successful subquery* (Godfrey, 1997; McSherry, 2004, 2005). Such a query consists of those elements that are not part of a corresponding minimal diagnosis. In the context of constraint-based systems (Tsang, 1993) diagnoses are also interpreted as a specific type of *explanation* (O'Sullivan et al., 2007).

Especially in interactive settings the calculation of all possible diagnoses is infeasible due unacceptable runtimes (Felfernig et al., 2009). Furthermore, it cannot be guaranteed that minimal-cardinality diagnoses lead the most interesting explanations for a user (O'Sullivan et al., 2007; Felfernig et al., 2009). The work of (O'Sullivan et al., 2007) is a first step toward the tailoring of the presented set of diagnoses in the sense that so-called *representative explanations* are determined. These explanations fulfill the criteria that each element part of a diagnosis is also contained in at least one of the diagnoses presented to the user. The work presented in

Reprint requests to: Alexander Felfernig, Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b, A-8010 Graz, Austria. E-mail: alexander.felfernig@ist.tugraz.at

Felfernig et al. (2009) takes one further step toward this direction by introducing personalization concepts that allow to determine personalized repair actions for inconsistent requirements in knowledge-based recommendation (Burke, 2000) where, in contrast to knowledge-based configuration scenarios, a fixed and predefined set of candidate products exists.

On the basis of related work in the field, we introduce a new algorithm for the personalized diagnosis of inconsistent user requirements that is especially tailored to knowledge-based configuration scenarios. The algorithm (PersDiag) performs a best-first search for diagnoses acceptable for the user where the decision on which nodes to expand during search is based on criteria often used in recommender systems development (Felfernig et al., 2007). The major contribution of this paper is to show how standard model-based diagnosis (MBD) approaches (Reiter, 1987; DeKleer et al., 1992) can be extended with intelligent personalization concepts that improve the *prediction quality of diagnosis selection* and reduce the *diagnosis calculation time* when searching for the *top-most-n relevant diagnoses*.

The remainder of this paper is organized as follows. In Section 2 we introduce a working example that will be used for illustration purposes throughout the paper. In Section 3 we discuss a basic approach to identify inconsistent user requirements (Felfernig et al., 2004) that is based on the concepts of MBD (Reiter, 1987; DeKleer et al., 1992). In Section 4 we present an algorithm (PersDiag) for the personalized identification of minimal sets of inconsistent user requirements. The results of empirical and performance evaluations are presented in Section 5. In Section 6, we discuss related and future work. We conclude the paper with Section 7.

2. WORKING EXAMPLE: COMPUTER CONFIGURATION

We will use *computer configuration* as a working example throughout this paper. The task of identifying a configuration for a given set of user requirements can be defined as follows (see Definition 1). This definition is based on the definition given in Felfernig et al. (2004) and, in contrast to the component-port based representation of a configuration problem (Felfernig et al., 2004), it relies on the definition of a constraint satisfaction problem (CSP; Tsang, 1993).

Definition 1 (configuration task). A configuration task can be defined as a CSP (V, D, C), where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of finite domain variables and $D = \{dom(v_1), dom(v_2), \ldots, dom(v_n)\}$ represents the domain of each variable v_i. Here, $C = C_{KB} \cup C_R$ is a set of all constraints, which can be divided into the configuration knowledge base (KB) $C_{KB} = \{c_1, c_2, \ldots, c_m\}$ and the set of specific user requirements (R) $C_R = \{c_{m+1}, c_{m+2}, \ldots, c_p\}$. ■

A simple example for a configuration task (V, D, C) is $V =$ ***{cpu***, ***graphic***, ***ram***, ***motherboard***, ***harddisk***, ***price}***, where ***cpu*** is the type of central processing unit, ***graphic*** represents the graphics card, ***ram*** represents the main memory specified in gigabytes, ***motherboard*** represents the type of motherboard, ***harddisk*** is the harddisk capacity in gigabytes, and ***price*** represents the overall price of the computer. These variables fully describe the potential set of requirements that can be specified by the user. The respective variable domains are $D = \{$dom(***cpu***) = {CPUA, CPUB}, dom(***graphic***) = {GCA, GCB, GCC, GCD}, dom(***ram***) = {1, 2, 3, 4}, dom(***motherboard***) = {MBX, MBY, MBZ, MBW}, dom(***harddisk***) = {200..700}, dom(***price***) = {300..600}$\}$. Note that for reasons of simplicity we do not explicitly discuss pricing constraints; the reader can assume that for each relevant variable value there is a corresponding specified price and that there is a set of constraints responsible for calculating the overall price of the configuration. The set of possible combinations of variable instantiations is restricted by the constraints in the configuration knowledge base $C_{KB} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$. In our working example these are simplified technical and sales constraints:

- c_1: ***cpu*** $= CPUA \Rightarrow$ ***graphic*** $\neq GCA$
- c_2: ***cpu*** $= CPUB \Rightarrow$ ***ram*** > 1
- c_3: ***motherboard*** $= MBY \Rightarrow$ ***ram*** > 1
- c_4: ***harddisk*** $= 700 \Rightarrow$ ***motherboard*** $= MBW$
- c_5: ***motherboard*** $= MBX \Rightarrow$ ***graphic*** $= GCB \vee$ ***graphic*** $= GCD$
- c_6: ***motherboard*** $= MBX \Rightarrow$ ***ram*** $= 1 \vee$ ***cpu*** $\neq CPUA$

For the purposes of our simple example, we assume that the following requirements have been specified by the user ($C_R = \{c_7, c_8, c_9, c_{10}, c_{11}, c_{12}\}$):

- c_7: ***cpu*** $= CPUA$
- c_8: ***graphic*** $= GCA$
- c_9: ***ram*** ≥ 2
- c_{10}: ***motherboard*** $= MBX$
- c_{11}: ***price*** ≤ 350
- c_{12}: ***harddisk*** ≥ 200

Based on this example of a configuration task, we can introduce a definition of a concrete configuration, that is, a solution for a configuration task.

Definition 2 (configuration). A configuration for a given configuration task (V, D, C) is an instantiation $I = \{v_1 = i_1, v_2 = i_2, \ldots, v_n = i_n\}$ of each variable v_j where $i_j \in dom(v_j)$. A configuration is *consistent* if the assignments in I are consistent with the constraints in C. Furthermore, a configuration is *complete* if all the variables in V are instantiated. Finally, a configuration is *valid*, if it is both consistent and complete. ■

In our working example, we assume that users already interacted with the computer configurator and created several configurations (CONFIGS $= \{conf_1, conf_2, conf_3, conf_4, conf_5, conf_6, conf_7\}$). These configurations are stored in a corresponding table (see Table 1). We will exploit this information for the determination of personalized diagnoses in Section 4.

Table 1. *User interaction data from configuration sessions (configuration log)*

	CPU	Graphic	RAM	Motherboard	Hard Disk	Price
$conf_1$	CPUA	GCB	1	MBX	200	350
$conf_2$	CPUB	GCA	3	MBY	500	400
$conf_3$	CPUA	GCD	1	MBX	200	450
$conf_4$	CPUA	GCC	3	MBZ	650	550
$conf_5$	CPUB	GCB	3	MBW	700	600
$conf_6$	CPUA	GCC	2	MBY	200	300
$conf_7$	CPUB	GCC	4	MBY	300	550

3. CALCULATING MINIMAL CARDINALITY DIAGNOSES

For the example configuration task specified in Section 2 we are *not* able to find a valid solution, for example, the processor type *CPUA* is incompatible with the graphic card *GCA* (a simple sales constraint). Therefore, we want to identify the minimal set of requirements ($c_i \in C_R$) that have to be relaxed in order to find a solution. For identifying such minimal sets, we exploit the concepts of MBD (Reiter, 1987; DeKleer et al., 1992). MBD starts with a system description, which in our case encompasses the configuration knowledge base C_{KB} that describes the set of possible product configurations. If the actual behavior of the system conflicts with its intended behavior (a corresponding configuration can be identified), the task of a diagnosis component is to determine those elements (in our case the elements are requirements in C_R) which, when assumed to be functioning abnormally, sufficiently explain the discrepancy between the actual and the intended behavior of the system. A diagnosis is a minimal set of faulty components (in our case requirements) that need to be relaxed in order to be able to identify a configuration.

On a more technical level, minimal diagnoses for faulty user requirements can be identified as follows. Let us assume the existence of a set $C_{KB} = \{c_1, c_2, \ldots, c_m\}$ of configuration constraints and a set $C_R = \{c_{m+1}, c_{m+2}, \ldots, c_p\}$ of user requirements (represented as constraints) *inconsistent* with C_{KB}, that is, no solution can be found for the constraints in $C_R \cup C_{KB}$. In such a situation, state-of-the-art configurators (Sinz & Haag, 2007) calculate a set of minimal diagnoses $DIAGS = \{diag_1, diag_2, \ldots, diag_k\}$, where $\forall diag_i \in DIAGS : C_{KB} \cup (C_R - diag_i)$ is consistent. A corresponding *User Requirements Diagnosis Problem* (*UR Diagnosis Problem*) can be defined as follows:

DEFINITION 3 (UR diagnosis problem). A UR diagnosis problem is defined as a tuple (C_{KB}, C_R) where C_{KB} represents the constraints of the configuration knowledge base and C_R is a set of user requirements. ■

Based on the definition of the UR diagnosis problem, a *UR diagnosis* can be defined as follows:

DEFINITION 4 (UR diagnosis). A UR diagnosis for (C_{KB}, C_R) is a set of constraints $diag \subseteq C_R$ such that $C_{KB} \cup (C_R - diag)$ is consistent. A diagnosis *diag* is minimal if there does not exist a diagnosis $diag' \subset diag$ such that $C_{KB} \cup (C_R - diag')$ is consistent. ■

Following the basic principles of MBD (Reiter, 1987; DeKleer et al., 1992), the calculation of diagnoses is based on the identification and resolution of conflict sets. A *conflict set* in the user requirements C_R can be defined as follows:

DEFINITION 5 (conflict set). A conflict set is defined as a subset $CS \subseteq C_R$ such that $CS \cup C_{KB}$ is inconsistent. CS is minimal if and only if there does not exist a conflict set CS' with $CS' \subset CS$. ■

In our simple working example, the user requirements $C_R = \{c_7, \ldots, c_{12}\}$ are inconsistent with the constraints in the configuration knowledge base $C_{KB} = \{c_1, \ldots, c_6\}$, that is, there does not exist a configuration (solution) that completely fulfills the requirements in C_R. The minimal conflict sets are $CS_1 = \{c_7, c_8\}$, $CS_2 = \{c_8, c_{10}\}$, and $CS_3 = \{c_7, c_9, c_{10}\}$, because each of these conflict sets is inconsistent with the configuration knowledge base and there do not exist conflict sets CS_1', CS_2', and CS_3' with $CS_1' \subset CS_1$, $CS_2' \subset CS_2$, and $CS_3' \subset CS_3$.

In MBD (Reiter, 1987; DeKleer et al., 1992) the standard algorithm for determining minimal diagnoses is the *hitting set-directed acyclic graph* (HSDAG) as described in Reiter (1987). User requirements diagnoses $diag_i \in DIAGS$ can be calculated by resolving conflicts in the set of requirements C_R. Because of its minimality property, one conflict can be resolved by deleting exactly one of the elements from the conflict set. After deleting at least one element from each identified conflict set we are able to present a diagnosis. The HSDAG algorithm employs breadth-first search where the resolution of all minimal conflict sets leads to the identification of all minimal diagnoses. In our working example the diagnoses derived from the conflict sets CS_1, CS_2, and CS_3 are $DIAGS = \{\{c_7, c_8\}, \{c_7, c_{10}\}, \{c_8, c_9\}, \{c_8, c_{10}\}\}$.

The construction of such a HSDAG is exemplified in Figure 1. The HSDAG algorithm assumes the existence of a component that is able to detect minimal conflict sets. Our implementation is based on a version of the QUICKXPLAIN

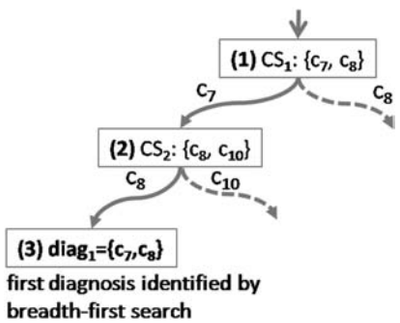

Fig. 1. Hitting set directed acyclic graph (Reiter, 1987) for the working example. The first identified diagnosis is $diag_1 = \{c_7, c_8\}$. The algorithm returns minimal diagnoses with increasing cardinality, that is, $diag_1 = \{c_7, c_8\}$ is a minimal cardinality diagnosis. The complete set of minimal diagnoses is $DIAGS = \{\{c_7, c_8\}, \{c_7, c_{10}\}, \{c_8, c_9\}, \{c_8, c_{10}\}\}$.

conflict detection algorithm introduced by Junker (2004). Following a breadth-first search regime with the goal of identifying a minimal diagnosis, we have to resolve the conflict set CS_1 by checking whether c_7 or c_8 already represent a diagnosis. Both alternatives to resolve the conflict do not lead to a diagnosis since $(C_R - \{c_7\}) \cup C_{KB}$ as well as $(C_R - \{c_8\}) \cup C_{KB}$ are still inconsistent. We now can switch to the next level of the search tree because breadth-first search inspects all nodes at level n of the search tree first and then extends the search to level $n + 1$. Let us assume that the next conflict set returned by QUICKXPLAIN is $CS_2 = \{c_8, c_{10}\}$. Now, $(C_R - (\{c_7\} \cup \{c_8\})) \cup C_{KB}$ does not trigger further conflicts, which means that $diag_1 = \{c_7, c_8\}$ has been identified as the first *minimal cardinality diagnosis*. Further details on the standard HSDAG algorithm can be found in Reiter (1987).

A major question to be answered is whether minimal cardinality diagnoses are leading to configurations of relevance, that is, have a high probability of being selected by the user. We will provide answers in the following sections.

4. CALCULATING PERSONALIZED DIAGNOSES

As the number of possible diagnoses can become large, and presenting such a large number of alternatives to the user is inappropriate, we want to systematically reduce the number of alternatives with the goal to identify relevant diagnoses for the user and keep the diagnosis evaluation process as simple as possible. A simple heuristic to identify such diagnoses has already been presented in Section 3, where diagnoses have been ranked to conform to their cardinality; in our working example $\{c_7, c_8\}$ has been identified as first minimal cardinality diagnosis. An alternative to this breadth-first search-based approach is to exploit recommendation techniques (Felfernig et al., 2007) for the identification of *relevant diagnoses*, that is, diagnoses that have a higher probability of being accepted by the user. In the following we will show how basic recommendation approaches can be exploited for the prediction of diagnoses that are relevant to the user. First, we will show how we can determine diagnoses leading to configurations that are similar to the original set of user requirements (*similarity-based diagnosis selection*). Second, we will introduce a utility-based approach that uses preference data for guiding the HSDAG construction (*utility-based diagnosis selection*).

4.1. Similarity-based diagnosis selection

The idea of similarity-based diagnosis selection is to prefer those minimal diagnoses that lead to configurations resembling the original user requirements. In order to derive such diagnoses, we can exploit information contained in already existing configurations (see, e.g., the configuration log in Table 1). For each entry in Table 1 we can calculate its similarity with the user requirements in C_R. The similarity values of our working example calculated on the basis of Eq. (4), $simrec(C_R, conf_k)$, $k = 1..7$, are $conf_1 = 0.45$, $conf_2 = 0.60$, $conf_3 = 0.43$, $conf_4 = 0.25$, $conf_5 = 0.30$, $conf_6 = 0.36$, $conf_7 = 0.14$. These values are calculated on the basis of the entries in Table 1 and the preferences of our example user, which are the importance values $w(c_i)$: $c_7 = 0.08$ (8%), $c_8 = 0.34$ (34%), $c_9 = 0.08$ (8%), $c_{10} = 0.17$ (17%), $c_{11} = 0.08$ (8%), $c_{12} = 0.25$ (25%).[1]

The calculation of similarity values is based on three attribute-level similarity measures (Konstan et al., 1997; Wilson & Martinez, 1997; McSherry, 2004). These measures calculate the similarity of a pair of attribute (a_i) of configuration $conf_k$ and the corresponding user requirement (c_i), for example, the similarity between attribute *ram* of configuration $conf_1$ and the user requirement c_9 ($ram \geq 2$) is 0.33, where we take the lower bound $ram = 2$ as basis for similarity calculation. Depending on the characteristics of the attribute, one of the three measures [Eqs. (1)–(3)] is chosen: *More-Is-Better* (MIB), *Less-Is-Better* (LIB) or *Nearer-Is-Better* (NIB; McSherry, 2004).

For attributes like *harddisk size* or the *ram size*, the higher the value the better it is for the user (MIB). For attributes like *price*, the lower the value the more satisfied the user is (LIB). When the user specifies a certain type of CPU (no intrinsic value scale), we suppose the most similar is the preferred one. In those cases, the NIB similarity measure is used.[2]

$$MIB: sim(c_i, a_i) = \frac{val(c_i) - min(a_i)}{max(a_i) - min(a_i)} \quad (1)$$

$$LIB: sim(c_i, a_i) = \frac{max(a_i) - val(c_i)}{max(a_i) - min(a_i)} \quad (2)$$

$$NIB: sim(c_i, a_i) = \begin{cases} 1 & if\ val(c_i) = val(a_i) \\ 0 & else \end{cases} \quad (3)$$

On the basis of the individual similarity values, Eq. (4) calculates the overall similarity value between the sequence of user requirements (c) and the sequence of attribute values of configuration a. In this context $w(c_i)$ denotes the importance of requirement c_i for our example user. The importance values can be directly specified by the user or derived by a learning algorithm, for example, a genetic algorithm.

$$simrec(c, a) = \sum_{i=1}^{n} \text{sim}(c_i, a_i) \times w(c_i) \quad (4)$$

The similarity values provided above will now be exploited for determining diagnoses in a personalized fashion (see Fig. 2).

For the similarity-based selection of diagnoses we again assume that the QUICKXPLAIN algorithm (Junker, 2004) returns as first conflict set $CS_1 = \{c_7, c_8\}$. Now there are two possibilities of resolving CS_1. If we delete c_7 from CS_1, the following configurations CONFIGS $= \{conf_2, conf_5, conf_7\}$ are consistent with c_7. This means that each of the configurations in CONFIGS is *inconsistent* with the requirement c_7 and

[1] Note that our approach does not rely on a specific preference elicitation method.

[2] For a detailed discussion of different types of similarity measures see, for example, McSherry (2004) and Wilson and Martinez (1997). In Eqs. (1)–(3), $val(c_i)$ denotes the value of user requirement c_i, $min(a_i)$ denotes the minimal possible value of configuration attribute a_i, and $max(a_i)$ denotes the maximal possible value of attribute a_i.

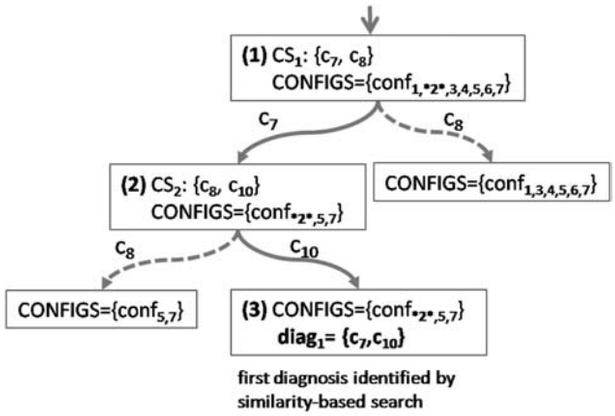

Fig. 2. Similarity-based selection of diagnoses with PERSDIAG.

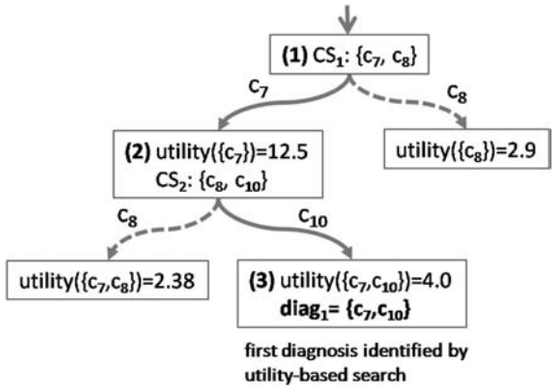

Fig. 3. Utility-based selection of diagnoses with PERSDIAG.

thus a potential candidate configuration for supporting diagnoses that include c_7. If we delete c_8 from CS_1, then CONFIGS $= \{conf_1, conf_3, conf_4, conf_5, conf_6, conf_7\}$. The configuration with the highest similarity compared to the original set of requirements $C_R = \{c_7, \ldots, c_{12}\}$ is $conf_2$ contained in node (2) of Figure 2. Consequently, node (2) of the HSDAG is further expanded, which results in the next conflict set $CS_2 = \{c_8, c_{10}\}$, because $C_{KB} \cup (C_R - \{c_7\})$ is still inconsistent. With this expansion we have identified two alternative diagnoses, namely, $\{c_7, c_8\}$ and $\{c_7, c_{10}\}$. The diagnosis $\{c_7, c_{10}\}$ will be rated higher because it is consistent with the configuration $conf_2$, the configuration with the highest similarity to the set of requirements, that is, $conf_2 \cup C_{KB} \cup (C_R - \{c_7, c_{10}\})$ is consistent. Note that in many configuration scenarios there exists a *ramp-up problem* (Burke, 2000) because initially no configuration data are available. An approach to deal with this situation is to define a *threshold value* that specifies an upper similarity limit for configurations to be accepted as similar to the original set of requirements. If no such configuration exists, a fallback solution is to present diagnoses resulting from breadth-first search or to apply the criteria presented in the following.

4.2. Utility-based diagnosis selection

The idea of utility-based diagnosis selection is to prefer those minimal diagnoses, which include requirements of low importance for the user. Following a utility-based approach (Winterfeldt & Edwards, 1986) we are summing up the individual importance values (see above) of the requirements part of a diagnosis in order to generate a corresponding ranking. The function $utility(C \subseteq C_R)$ returns a utility score for a specific set C that is a subset of the user requirements C_R [see Eq. (5)].

$$utility(C \subseteq C_R) = \frac{1}{\sum_{c_i \in C} w(c_i)} \tag{5}$$

For the utility-based selection of diagnoses we again assume that QUICKXPLAIN returns as first conflict set $CS_1 = \{c_7, c_8\}$ (see Fig. 3). The importance value for c_7 is 0.08, whereas the importance value for requirement c_8 is 0.34 (see above). By applying Eq. (5) we derive the corresponding utility values, for example, utility($\{c_7\}$) = 1/0.08 = 12.5 and utility($\{c_8\}$) = 1/0.34 = 2.9. Because resolving the conflict set $\{c_7, c_8\}$ by deleting c_7 has a higher utility [application of Eq. (5)], the search for a diagnosis is continued with $C_R - c_7$, which results in the second conflict set returned by QUICKXPLAIN ($CS_2 = \{c_8, c_{10}\}$). Again, we sort the utility values for all nodes in the fringe of the search tree and come to the conclusion that extending the path $\{c_7, c_{10}\}$ is the best choice ($utility(\{c_7, c_{10}\}) = 4.0$). Because $(C_R - \{c_7, c_{10}\} \cup C_{KB})$ is consistent, $diag_1 = \{c_7, c_{10}\}$ is the first diagnosis identified (in this case the result is the same as the one determined by the similarity-based approach).

4.3. Algorithm for calculating personalized diagnoses

The algorithm for calculating best-first minimal diagnoses for inconsistent user requirements is the following (Algorithm 1, PERSDIAG). We keep the description of the algorithm on a level of detail, which has been used in the description of the HSDAG algorithm (Reiter, 1987). In PERSDIAG, the different paths of the HSDAG are represented as separate elements in a collection structure H that is initially empty. H stores all paths of the search tree in a best-first fashion, where the currently best path *(h)* is the one with the most promising (partial) diagnosis. If the theorem prover (TP) call TP($(C_R - h) \cup C_{KB}$) does not detect any further conflicts for the elements in h ($isEmpty$(CS)), a diagnosis is returned. The major role of the TP *is to check whether there exists a configuration for* C_R, disregarding the already resolved conflict set elements in h. If the theorem prover call TP($(C_R - h) \cup C_{KB}$) returns a nonempty conflict set CS, h is expanded to the paths containing exactly one element of CS each. In case that h is expanded, the original h must of course be removed from H ($delete(h, H)$). Afterward, the new elements have to be inserted into H. This collection (H) is then finally sorted ($sort(H, k)$) according to the criteria defined in k.[3] In this context, k repre-

[3] Note that the HSDAG pruning is implemented by the functionalities of $sort(H, k)$.

sents the criteria used for selecting the next node to be expanded in the search tree that could be *breadth-first*, *similarity-based*, or *utility-based*.

Algorithm 1 PERSDIAG(C_R, C_{KB}, H, k)
{C_R: set of user requirements}
{C_{KB}: the configuration knowledge base}
{H: collection of all paths in the search tree (initially empty)}
{k: node evaluation criteria used by *sort*(H, k)}
{h: diagnosis returned}
$h \leftarrow$ *first*(H)
CS $\leftarrow$ TP($(C_R - h) \cup C_{KB}$)
if *isEmpty*(CS) **then**
  return h
else
  for all X *in* CS **do**
    $H \leftarrow H \cup \{h \cup \{X\}\}$
  end for
  $H \leftarrow$ *delete*(h, H)
  $H \leftarrow$ *sort*(H, k)
  PERSDIAG(C_R, C_{KB}, H, k)
end if

5. EVALUATION

5.1. Evaluation of prediction quality

To demonstrate the improvements achieved by our approach, we conducted an empirical study. Configuration data were gathered on the basis of an online user study conducted at the Graz University of Technology with 415 participants (82.4% male, 17.6% female) conform to the basic structure of Table 1. Each participant had to define his/her requirements [including the corresponding importance values—see Eq. (4)] regarding a predefined set of *12 computer attributes* (*price, type of central processing unit, operating system, operating system language, amount of main memory, screen size, harddisk capacity, type of DVD drive, Web cam, type of graphic card, amount of graphic card memory, and type of service*). After this requirements specification phase participants were informed about the fact that for the specified set of requirements no solution could be found (the goal was to confront each participant with such a situation). The system then presented a list of a maximum of 50 alternative configurations (only those repair configurations *inconsistent* with the current set of requirements) that have been calculated by a computer configuration knowledge base built for the product set offered by a commercial website.[4] The ordering of the configurations in this list was randomized and the participants were enabled to navigate in the list and to order the configurations regarding different criteria such as the price (LIB), the size of the hard disk (MIB), or the number of fulfilled requirements (MIB). The participants then had the task to select one out of the presented repair configurations that appeared to be the most acceptable one for them.

Based on the data collected in the user study we evaluated the three presented approaches with respect to their capability of predicting diagnoses that are acceptable for the user. The first approach is based on the algorithm proposed by Reiter (1987), where diagnoses are ranked according their *cardinality* and diagnoses of the same cardinality are ranked according to their calculation order (see Section 3). The second approach identifies personalized diagnoses on the basis of a *similarity*-based node expansion strategy in HSDAG construction (see Section 4). The third approach uses a *utility* measure to find relevant diagnoses for the user (see Section 4). Because of the fact that no solution was made available for the original set of requirements, for each such set of requirements we could determine conflicts and a set of corresponding diagnoses that indicated which of the requirements had to be relaxed in order to be able to identify a solution (conflicts were induced by excluding those configurations from the set of possible configurations that are *consistent* with a given set of requirements). Figure 4 depicts the distribution of diagnoses with respect to their cardinality. Most of the diagnoses contained about five elements (diagnoses of cardinality 5), the average number of diagnoses per set of user requirements was 5.32.

We were then interested in the *prediction accuracy* of the three different diagnosis approaches (*cardinality based*, *similarity based*, and *utility based*). First, we analyzed the distance between the *predicted position* of diagnoses leading to a selected repair proposal and their *expected position* (which is 1). We measured this distance in terms of the *root mean square deviation* [RMSD; see Eq. (6)], where *predicted position* is the ranking determined by the diagnosis approach and *expected position* is 1; that is, it is expected that the algorithm correctly predicts the diagnosis. The utility-based diagnosis approach has the lowest RMSD, which is 0.97. The similarity-based approach shows a similar RMSD value (1.03), and the cardinality-based approach shows the worst performance (RMSD = 1.64).

$$\mathrm{RMSD} = \sqrt{\frac{1}{n}\sum_{1}^{n}(\mathit{predicted\,position} - \mathit{expected\,position})^2} \quad (6)$$

Although RMSD is a good-quality estimate, it provides only limited information about the precision of the prediction. Therefore, we analyzed the precision of the three diagnosis approaches; the precision measure is shown in Eq. (7). The basic idea is to provide a measure on how often a diagnosis that leads to the repair configuration selected by the participant is among the top-n ranked diagnoses. As can be seen in Table 2, the utility-based approach has the highest prediction accuracy in terms of precision, followed by the similarity-based diagnosis approach. The cardinality-based approach has the worst performance in terms of prediction accuracy. We were interested whether we could detect a statistically sig-

[4] The knowledge base has been implemented for the 50 configurations extracted from www.dell.at. We chose this simple knowledge base in order to avoid biases, for example, in terms of presenting only solutions that are near the original set of requirements.

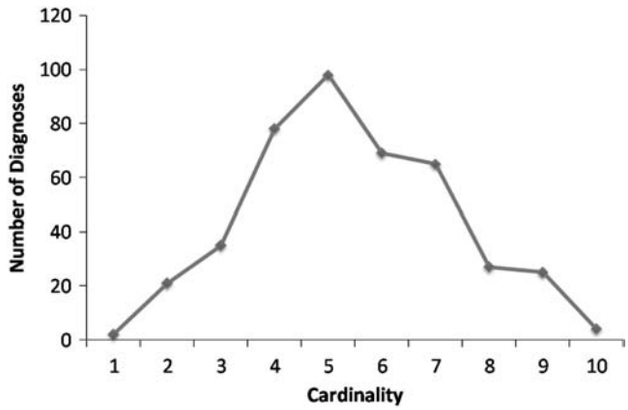

Fig. 4. Overall distribution of diagnoses in empirical study; average number of diagnoses per set of user requirements = 5.32 (SD = 1.67).

nificant difference between the three diagnosis approaches in terms of prediction accuracy. Therefore, we conducted a pairwise comparison between the diagnosis approaches on the basis of a Mann–Whitney U test. We could detect a significant difference between the prediction accuracy of *utility-based* diagnosis and *cardinality-based* diagnosis ($p = 5.69e^{-9}$) and between *similarity-based* and *cardinality-based* diagnosis ($p < 2.2e^{-16}$). There was no significant difference between *utility-based* and *similarity-based* diagnosis in terms of prediction accuracy ($p = 0.5952$).

$$precision = \frac{|correctly\,predicted\,diagnoses|}{|\,predicted\,diagnoses|} \tag{7}$$

5.2. Performance evaluation

The PersDiag algorithm has been implemented on the basis of the standard hitting set algorithm introduced in Reiter (1987). The algorithm is NP-hard in the general case (Friedrich et al., 1990) but is applicable for interactive configuration settings (see the following evaluation). In our implementation, the determination of minimal conflict sets is based in QuickXPlain (Junker, 2004). In the worst case, QuickXPlain needs $O(2k \times \log(n/k) + 2k)$ consistency checks to compute one minimal conflict set of size k (given an inconsistent constraint set of cardinality n).

In order to be able to conduct an in-depth performance analysis, we based our evaluation on different generated settings characterized by a varying number of conflict sets (1–5 conflict sets of cardinality 1–4) and corresponding diagnoses (3–22). As configuration engine we used the constraint solver Choco (choco.emn.fr), the performance evaluation has been conducted on a standard PC (Intel Core2 Quad QD9400 2.66-GHz CPU with 2 GB of RAM). The solver had to conduct consistency checks on knowledge bases with $n = 100$ variables, $t = 100$ constraints in C_{KB}, and $q = 5..20$ user requirements (C_R) inconsistent with C_{KB} (we did not optimize the knowledge bases in terms of, for example, *variable selection* or *value selection*). Based on this setting we compared the performance of the best-first based diagnosis approaches (*similarity-based* and *utility-based*) with the performance of the standard breadth-first search approach (cardinality-based) when calculating the *topmost-n relevant diagnoses* (for $n = 5.10$, see Fig. 5). Best-first based diagnosis clearly outperforms the breadth-first one because the latter has to determine all diagnoses to be able to achieve a comparable prediction quality.

Table 2. *Precision of the three diagnosis approaches*

	top-1	top-2	top-3
Cardinality based	0.51	0.75	0.87
Similarity based	0.70	0.87	0.97
Utility based	0.74	0.89	0.96

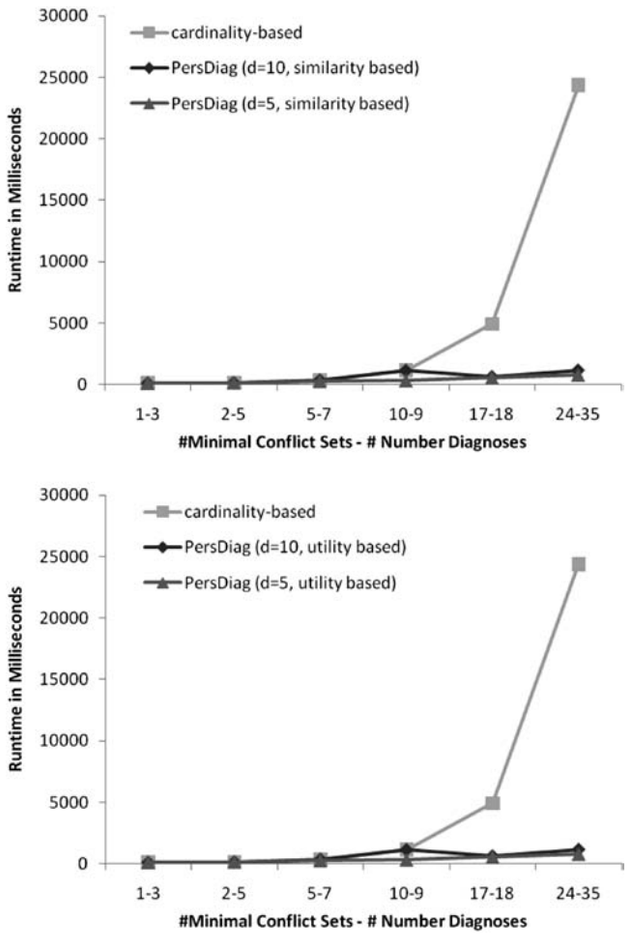

Fig. 5. The performance of the cardinality-based (breadth-first) diagnosis approach compared to personalized approaches for the topmost-n relevant diagnoses for typical combinations of #conflict sets and #diagnoses (Felfernig et al., 2004). Personalized approaches are significantly more efficient (compared to the cardinality-based approach) and show similar performance among themselves.

6. RELATED AND FUTURE WORK

6.1. Knowledge-based configuration

Configuration is one of the most successful application areas of artificial intelligence (Stumptner, 1997). One of the first configuration systems was R1/XCON, which has been developed by John McDermott on the basis of the OPS5 language (McDermott, 1982). A detailed analysis and discussion of the experiences with R1/XCON is provided in Barker et al. (1989). In productive use, the system included ∼31,000 components and ∼17,500 rules. R1/XCON was a rule-based system that triggered enormous maintenance problems because of the intermingling of product domain and problem solving knowledge. Acquisition and maintenance processes for knowledge bases have been significantly improved by the development of model-based knowledge representations with a strict separation of problem solving and domain knowledge (Mittal & Frayman, 1989, 1990). Most of today's available configuration systems are based on such a model-based approach: examples of corresponding configuration environments are SAP (Haag, 1998), SIEMENS (Fleischanderl et al., 1998), and TACTON (Orsvarn, 2005). The diagnosis concepts presented in this paper are focusing on the mentioned model-based knowledge representations and consequently provide an important contribution to the improvement of commercial systems in terms of usability.

6.2. MBD

The increasing size and complexity of configuration knowledge bases motivated the application of MBD (Reiter, 1987; DeKleer et al., 1992) for testing and debugging purposes (Felfernig et al., 2004). Similar reasons led to the application of MBD in technical domains such as hardware designs (Friedrich et al., 1999) and onboard diagnosis for automotive systems (Sachenbacher et al., 2000). The work presented in Felfernig et al. (2004) has a special relationship to the concepts presented in this paper: Felfernig et al. (2004) focus on the application of MBD to the identification of faults in configuration knowledge bases where test cases are used to induce conflicts in a given configuration knowledge base. In addition, a first approach to calculate diagnoses for inconsistent user requirements is presented, which is based on breadth-first based HSDAG construction. In this paper we have shown how to apply basic recommendation algorithms (similarity based and utility based) to improve the diagnosis algorithms in terms of prediction accuracy and performance.

6.3. Conflict detection

Diagnosis calculation for inconsistent user requirements relies on minimal conflict sets. Such conflict sets can be determined, for example, on the basis of QuickXPlain (Junker, 2004), which is a frequently applied divide and conquer algorithm. Alternative approaches to the identification of conflicts have been developed in the context of knowledge-based recommendation (Schubert et al., 2009, 2010). These approaches cannot be applied in knowledge-based configuration scenarios, because due to the size and complexity of the underlying products, knowledge-based configurators typically do not operate on a predefined set of products. The existence of predefined item sets is a major precondition for applying the conflict detection algorithms introduced in Schubert et al. (2009, 2010).

6.4. Diagnosing inconsistent requirements

An approach to suggest personalized repair actions for inconsistent requirements in the context of knowledge-based recommendation tasks has been introduced by Felfernig et al. (2009). The underlying idea is to apply the concepts of MBD (Reiter, 1987; DeKleer et al., 1992) to determine change proposals (minimal sets of inconsistent requirements) in the case of a given predefined list of products. In O'Sullivan et al. (2007) such minimal sets are denoted as minimal exclusion sets. In case-based recommendation scenarios (Godfrey, 1997; McSherry, 2004, 2005) the complement of a minimal exclusion set is denoted as maximally successful subquery. The concept of representative explanations has been introduced by (O'Sullivan et al., 2007). Representative explanations follow the idea of generating diversity in sets of diagnoses (minimal exclusion sets). The approach does not explicitly take into account the preference structure of the current user but rather tries to determine diagnosis sets that satisfy the requirement that each element (constraint) part of at least one diagnosis is also contained in at least one of the diagnoses presented to the user. Note that the scenario presented in this paper is based on the assumption of an *open configuration* approach where the user is free to specify requirements and the system provides feedback in the form of explanations in the case of inconsistencies. Alternatively, configurators precalculate still possible options and dim options that cannot be selected in the current context. In such a scenario our diagnosis approach could be used for intentionally exploring trade-offs in the set of user requirements (a kind of specific exploration mode in addition to the standard mode where still valid options are predetermined).

6.5. Assumption-based truth maintenance based approaches

The notion of *conflict sets* used in the context of MBD (Reiter, 1987; DeKleer et al., 1992) corresponds to the notion of nogoods in **a**ssumption-based truth maintenance approaches to calculate explanations (Haag, 1998; Sinz & Haag, 2007). On the basis of the conjunctive normal form of the set of nogoods we can easily determine the corresponding set of diagnoses by transforming the conjunctive normal form into a corresponding disjunctive normal form.

6.6. Future work

Future work will include the evaluation of other potential prediction techniques for user requirements diagnoses such as probability-based prediction or similarity-based prediction using

local search-based learning of attribute weights. Furthermore, we are interested in developing mechanisms that support the calculation of preferred diagnoses in the case of complex requirement structures, for example, structures such as *x or y should be fulfilled*. We are also interested in the calculation of personalized recommendations of repair proposals for inconsistent requirements, that is, we want to extend the concepts presented in this paper with the determination of concrete change proposals (repairs related to diagnoses) for inconsistent user requirements in knowledge-based configuration scenarios.

7. CONCLUSION

In this paper we introduced an algorithm (PersDiag) for the determination of personalized diagnoses. The algorithm significantly improves the prediction quality compared to state of the art diagnosis approaches. PersDiag follows a best-first search regime and can be parametrized with different kinds of selection strategies regarding the expansion of the search tree. We have compared different expansion strategies (cardinality based, similarity based, and utility based) within the scope of an empirical study. The results of this study show the advantages of personalized diagnosis calculation compared to existing breadth-first based search in terms of prediction quality and efficiency. These results provide a solid basis for improving existing industrial applications regarding the determination of diagnoses for inconsistent requirements.

REFERENCES

Barker, V., O'Connor, D., & Soloway, E. (1989). Expert systems for configuration at digital—XCON and beyond. *Communications of the ACM 32(3)*, 298–318.

Burke, R. (2000). Knowledge-based recommender systems. *Library and Information Systems 69(32)*, 180–200.

DeKleer, J., Mackworth, A., & Reiter, R. (1992). Characterizing diagnoses and systems. *AI Journal 56(2–3)*, 197–222.

Felfernig, A., Friedrich, G., Jannach, D., & Stumptner, M. (2004). Consistency-based diagnosis of configuration knowledge bases. *AI Journal 152(2)*, 213–234.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., & Zanker, M. (2003). Configuration knowledge representations for semantic web applications. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing 17(2)*, 31–50.

Felfernig, A., Friedrich, G., & Schmidt-Thieme, L. (2007). Introduction to the IEEE intelligent systems special issue: recommender systems. *IEEE Intelligent Systems 22(3)*, 18–21.

Felfernig, A., Friedrich, G., Schubert, M., Mandl, M., Mairitsch, M., & Teppan, E. (2009). Plausible repairs for inconsistent requirements. *Proc. 21st Int. Joint Conf. Artificial Intelligence (IJCAI09)*, pp. 791–796, Pasadena, CA.

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

Friedrich, G., Gottlob, G., & Neijdl, W. (1990). Physical impossibility instead of fault models. *Proc. 8th National Conf. Artificial Intelligence AAAI/IAAI90*, pp. 331–336, Boston.

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

Godfrey, P. (1997). Minimization in cooperative response to failing database queries. *International Journal of Cooperative Information Systems 6(2)*, 95–149.

Haag, A. (1998). Sales configuration in business processes. *IEEE Intelligent Systems 13(4)*, 78–85.

Junker, U. (2004). QuickXPlain: preferred explanations and relaxations for over-constrained problems. *Proc. 19th National Conf. Artificial Intelligence (AAAI04)*, pp. 167–172, San Jose, CA.

Konstan, J., Miller, B., Maltz, D., Herlocker, J., Gordon, L., & Riedl, J. (1997). Grouplens: applying collaborative filtering to usenet news. *Communications of the ACM 40(3)*, 77–87.

McDermott, J. (1982). R1—a rule-based configurer of computer systems. *Artificial Intelligence 19(1)*, 39–88.

McSherry, D. (2004). Maximally successful relaxations of unsuccessful queries. *Proc. 15th Conf. Artificial Intelligence and Cognitive Science*, pp. 127–136, Galway, Ireland.

McSherry, D. (2005). Retrieval failure and recovery in recommender systems. *Artificial Intelligence Review 24(3–4)*, 319–338.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. 8th National Conf. Artificial Intelligence, IAAI/AAAI90*, pp. 25–32, Boston.

Mittal, S., & Frayman, F. (1989). Towards a generic model of configuration tasks. *Proc. 11th Int. Joint Conf. Artificial Intelligence (IJCAI89)*, pp. 1395–1401, Detroit, MI.

Orsvarn, K. (2005). Tacton configurator—research directions. *Proc. IJCAI 2005 Workshop on Configuration*, p. 75, Edinburgh, Scotland.

O'Sullivan, B., Papdopoulos, A., Faltings, B., & Pu, P. (2007). Representative explanations for over-constrained problems. *Proc. 22nd National Conf. Artificial Intelligence (AAAI07)*, pp. 323–328, Vancouver, Canada.

Reiter, R. (1987). A theory of diagnosis from first principles. *AI Journal 23(1)*, 57–95.

Sabin, D., & Weigel, R. (1998). Product configuration frameworks—a survey. *IEEE Intelligent Systems 13(4)*, 42–49.

Sachenbacher, M., Struss, P., & Carlen, C. (2000). Prototype for model-based onboard diagnosis of automotive systems. *AI Communications 13(2)*, 83–97.

Schubert, M., Felfernig, A., & Mandl, M. (2009). Solving over-constrained problems using network analysis. *Proc. Int. Conf. Adaptive and Intelligent Systems*, pp. 9–14, Klagenfurt, Austria.

Schubert, M., Felfernig, A., & Mandl, M. (2010). Fastxplain: conflict detection for constraint-based recommender problems. *Proc. 23rd Int. Conf. Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pp. 621–630, Cordoba, Spain.

Sinz, C., & Haag, A. (2007). Configuration. *IEEE Intelligent Systems 22(1)*, 78–90.

Stumptner, M. (1997). An overview of knowledge-based configuration. *AI Communications 10(2)*, 111–125.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Reading, MA: Academic Press.

Wilson, D., & Martinez, T. (1997). Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, *6*, 1–34.

Winterfeldt, D., & Edwards, W. (1986). *Decision Analysis and Behavioral Research*. Cambridge: Cambridge University Press.

Alexander Felfernig is a Professor of applied software engineering at Graz University of Technology. Alexander is also Cofounder and Director of ConfigWorks, a company focused on the development of knowledge-based recommendation technologies. Prof. Felfernig's research focuses on intelligent methods and algorithms supporting the development and maintenance of complex knowledge bases. Furthermore, he is interested in the application of AI techniques in the software engineering context, for example, the application of decision and recommendation technologies to make software requirements engineering processes more effective. In 2009 Dr. Felfernig received the Heinz–Zemanek Award from the Austrian Computer Society for his research.

Monika Schubert is a PhD student in the group of Applied Software Engineering at Graz University of Technology. Ms. Schubert received her MS in software engineering and economy from Graz University of Technology. Her research focuses on knowledge-based systems, intelligent product configuration, MBD, and product recommendation. She is also interested in user interaction with complex knowledge bases.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 185–195.

doi:10.1017/S0890060410000624

Adaptive attribute selection for configurator design via Shapley value

YUE WANG AND MITCHELL M. TSENG
Advanced Manufacturing Institute, Hong Kong University of Science and Technology, Hong Kong

(RECEIVED April 4, 2010; ACCEPTED October 29, 2010)

Abstract

Configurators have been generally accepted as important tools to elicit customers' needs and find the matches between customers' requirements and company's offerings. With product configurators, product design is reduced to a series of selections of attribute values. However, it has been acknowledged that customers are not patient enough to configure a long list of attributes. Therefore, making every round of configuring process productive and hence reducing the number of inputs from customers are of substantial interest to academic and industry alike. In this paper, we present an efficient product configuration approach by incorporating Shapley value, which is a concept used in game theory, to estimate the usefulness of each attribute in the configurator design. This new method iteratively selects the most relevant attribute that can contribute most in terms of information content from the remaining pool of unspecified attributes. As a result from product providers' perspective, each round of configuration can best narrow down the choices with given amount of time. The selection of the next round query is based on the customer's decision on the previous rounds. The interactive process thus runs in an adaptive manner that different customers will have different query sequences. The probability ranking principle is also exploited to give product recommendation to truncate the configuration process so that customers will not be burdened with trivial selection of attributes. Analytical results and numerical examples are also used to exemplify and demonstrate the viability of the method.

Keywords: Attribute Selection; Configurator; Probability Ranking Principle; Shapley Value

1. INTRODUCTION

Today, global marketplace is becoming increasingly competitive and diversified. Offering tailored products and in the mean time delivering products quickly to customers become major challenges for current manufacturing industry (McCutcheon et al., 1994). In this new situation, product configurator systems have been explored to handle the so-called "customization–responsiveness squeeze" phenomena, that is, providing tailored products with short delivery time (Schierholt, 2001; Salvador & Forza, 2004). Basically, a product configurator consists of a set of predefined attributes for customers to choose from. Some constraints on these attributes are also included to ensure that the selected attributes work compatibly. It takes a customer's specifications as input and the output is the customer's target product(s). With product configurators, product design is reduced to a series of selections of attribute values (Darr & Birmingham, 2000). In the studies of customers' decision-making processes, it has been shown that customers have higher satisfaction with the outcomes of configuring process than traditional selection process (Kurniawan et al., 2003). Today, product configurators have not only been studied in academia but also been widely adopted in industries. It is reported that configurators have greatly improved manufacturers' responsiveness in product customization and reduced the cost of customer integration (Piller, 2004). Product configuration systems have been accepted as a viable strategy to bridge the gap between customers' needs and companies' offerings.

The history of product configurators can be traced back to 1970s. Digital Equipment Corporation (DEC) developed a program called R1 (later called XCON) in 1978 to configure VAX computer systems to customer specifications (McDermott et al., 1980). It was first put to use in 1980, and by 1986, it had processed 80,000 orders, achieving 95–98% accuracy. It was estimated to be saving DEC $25 million a year by reducing the need to give customers free components when technicians made errors, by speeding up the assembly process, and by increasing customer satisfaction. Ever since then, a large number of configuration expert systems had been developed and put into use, such as Cossack (Frayman et al., 1987), BLADES (Elturky et al., 1986), and MICON

Reprint requests to: Yue Wang, Advanced Manufacturing Institute, RM2591, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: yacewang@ust.hk

(Birmingham et al., 1988). Because of the development of information technology in the past decade, it is possible for companies to acquire immediate information about customers' requirements and meet them by delivering customized products or related information efficiently. One of the most cited successful modern configurators cases is Dell Computer, which is able to deliver customized personal computers and notebooks within 1 week, with prices lower than its mass producing competitors. By using online configurator-based product customization system, Dell Computer has gained the so-called first-mover advantage and maintained high profitability and growth in a hypercompetitive industry for a long period.

Using configurators can streamline and automate the configuring process, reduce configuration errors, and enhance flexibility and responsiveness (Sabin & Weigel, 1998). However, there are still some limitations and shortcomings that have not been paid enough attention in previous research (Tseng & Piller, 2003). First, most product configurators still rely on fixed query sequences that entail sets of rigid interactive procedures. Although some configurators can capture customers' specifications in the order determined by customers, there is still no systematic study on adaptively eliciting customer needs according to customers' specifications in previous configuration steps. Therefore, the configuration process can be characterized as a one-way information flow from customers to designers, instead of an interactive and adaptive customer needs elicitation process. Second, product configuring process can be tedious and time consuming, especially when the product is complex. The configuring procedure may require seemingly redundant or trivial dialogues between customers and product development team. However, it has been widely acknowledged that customers are impatient to specify a long list of attributes (Enos, 2001). Therefore, it is necessary to elicit customers' needs in an efficient manner. Third, customers may have little knowledge about what a manufacturer is offering, including products features, design, limitations, cost, and delivery. Furthermore, they may even be unable to articulate their needs. Sometimes they are unclear about what they really want when facing a large number of options provided by companies. Customers may fail to understand or appreciate manufacturers' offerings (Simonson, 2005). They may find the configuration process unpleasant or even stressful (Schwartz, 2004). In summary, it is crucial for configurators to capture customers' specifications efficiently with less demand for customers' attention and time.

To overcome the limitations, we develop an approach for attribute selection task in product configuration process. Product configuring is considered as a sequential Q&A process. From designers' perspective, a customer's specifications to a product's attributes are unknown before the configuration process. Designers' objective is to elicit the customer's needs efficiently and accurately. During configuring process, designers can discover the customer's needs gradually based on the customer's partial specifications to some attributes. The more attributes the customer configures, the more information about the customer's needs is obtained. Thus, in product configuration process, designers' uncertainty about the customer's needs is decreasing. In this sense, the configuration process is an *uncertainty elimination process* from designers' point of view. We want to eliminate the most uncertainty about a customer's needs in each configuration round so that designers can capture the customer's needs efficiently. In this paper, Shapley value is deployed to evaluate the relevance or usefulness level of each attribute. The method iteratively selects the most relevant attribute from the unspecified attributes pool and proposes it for the customer to configure. The selection of the next round query is based on the customer's decision in previous rounds. Thus, it solicits customers' specifications in an adaptive manner in the sense that different customers may have different query sequences. The customized one-to-one configuring procedure is presented and the final configuration can converge to a customer's target with fewer interactions between the customer and designers. In this sense, the configuration process is no longer a traditional process of passively accepting customers' specifications, but a bidirectional information flow procedure. This paper extends the methods in Wang and Tseng (2007, 2009) by presenting an analytical framework of attributes selection and product recommendation. Numerical studies are also conducted to verify the proposed approach.

The remainder of the paper is organized as follows. Related literature will be briefly reviewed in Section 2. In Section 3 we introduce the methodology for attribute selection from coalition game's point of view. A product recommendation approach is elaborated in Section 4. Section 5 presents the detailed procedure for attribute selection. A numerical example is presented in Section 6 to verify the proposed approach. Session 7 concludes the whole paper and points out some further research directions.

2. LITERATURE REVIEW

A configurator design can be considered as a reasoning task in its nature. Existing configuration design methodologies can be generally classified into rule-based, model-based, and case-based methodologies, depending on the reasoning techniques used (Sabin & Weigel, 1998).

In a rule-based system, design knowledge is codified as configuration rules or constraints. Most of early configuration systems fall in this category, like R1/XCON, Cossack, BLADES, and MICON. They derive solutions in a forward chaining manner. This kind of systems often suffers from the maintenance issues because of the lack of separation between domain knowledge and control strategy, especially when the configurator system is complex (Yu et al., 1998).

In a model-based system, design knowledge is contained in a system *model*, which consists of decomposable entities and interactions between their elements. The most extensively studied approach is probably constraint-based approach. Mittal and Frayman (1989) first treated configuration tasks as constraint satisfaction problems (CSPs). In this framework, a configuration task is to assign values to all the variables without violating any constraints. Mittal and Falkenhainer also modeled the configuration task as a sequence of dynamic

CSPs to cope with the change of attributes set during product configuration process. Both the ports and design alternatives were considered as variables in a CSP domain (Mittal & Falkenhainer, 1990). Not only compatibility constraints but also activity constraints were introduced into the extension to specify conditions under which a configurator can dynamically include or exclude component based on current selections. Sabin and Freuder (1996, 1998) proposed an idea that represented the configuration task as a new class of nonstandard CSP called composite CSP. It provided a more comprehensive and efficient basis for formulating and solving configuration problems (Sabin & Freuder, 1996, 1998). Gelle and Faltings developed a general framework to handle both continous and discrete variables in configuration task that is called *mixed* and *conditional CSP* problems (Gelle & Faltings, 2003). A generative CSP framework was also defined that can support resource-balancing constraints (Mailharro et al., 1998; Stumptner et al., 1998). In this framework, component attributes were used to represent the resource demands and supplies.

In a case-based system, the basic idea is to compute the similarity between the input queries and existing product cases. Then the existing configurations that are likely to satisfy the input queries are refined according to customers' particular needs. As pointed by Wielinga and Schreiber (1997), the key issue in case-based configuration is how to retrieve the best configuration from the database and identify aspects that cause violation of constraints or requirements. Different methods have been used to tackle the issue. Rahmer and Voß (1996) used resource-oriented scheme to deal with case adaptation for telecooperation system. Löckenhoff and Messer (1994) presented detailed knowledge engineering based models for case-based configuration. It is a structure-oriented approach where a taxonomical structure of components is mapped onto a graph. The approach is also resource-oriented based on balancing of resource requesting and production model of components. Hüllermeier (1997) recast case-based reasoning task from combination optimization perspective. A combination optimization based approach was applied to solve configuration problems. Critiquing is also a method for case-based reasoning systems by using customers' feedback information (Burke, 2002; Burke et al., 1996, 1997). Customers only need to indicate a directional preference for a feature instead of inputting detailed feature value. Traditional critiquing only copes with single feature. Reilly et al. (2004) extended the technique to multiple features case, which is called compound critiques. They argued that the methods can offer explanatory benefits to help users better understand the structure of the recommendation mechanism and improve the performance of case-based recommendations. Tseng et al. (2005) also used case-based reasoning to construct a new bill of materials to reduce the time and cost of product design.

Table 1 gives an overview of different configuration methods, including the advantages and limitations of each approach. It is worth noting that the proposed method in this paper does not belong to any of the three categories. This paper does not present a holistic configuration design method. It is mainly concerned with the task of how to present attributes for customers to configure which is a major step for configuration tasks. In this sense, the proposed approach can be applied to any product configuration system.

Table 1. *Overview of different configuration approaches*

Approach	Advantages	Limitations
Rule-based configurator	• Intuitive way of presenting design knowledge and configurations • Good at representing and evaluating heuristic relations	• Difficult to maintain when the product is complex
Model-based configurator	• Good at handling constraints within the configuration rules	• Limited efficiency when solving constraints
Case-based configurator	• More efficient by using existing cases	• The results may be unreliable and inaccurate • Cannot adapt to new product configuration

3. ATTRIBUTE SELECTION AS A COALITION GAME

As stated in Section 1, it is crucial for configurators to capture customers' specifications efficiently, because customers are not patient enough to specify a long list of attribute. From a designer's point of view, the configuration design task is to select the attributes and the way of configuring them (Yu et al., 1998). During product configuration process, an attribute is presented for a customer to specify in each configuration round. A well-designed series of attributes could potentially shorten the lengthy iterative information exchange procedure. Therefore attribute selection serves as a critical factor for improving the efficiency of configurators. In this section we introduce a coalition game-based attribute selection criteria to accelerate the product configuration procedure.

3.1. Preliminaries

In this section we will recast the attribute selection problem from game theory point of view, particularly coalition game. In a coalitional game, a set of players have certain payoff functions that represent the benefit achieved by different subcoalitions in the game. In a formal language, a coalition game is defined by (N, v) where N is a set of players and v is a worth function of any subset of N, that is, the coalition. The meaning of the worth function is that if S is a coalition of players who agree to cooperate in a game, then v is the expected benefit they can get from the cooperation. Here, v is assumed to be monotone, that is, $v(S') \geq v(S)$ for $S \subset S'$ and $v(\phi) = 0$. Let

$\{x_i\}_{i\in N}$ be a partition of $v(N)$, that is,

$$v(N) = \sum_{i\in N} x_i,$$

where x_i is the benefit that player i can get from the cooperation. The marginal benefit for player i with respect to $S \in N$, $i \notin S$ is $\Delta_i(S) = v(S \cup \{i\}) - v(S)$. Intuitively, the worth function and marginal benefit have the following properties:

- Dummy axiom: if player i is a dummy player, then $x_i = 0$. It means that if a player contributes nothing in the game, he should not receive any benefit.
- Symmetry axiom: if $i \neq j$ such that $\Delta_i(S) = \Delta_j(S)$ for all $i, j \notin S$, then $x_i = x_j$. It means that if two players contribute equally in the game, they should receive the same amount of benefit.
- Linearity axiom: if $v(S) = v_1(S) + v_2(S)$ where v_1 and v_2 are also nonnegative monotone function satisfying $v_1(\phi) = v_2(\phi) = 0$, then $x_i = x_i^1 + x_i^2$ where x_i^j is the cost share for v_j. This axiom means that two coalition games can be combined.

Then the Shapley value for the ith player is defined as the expectation $E(X_i)$ where $X_i = v((\sigma_1, \sigma_2, \ldots, \sigma_i)) - v((\sigma_1, \sigma_2, \ldots, \sigma_{i-1}))$ and $(\sigma_1, \sigma_2, \ldots, \sigma_i)$ is a permutation of $(1, 2, \ldots, i)$, where σ_j {can be any number in the set $(1, 2, \ldots, i)$. For example, $(\sigma_1, \sigma_2, \sigma_3)$ can be any element in the set of {(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)}. Shapley value is the expected marginal worth of a player over all the possible sets of coalitions. The expectation is calculated with respect to all the possible permutations with equal probability. Shapley (1953) proved that Shapley value is the only value that satisfies the three axioms.

3.2. Estimating attribute contribution by Shapley value

As stated above, customers are not patient enough to specify a long list of items. In addition, the attributes to be specified differ a lot in terms of the usefulness to reveal customers' needs. We want to estimate the usefulness of each attribute and ask customers to specify the most useful one. In this paper, the Shapley value of each attribute is used to measure the usefulness level.

The calculation of the Shapley value is usually time consuming for the attribute selection problem because it requires summing over all the possible subsets of coalitions and permutation on them. Keinan et al. (2004) presented an unbiased estimator to calculate the Shapley value by uniformly sampling from the permutation over N. Still, the estimator considers both large and small attributes sets to calculate the contribution values. Cohen et al. (2005) found that in practical problems, the contributions of players in a coalition formed by a subset of N are not as significant as that in coalition N, the coalition formed by all the players. They calculated the contribution value of each player only with respect to N. Thus, the computational complexity is reduced because coalitions with size smaller than N are not taken into consideration. In this sense, the Shapley value of the ith attribute can be approximated by

$$E(X_i) \approx \frac{1}{n} \times \Delta_i(N \backslash \{i\}).$$

Because the parameter $1/n$ is the same for all the attributes, we only need to consider $\Delta_i(N \backslash \{i\})$ to select attribute. During each configuration round, the Shapley value of each unconfigured attribute will be calculated and the attribute with the biggest Shapley value is presented to for the customer to configure.

In this paper, we try to eliminate the most uncertainty about a customer's needs in each configuration round. Hence, the amount of uncertainty about customers' needs is adopted as the evaluation criterion. Because one of the key concerns in configurator design is to achieve product configuration quickly and accurately, the most informative attribute should be selected from the remaining unspecified attributes pool. Bearing this in mind, we take Δ_i as the form of

$$\Delta_i(C) = \text{entropy}(C) - \text{entropy}(C|i), \tag{1}$$

where C is the set containing all the end products. Each product has certain probability to be a customer's target. After knowing the value of the ith attribute, the set C will be reduced to a subset $C|i$; C is a discrete random variable with possible states $1, \ldots, n$. Its entropy is defined as

$$\text{entropy}(C) = -\sum_{k=1}^{n} p_k \log_2 p_k,$$

where p_k is the probability that C is in state k (Shannon, 1948). The concept of entropy in information theory describes how much uncertainty there is in a signal or random event. Similarly, the entropy of $C|i$ is

$$\begin{aligned} \text{entropy}(C|i) &= E_i(\text{entropy}(C|i = i_k)) \\ &= \sum_{i_k} P(i = i_k) \times \text{entropy}(C|i = i_k), \end{aligned} \tag{2}$$

where i_k is the kth alternative of the ith attribute.

In summary, from designers' perspective the Shapley value of an attribute is the amount of uncertainty that the attribute can eliminate after getting its value. Shapley value is deployed to select attribute for a customer to configure in each configuration round. The unspecified attribute that can eliminate the most uncertainty will be chosen for the customer to specify.

3.3. Estimation of parameters

To calculate the Shapley value of each attribute, we need to know the probability (conditional probability) that each end product meets a customer's needs. Given enough customers' choices data, the probabilities can be estimated from the data.

In this paper, we use the frequency that each end product being selected by customers to approximate the probability that the product will meet customers' needs. It can be proven that it is a maximum likelihood estimator of the probability. The estimation is

$$P(A_i = a_{ij} \,|\, A_k = a_{kl}) = \frac{|a_{ij} \cap a_{kl}|}{|a_{kl}|},$$

where a_{ij} and a_{kl} are the alternatives of attribute A_i and A_k, respectively; $|a_{ij} \cap a_{kl}|$ is the number of cases with attribute A_i having value a_{ij} and in the mean time attribute A_k having value a_{kl} in existing configuration data; and $|a_{kl}|$ is the number of cases with attribute A_k having value a_{kl}. To avoid zero probability caused by data sparsity, we apply the widely used smoothing technique by adding constants to both numerator and denominator (Cox, 1972). Then the estimation becomes

$$P(A_i = a_{ij} \,|\, A_k = a_{kl}) = \frac{|a_{ij} \cap a_{kl}| + 1}{|a_{kl}| + r}, \tag{3}$$

where r is the number of alternatives of attribute A_i. If $|a_{ij} \cap a_{kl}| = |a_{kl}| = 0$,

$$P(A_i = a_{ij} \,|\, A_k = a_{kl}) = \frac{1}{r}. \tag{4}$$

It means that each alternative of attribute A_i is equally likely to be selected by the customer, that is, we assume the prior probability distribution of customers' choices is uniform. When sufficient data are obtained, the knowledge discovered from data will dominate the prior probability, because if $|a_{ij} \cap a_{kl}|$ is large enough,

$$P(A_i = a_{ij} \,|\, A_k = a_{kl}) = \frac{|a_{ij} \cap a_{kl}| + 1}{|a_{kl}| + r} \approx \frac{|a_{ij} \cap a_{kl}|}{|a_{kl}|}. \tag{5}$$

Therefore, this estimation is actually the compromise between the knowledge discovered from the data and the prior belief about the probability distribution of customers' potential likelihood toward different attributes.

Both A_i and A_k can be generalized from one single attribute to a set of attributes. The idea is also to use the frequency of each event to approximate the true probability that we are interested in. By law of large number, the estimation will converge to the true probability when the data size increases. In this way, the likelihood of customers' choices dependency among different attributes will be quantified by conditional probabilities. The conditional probabilities will be deployed for the calculation of Shapley value.

4. RECOMMENDATION OF PRODUCT CONFIGURATIONS

The presented product configuration method aims at helping customers find their target products quickly. To further improve the efficiency of product configuration, a product recommendation module is also used to present the most likely accepted product and thus truncate the product configuration process as early as possible. After a customer configures an attribute, certain number of products will be recommended. If the customer is satisfied with one of them, he can select it and terminate the configuration process ahead of time. Basically a recommendation method is to find the most likely accepted product configuration, that is, the one with the highest probability to meet a customer's requirements, based on the incomplete specifications. In this section, two basic questions regarding product recommendation will be answered, namely, how to calculate the probability of relevance for each product, that is, the product's probability of being a customer's target and in what order to present the recommendations if multiple products are recommended.

Let E be the set of configured attributes and Q be the set of attributes that are not specified. Let R denote the recommendation that is an instantiation of Q plus the specified attributes set E. Then we have

$$R = E \cup \left\{ \arg\max_{Q} P(Q|E) \right\}.$$

By Bayes rule,

$$P(Q|E) = \frac{|Q \cap E|}{|E|},$$

where $|Q \cap E|$ is the number of configurations with attributes Q and E and $|E|$ is the number of configurations with attributes E.

If we assume customers' choices among different attributes are independent, the recommendation can be simplified to

$$R = E \cup \left\{ \bigcup_{q_i \in Q} \arg\max_{q_i} P(q_i|E) \right\}, \tag{6}$$

where q_i is the ith unspecified attribute and $P(q_i|E)$ can be estimated by

$$P(q_i|E) = \frac{|q_i \cap E|}{|E|}.$$

This independence assumption is often referred to as "local independence" and has been applied in marketing research (Kamakura & Wedel, 1995).

In this way, each end product's probability of meeting the customer's needs can be calculated. A very natural way to present the recommendations is based on the ranking of the probabilities, which is known as probability ranking principle (PRP; van Rijsbergen, 1979).

In information retrieval literature, expected search length is used to measure the efficiency of a retrieval approach (van Rijsbergen, 1979). It refers to the expected number of items that a customer has screened when he finds a target one. Let esl(S) represent the expected search length given a set of specifications S. Then,

$$\text{esl}(S) = E[i] = \sum_{i=1}^{N} p_i \times i,$$

where i is the search length and p_i represents the corresponding probability of occurrence of this search length. It is worth noting that p_i is a function of S. To simplify the notations, we just use p_i here. N is the number of possible search lengths and is bounded by the total number of product configurations. It can be proven that PRP guarantees the smallest expected search length and thus the highest efficiency.

PROPOSITION 1. *The PRP results in a minimal expected search length.* ■

Proof: Given a set of partial specifications S from a customer, let $R_1 = (r_{11}, r_{12}, \ldots, r_{1m})$ be the recommendation based on PRP with corresponding probabilities of meeting a customer's needs $(P_{11}, P_{12}, \ldots, P_{1m})$, which satisfies $P_{11} \geq P_{12} \geq \cdots \geq P_{1m}$. Let us consider another recommendation approach that proposes the same m recommendations in another order $R_2 = (r_{21}, r_{22}, \ldots, r_{2m})$ with probabilities of meeting a customer's needs $(Q_{11}, Q_{12}, \ldots, Q_{1m})$, which is a permutation of $(P_{11}, P_{12}, \ldots, P_{1m})$. The expected search length can be reformulated as

$$\mathrm{esl}(S) = E[i] = \sum_{i=1}^{m} p_i \times i = \sum_{n=1}^{m} P(i \geq n),$$

where $P(i \geq n)$represents the probability that the first $n - 1$ recommendations are not the target product (Durrett, 2003). Then we can yield

$$P_{\mathrm{PRP}}(i \geq n) = \prod_{k=1}^{n-1} (1 - P_{1k}).$$

Similarly,

$$P_{\mathrm{others}}(i \geq n) = \prod_{k=1}^{n-1} (1 - Q_{2k}).$$

Because $P_{11} \geq P_{12} \geq \cdots \geq P_{1m}$ and $(Q_{11}, Q_{12}, \ldots, Q_{1m})$ is a permutation of $(P_{11}, P_{12}, \ldots, P_{1m})$, we can get

$$\begin{aligned} P_{\mathrm{PRP}}(i \geq n) &= \prod_{k=1}^{n-1} (1 - P_{1k}) \leq \prod_{k=1}^{n-1} (1 - Q_{2k}) \\ &= P_{\mathrm{others}}(i \geq n) \end{aligned}$$

for any n. By applying rearrangement inequality, we arrive at

$$\begin{aligned} \mathrm{esl}_{\mathrm{PRP}}(S) &= \sum_{n=1}^{m} P_{\mathrm{PRP}}(i \geq n) \leq \sum_{n=1}^{m} P_{\mathrm{others}}(i \geq n) \\ &= \mathrm{esl}_{\mathrm{others}}(S). \end{aligned}$$

Because R_2 is selected arbitrarily, the PRP results in the minimal expected search length comparing with any other recommendation approaches. ■

It is worth noting that we assume each customer has certain target product(s) in mind that is unknown to designers. However, designers can *guess* which configuration(s) may be the target product(s) based on the customer's partial specifications during product configuring process. This proposition states that the best strategy for designers to present recommendations is based on the probability of relevance. Customers' expected search length can be minimized in this way.

5. THE PROCESS OF ADAPTIVE ATTRIBUTE SELECTION FOR CONFIGURATOR DESIGN

The configuring process is an interactive and adaptive communication procedure. A schematic configuration process is shown in Figure 1. The conditional probabilities of customers' choices are estimated offline from existing configuration data. Once the probabilities are estimated, they will serve as the supporting base for product configuration procedure, particularly the attribute selection and recommendation. It sequentially selects the most relevant item from the remaining attributes pool for a customer to configure and recommendations will be presented afterward. The whole operation process can be summarized as follows:

1. Put all of the unspecified attributes into candidate set CS.
2. For each unspecified attributes q_i, calculate the conditional probability $P(q_i|E)$ according to Eq. (3), where E is the set of all the combination of the remaining attributes.
3. For each unspecified attribute q_i, calculate its Shapley value $E(X_i)$ based on Eq. (1).
4. Select the attribute with the biggest Shapley value and present it to the customer to configure.
5. Get the customer's specification and remove the attribute from the candidate set CS.

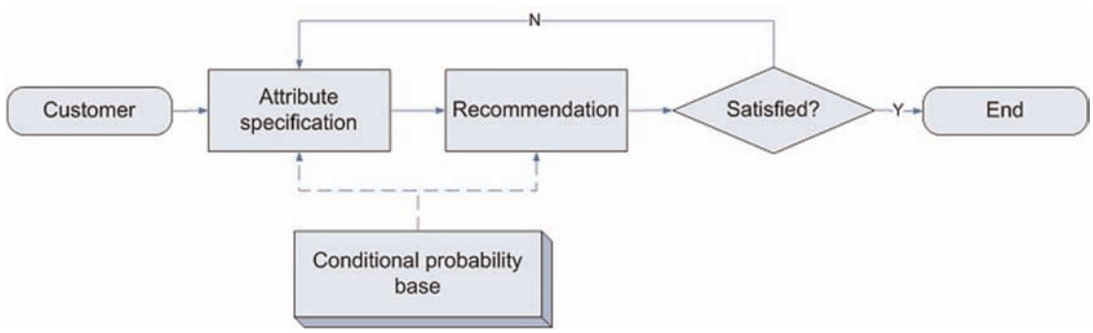

Fig. 1. The product configuration process. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

6. Calculate the probability that each end product meets the customer's needs

$$P\left(\bigcup_i \{q_i\}|E\right) = \prod_i P(q_i|E)$$

according to Eq. (6) and present recommendations according to the ranking of the probabilities.
7. Get the customer's feedback toward the recommendations. If the customer is satisfied with one recommendation, end.
8. If CS = ϕ, end. Otherwise goes to step 3.

Note that we assume the customer's target product is in current product family. Because the purpose of this paper is to elicit customer needs and provide customer's target product efficiently, the case that no target product exists in current product family is not considered here (Fig. 1).

6. CASE STUDY

This section uses the configuration process of a simplified personal computer as an example to illustrate the ideas proposed in this paper. The set of components and their alternatives are listed in Table 2. Here we use a sixtuple to represent one PC configuration. For example, <1, 2, 2, 3, 2, 2> stands for the configuration containing the components A1, B2, C2, D3, E2, and F2. A survey was conducted in an East Asian university and 69 customers' preferred configurations data were obtained. We want to use them to estimate the conditional probabilities $P(q_i|E)$ that we need to run our method. However, the sample size is too small for the scale of the configuration task. To handle the data sparsity issue, we generated 1380 configuration data as training data to estimate the conditional probability and 345 testing data by a perturbative bootstrap approach. Bootstrap is a powerful data resampling method in statistics (Efron, 1979). It generates samples from an existing data set, where each sample is obtained by random sampling with replacement from the data set. Considering that customers may be flexible to some choices (Lilien et al., 1992), we add some variants when generating resamples. We call the resampling method perturbative bootstrap.

Table 2. *List of components and their alternatives for PC*

Component	Code	Description
Processor (A)	A1	Intel Core 2 duo 3.16 GHz
	A2	Intel Core 2 duo 2.66 GHz
	A3	Intel Core 2 duo 2.8 GHz
	A4	Intel Pentium Dual-Core 2.6 GHz
	A5	Intel Core 2 quad processor 2.5 GHz
	A6	Intel Core 2 quad processor 2.6 GHz
Memory (B)	B1	2 GB DDR2
	B2	4 GB DDR2
	B3	6 GB DDR2
	B4	8 GB DDR2
Monitor (C)	C1	17-in. LCD
	C2	19-in. LCD
	C3	20-in. LCD
	C4	22-in. LCD or above
Hard disk (D)	D1	160 GB
	D2	250 GB
	D3	500 GB
	D4	750 GB
Disk driver (E)	E1	16X DVD+/−RW*
	E2	Blu-ray disk
	E3	Blu-ray disk + 16X DVD+/−RW*
Display card (F)	F1	Intel GMA 3100
	F2	512-MB NVIDIA GeForce 9800GT
	F3	256-MB ATI Radeon HD 3450 LE
	F4	256-MB ATI Radeon HD 3650
	F5	512-MB ATI Radeon HD 4670

Before conducting the resampling method, each attribute alternative's substitutes are identified according to the similarity of performance, price and other characteristics. To make the algorithm more general, let us suppose a product's attributes set is $\{A_i : 1 \le i \le k\}$, where $k$is the number of attributes. Each attribute A_i has a set of alternatives $a_{ij} : 1 \le i \le k$, $1 \le j \le n_i$, where n_i is the number of alternatives for the ith attribute. Each attribute alternative a_{ij} has a substitute set $\overline{a_{ij}}$ determined beforehand. The substitute set $\overline{a_{ij}}$ can be empty if there is no proper substitute for a_{ij}. The detailed data resampling algorithm is as follows:

For $l = 1$ to m // m is the data size of the original data set
  For $i = 1 : k$
    For $j = 1 : n_i$
    u = rand(0,1); // generating a random number following uniform(0,1)
    if $u > h$ and $\overline{a_{ij}}$ is not empty, a_{ij}^* equals to one value in $\overline{a_{ij}}$ with probability $\frac{1}{|\overline{a_{ij}}|}$ where $|\overline{a_{ij}}|$ is the cardinal of set $\overline{a_{ij}}$ and h is a predetermined threshold;
    else $a_{ij}^* = a_{ij}$;
    end
  end
end

In this numerical example, h is set to 0.9. After running the algorithm once, a new set of configuration data are generated. It has the same size as the original data set. We repeat the process 25 times and generate 1380 training data and 345 testing data. Because of the limit of pages, the conditional probabilities estimated from the training data set are omitted here.

6.1. A product configuration example by applying the proposed approach

Suppose a new customer's target configuration is <1, 2, 2, 3, 2, 2> that is unknown to designers before the product configuring process.

STEP 1. The probabilities $P(q_i|E)$ are calculated according to Eq. (3). Because no attributes are specified at the beginning, $P(q_i)$ is used instead.

STEP 2. The Shapley values of each attribute are calculated according to Eq. (1). For example,

$$\begin{aligned}\Delta_A(S) &= \text{entropy}(S) - \text{entropy}(S|A)\\ &= -\sum_{i=1}^{69} 0.014 \times \log 0.014\\ &\quad -\left(-0.246\sum_{i=1}^{17} 0.059 \times \log 0.059\right.\\ &\quad -0.159 \times \sum_{i=1}^{11} 0.091 \times \log 0.091\\ &\quad -0.246 \times \sum_{i=1}^{17} 0.059 \times \log 0.059\\ &\quad -0.087 \times \sum_{i=1}^{6} 0.167 \times \log 0.167\\ &\quad -0.159 \times \sum_{i=1}^{11} 0.091 \times \log 0.091\\ &\quad \left.-0.101 \times \sum_{i=1}^{7} 0.143 \times \log 0.143\right)\\ &= 2.48.\end{aligned}$$

Similarly, we can get the Shapely values of other attributes.

$$\Delta_B(S) = \text{entropy}(S) - \text{entropy}(S|B) = 1.76;$$

$$\Delta_C(S) = \text{entropy}(S) - \text{entropy}(S|C) = 1.75;$$

$$\Delta_D(S) = \text{entropy}(S) - \text{entropy}(S|D) = 1.69;$$

$$\Delta_E(S) = \text{entropy}(S) - \text{entropy}(S|E) = 1.53;$$

$$\Delta_F(S) = \text{entropy}(S) - \text{entropy}(S|F) = 2.26.$$

As a result, we present the attribute A that has the highest Shapley value to the customer.

STEP 3. After getting the customer's specification A1, we can present the corresponding recommendation just by checking the conditional probability table. The independence assumption stated in the last session is used here to present the recommendation. The following recommendation can be yielded:

$$\arg\max_B P(B_i|A = A1) = B2; \quad \arg\max_C P(C_i|A = A1) = C3;$$

$$\arg\max_D P(D_i|A = A1) = D3; \quad \arg\max_E P(E_i|A = A1) = E2;$$

$$\arg\max_F P(F_i|A = A1) = F3.$$

Because the output recommendation <1, 2, 3, 3, 2, 3> differs from the target configuration <1, 2, 2, 3, 2, 2 >, further processing is required.

STEP 4. the Shapley values given that A1 is selected are calculated and result to

$$\Delta_{B|A1}(S) = \text{entropy}(S|A1) - \text{entropy}(S|A1, B) = 1.45;$$

$$\Delta_{C|A1}(S) = \text{entropy}(S|A1) - \text{entropy}(S|A1, C) = 1.61;$$

$$\Delta_{D|A1}(S) = \text{entropy}(S|A1) - \text{entropy}(S|A1, D) = 1.55;$$

$$\Delta_{E|A1}(S) = \text{entropy}(S|A1) - \text{entropy}(S|A1, E) = 1.16;$$

$$\Delta_{F|A1}(S) = \text{entropy}(S|A1) - \text{entropy}(S|A1, F) = 2.26.$$

The highest Shapley value is the one for the attribute F, which we therefore present to the customer to configure.

STEP 5. After getting the customer's specification F2, the following recommendation can be reached by checking the conditional probability table.

$$\arg\max_B P(B_i|A = A1, F2) = B2;$$

$$\arg\max_C P(C_i|A = A1, F2) = C3;$$

$$\arg\max_D P(D_i|A = A1, F2) = D3;$$

$$\arg\max_E P(E_i|A = A1, F2) = E3.$$

Thus, <1, 2, 3, 3, 3, 2> is recommended.

STEP 6. Because the recommendation is still not satisfactory, the previous attribute selection process should be repeated until the recommendation meets the requirement.

In summary, the configuring process for this customer is shown in Table 3. After three configuration rounds, the customer gets his target PC. Section 6.2 presents the experiment results to show the advantage of the proposed method.

6.2. Performance comparison

In this section, we compare the performance of the proposed approach with other attribute selection and recommendation methods. Let "FixSeq + PRP" represent the method using fixed query sequence and PRP based recommendation. Ac-

Table 3. *The specification defining process for the customer with preference {A1, B2, C2, D3, E2, F2}*

Specification Round	Information Gain	Proposed Item	Recommendation	Target
1	$E_A = 2.48$, $E_B = 1.76$, $E_C = 1.75$, $E_D = 1.69$, $E_E = 1.53$, $E_F = 2.26$	CPU (A)	<1, 2, 3, 3, 2, 3 >	No
2	$E_B = 1.45$, $E_C = 1.61$, $E_D = 1.55$, $E_E = 1.16$, $E_F = 2.26$	Display card (F)	<1, 2, 3, 3, 3, 2 >	No
3	$E_B = 0.92$, $E_C = 0.92$, $E_D = 0$, $E_E = 0.92$	Monitor (C)	<1, 2, 2, 3, 2, 2 >	Yes

cording to the number of attributes, there are $6! = 720$ possible query sequences for this PC configurator. We calculate the average configuration rounds of all the possible sequences (AverFixSeq + PRP). The results of two arbitrarily selected sequences ("FixSeq1 + PRP" and "FixSeq2 + PRP") are also presented for comparison. The other approach uses the Shapley value based attributes selection method addressed in this paper but recommends the configuration randomly (Shapley + Rand). The proposed approach is abbreviated as "Shapley + PRP."

The products in the testing set generated by perturbative bootstrap are used as customers' targets. In previous section's example, only one recommendation is provided in each round. In this numerical analysis, multiple products are recommended according to their probabilities of relevance. If a customer's target PC is in the set of recommendations, the process will end and the corresponding configuration rounds will be recorded. The number of configuration rounds is used as the measure of efficiency. It can be anticipated that if the whole framework performs better, then fewer rounds of communications will occur. Figure 2 shows the experiment results under different approaches.

The x axis represents the number of recommendations in each round and y axis is the number of recommendations rounds needed for the customer to find the target product. Because there are six attributes altogether in this example, the worst case requires six configuration rounds. We can see that the "Shapley + PRP" approach proposed in this paper outperforms others. When more recommendations are presented, the configuration rounds are also decreased, because bigger recommendation set is more likely to contain the target product. The experiment results show that our approach provides a promising direction of improving the efficiency of product configuration.

7. CONCLUSION

This paper recasts the attribute selection task in configurator design from game theory's point of view. Shapley values are adopted to measure the usefulness of different attributes.

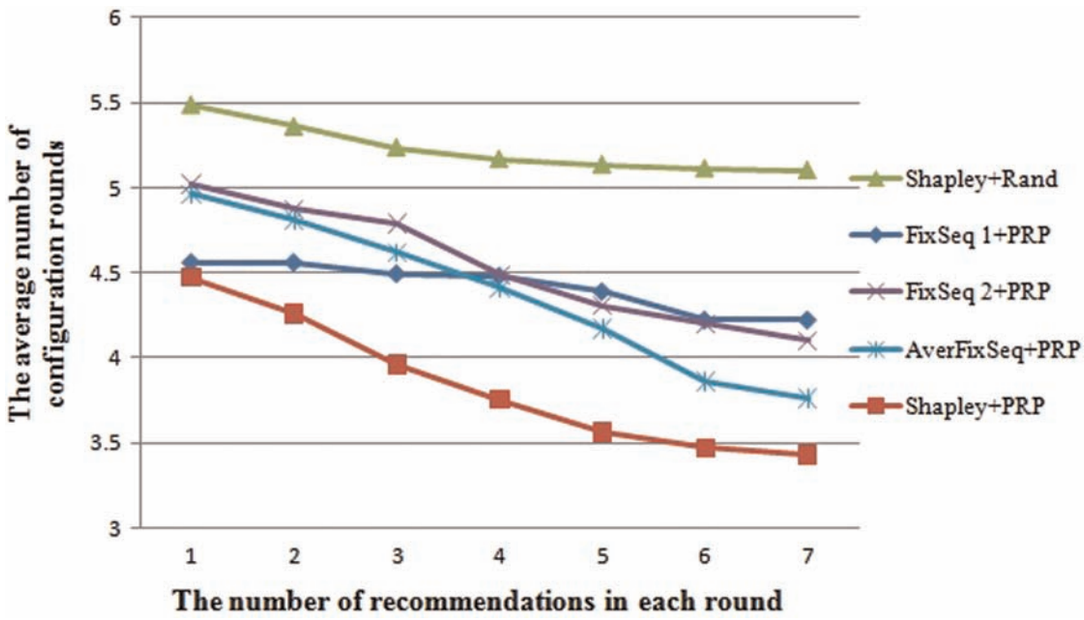

Fig. 2. A comparison of the configuration approaches. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

The most relevant attribute is selected for customers to configure during product configuring procedure. The main contributions are as follows:

1. A product configuring process is treated as a sequential decision making procedure. In each configuring round, the most uncertainty about a customer's needs is eliminated. The interactive process runs in an adaptive manner in the sense that different customers will have different query sequences. This offers a brand new perspective for us to understand configurator issues in product customization context.
2. PRP is adopted for product recommendation. It could shield customers from the tedious process of screening and making a selection from a vast number of products and thus overcome the information overload issue. Analytical results show that PRP is optimal with respect to expected search length. The efficiency of matching between demand and supply is thus enhanced.

The presented method also has some limitations and can be enriched along several dimensions. It functions well for customers who have enough expertise and can clearly configure each attribute. For customers who only have vague functional requirements, there are no links between the customers' needs and the detailed attributes in this framework. The configuration task is hard to conduct. How to incorporate customer needs in fuzzy functional requirements form into the configuration task is a future research direction. In addition, this paper assumes the product configuration space is fixed. Innovation and evolvement in a product family are not considered. Apparently, it is not profitable to start from scratch again to collect data and implement the approach addressed in this paper. One potential solution is to adapt previous configuration data to the updated product family via some econometric methods. Another direction is to improve the computational complexity of the configuration design task. If the product contains m attributes and each attribute has n attribute alternatives, then the computational complexity of the proposed attribute selection task is $O(m^2n)$. How to improve the computational complexity remains to be a practical and significant research issue.

ACKNOWLEDGMENTS

This research is supported by the Hong Kong Research Grants Council (RGC CERG HKUST 620308 and 620609).

REFERENCES

Birmingham, W., Brennan, A., & Siewiorek, D. (1988). MICON: a single board computer synthesis tool. *IEEE Circuits and Devices Magazine 4(1)*, 37–46.

Burke, R. (2002). Interactive critiquing for catalog navigation in E-Commerce. *Artificial Intelligence Review 18(3–4)*, 245–267.

Burke, R., Hammond, K., & Young, B. (1996). Knowledge-based navigation of complex information spaces. *Proc. 13th National Conf. Artificial Intelligence*, pp. 462–468. Portland, OR: AAAI Press/MIT Press.

Burke, R., Hammond, K., & Young, B. (1997). The Findme approach to assisted browsing. *Journal of IEEE Expert 12(4)*, 32–40.

Cohen, S., Dror, G., & Ruppin, E. (2007). Feature selection via coalitional game theory. *Neural Computation 19*, 1939–1961.

Cox, D.R. (1970). *The Analysis of Binary Data*. London: Methuen.

Darr, T., & Birmingham, W.. (2000). Part-selection triptych: a representation, problem properties and problem definition, and problem-solving method. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 14(1)*, 39–51.

Durrett, R. (2003). *Probability: Theory and Examples*, 3rd ed. New York: Thomson Learning/Brooks–Cole.

Efron, B. (1979). Bootstrap methods: another look at the jackknife. *Annals of Statistics 7(1)*, 1–26.

Elturky, F., & Nordin, R. (1986). BLADES: an expert system for analog circuit design. *Proc. IEEE Symp. Circuit and System*, pp. 552–555, San Jose, CA.

Enos, L. (2001). Report: five keys for e-tail success. *E-Commerce Times.* Accessed at http://www.ecommercetimes.com/story/7743.html

Frayman, F., & Mittal, S. (1987). Cossack: a constraint based expert system for configuration task. In *Knowledge-Based Expert Systems in Engineering: Planning and Design* (Sriram, D., & Adey, R., Eds.), pp. 143–166. Boston: Computational Mechanics Publication.

Gelle, E., & Faltings, B. (2003). Solving mixed and conditional constraint satisfaction problems. *Constraint 8(2)*, 107–141.

Hüllermeier, E. (1997). *Case-based search techniques for solving configuration problems.* Accessed at http://citeseer.ist.psu.edu/old/79018.html

Kamakura, W.A., & Wedel, M. (1995). Life-style segmentation with tailored interviewing. *Journal of Marketing Research 32*, 308–317.

Keinan, A., Sandbank, B., Hilgetag, C., Meilijson, I., & Ruppin, E. (2004). Fair attribution of functional contribution in artificial and biological networks. *Neural Computation 16(9)*, 1887–1915.

Kurniawan, S., Tseng, M., & So, R. (2003). Consumer decision making process in mass customization. *Proc. 2003 World Congress on Mass Customization and Personalization*, p. 38, Munich.

Lilien, G., Kotler, P., & Moorthy, K. (1992). *Marketing Model*. Englewood Cliffs, NJ: Prentice–Hall.

Löckenhoff, C., & Messer, T. (1994). Configuration. In *CommonKADS Library for Expertise Modelling* (Breuker, J., & Van de Velde, W., Eds.), pp. 197–212. Amsterdam: IOS Press.

Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12(4)*, 383–397.

McCutcheon, D., Raturi, A., & Meredith, J. (1994). The customization-responsiveness squeeze. *Sloan Management Review 35(2)*, 89–99.

McDermott, J. (1980). R1: an expert in the computer systems domain. *Proc. 1st Annual National Conf. Artificial Intelligence*, pp. 269–271, Stanford University.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. American Association for Artificial Intelligence*, pp. 25–32, Boston.

Mittal, S., & Frayman, F. (1989). Towards a generic model of configuration task. *Proc. Int. Joint Conf. Artificial Intelligence*, pp. 1395–1401. San Mateo, CA: Morgan Kaufmann.

Piller, F., & Moeslein, K. (2004). Does mass customization pay? An economic approach to evaluate customer integration. *Production Planning & Control 15(4)*, 435–444.

Rahmer, J., & Voß, A. (1996). *Case-based reasoning in the configuration of telecooperation systems*, pp. 93–98, AAAI Technical Report FS-96-03. Menlo Park, CA: AAAI Press.

Reilly, R., McCarthy, K., McGinty, L., & Smyth, B. (2004). Dynamic critiquing. *Advances in Case-Based Reasoning*, pp. 763–777. Berlin: Springer.

Sabin, D., & Freuder, E. (1996). Configuration as composite constraint satisfaction. *Proc. 1st Artificial Intelligence and Manufacturing Research Planning Workshop*, pp. 153–161. Menlo Park, CA: AAAI Press.

Sabin, M., & Freuder, E. (1998). *Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems.* Accessed at http://citeseer.ist.psu.edu/sabin98detecting.html

Sabin, D., & Weigel, R. (1998). Product configuration frameworks—a survey. *IEEE Intelligent Systems 13*, 42–49.

Salvador, F., & Forza, C. (2004). Configuring products to address the customization-responsiveness squeeze: a survey of management issues and opportunities. *International Journal of Production Economics 91(3)*, 273–291.

Schierholt, K. (2001). Process configuration: combining the principles of product configuration and process planning. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 15(5)*, 411–424.

Schwartz, B. (2004). *The Paradox of Choice: Why More Is Less*. New York: ECCO.

Shannon, C.E. (1948). A mathematical theory of communication. *Bell System Technical Journal 27(7)*, 379–423.

Shapley, L. (1953). A value for n-person games. In *Contributions to the Theory of Games*, Vol. 1 (Kuhn, H.W., & Tucker, A.W., Eds.). Princeton, NJ: Princeton University Press.

Simonson, I. (2005). Determinants of customers' responses to customized offers: conceptual framework and research propositions. *Journal of Marketing 69(1)*, 32–45.

Stumptner, M., Friedrich, G., & Haselböck, A. (1998). Generative constraint-based configuration of large technical systems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12(4)*, 307–320.

Tseng, H., Chang, C., & Chang, S. (2005). Applying case-based reasoning for product configuration in mass customization environments. *Expert Systems With Applications 29(4)*, 913–925.

Tseng, M., & Piller, F. (2003). New direction for mass customization. In *The Customer Centric Enterprise* (Piller, F., & Tseng, M., Eds.), pp. 519–535. Berlin: Springer.

van Rijsbergen, C.J. (1979). *Information Retrieval*. London: Butterworths.

Wang, Y., & Tseng, M.M. (2008). Incorporating probabilistic model of customers' preferences in concurrent engineering. *Annals of the CIRP 58(1)*, 137–140.

Wang, Y., & Tseng, M.M. (2009). Attribute selection for configurator design based on Shapley value. *Proc. ASME 2009 Int. Design Engineering Technical Conf., Computers and Information in Engineering Conf. (IDETC/CIE 2009)*, Paper No. DETC2009-86904.

Wielinga, B., & Schreiber, G. (1997). Configuration-design problem solving. *IEEE Expert: Intelligent Systems and Their Applications 12(2)*, 49–56.

Yu, B., Skovgaard, H.. & Baan Frond Office Systems. (1998). A configuration tool to increase product competitiveness. *IEEE Intelligent Systems 13(July/August)*, 34–41.

Yue Wang received his PhD from the Department of Industrial Engineering and Logistics Management at the Hong Kong University of Science and Technology and his BS and MS degrees in electronic engineering from Peking University, Beijing. Dr. Wang's research interest is focused on engineering design and manufacturing, artificial intelligence, and applied statistics.

Mitchell M. Tseng is a Professor and the Director of the Advanced Manufacturing Institute and Zhejiang Advanced Manufacturing Institute of Hong Kong University of Science and Technology. Prof. Tseng joined Hong Kong University of Science and Technology as the founding department head of industrial engineering in 1993 after holding executive positions at Xerox and DEC. He previously held faculty positions at the University of Illinois at Urbana–Champaign and the Massachusetts Institute of Technology. Dr. Tseng received MS and PhD degrees in industrial engineering from Purdue University and a BS degree from National Tsing Hua University. He is a fellow of the International Academy of Production Engineers (CIRP) and ASME.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 197–206.

doi:10.1017/S0890060410000636

The impact of product configurators on lead times in engineering-oriented companies

ANDERS HAUG,[1] LARS HVAM,[2] AND NIELS HENRIK MORTENSEN[3]

[1]Department of Entrepreneurship and Relationship Management, University of Southern Denmark, Kolding, Denmark
[2]Department of Management Engineering, Technical University of Denmark, Lyngby, Denmark
[3]Department of Management, Technical University of Denmark, Lyngby, Denmark

(RECEIVED March 17, 2010; ACCEPTED October 29, 2010)

Abstract

This paper presents a study of how the use of product configurators affects business processes of engineering-oriented companies. A literature study shows that only a minor part of product configuration research deals with the effects of product configuration, and that the ones that do are mostly vague when reporting the effects of configurator projects. Only six cases were identified, which provide estimates of the actual size of lead time reduction achieved from product configurators. To broaden this knowledge, this paper presents the results of a study of 14 companies concerning the impact of product configurators on business processes related to the creation of quotes and detailed product specifications. The study documents impressive results of the application of configurator technology. For example, in the data retrieved the use of configurators was estimated to have implied up to a 99.9% reduction of the quotation lead time with an average estimated reduction of 85.5%.

Keywords: Configurator; Lead Times; Process Reengineering; Product Configuration; Sales Configuration

1. INTRODUCTION

Product configurators represent one of the most successful applications of artificial intelligence principles (Stumptner, 1997; Sabin & Weigel, 1998; Blecker et al., 2004). A product configurator is a software-based expert system that supports the users in the specification of customized products by providing design choices for the user while restricting how different elements and their properties may be combined. Thus, the use of configurator technology means that product specification tasks, which normally require human experts, can be automated. In many cases, product configurators have been used for automating the creation of quote prices, sales prices, bills of materials, and other product specifications.

Product configurators can be divided into two main classes: those used for the specification of products that are traditionally mass produced and those aimed at products that are traditionally engineered (Haug et al., 2009). Configuration of products that are traditionally mass produced implies very little complexity of the knowledge base of the configurator compared to configurators aimed at engineered products, which can include thousands of rules for how elements and properties may be combined. The focus of this paper is on configurators that support products that typically require engineering work for each customer order. In engineering-oriented companies, the use of product configurators has resulted in a range of benefits such as shorter lead times, improved quality of product specifications, preservation of knowledge, use of fewer resources for specifying products, optimized products, less routine work, improved certainty of delivery, and less time needed for training new employees (Felfernig et al., 2000; Forza & Salvador, 2002*a*; Ardissono et al., 2003; Hvam, 2004; Piller et al., 2004; Helo, 2006).

Configurators can automate much of the work of human experts in sales and design processes, which implies that large reductions of lead times can be achieved. Lead time reduction is actually one of the most mentioned effects of using product configurators, as the literature review in the subsequent section of this paper shows. However, although this type of effect is often mentioned, only little empirical evidence has been provided. It seems that no major studies that investigate such effects in a detailed manner have been carried out. Furthermore, the few studies that do provide quantitative descriptions of lead time reductions as a consequence of using configurators are not fully comparable because of unclear research methods and different focus. Thus, existing research does not provide a basis for making solid generalizations about lead time reduc-

Reprint requests to: Anders Haug, Department of Entrepreneurship and Relationship Management, University of Southern Denmark, Engstien 1, Kolding 6000, Denmark. E-mail: adg@sam.sdu.dk

tions in successful product configurator projects. To contribute to the knowledge of the effects of configurators on lead times, this paper answers the question: What are the effects of using configurators in terms of reduction of lead time duration and man hours in engineering-oriented companies? The question is answered based on studies of 14 companies.

The remainder of the paper is structured as follows. Section 2 investigates relevant literature on reduction of lead times as a consequence of using product configurators, and Section 3 discusses the changes of business processes implied by the use of configurators. Section 4 describes the method applied for conducting the study of 14 companies, an Section 5 presents the results of the study. The paper ends with a conclusion in Section 6.

2. LITERATURE STUDY

The literature study has the purpose of clarifying what existing configurator research has to say about the effects of product configurators. The literature was found by searching relevant databases of academic journals (including all Institute for Scientific Information indexed papers), conference proceedings, and PhD projects. The search terms used were "configurator" and "product configuration," with results delimited to relevant areas of research. More than 100 configuration-related papers were found. However, the majority of this literature deals with proposition of methods, tools, and techniques, whereas empirically based studies of the effects of configurators in the companies using this technology are rare. The literature presented in the following subsection is based on two delimitations: it includes only literature that deals explicitly with product configuration/configurators; it includes only literature that deals with the effects of configurators in engineering-oriented companies. To illustrate the vagueness of descriptions of the effects of configurators found in most relevant research, the following subsection provides quotes from the relevant papers.

2.1. Literature

Barker et al. (1989) describe the configurators at Digital Equipment Corporation. These configurators are used to validate the technical correctness (configurability) of customer orders and to guide the actual assembly of these orders, that is, computers and computer room layout and networks. They mention that "overall the net return to Digital is estimated to be in excess of $40 million per year." Furthermore, the effects of the configurator are mentioned: "contributing to customer satisfaction, lower costs, and higher productivity"; "ensures that complete, consistently configured systems are shipped to the customer"; "simplifies field and manufacturing training needs and avoids confusion about new products that can delay time-to-market significantly"; "increases manufacturing's flexibility"; "increased the technical accuracy of orders entering manufacturing"; "assures that when the components of the order come together for the first time at the customer site the system will work"; and "major positive impact on cycle times, inventory levels, and manufacturing costs."

Heatley et al. (1995) describe the case of Carrier Corporation, a major air-conditioning manufacturer. Carrier introduced a configurator that is capable of configuring a set of part numbers for a particular air-conditioning equipment series based on customer request. The configurator was conceived for use by salespeople to support the process of filling orders. Heatley et al. describe a number of effects of the configurator, for example, the order throughput cycle was reduced from 6 days to 1 day, the number of manufacturable orders increased from 40% to 100%, and the incidence of pricing errors in orders was reduced from 80% to none. In relation to sales, among others, the following benefits are mentioned: elimination of non-value added overhead, reduced warranty and factory rework by $100,000 annually, improved customer satisfaction, improved morale of sales force, and a doubling of the sales engineers' selling time.

Ariano and Dagnino (1996) describe a case in which a manufacturer of modular wooden office furniture applies a configurator for the creation of bills of materials. They mention the following benefits achieved from the configurator: "a new and more organized way of structuring the company's product line"; "allows for a more consistent, faster, easier, and more comprehensive way to enter an order"; "while the order is entered, the system verifies that the configuration of the products is correct and compatible with the company's offerings"; "helps in quoting an accurate pricing to the company's products"; and "implies a reduction in the duplication of information, pricing deviations, and configuration inconsistencies."

Fleischanderl et al. (1998) from Siemens describe the use of the Lava configurator for configuring large telecommunication systems. They state that process gains implied a return on investment within the first year of use. In addition, they claim that the configurator has "improved the quality of the configuration results," helps with "avoiding error-prone manual editing of parameters," has "revealed numerous errors, such as cables having wrong length codes," and "makes the knowledge about the EWSD [telecommunication systems] configuration explicit."

Forza and Salvador (2002*a*) present a case study of a small company that produces voltage transformers. They mention the following effects of the introduction of a configurator: a "reduction to almost zero of the errors in the configurations released by the sales office"; "reducing the total time necessary for generating the tender"; made it "possible to recover a notable volume of man-hours, which freed part of the sales personnel for tasks with greater additional value"; "made it possible to increase technical productivity, both as regards product documentation release and design activities"; an "increase in technical department productivity"; a "formalization of the company knowledge"; and enabling "the transformation of individual competencies into organizational competencies." Finally, they state that "product configurators reduce the risk to lose a strategic competence because of departure of a key employee."

Forza and Salvador (2002*b*) present a case study concerning the implementation of product configuration software in a small manufacturing company that produces mold bases for plastics molding and punching bases for metal sheet punching. The implementation of the product configuration software resulted in two main kinds of advantages: reduction of manned activities in the tendering process (tendering lead time from 5–6 days to 1 day) and an increase in the level of correctness of product information (almost 100%). They state that the configurator "in turn would reduce the eventual distortions in the company–customer communication channel, reducing the chance of delivering a product that does not conform to the customer needs" and "the pay-off for the customers, besides the positive effect of better coordination, is the reduced time in generating product specifications and drawings." Finally, they argue that the case study shows that the company obtained both a rapid payback of the investment in configuration technology as well as a competitive advantage, and the configurator can be propagated to departments not directly involved in the implementation. In addition, the resulting new workflow can also affect the organization of the customers, that is, interfirm coordination.

Raatikainen et al. (2004) present results of a case study undertaken in two companies that develop and deploy configurable software product families. They state that for both companies the configurable software product family approach "seemed an efficient way to systemize the software development and enable an efficient control of versions and variants in a set of systems," but that "neither of the companies had estimates of investment payback times or other economic justifications when compared with, for example, project-based software development." They further claim that the configurable software product family approach "enabled the companies to delay variability binding to installation and even operation time" and "by using the configurators, the companies were able to deploy products in such a way that, in practice, there is no software engineering knowledge needed." Finally, they argue that their study shows that it is feasible to systematically develop families of software and manage the variability within the software family.

Hvam et al. (2002, 2004) describe the configurator project of Demex Electric, a Danish manufacturer of electronic switchboards. Hvam et al. (2002) summarize the effects of introducing a configurator in Demex Electric/Solar A/S as a "reduction of lead time from 3–4 days to 10 minutes when generating quotes," "up to 10% reduction of materials," and a "huge reduction in specification hours."

Hvam (2004, 2006*b*) describes the case of American Power Conversion (APC), a producer of data center infrastructure such as uninterruptible power supplies, battery racks, power distribution units, racks, cooling equipment, and accessories. APC uses configurators for working out quotations and manufacturing specifications. On the effects of the configurators it is mentioned that "products are sold through the product configuration system, which makes it possible for APC to control a huge amount of sales personnel around the world"; "the product configuration, including the work out of quotations and manufacturing specifications, is carried out by the configuration system saving considerably resources"; "the lead time for making quotations and manufacturing specifications is reduced from weeks to hours"; and "the product configuration systems make it easier to introduce new versions of the products to the sales personnel and the customers." In the context of large complex infrastructure systems for data centers, Hvam (2006*b*) states that the use of mass customization and configurators has implied a "reduction of the overall delivery time for a complete system from around 400 to 16 days."

Hvam (2004, 2006*a*) and Hvam et al. (2006) describe the case of FLSmidth, a manufacturer of large processing plants for cement production. Hvam (2006*a*) states that the application of a configurator "has enabled FLSmidth to reduce resources for the elaboration of quotations by 50%"; "means that sales representatives do not have to burden engineering specialists with the elaboration of budget quotations"; implied that "the period from a client request to the signing of the final contract has been considerably reduced"; "enables FLSmidth to respond to all requests with a quotation"; implies "more structured negotiations with the customer"; implies that "budget quotations become more homogeneous and of better quality"; "ensures that the sales person obtains all the necessary information before the budget quotation is made"; "means that a quotation can be made at an early stage with only very little customer input"; implies "it becomes possible to simulate different solutions for the customer"; "enables the company to optimise the cement plant in relation to parts already constructed and in use within the FLS group"; implies that "customers can be led to select FLSmidth standard solutions instead of specialised/customised solutions"; and is "a major means of internal knowledge sharing." Hvam (2004) states that "a gap analysis indicated that the lead time for making budget quotations could be reduced from 3–5 weeks to 1–2 days, the resources spent could be reduced from 15–25 man-days to 1–2 man-days," and Hvam et al. (2006) state that "the usage of engineering resources for developing a budget quotation is reduced from 5 MW to 0.2 MW, and the lead time is lowered from several weeks to a few days."

A contribution to a more general picture of the effects of product configurators in engineering-oriented companies is offered by a research project on product configuration / that emerged from this project. (Edwards et al., 2005) and the papers Danish on product configuration / that emerged from this project. The project was carried out during the period of 2003 to 2004 and includes studies of 12 Danish firms that were using product configurators at the time of the investigation. Based on this project, Pedersen and Edwards (2004) present the results of the 12 companies' answers to the realized effects of their configurator projects, as shown in Figure 1. The firms gave scores from 1 to 5 (1 = *very small* and 5 = *very large*) and 0 = *without influence*. As seen, the three top scorers are lower turnaround time (average $\sim$ 3.6), improved quality (average $\sim$ 4.4), and less use of resources (average $\sim$ 3.3).

Forza et al. (2006) present a case study of a company that produces electric motors. The case shows how the right

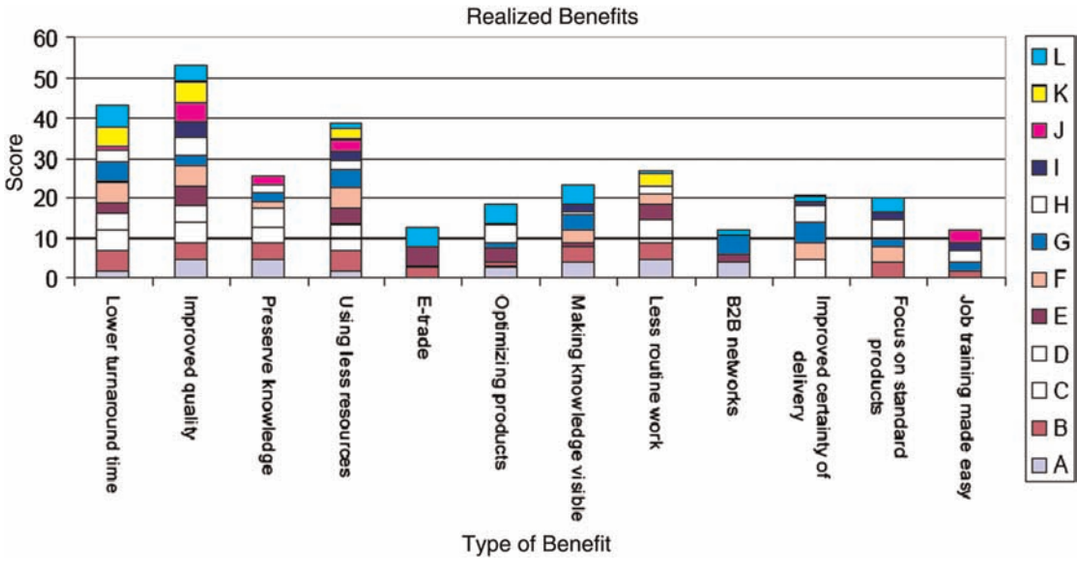

Fig. 1. The benefits realized from configurator projects (Pedersen & Edwards, 2004). [A color version of this figure can be viewed online at journals.cambridge.org/aie]

grouping of components (into kits) has enabled the company to implement a product configurator and to postpone product differentiation along the material flow. They state that the configurator "enhances product assortment communication"; "makes it easier and faster to explore the solution space offered by the company"; "enables a faster, accurate generation of a feasible offer without consulting the technical office"; "enables a faster, accurate creation of product code, BOM, and production cycle"; "allows storage of a large amount of customer data collected during the exploration and configuration phases"; and "allows rapid retrieval of past configurations for maintenance or repair purposes."

Petersen et al. (2007) describe the case of Aalborg Industries, a company that specializes in steam and heat generating equipment for maritime and industrial applications. The company has implemented a product configurator to render the sales-delivery process more efficient. Petersen at al. (2007) state that because of the configurator the company is "gaining significant benefits, and has learned much about the challenges of implementing product configuration in ETO."

Hong et al. (2008) describe the case of Gienow Windows and Doors, a Canadian manufacturing company of windows and doors. This company has introduced a configurator with the purpose of modeling the designs based on customer needs, creating requirements of materials, machines, and personnel, and identifying the optimal production schedule. Hong et al. (2008) state that "the lead time from a customer order to the product delivery has been reduced to 3 weeks compared to the average of 2 months in this industry."

Ladeby (2009) describes the configurator project at NNE Pharmaplan, a Danish supplier of systems, consultancy, and engineering services to the international pharmaceutical and biotechnical industry. The configurator is primarily a 3D visualization system for plant layout, and it does not produce prices or detailed bills of materials. It is stated that a main benefit of the system is that "a customer should not wait for weeks before he sees drawings and illustrations of what has been agreed upon."

Ladeby (2009) describes the configurator project of GEA Niro, an international engineering company within the area of design and supply of spray drying plants. According to Ladeby (2009), the configurator of GEA Niro focuses on the quotation phase, and it is used in about 50% of the first quotations sent out to customers. He states that "the process of making quotations has become more standardised and formalised," "product knowledge has become more standardised," and the sales person "gets the whole quotation served on a plate and sends it to the customer." It is also mentioned that "preservation of knowledge has been a motivation for the configurator project."

2.2. Literature summary

As shown in the literature review in the previous section, most literature based on studies of configurator projects is rather vague when it comes to describing the effects of such projects in terms of the effects on business process length and resource consumption. Although much such literature talks about large reductions of lead times and similar, it is actually unclear if such reductions represent, for example, 10% or 99%. The six cases with the most accurately described effects of configurators on lead times are summarized in Table 1.

In the first four of the six cases in Table 1, it is clear that configurators have had massive effects on lead times in the quotation phase, that is, configurators in these cases are estimated to have reduced lead times between 65% and 99.4%. The last two cases refer to other types of lead times for which

Table 1. *Literature with quantified estimates of lead time reductions*

Literature	Case	Lead Time Effect
Heatley et al. (1995)	Carrier Corporation	Order throughput cycle from 6 days to 1 day
Forza & Salvador (2002*b*)	Voltage transformer manufacturer	Quote lead time from 5–6 days to 1 day
Hvam et al. (2002), Hvam et al. (2004)	Demex Electric	Creation of quote lead time from 3–4 days to 10 min
Hvam (2004, 2006*a*), Hvam et al. (2006)	FLSmidth	Creation of quote lead time gap analysis indicated a reduction from 3–5 weeks to 1–2 days (Hvam, 2004) and a reduction from several weeks to a few days (Hvam et al., 2006)
Hvam et al. (2004, 2006)	American Power Conversion	Overall delivery time for a complete system estimated to be reduced from around 400 to 16 days
Hong et al. (2008)	Gienow Windows and Doors	Lead time from a customer order to the product delivery reduced to 3 weeks compared to an average of 2 months in the industry

reason the reductions are not directly comparable. Thus, only four comparable cases were found.

3. DEFINITIONS

As a basis for carrying out the studies of the effects of product configurators, first some basic definitions were set forth on how relevant business processes could be affected. As mentioned, this paper focuses on what is referred to as "engineering-oriented companies." In this paper the term is used to describe engineer-to-order (ETO) companies, and the part of the assembly-to-order (ATO) and make-to-order (MTO) companies in which each customer order requires some engineering work, that is, companies that are not pure ATO or MTO but in between ETO and ATO or MTO (Olhager, 2003). Figure 2 shows the four traditional types of product delivery strategies.

In ETO companies a product is often defined in two major turns, namely, a high-level design in the sales phase and a detailed design phase upon acceptance of an offer. Typically configurators are only used for high-level design in ETO companies, because it would be extremely time consuming to define the solution space at a detailed level. An example of this is the FLSmidth case (Hvam, 2004, 2006*a*). At FLSmidth, potential customers provide some requirements (i.e., a form of high-level design) that FLSmidth, by using a configurator, converts into a high-level design and based on this a quote. If the quote is accepted, detailed design is initiated. In contrast, often in ATO companies that typically deal with somewhat simpler products, the detailed design is defined during the sales phase as a basis for calculating the price. An example of such a company is APC (Hvam, 2004, 2006*b*). At APC, configurators can produce a quote at a detailed design level, for example, for some of their emergency power supply systems (Hvam, 2006*b*). When focusing on the process from RFQ (request for quotation) to production planning, the two discussed process types can be illustrated in a principle manner as seen in Figure 3. The gray boxes symbolize the processes normally automated (or at least partly) by configurator technology, wheres the black boxes show processes that in principle could be automated but typically are not (Hvam et al., 2004, 2008; Edwards et al., 2005). The color of the process "detailed design" is gray and black because the "simple" part of the engineering work in this process is often automated due to the overlap between high-level and detailed design.

As seen in Figure 3, at the quote creation stage of process type 1, only high-level design has been carried out, whereas in process type 2 detailed design has been made at this stage. To illustrate the difference between the configurator outputs of such processes, this paper proposes a division based on the level of detail of the output, which is illustrated in a class diagram in Figure 4.

In engineering-oriented companies, the quote creation phase often lasts days or weeks without the use of configurators. For example, at APC the quote process for emergency power supply systems lasted weeks before the use of configurators, and the gains from automating the quotation process at APC implied a reduction of the quote lead time of more than 90% (Hvam et al., 2006*b*). This reduction can be explained by the

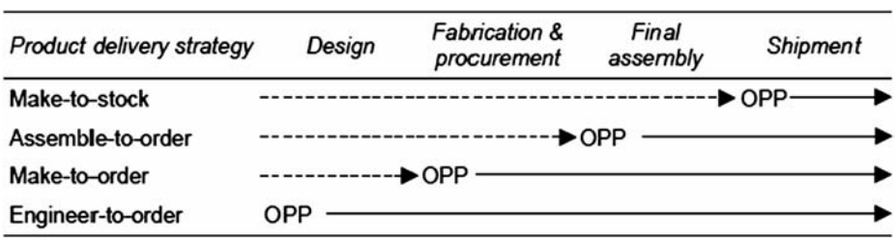

Fig. 2. The order penetration point (OPP; Olhager, 2003).

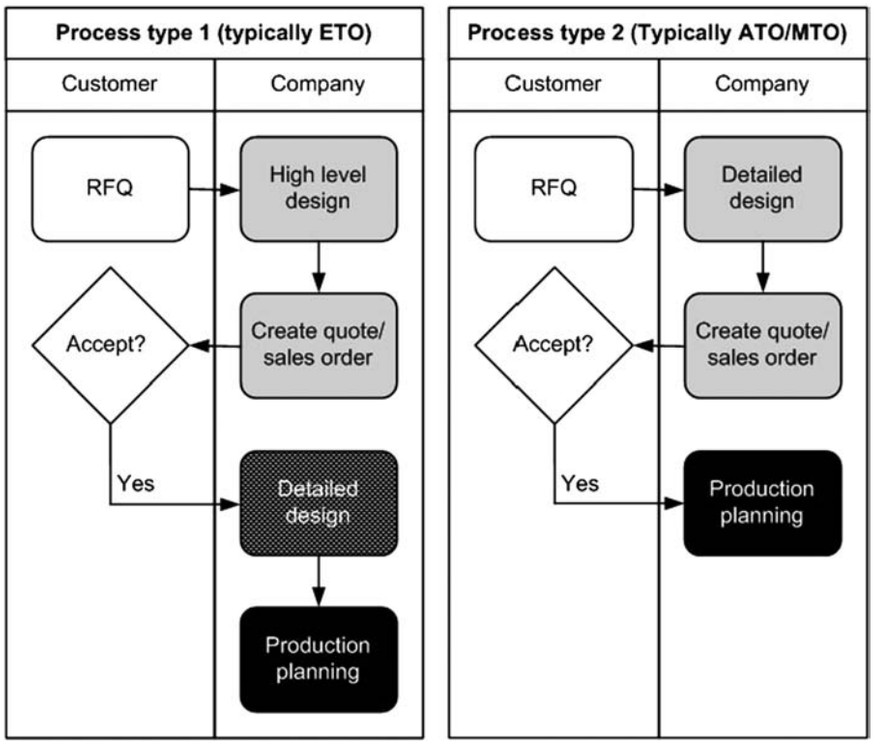

Fig. 3. Engineering-oriented customizers' processes. ETO, engineer-to-order companies; ATO, assembly-to-order companies; MTO, make-to-order companies; RFQ, request for quotation.

fact that the majority of the work associated with this process has been automated. However, the question is exactly which processes do configurators automate or reduce the duration of? To understand this question better, Figure 5 shows a generalized illustration of a quotation process. This is subsequently discussed.

As seen in Figure 5, the quotation generation process can be divided into three phases: initial product specification, further product specification, and quote creation. Besides the duration of the processes included in the figure, time can be added for waiting, handovers, and other internal communication. The use of configurators can affect all these activities. In

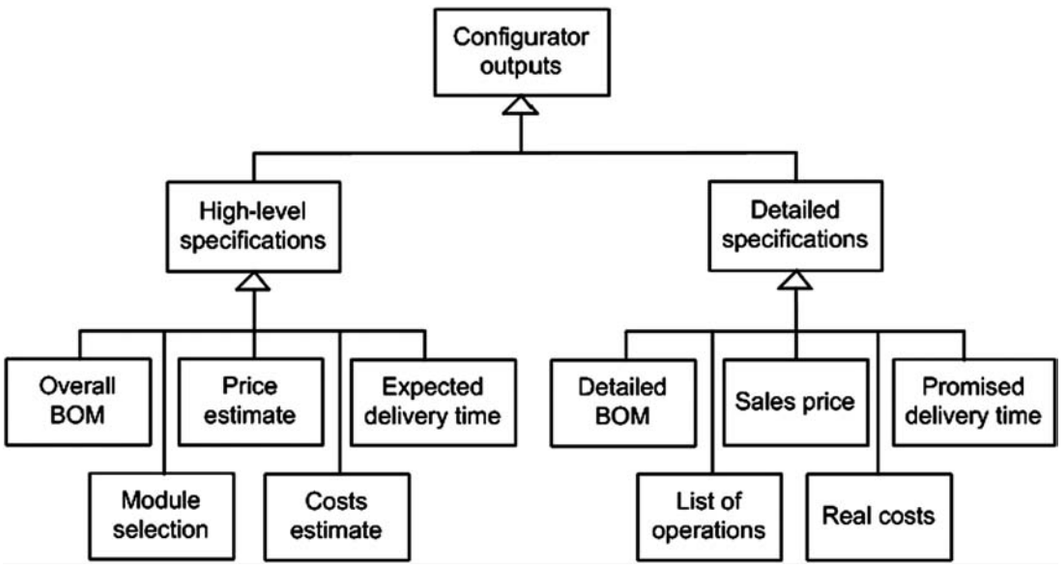

Fig. 4. The main configurator output types.

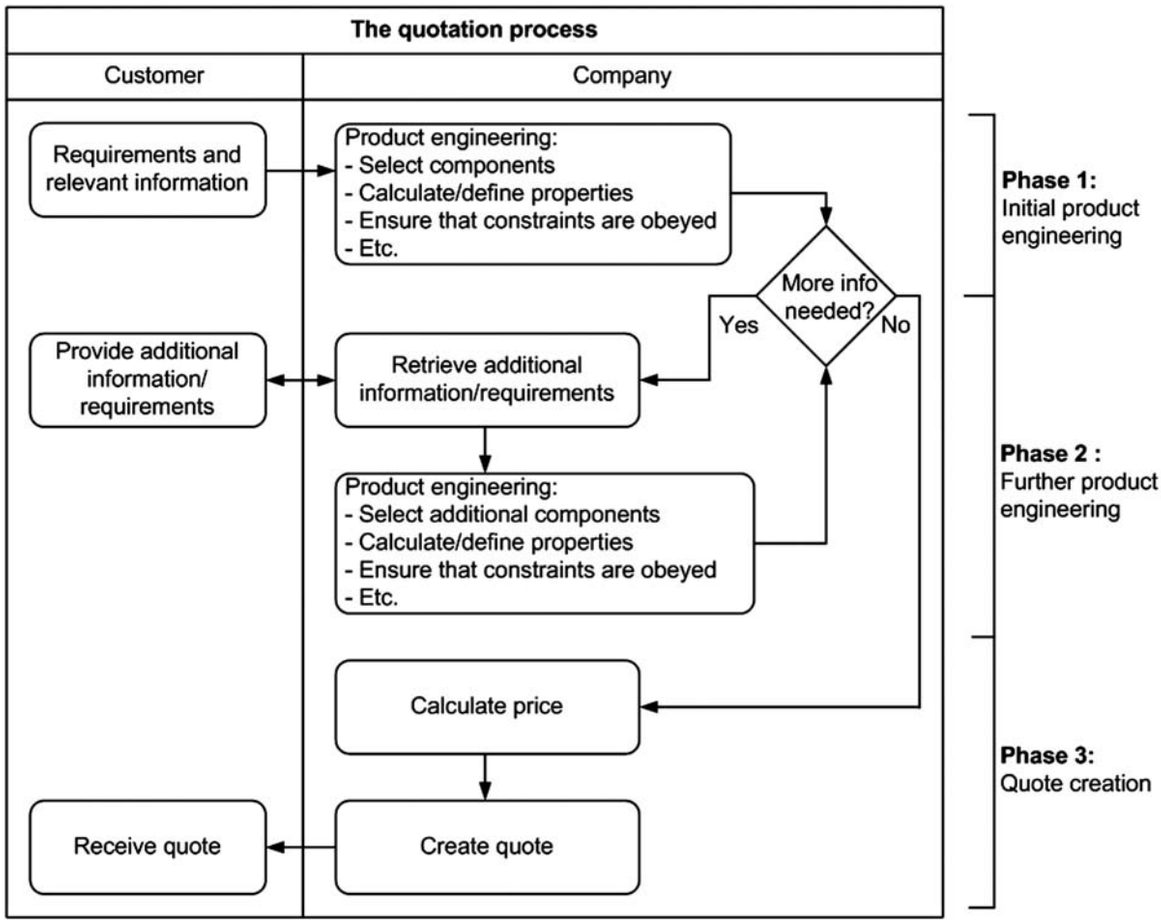

Fig. 5. The quotation process.

phase 1, a configurator directly reduces the duration of the process termed "product engineering." In many cases, configurators almost fully automate this phase. Subsequently, the duration of this process is reduced to the time it takes for the configurator to generate the relevant specifications, that is, minutes or seconds. In many cases, phase 2 comes into play when a need for further information occurs during the product engineering. Because a configurator works as a check list for needed information, a configurator may imply that phase 2 can be avoided or at least limit the number of loops in this phase. Normally phase 3 is also automated by a configurator, at least the part of calculating sales prices. Thus, it is also possible to significantly reduce the duration of phase 3.

4. RESEARCH METHOD

To investigate the effects of product configurators on the lead times of engineering-oriented companies, a study of the use of product configurators in the Danish industry was carried out during spring and early summer of 2009. The study was carried out as structured interviews of employees with knowledge of the configurator projects. The main reason for using interviews instead of a Web-based or paper-based questionnaire survey is that the area in focus is characterized by much unclear terminology. Therefore, the chosen approach allowed for the interviewer to clarify the meaning of questions not understood.

A total of 26 companies were interviewed. For this paper a sample of 14 companies was selected based on ability to estimate the effects on lead times from the use of configurators, and use of configurators that focus on products that are traditionally engineered. All these 14 companies produce business-to-business products, and in 9 of the 14 companies, several configurators were in operation. In the context of counting the number of configurators, a single configurator was defined as an operable software application that has an individual knowledge base. In most cases, such configurators were created by using the same standard configurator software shells. To be able to compare the data obtained from the different companies while focusing on relatively recent projects, the companies were told to focus on a configurator developed recently, preferably as complex and widely used as possible.

Table 2 shows the background information on the companies included and their configurators. As seen in Table 2, 11 of the 14 companies apply configurators for quotations and for creating the manufacturing basis, but 3 of these only use

Table 2. *Background information*

ID	Approx. No. of Employees	Product Types	Year of 1st Config.	No. of Config.	Turnover		Config. Support for Quotation Process	Config. Product Detail for Production
					Share From Customized Products	Part From Configured Products		
1	4500 (WW)	Plants, systems, components	2004	6	90	30	Yes	No
2	600 (DK)	Plants, systems, components	1985	10	90	50	Yes	Yes
3	50 (DK)	Components	1997	2	70	10	Yes	Yes
4	2800 (WW)	Systems	2001	1	90	30	Yes	Yes
5	5000 (WW)	Building parts and systems	2003	5	90	30	Yes	Yes
6	18,000 (WW)	Systems, components	2001	2	30	10	Yes	Yes
7	5000 (WW)	Plants, systems	1997	4	10	10	Yes	Yes
8	26,000 (WW)	Systems, components, spare parts	1995	2	5	85	Yes	No
9	10,000 (WW)	Building parts and systems	1999	1	10	100	Yes	Yes
10	3000 (WW)	Systems, components	1999	6	70	70	Yes	Yes
11	75 (DK)	Building parts and systems	2006	1	50	40	Yes	Yes
12	30 (DK)	Components	2006	1	50	20	Yes	No
13	2400 (WW)	Building parts and systems, service	1999	1	70	70	Yes	Yes
14	750 (WW)	Systems	2000	1	10	10	Yes	Yes

Note: WW, worldwide; DK, Denmark.

configurators for the quotation phase. Table 2 also shows the ratio between configured products (i.e., defined by use of a configurator) and customized products (i.e., user-specific products, not necessarily defined by using a configurator). Note that the percentage of configured products in 8 of the cases is smaller than the number of customized products, in 4 cases all customized products are configurable, and in 2 cases the number of configured products are higher than the percentage of customized products. The explanation for the latter phenomenon is that standard products are sold via a configurator; that is, it is the combination of products that is customized, not the products themselves.

5. RESULTS

This section presents the results of the study related to quotation and manufacturing, which are subsequently discussed.

Table 3 and Table 4 show the companies' answers to the effects of configurators for the creation of quotes as measured in duration and man-hour consumption, respectively. Note that the numbers represent generalized estimates made by key personnel.

As seen in Table 3, the lead time reductions estimated were rather significant. More specifically, 12 of the 13 companies that were able to answer this question provided estimates of a 75% to 99.9% reduction of quotation lead time. One company estimated a 50% reduction, but it should be noted that the original lead time in this case was only 60 min; therefore, the 30 min remaining may be related mainly to customer in-

Table 3. *Effects of the process duration on the quotation process*

ID	Without Config.[a]	With Config.[b]	Duration Reduction[c]
1	Do not know	Do not know	Do not know
2	2 days	9 min	99.9%
3	30 days	2 days	93.3%
4	5 days	2–8 h	80–95%
5	2–15 (avg. 8.5) h	10–180 min	64.7–98.0%
6	14 days	1 day	92.9%
7	50 days	2 days	96%
8	3 days	20 min	98.6%
9	2–5 (avg. 3.5) days	5 min to 8 h	71.4–99.7%
10	120 min	5 min	95.8%
11	60 min	30 min	50%
12	60 min	15 min	75%
13	4 h	1 h	75%
14	5 days	1 day	80%
Average			85.5%

[a]Duration of the creation of a quote before using the configurator.
[b]Duration of the creation of a quote using the configurator.
[c]Reduction of the duration of the creation of a quote.

Table 4. *Effects on the quotation process in man-hours*

ID	Without Config.[a]	With Config.[b]	Man-Hour Reduction[c]
1	20 h	2 h	90%
2	3–4 h	9 min	95.0–96.3%
3	Do not know	Do not know	Do not know
4	37 h	2–3 h	91.9–94.6%
5	2–15 (avg. 8.5) h	10–180 min	64.7–98.0%
6	4–5 h	1 h	75–80%
7	150 h	6 h	96%
8	1.5 h	20 min	77.8%
9	1 h	5–10 min	83.3–91.6%
10	2 h	5 min	95.8%
11	1 h	30 min	50%
12	2 h	1 h	50%
13	8 h	4 h	50%
14	5 days	1 day	80%
Average			78.8%

[a]Man-hours used for the creation of a quote before using the configurator.
[b]Man-hours used for the creation of a quote using the configurator.
[c]Reduction of the man-hours used for the creation of a quote.

Table 5. *Effects on the process of creating product specifications for manufacturing*

ID	Without Config.[a]	With Config.[b]	Duration Reduction[c]
1			
2	Do not know	Do not know	Do not know
3	3–13 (avg. 8) days	0–3 days	62.5–100%
4	Do not know	Do not know	Do not know
5	3–4 (avg. 3.5) h	0	100%
6	5 h	3–5 min	98.3–99.0%
7	Do not know	Do not know	Do not know
8			
9	Avg. 4 h	0	100%
10	60 min	30 min	50%
11	60–120 (avg. 90) min	30 min	66.7%
12			
13	1 day	1 min	99.8%
14	10 days	1.5 days	85%
Average			85.2%

[a]Duration of the creation of the product specifications for manufacturing before using the configurator.
[b]Duration of the creation of the product specifications for manufacturing using the configurator.
[c]Reduction of the duration of time for creation of the product specifications for manufacturing.

teraction. In general, the number of man-hours saved is a little lower than the lead time reduction, that is, 85.5% versus 78.8%. This difference may be explained by the fact that configurators cause fewer handovers. Fewer handovers imply less waiting time between tasks, which is time wherein no human action is required. Thus, the lead time is reduced whereas the use of man-hours is not, under the assumption that relevant personnel carries out other tasks while waiting.

Table 5 shows the estimates of the effects configurator use on the duration of the process of creating detailed product specifications as a basis for production.

As can be seen, 8 of the 11 companies that use configurator output as a basis for manufacturing were able to answer the question of lead time reduction in the product specification part of the production process. The average of these reductions was 85.2%. The high average reduction estimated can be explained by the fact that the configurator in the 8 companies produces most (if not all) of the product specifications needed for production during the quotation phase.

6. CONCLUSION

This paper has provided new insight into how product configurators may have an impact on business processes in engineering-oriented companies, more specifically, the impact of configurators on the quotation and production preparation processes.

This paper first reviewed the literature in order to clarify the effects of product configurators on lead times. However, despite including more than 100 papers in the review, only six cases were identified in which reports of lead time reductions in a quantitative manner were provided. Next the paper presented the result of structured interviews with 14 companies on the effects of product configurators on lead times. For quotation processes, the use of configurators in 12 of the 14 companies implied a 75% to 99.9% reduction of quotation lead time, whereas the last 2 companies experienced a 50% reduction or were not able to answer, respectively. The average lead time reduction relating to the creation of quotes was 85.5%, and the average man-hours saved represented a reduction of 78.8%. Concerning the creation of detailed product design specifications, 11 companies had this focus, and of these, 8 were able to answer the question of lead time reduction. The average size of these estimates of lead time reductions was 85.2%.

The results of this paper clearly show that engineering companies that successfully implement product configurators can achieve significant lead time and man-hour reductions in processes relating to quotation and production preparation. However, the creation of configurators is often a risky and highly time-consuming project. Thus, even if a 90% reduction of lead time and man-hours is achieved, this may still be an unprofitable project if the costs of achieving this are too high. In this context, note that configurators need continuous updates as the product assortment changes that may result in high maintenance costs. Therefore, instead of being blinded by the impressive effects of configurators documented by this study, this paper recommends to carefully estimate the expected costs before initiating such a project.

Based on the configurator literature identified in the literature review, it seems that the study presented in this paper is the first major study that has investigated in a detailed manner the impact of product configurators on business processes related to the creation of quotes and manufacturing-related product specifications. More specifically, only 6 cases in

which such effects have been quantified were found in literature. Thus, the data from the 14 cases of this paper represent a significant contribution to configuration literature.

REFERENCES

Ardissono, L., Felfernig, A., Friedrich, G., Goy, A., Jannach, D., Petrone, G., Schäfer, R., & Zanker, M. (2003). A framework for the development of personalized, distributed web-based configuration systems. *AI Magazine 24(3)*, 93–108.

Ariano, M., & Agnino, D. (1996). An intelligent order entry and dynamic bill of materials system for manufacturing customized furniture. *Computers and Electrical Engineering 22(1)*, 45–60.

Barker, V.E., O'Connor, D.E., Bachant, J., & Soloway, E. (1989). Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM 32(3)*, 298–318.

Blecker, T., Abdelkafi, N., Kreutler, G., & Friedrich, G. (2004). Product configuration systems: state of the art, conceptualization and extensions. *Proc. 8th Maghrebian Conf. Software Engineering (MCSEAI 2004)*, pp. 25–36.

Edwards, K., Hvam, L., Pedersen, J.L., Møldrup, M., & Møller, N. (2005). *Udvikling og implementering af konfigureringssystemer: Økonomi, Teknologi og Organisation* [Final Report From PETO Research Project]. Lyngby, Denmark: Technical University of Denmark, Department of Manufacturing Engineering and Management.

Felfernig, A., Jannach, D., & Zanker, M. (2000). Contextual diagrams as structuring mechanisms for designing configuration knowledge bases in UML. In *UML 2000—The Unified Modeling Language, Advancing the Standard, Int. Conf.* (Kent, S., & Selic, B., Eds.), LNCS, Vol. 1939, pp. 240–254. Heidelberg: Springer.

Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H., & Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems 13(4)*, 59–68.

Forza, C., & Salvador, F. (2002*a*). Managing for variety in the order acquisition and fulfilment process: the contribution of product configuration systems. *International Journal of Production Economics 76(1)*, 87–98.

Forza, C., & Salvador, F. (2002*b*). Product configuration and inter-firm coordination: an innovative solution from a small manufacturing enterprise. *Computers in Industry 49(1)*, 37–46.

Forza, C., Trentin, A., & Salvador, F. (2006). Supporting product configuration and form postponement by grouping components into kits: the case of MarelliMotori. *International Journal of Mass Customization 1(4)*, 427–444.

Haug, A., Ladeby, K., & Edwards, K. (2009). From engineer-to-order to mass customization. *Management Research News 32(7)*, 633–644.

Heatley, J., Agraval, R., & Tanniru, M. (1995). An evaluation of an innovative information technology—the case of Carrier EXPERT. *Journal of Strategic Information Systems 4(3)*, 255–277.

Helo, P.T. (2006). Product configuration analysis with design structure matrix. *Industrial Management & Data Systems 106(7)*, 997–1011.

Hvam, L. (2004). A multi-perspective approach for the design of Product Configuration Systems—an evaluation of industry applications. *Proc. Int. Conf. Economic, Technical and Organizational aspects of Product Configuration Systems (PETO)*, pp. 13–25.

Hvam, L. (2006*a*). Mass customisation of process plants. *International Journal of Mass Customisation 1(4)*, 445–462.

Hvam, L. (2006*b*). Mass customisation in the electronics industry based on modular products and product configuration. *International Journal of Mass Customisation 1(4)*, 410–426.

Hvam, L., Malis, M., Hansen, B., & Riis, J. (2004). Reengineering of the quotation process—application of knowledge based systems. *Business Process Management Journal 10(2)*, 200–213.

Hvam, L., Mortensen, N.H., & Riis, J. (2008). *Product Customization*. Berlin: Springer–Verlag.

Hvam, L., Pape, S., & Nielsen, M.K. (2006). Improving the quotation process with product configuration. *Computers in Industry 57(7)*, 607–621.

Hvam, L., Riis, J., & Malis, M. (2002). A multi-perspective approach for the design of configuration systems. *Proc. 15th European Conf. Artificial Intelligence.* Accessed at www.produktmodeller.dk.

Hong, G., Hu, L., Xue, D., Tu, Y.L., & Xiong, Y.L. (2008). Identification of the optimal product configuration and parameters based on individual customer requirements on performance and costs in one-of-a-kind production. *International Journal of Production Research 46(2)*, 3297–3326.

Ladeby, K.R. (2009). *Applying product configuration systems in engineering companies: motivations and barriers for configuration projects*. PhD Thesis. Technical University of Denmark, Department of Management Engineering and Operations Management.

Olhager, J. (2003). Strategic positioning of the order penetration point. *International Journal of Production Economics 85(3)*, 319–329.

Pedersen, J.L., & Edwards, K. (2004). Product configuration systems and productivity. *Proc. Int. Conf. Economic, Technical and Organizational Aspects of Product Configuration Systems (PETO)*, pp. 165–176.

Petersen, T.D., Jorgensen, K.A., Hvolby, H.H., & Nielsen, J.A. (2007). Multi level configuration of ETO products. *Proc. 4th Int. Conf. Product Lifecycle Management: Assessing the Industrial Relevance*, pp. 293–302.

Piller, F.T., Moeslein, K., & Stotko, C.M. (2004). Does mass customization pay? An economic approach to evaluate customer integration. *Production Planning & Control 15(4)*, 435–444.

Raatikainen, M., Soininen, T., Männistö, T., & Matilla, A. (2004). A case study of two configurable software product families. In *Software Product-Family Engineering, 5th Int.Workshop, PFE 2003* (van der Linden, F., Ed.), LNCS, Vol. 3014, pp. 403–421. New York: Springer–Verlag.

Sabin, D., & Weigel, R. (1998). Product configuration frameworks—a survey. *IEEE Intelligent Systems and Their Applications 13(4)*, 42–49.

Stumptner, M. (1997). An overview of knowledge-based configuration. *AI Communications 10(2)*, 111–126.

Anders Haug is an Assistant Professor in the Department of Entrepreneurship and Relationship Management at the University of Southern Denmark. He received his PhD from the Technical University of Denmark. Dr. Haug's research focuses on information systems, knowledge engineering, product configuration, and knowledge management from an industrial perspective. He has published a long list of papers on these topics in international journals and at international conferences and has years of practical experience from projects in these areas.

Lars Hvam is a Professor at the Technical University of Denmark. He has been working on product configuration for more than 15 years as a teacher, researcher, and consultant for more than 15 configuration projects in large industrial companies. He has supervised 8 PhD projects on the construction and application of configuration systems and has been the project leader for 4 large research projects on product configuration. Dr. Hvam is also the founder and current chairman of the Product Modelling Association (www.productmodels.org), whose aim is to disseminate knowledge of the possibilities offered by product configuration.

Niels Henrik Mortensen is a Professor at the Technical University of Denmark. He has been engaged in research into and teaching of product configuration for 10 years. Dr. Mortensen has also been a consultant for more than 15 configuration projects for companies in Denmark and abroad. He is the supervisor for six PhD students within this field. Prof. Mortensen is a member of the board of the Product Modelling Association.

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 207.

doi:10.1017/S0890060410000648

CALL FOR PAPERS

AI EDAM Special Issue, August 2012, Vol. 26, No. 3
Sketching and Pen-Based Design Interaction

Guest Editors: Levent Burak Kara & Maria C. Yang

Sketching is a primary medium for ideation and communication among humans, and it is widely used in tasks involving both synthesis (design, creation) and analysis (problem solving, modeling, editing, marking). The increased availability of supporting digital tools has caused interest in sketch-based interaction to surge considerably, and researchers have developed computational systems and applications that are capable of working from such input. The challenges along the way, however, have forced us to take a closer look at our knowledge of sketching and our interactions with sketches, because we realize how little we understand the way they facilitate and contribute to our creative tasks.

This Special Issue presents new research that will push the field forward and establish future directions in sketching and pen-based interaction. Thus, we are interested in theories, methods, systems, and experiments that shed light on the knowledge contained and communicated in sketches; the role of sketches in design, creativity, and problem solving; and the utility of sketches in both human–human and human–computer interaction.

We seek contributions from a variety of fields including engineering, computer science, cognitive sciences, psychology, architecture, and art. Accepted papers are expected to provide new insights and approaches to existing problems or identify new theories and problems that will fill the knowledge gap in the field. Both theoretical and computational studies are welcome.

Topics of interest include, but are not limited to, the following:

- Knowledge representation, capture, and reuse in sketching
- The role of sketching in design, creativity, problem solving, and cognition
- Sketch-based generative design
- Sketching in collaborative design
- Sketching and aesthetics
- Sketch recognition
- Sketch-based computer-aided design and solid modeling
- Novel sketch interfaces, visualization, and implications
- Applications in engineering, computer graphics, art, architecture, medicine, and so forth

All submissions will be anonymously reviewed by at least three expert reviewers, and the selection for publication will be made on the basis of these reviews. The criteria for acceptance will be based on the importance of the problem and the review of the literature, as well as the scientific approach, experiments or evaluations, and the significance of the results.

Information about the format and style required for *AI EDAM* papers can be found at www.cs.wpi.edu/~aiedam/Instructions/

Note that all inquiries and submissions for Special Issues go to the Guest Editors, **not** to the Editor in Chief.

Important Dates

Intent to submit (Title and Abstract):	As soon as possible
Submission deadline for full papers:	1 May 2011
Reviews due:	30 August 2011
Notification and reviews to authors:	30 September 2011
Revised version submission deadline:	15 January 2012

Guest Editors

Levent Burak Kara
Mechanical Engineering Department
Carnegie Mellon University
Scaife Hall 315
5000 Forbes Avenue
Pittsburgh, PA 15213
E-mail: lkara@cmu.edu

Maria C. Yang
Engineering Systems Division
Massachusetts Institute of Technology
Room 3-449B
77 Massachusetts Avenue
Cambridge, MA 02139-4307
E-mail: mcyang@mit.edu

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 209–210.

doi:10.1017/S089006041000065X

CALL FOR PAPERS

AI EDAM Special Issue, May 2013, Vol. 26, No. 4
Intelligent Decision Support and Modeling

Guest Editors: Andy Dong & Julie Jupp

Understanding how decisions are made in risky situations with incomplete, imperfect, and uncertain information continues to be a critical interdisciplinary research question that has far-reaching implications in fields ranging from engineering to economics to public policy. In situations where multiple alternatives to a particular problem exist, each with uncertain variables and payoffs that must be analyzed and decided upon, the aim is to improve decision making so that goals can be attained while minimizing undesirable, unintended consequences.

Concurrent with the problem of decision making is forecasting the effects of decisions. Both of these matters are complicated by the realities of collective decision making of increasing scale and complexity that is typical of highly complex engineering design problems. Decision-making research is also progressively turning to the problem of the complex interplay of stakeholders, each with differing authority and information on which to make decisions and who have competing beliefs and incentives. All of these facets of decision make this an exciting area of research.

In order to tackle these matters, research methods in decision making now range from formal, mathematical modeling to statistical mechanics based models to agent-based modeling and simulation to empirical, behavioral research. Research in this area is reaching beyond normative models of decision making to examine cognitive (e.g., frames), emotional (e.g., beliefs), and social factors (e.g., herding) that influence decision making.

This Special Issue is aimed at disseminating the state-of-the-art research and applications, addressing the major challenges and issues of decision modeling, and developing and applying intelligent decision support systems. The Guest Editors invite authors to submit original papers to this Special Issue. We are also interested in authoritative reviews of the state of the art and directions for future research in the area.

The Special Issue will cover, but is not limited to, the following topics:

- Decision support systems and decision process modeling
- Empirical studies in decision making including handling risk, uncertainty, and imperfect information in individual, small group, and collective decision making
- Decision analysis including new computational methods for analyzing large-scale decision networks
- Decision theories, including game theory, utility theory, probability theory, fuzzy set theory, Bayesian theory, among others
- Approaches to decision-based design

All submissions will be anonymously reviewed by at least three reviewers. The selection for publication will be made on the basis of these reviews. High-quality papers not selected for this Special Issue may be considered for standard publication in *AI EDAM*.

Information about the format and style required for *AI EDAM* papers can be found at www.cs.wpi.edu/~aiedam/Instructions/
Note that all inquiries and submissions for Special Issues go to the Guest Editors, **not** to the Editor in Chief.

Important Dates

Intent to submit (Title and Abstract):	As soon as possible
Submission deadline for full papers:	15 September 2011
Reviews due:	15 December 2011
Notification and reviews to authors:	15 January 2012
Revised version submission deadline:	1 April 2012

Guest Editors

Andy Dong
Faculty of Architecture, Design and Planning
Room 275, G04 Wilkinson
University of Sydney
Sydney, NSW 2006, Australia
E-mail: andy.dong@sydney.edu.au

Julie Jupp
School of the Built Environment
702–730 Harris Street
University of Technology, Sydney
Ultimo, NSW 2007, Australia
E-mail: julie.jupp@uts.edu.au

Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 211–212.

doi:10.1017/S0890060410000661

CALL FOR PAPERS

AI EDAM Special Issue, May 2013, Vol. 27, No. 2
Studying and Supporting Design Communication

Guest Editors: Maaike Kleinsmann & Anja Maier

Communication is an essential part of any design process. Problems in design communication can lead to delays, mistakes, and even the ultimate failure of projects. Design communication is a multifaceted and complex phenomenon to study. It is about products and services that may or may not yet exist and includes abstraction to possible future situations. Communication can be formal or informal. For example, it can happen at the same time (synchronously) or at different times (asynchronously) and it has different directions, such as from manager to designer (top-down), from designer to manager (bottom-up), between designers, and between designers and the users. Transmitted information can take many different forms. It can be spoken, written, or drawn and can be sent and received using different media. Further, a designer may work alone. More likely, however, the design process is executed in large teams with members from differing backgrounds.

This Special Issue encourages investigation of a number of focus areas, including the following:

- design communication during different design stages of the product;
- design communication in different situations, for example, critical situations;
- interface communication (between a product and a designer, between designers, between design teams, between companies, between designers and society as a whole);
- organization of a design team to enable adequate communication, for example, the impact of team diversity or remote or colocated teams on the design process;
- emergence of shared understanding through design communication;
- communication patterns in design meetings;
- impact of affective design communication on the design process;
- nature of informal and formal communication in the design process;
- visualization of design rationale as design communication;
- interpretation of intent from sketches and other forms of representation;
- using artifacts, such as drawings and prototypes, as media in the design process;
- the role and importance of the shape of products, for example, product language;
- understanding and supporting the information requirements of a design engineer;
- multimodal design communication; and
- the future of design communication in practice and research into design communication.

In investigating the topics listed above, we often draw on insights and use methods from a number of scholarly disciplines to frame the phenomenon observed, to analyze our findings, and to draw our conclusions. Conscious or not, explicit or not, we as design researchers view the subject matter from a certain disciplinary angle. Perhaps we even use several. Ideally, the authors of this Special Issue will draw out the angle chosen and make its applicability and usefulness to design practice and research explicit.

We welcome papers that are empirical, conceptual, theoretical, or speculative.

- Empirical papers perhaps report on the practice of communication in design using field studies or experiments.
- Conceptual papers might reflect on existing discussions in the literature.
- Theoretical papers may explore one perspective or create an in-depth comparison between different theories of communication and their application to designing products.
- Speculative papers might describe the nature and future of design communication.

Together, the papers are intended to show an overview of the fields of research that contribute to the study of communication in design.

All submissions will be anonymously reviewed by at least three reviewers. The selection for publication will be made on the basis of these reviews. High quality papers not selected for this Special Issue may be considered for standard publication in *AI EDAM*.

Information about the format and style required for *AI EDAM* papers can be found at www.cs.wpi.edu/~aiedam/Instructions/

Note that all inquiries and submissions for Special Issues go to the Guest Editors, **not** to the Editor in Chief.

Important Dates

Intent to submit (Title and Abstract):	Before 31 January 2011
Submission deadline for full papers:	1 September 2011
Notification and reviews to authors:	15 January 2012
Revised version submission deadline:	1 May 2012
Final version submission deadline:	1 October 2012

Guest Editors

Maaike Kleinsmann
Department of Product Innovation Management
Faculty of Industrial Design Engineering
Delft University of Technology
Landbergstraat 15, Delft 2628CE
The Netherlands
E-mail: M.S.Kleinsmann@tudelft.nl

Anja Maier
Department of Management Engineering
Technical University of Denmark
Produktionstorvet, Building 425
DK-2800 Kgs. Lyngby
Denmark
E-mail: amai@man.dtu.dk