Configuration Knowledge 
Representation and Reasoning

Lothar Hotz*, Alexander Felfernig†, Markus Stumptner‡, Anna Ryabokon§, Claire Bagley¶ and Katharina Wolter*

*HITeC e.V., University of Hamburg, Hamburg, Germany 
†Graz University of Technology, Graz, Austria 
‡University of South Australia, Adelaide, SA, Australia 
§Alpen-Adria Universität Klagenfurt, Klagenfurt, Austria 
¶Oracle Corporation, Burlington, MA, USA

Abstract: Configuration models specify the set of possible configurations (solutions). A configuration model together with a defined set of (customer) requirements are the major elements of a configuration task (problem). In this chapter, we discuss different knowledge representations that can be used for the definition of a configuration model. We provide examples that help to further develop the understanding of the underlying concepts and include a UML-based personal computer (PC) configuration model that is used as a reference example throughout this book.

6.1 Introduction

There is quite a long history of research dedicated to the development of configuration knowledge representation languages (Felfernig et al., 2003; Günter and Kühn, 1999; Jüngst and Heinrich, 1998; Junker, 2006; Mailharro, 1998; McDermott, 1982; McGuinness and Wright, 1998; Mittal and Falkenhainer, 1990; Mittal and Frayman, 1989; Soininen et al., 1998). One of the first configuration systems was R1/XCON, which was built upon a rule-based representation language (McDermott, 1982; Soloway et al., 1987). Rule-based systems operate on a working memory that stores assertions made by rules. Rules are represented in an if-then style. If a rule is activated (given that the if condition is fulfilled) it can modify the contents of the working memory. Experiences showed that a major problem with rule-based systems is the intermingling of (product) domain knowledge and problem solving knowledge (see also Friedrich et al., 20141). A consequence is that if product knowledge is changed, rules related to the sequence in which components are integrated in a configuration have to be adapted as well, which triggers enormous maintenance overheads.2 The shortcomings of rule-based approaches were the motivation for the development of model-based knowledge representations that support a clear separation of domain and problem solving knowledge. Mittal and Frayman (1989) introduced a model-based definition of a

1Chapter 11.
2Rules are not recommended for larger industrial settings (Faltings and Weigel, 1994) and will not be discussed in detail within the scope of this book.
configuration task that is characterized by: (1) a description of generic components in terms of their properties and relationships and (2) user requirements and preferences regarding functional properties of the solution (configuration). This characterization is a major foundation for configuration research and industry (Junker, 2006). It also includes the idea of separating functional requirements (customer requirements) from the technical properties of a configurable product.

A major representative of model-based approaches are constraint-based systems (see, e.g., Freuder, 1997; Tsang, 1993). Related knowledge representations range from static constraint satisfaction (variables and constraints are predefined and are part of the search process from the very beginning (see, for example, Tsang, 1993), dynamic constraint satisfaction (variables are not necessarily active from the very beginning but can be activated within the scope of the search process; Mittal and Falkenhainer, 1990), to generative constraint satisfaction where components and related constraints are generated on-demand within the scope of the search process (see Stumptner et al., 1998). Generative approaches can also be denoted as component-oriented since component types (instead of variables) and meta (generic) constraints (instead of constraints) are taken into account as first-class entities. Examples of configuration knowledge representation in terms of a constraint satisfaction problem (CSP) will be provided in Section 6.2.

In the following, graphical representations have been developed with the goal of improving the accessibility (in terms of, e.g., understandability and maintainability) of configuration knowledge, which is crucial especially in commercial settings. One approach to the graphical representation of configuration models are feature models (see, e.g., Kang et al., 1990). Feature models can be represented in the form of a static constraint satisfaction problem (CSP; Tsang, 1993) or in the form of a SAT problem (Gomes et al., 2008). Neumann (1988) and Soininen et al. (1998) introduced configuration ontologies, which define relevant modeling concepts for configuration knowledge representation. These ontologies include modeling concepts that are typically provided in today’s commercial configuration environments; for example, component types, ports, and constraints. Similar ontological concepts as introduced, for example, by Neumann (1988) and Soininen et al. (1998) in terms of a UML profile for configuration knowledge representation, and a formalization of these concepts in first-order logic as well as in description logic has been proposed by Felfernig (2007), and Felfernig et al. (2000a, 2003). Note that in this chapter we focus on the representation of product-related configuration knowledge; knowledge representations related to incremental configuration processes (Sabin and Weigel, 1998) are discussed in detail in Leitner et al. (2014), Hotz and Wolter (2014), and Hotz and Günter (2014).^5

With this chapter, we provide an overview of modeling concepts that are relevant for configuration knowledge representation. In Subsection 6.2.1, we introduce static constraint satisfaction problems as a basic representation mechanism for configuration knowledge. In Subsection 6.2.2, we introduce two types of advanced CSP representations that are dynamic and generative constraint satisfaction problems. Thereafter, in Section 6.3, we discuss graphical configuration knowledge representations (feature models and UML models). On the basis of the latter we introduce a personal computer configuration model. In the other chapters of this book, this UML model is adapted and extended where needed. In Section 6.4,
we give an overview of different logic-based approaches to configuration knowledge representation. This chapter concludes with a discussion of major advantages and disadvantages of different types of configuration knowledge representations.

### 6.2 Constraint-Based Knowledge Representation

#### 6.2.1 Static Constraint Satisfaction

“Constraint technologies are one of the closest approaches computer science has yet made to the Holy Grail of programming: a user states the problem, the computer solves it.” (Freuder, 1997). Constraint technologies are one of the most successfully applied technologies of Artificial Intelligence (AI) in commercial settings (Brailsford et al., 1999). With these technologies, the knowledge acquisition bottleneck is tackled by the means of an easy to understand and easy to use technology. Application examples are configuration (Fleischanderl et al., 1998; Mittal and Frayman, 1989), scheduling (Pinedo, 2012), and recommender systems (Felfernig and Burke, 2008; Jannach et al., 2010). Constraint-based representations are widely used for the definition of configuration models (see, Junker and Mailharro, 2003; Mailharro, 1998). We first focus on static constraint satisfaction problems (SCSPs).

**Constraint Satisfaction Problem and Solution**

We will now introduce a definition of a *constraint satisfaction problem* (CSP) and its solution. This definition will then be used as a basis for defining a *configuration task* (problem) and a corresponding *configuration (solution).*

**Definition (Constraint Satisfaction Problem – CSP).** A constraint satisfaction problem (CSP) can be defined by a triple \((V, D, C)\) where \(V\) is a set of finite domain variables \(\{v_1, v_2, \ldots, v_n\}\), \(D\) represents variable domains \(\{\text{dom}(v_1), \text{dom}(v_2), \ldots, \text{dom}(v_n)\}\), and \(C\) represents a set of constraints defining restrictions on the possible combinations of variable values (\(\{c_1, c_2, \ldots, c_m\}\)).

A solution for a given CSP = \((V, D, C)\) can be defined as follows.

**Definition (CSP Solution).** A solution for a given CSP = \((V, D, C)\) is represented by an assignment \(S = \{ins(v_1), ins(v_2), \ldots, ins(v_k)\}\) where \(ins(v_i) \in dom(v_i)\). \(S\) is required to be complete; that is each variable of the CSP definition has a value in \(S\) and is consistent (i.e., \(S\) fulfills the constraints in \(C\)).

**Configuration Task and Solution**

Based on this definition of a CSP, we can now introduce the definition of a *configuration task* (problem) and a corresponding *configuration (solution).* We introduce two sets of constraints that describe (1) the configuration knowledge base \((C_{KB})\) and (2) the user requirements \((REQ)\).

**Definition (Configuration Task).** A configuration task can be defined as a CSP \((V, D, C)\) where \(V = \{v_1, v_2, \ldots, v_n\}\), \(D = \{\text{dom}(v_1), \text{dom}(v_2), \ldots, \text{dom}(v_n)\}\), and \(C = C_{KB} \cup REQ\). \(C_{KB}\) represents the configuration knowledge base (the configuration model) and \(REQ\) represents a set of user (customer) requirements.

---

6Efforts triggered by the communication between domain experts and engineers.
Definition (Configuration). A configuration (solution) \( S \) for a given configuration task \((V, D, C_{KB} \cup REQ)\) is represented by an assignment \( S = \{ins(v_1), ins(v_2), \ldots, ins(v_k)\} \) where \( ins(v_i) \in dom(v_i) \) and \( S \) is complete and consistent with the constraints in \( C_{KB} \cup REQ \).

This definition of a configuration task and its solution is exploited by some of the chapters in this book. This approach to represent a configuration task is primarily used in situations where all variables of the problem definition are relevant for the solution. To better understand the pragmatics of these definitions, we will now present two example configuration models: a configuration model for map coloring and a mobile phone configuration model.

Example: Map Configuration (Map Coloring)

Map-coloring can be interpreted as a simple configuration task (problem). The goal of this configuration task is to assign a color to each of the regions of a map in such a way that for each region \( x \) on the map the following property holds: all regions \( y \neq x \) that are direct neighbors of \( x \) must have a different color (different from the color of \( x \)).

As an example, we take the map of Australia (see also Russel and Norvig, 2003), which includes the regions of Western Australia (WA), Northern Territory (NT), South Australia (SA), Queensland (Q), New South Wales (NSW), Victoria (V), and Tasmania (T) (see Figure 6.1). These regions must be colored with the colors red, green, and black.

Based on this characterization of a map coloring configuration problem, we can define the corresponding configuration task \((V, D, C)\).

\[
V = \{WA, NT, SA, Q, NSW, V, T\}
\]
\[
D = \{dom(WA)=\{r,g,b\}, \ dom(NT)=\{r,g,b\}, \ dom(SA)=\{r,g,b\}, \ dom(Q)=\{r,g,b\}, \ dom(NSW)=\{r,g,b\}, \ dom(V)=\{r,g,b\}, \ dom(T)=\{r,g,b\}\}
\]

Figure 6.1

Territory map of Australia.
Finally, we introduce the set of constraints $C = C_{KB} \cup \text{REQ}$ to be taken into account when configuring (coloring) a map. The elements of $C_{KB}$ are those constraints that are defining the restrictions regarding the color combinations of neighborhood territories.

$C_{KB} = \{WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q, SA \neq NSW, SA \neq V, Q \neq NSW, NSW \neq V\}$

A CSP can also be represented graphically in terms of nodes (the variables) and the corresponding constraints (arcs between the variables); Figure 6.2 shows a graphical depiction of the map coloring configuration model.

Customer requirements in this context can be preferences (in terms of initial settings) regarding the coloring of the map. For example, we could introduce the singleton requirement $\text{REQ} = \{WA = r\}$; that is, the color of Western Australia (WA) should be red ($r$). Then $\text{REQ} \cup C_{KB}$ is consistent; that is, we are able to determine a solution for the map coloring (configuration) problem.

$S = \{WA = r, NT = g, SA = b, Q = r, NSW = g, V = r, T = r\}$

**Example: Mobile Phone Configuration**

The following configuration model of a mobile phone is based on the model of a mobile phone software product lines (adapted from Benavides et al., 2010). We will provide a formalization of this model on the basis of the definition of a configuration task given in Subsection 6.2.1. See Figure 6.3 for a graphical depiction of the mobile phone configuration model.

Our Mobile Phone (MP) model consists of the following elements: each mobile phone has two mandatory elements that are the support of Calls (CA) and the availability of a Screen (SC). A Screen can either be of type Basic (BA), Color (CO), or HighResolution (HR). A mobile phone can (but does not have to) be equipped with a GPS (GPS) feature. Furthermore, a mobile phone can include additional media equipment (Camera (CM) and MPX Player (MPX)). When equipped with a

---

7In Subsection 6.3.1 we will show how to represent similar settings with the notation of feature models (Kang et al., 1990).
FIGURE 6.3
Simple Mobile Phone configuration model (represented as CSP). An abbreviation is used for the constraint representation; for example, \( CA \leftrightarrow MP \) is the short form of \( CA = 1 \leftrightarrow MP = 1 \). \( MP \) = Mobile Phone, \( CA \) = Calls, \( GPS \) = GPS, \( SC \) = Screen, \( CM \) = Camera, \( MPX \) = MPX Player, \( BA \) = Basic, \( CO \) = Color, \( HR \) = High Resolution.

\[
\text{GPS, then the phone's Screen can't be of type Basic. A phone with a Camera must be equipped with a HighResolution Screen.}
\]

Based on this characterization of the basic properties of our example mobile phone, we can introduce the following configuration task \((V, D, C)\).

\[
V = \{\text{MP, CA, GPS, BA, CO, HR, SC, CM, MPX}\}
\]

Since we are interested in defining the possible combinations of mobile phone features, we assign to each of the feature variables in \( V \) the domain \([1, 0]\) \((\{\text{true, false}\} \text{ is possible as well})\. The semantics of 1 (resp. 0) is that a feature is active (resp. inactive); that is, included or not included in a configuration.

\[
D = \{\text{dom(MP)} = \{1, 0\}, \text{dom(CA)} = \{1, 0\}, \text{dom(GPS)} = \{1, 0\}, \text{dom(BA)} = \{1, 0\}, \text{dom(CO)} = \{1, 0\}, \text{dom(HR)} = \{1, 0\}, \text{dom(SC)} = \{1, 0\}, \text{dom(CM)} = \{1, 0\}, \text{dom(MPX)} = \{1, 0\}\}
\]

Finally, we specify the constraints \( C = C_{KB} \cup REQ \). Since we are not interested in empty configurations, we introduce the constraint \( \text{MP} = 1 \).

\[
C_{KB} = \{\text{MP} = 1, \text{CA} \leftrightarrow \text{MP}, \text{GPS} \rightarrow \text{MP}, \text{SC} \leftrightarrow \text{MP}, \text{CM} \rightarrow \text{MP}, \text{MPX} \rightarrow \text{MP}, (\text{BA} \leftrightarrow (\neg \text{CO} \land \neg \text{HR} \land \text{SC})) \land (\text{CO} \leftrightarrow (\neg \text{BA} \land \neg \text{HR} \land \text{SC})) \land (\text{HR} \leftrightarrow (\neg \text{CO} \land \neg \text{BA} \land \text{SC})), \neg \text{BA} \lor \neg \text{GPS}, \text{CM} \rightarrow \text{HR}\}
\]

Customer requirements regarding a concrete mobile phone configuration could be the following: \( REQ = \{\text{GPS} = 1\}; \) that is, \( GPS \) should be included. In our example, \( REQ \cup C_{KB} \) is consistent and a possible configuration (solution) \( S \) is:

\[
S = \{\text{MP} = 1, \text{CA} = 1, \text{GPS} = 1, \text{BA} = 0, \text{CO} = 1, \text{HR} = 0, \text{SC} = 1, \text{CM} = 0, \text{MPX} = 0\}\]

\(^8\text{CSP solvers usually support the integration of additional constraint types such as arithmetic constraints (e.g., } x = y + z\), meta-constraints (e.g., all different \((x_1, \ldots, x_k)\)), and relational constraints (e.g., } x \leq y\).
Solution Search in Static CSPs

There is plenty of literature on algorithms that can be used for solving static constraint satisfaction problems (e.g., Mackworth, 1977; Rossi et al., 2006; Russel and Norvig, 2003; Tsang, 1993). The basic approach of CSP algorithms is to apply backtracking in combination with a couple of additional mechanisms that altogether focus on variable domain size reduction during search (Russel and Norvig, 2003). One basic principle for reducing the domain size of variables is node-consistency, which requires that those values of a variable domain that are inconsistent with an unary constraint are eliminated as options (for these options there does not exist a solution). Arc-consistency (AC) is a property that must be fulfilled by combinations of variables $v_1$ and $v_2$ that are connected via a binary constraint $c_{b9}$: each value in the domain of $v_1$ must have at least one corresponding value in the domain of $v_2$ such that $c_{b9}$ is fulfilled. Note that arc-consistency is directed; if variable $v_1$ is arc-consistent with variable $v_2$ this does not necessarily mean that $v_2$ is arc-consistent with $v_1$. In the following example we take a look at the variables WA and NT of our map configuration (coloring) example (see Figure 6.2). Although AC is restricted to binary constraints, this does not restrict its applicability since every n-ary constraint can be transformed into a corresponding set of binary constraints (e.g., Bacchus and vanBeek, 1998).

Example of node-consistency and arc-consistency. The original domain of WA = \{r,g,b\}. Since we defined \{WA = r\} as a requirement, WA is node-consistent and the values \{g,b\} get removed from its domain. In order to ensure arc-consistency between WA and NT, the value \{r\} needs to be removed from NT’s domain definition. Now, each value in the new domain of WA has a consistent corresponding value in the domain of NT and vice versa.

Another important principle in variable domain size reduction is forward checking: Depending on the value selected for the current variable, the values of the instantiated variables that can’t be a support for the selected value are eliminated from their corresponding domains. An example is depicted in Figure 6.4.

Search algorithms for CSPs are able to support optimization, which helps to minimize (e.g., the overall price of a configuration) or maximize (e.g., the failure safety of an equipment) solution parameters (Pitiot et al., 2013).

---

**FIGURE 6.4**

A simple example of forward checking. The variables WA and NT already have assigned values (r and g). The only possible remaining value for SA is b; r and g do not have to be checked for consistency with the settings of WA and NT.

---

9 A constraint that refers to exactly two variables.
Finally, tuning the performance of solution search is typically done with basic strategies of variable and value orderings (Russel and Norvig, 2003), with advanced approaches such as symmetry breaking (Kiziltan and Hnich, 2001) and machine learning based acquisition of search strategies (Terashima–Marin et al., 2008), with local search algorithms (Li et al., 2006), and different techniques of knowledge compilation such as binary decision diagrams (BDDs; Andersen et al., 2010) and solution automata (Amilhastre et al., 2002). For further literature on related algorithms and optimization approaches we refer to Brailsford et al. (1999); Rossi et al. (2006); Russel and Norvig (2003), and Tsang (1993).

Excursus: SAT Problems

For a given propositional formula, the goal of SAT solving (see, e.g., Claessen et al., 2008) is the determination of a variable assignment such that the formula evaluates to true. A propositional logic formula is in a conjunctive normal form (CNF) when it is represented in the form of conjunctions of disjunctions of literals. For a Boolean variable \( X \), a literal is defined as \( X \) or its negation \( \neg X \). Each of the disjunctions is denoted as a clause. A SAT problem (satisfiability problem) is to identify an assignment to the given Boolean variables in such a way that the mentioned formula in CNF evaluates to true. This is the case if each clause has at least one literal that evaluates to true. For a more detailed discussion of SAT solving and related algorithms, refer to Claessen et al. (2008). An example of the translation of a given static CSP into a corresponding SAT representation follows.

Example: SAT encoding of a static CSP. In order to show the basic principle of how to translate a static CSP into a corresponding SAT representation, we take the variables \( WA \) and \( NT \) from our working example (map coloring). In a static CSP representation the variable domains can be specified as \( \text{dom}(WA) = \{r, g, b\} \) and \( \text{dom}(NT) = \{r, g, b\} \). In the corresponding SAT representation we have to introduce Boolean variables where each domain value of the original variables is represented by its own (Boolean) variable. In our example this translation results in the variables \( WA_r, WA_g, WA_b, NT_r, NT_g, NT_b \). Each of these variables has the domain \{true, false\} (\{1,0\}). The translation of the constraint \( WA \neq NT \) into a clausal form could look like:

\[
\neg WA_r = 1 \lor \neg NT_r = 1 \land (\neg WA_g = 1 \lor \neg NT_g = 1) \land (\neg WA_b = 1 \lor \neg NT_b = 1) \land (\neg WA_r = 0 \lor \neg WA_g = 0 \lor \neg WA_b = 0) \land (\neg NT_r = 0 \lor \neg NT_g = 0 \lor \neg NT_b = 0).
\]

With this, we assure that (1) the two territories do not have the same color and (2) both territories have a color assigned.

As can easily be seen, our Mobile Phone configuration model can directly be translated into a corresponding SAT representation, since the variable domains in the CSP are already of type Boolean. For further related work refer to Janota (2008) and Janota et al. (2010).

6.2.2 Advanced CSP Approaches

On the basis of the static CSP approach, numerous extensions have been implemented in order to improve expressiveness and the efficiency of the underlying reasoning algorithms. An in-depth analysis of existing CSP variants is beyond the scope of this book; however, we will now discuss some representative ones that have found their way into commercial configuration environments.

Dynamic Constraint Satisfaction

Configuration tasks very often include variables that do not have to be taken into account in specific configuration contexts. For instance, in our mobile phone configuration example, the variable Camera is not relevant if the user is interested only in monochrome (basic) displays. In the mobile phone
configuration example in Subsection 6.2.1, the property of variable irrelevance is implemented in terms of feature deactivation; that is, if no Camera is part of the configuration, the value of this variable is set to 0 (false). However, if we want to introduce a variable Camera with the domain \{TypeA, TypeB, TypeC\} representing three different camera types, we would have to introduce an additional domain value noval that expresses the fact that the camera is not part of the configuration. As can be seen from this simple example, there is a need to include concepts that support reasoning over variable states (active, inactive). In order to take into account the requirement of leaving out irrelevant variables during the search process, Mittal and Falkenhainer (1990) introduced the concept of dynamic constraint satisfaction (DCSP), where a constraint solver is also able to reason about the activity states of variables. The following example shows such an activation constraint.

**Example of an activation constraint.** For a variable CM (the camera) with the domain \{TypeA, TypeB, TypeC\} we could define the following activation constraint: HighResolution (HR) = yes $\rightarrow$ active(CM). If the precondition is not fulfilled, there is no need for CM to get assigned a value (i.e., be activated).

Such a reasoning about activity states in DCSPs is based on predefined activity constraints. A formal discussion of the properties of DCSPs can be found in Bowen and Bahler (1991). The concept of variable activation has also been applied in other approaches. For example, generative constraint satisfaction (GCSP) based systems (Stumptner et al., 1998) generalize the concept of variable activation to the concept of component and constraint activation. The configuration environment of SIEMENS is based on such a generative approach (see also Falkner and Schreiner, 2014).10

**Generative Constraint Satisfaction**

Major drawbacks of static (and dynamic) CSPs include representational limits in terms of dealing with variables and variable values instead of component types and corresponding components (instances). Another disadvantage is that static CSPs are limited with regard to the description of configuration problems in which the size and structure of configurations cannot be estimated beforehand. Generative constraint satisfaction (GCSP) helps to overcome these problems by moving variables and corresponding constraints to the level of component types and meta-constraints (generic constraints). Generic constraints (constraint schemata) help to identify and generate relevant additional components and variables that are relevant in the current configuration context.

Constraints are referred to as “generic” since they apply individually to all components of the types on which the constraints are defined. For example, a constraint preventing a particular type of CPU from being placed on a particular type of Motherboard applies to all CPUs of that type although it is only defined in the knowledge base once. Components in GCSP settings are represented in terms of a finite domain variable (the domain of the variable corresponds to the different types a component can take). Activation constraints are used, for example, to generate new property variables based on the assignment of a type to a component variable. A Generative Constraint Satisfaction Problem (GCSP) can be defined as follows (Stumptner et al., 1998).

**Definition (Generative Constraint Satisfaction Problem – GCSP).** A generative constraint satisfaction problem can be defined by a triple (CV, P, Γ) where CV is a possibly infinite set of component variables. The domain of each variable from CV is a finite set $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of component types. $P = \{p_1, p_2, \ldots, p_n\}$ is a set of property (attribute) names. Furthermore, there exists a set

10 Chapter 16.
$D = \{CV, D_1, D_2, \ldots, D_n\}$ of domains and a function $Dom$ that maps pairs of the form ($property\ name$, $type$) to elements of $D$. The sets $D_1, D_2, \ldots, D_n$ are called atomic domains (i.e., they contain numeric or symbolic values, but are distinct from $CV$). The name of a property variable is composed from the name of the originating component and the property name; for example, if property $p \in P$ is defined for component $c \in CV$, then the property variable is named $c.p$. Finally, $\Gamma$ is a finite set of generic constraints including activation and resource constraints.

GCSP representations allow the definition of ports that represent connection points between different components. In GCSPs (Stumptner et al., 1998), ports are represented by properties (attributes) that permit other components as values. Partonomies (part-of structures; see Subsection 6.3.2) can be represented on the basis of such port definitions. Inheritance relationships (related to a generation hierarchy) are represented on the basis of (generic) constraints that refer to component type domains.

**Example GCSP component type.** A component type $PC$ ($P$) can be represented as follows using a GCSP-style notation (see, e.g. Stumptner et al., 1998). This definition of the basic properties of a PC is related to the graphical configuration model depicted in Figure 6.7. As can easily be seen, GCSP approaches have a more intuitive knowledge representation, which is inspired by object-oriented data models combined with constraints (see, e.g., Bowen and Bahler, 1991; Neumann 1988; Puget and Albert, 1991).

```
PC P DESCRIPTION:
P.name := [String];
P.price := [Integer];
P.usage := {'internet','scientific','multimedia'};
P.efficiency := {'A','B','C'};
P.PORTS := {screen-of-pc-1[Screen], screen-of-pc-2[Screen],
hdunit-of-pc-1[HDUnit], ...};
P.screens := <screen-of-pc-1,screen-of-pc-2>;
P.mb := <mb-of-pc-1>;
P.hdunits := <hdunit-of-pc-1,hdunit-of-pc-2>;
```

GCSP constraints are represented in the context of a specific component type. As such this approach is very similar to constraint-including object-oriented languages such as the Unified Modeling Language (Booch et al., 2005) that includes the Object Constraint Language (OCL; Warmer and Kleppe, 2003). For an overview of the application of UML and OCL for configuration knowledge representation refer to Felfernig (2007) and Felfernig et al. (2000a). An example of a UML-based configuration model is introduced in Subsection 6.3.2.

**Example GCSP constraint.** A GCSP constraint that describes the fact that the energy efficiency required by the user must be supported by the motherboard $mb$, can be represented as follows. This example shows how constraints can “navigate” through the component partonomy defined for the GCSP.

```
PC P DESCRIPTION:
P.efficiency = P.mb.efficiency;
```

Usually, GCSP environments include a graphical front end that helps to define the configuration model more easily in an object-oriented fashion. Such an abstraction level improves the quality of
knowledge acquisition support. For further information regarding GCSP-based configuration knowledge representations refer to Stumptner et al. (1998). Note that we do not need to introduce a specific definition of a configuration problem for GCSPs since we can reuse the definition given in Subsection 6.2.1.

**Solution Search in Advanced CSPs**

Constraint solving in other domains is often not confronted with configuration-specific requirements. For example, a user interacting with a configurator is in the need of *explanations* (Junker, 2004; Sinz et al., 2007) in terms of why a solution has been identified or why no solution could be found (Falkner et al., 2011; Junker, 2004). In situations where no solution can be found for a given set of requirements, *explanations* represent minimal sets of requirements that have to be adapted such that a solution can be identified (Felfernig et al., 2004, 2012; Reiter, 1987).

In contrast to static CSPs, generative CSPs not only need to choose values for individual variables but also need to manage the generation of variables (components) and constraints within the scope of a search process. Decisions regarding the generation of variables are quite important since they directly influence solution quality. For example, if three hard disk drives are generated, although two is more than enough for the given set of requirements, the resulting configuration is clearly suboptimal. This is also referred to as *over-fulfilled function* (see, e.g., Junker, 2006). In order to avoid such problems, decisions regarding variable (component) instantiation have to follow a set of properties (heuristics). For example, it often makes sense to configure a product top down; that is, to add new components (variables) for a lower level of the partonomy, if components on a higher level have already been instantiated. For a detailed discussion of instantiation strategies refer to Junker (2006).

**Example of Dynamic Component Creation.** Assume a PC configuration situation where pc-1 (a PC instance) has (e.g., by explicit user interaction) a HDUnit (hdunit-1) and two HDisks (harddisks: hdisk-1 and hdisk-2) built in, giving a total capacity of 2 TB (left-hand side of Figure 6.5).

However, a resource constraint (the one in Figure 6.5) requires a total capacity of at least 4 TB for each PC. Since no HDUnit with a free disk port is available, a new one must be created. The system can then create new HDisks (hdisk-3 and hdisk-4) with a corresponding HDUnit (hdunit-2) such that the capacity is satisfied (let us assume for simplicity that two HDisks with capacity 1 TB are chosen). Because they are of that type, the port and attribute variables defined for that type are created. This includes capacity (assigned 1 TB) and a port called hdunit-of-hdisk that is used to express the part hierarchy. The definition of HDisk would specify that the hdunit-of-hdisk port needs to be connected to a HDUnit component. The port in a HDUnit that could take the other end of the connection would be the hdisk-of-hdunit port. For a more detailed discussion of related reasoning approaches refer to Junker (2006), Mailharro (1998), and Stumptner et al. (1998).

One of the major ideas of generative constraint satisfaction (see Stumptner et al., 1998) is to generalize the configuration problem representation from individual variables and constraints toward problem definitions with component types and generic constraints. Such an approach is also known as *component-based or component-oriented configuration* where configuration domain-specific concepts (e.g., components, attributes, and part-of hierarchies) can directly be used when designing a configuration model. Approaches supporting component-oriented configuration can be seen as an important step toward the development of configuration knowledge representations that help to improve the efficiency of model development as well as to increase the accessibility of configuration knowledge for product experts (with no configuration-related technical background). As already mentioned, graphical knowledge
FIGURE 6.5
Simple GCSP dynamic component creation example. The constraint representation is simplified (i.e., it does not directly correspond to the GCSP constraint representation used in Stumptner et al., 1998).

representations have been developed to more easily represent and manage configuration knowledge and increase its accessibility, which is crucial for commercial configurator projects. In the following Subsections (6.3.1–6.3.2) we introduce two basic approaches to a graphical configuration knowledge representation.

6.3 Graphical Knowledge Representation

6.3.1 Feature Models

Feature models have been developed in the Software Engineering community with the goal of representing variability properties in software product lines (see, e.g., Batory, 2005; Czarnecki et al., 2005; Felfernig et al., 2013a; Kang et al., 1990). We show how feature models can be used to represent static CSP-based configuration models. Note that each static CSP can be easily transformed into a corresponding Boolean CSP (Gomes et al., 2008) by translating individual variable domain values into corresponding Boolean variables; the constraints have to be adapted correspondingly (for an example see Subsection 6.2.1).

In this section we show how to represent the mobile phone configuration model (Subsection 6.2.1) as a feature model (see Figure 6.6). Feature models are based on a knowledge representation that
Feature model of a mobile phone (adapted version of Benavides et al., 2010). This feature model is equivalent to the constraint-based representation of Figure 6.3.

can be used to model the variability of software product lines. The idea is to represent all allowed variants of a configurable product in one integrated graphical model. If we compare the constraint-based representation of Figure 6.3 with the feature model representation of Figure 6.6 it becomes clear that the latter has a more compact and understandable representation.\(^{11}\)

The feature model (configuration model) of a mobile phone (see also Subsection 6.2.1) consists of two parts. The *structural part* defines a hierarchical relationship between the features of the feature model. The *constraint part* integrates additional constraints that define so-called cross-hierarchy restrictions. In feature models, all these additional constraints are defined on a graphical level and, typically, no additional constraints are needed to specify variability properties. Structural properties in a feature model are defined in terms of features and their relationships (*mandatory*, *optional*, *alternative*, and *or*). *Requires* and *excludes* are the two constraint types that are used for the specification of feature models. We will now discuss in an informal fashion the semantics of the modeling concepts offered by feature models. We will do this on the basis of our working example of mobile phone configuration (the corresponding Mobile Phone feature model is depicted in Figure 6.6).

**Mobile Phone Feature Model: Structure**

The structure of a feature model is defined on the basis of *features*, *relationships*, and *constraints*. The following *modeling concepts* (*modeling facilities*) can be used to develop a feature model (Kang et al., 1990).\(^{12}\) Compared to static CSP representations, feature models are less expressive. For example, basic arithmetic expressions, meta constraints, and relational expressions cannot be directly included in feature models.

---

\(^{11}\)For large and complex graphical models, additional structuring mechanisms are needed. For a detailed discussion of structuring mechanisms for configuration knowledge, refer to Felfernig et al. (2000b) and Felfernig and Zanker (2000).

\(^{12}\)For further feature model notations refer to Batory (2005) and Czarnecki et al. (2005).
• **Feature:** A feature has a unique name and is used to describe the possible states of a feature (included *in* or excluded *from* a certain configuration). The root feature (i.e., the feature `Mobile Phone` in our working example) is contained in every configuration.

• **Mandatory Relationship:** A mandatory relationship between two features describes the fact that if the higher-level feature is part of a configuration, then the lower-level feature (also called *subfeature*) must be part of the configuration. The lower-level feature can only be part of the configuration, if the higher-level feature is included. For example, if the feature `Mobile Phone` is part of the configuration, the feature `Calls` must be part of the configuration as well. Since `Mobile Phone` is the root feature, `Calls` and `Screens` must be part of every configuration.

• **Optional Relationship:** An optional relationship between two features describes the fact that if the higher-level feature is part of the configuration, the lower-level feature *can* be part of the configuration. The lower-level feature can only be part of the configuration if the higher-level feature is part of the configuration. An example of an optional feature is `Camera`.

• **Alternative Relationship:** An alternative relationship between an upper-level feature `u` and the lower-level features `l_1, l_2, \ldots, l_n` describes the fact that if `u` is part of the configuration, then *exactly one* of the lower-level features must be part of the configuration. The lower-level feature can only be part of the configuration if the higher-level feature is included. In our example, `Screen` and its subfeatures represent an alternative relationship.

• **Or Relationship:** An or relationship between an upper-level feature `u` and lower-level features `l_1, l_2, \ldots, l_n` describes the fact that if `u` is in a configuration, *at least one* of the lower-level features must be part of the configuration. Such a setting can easily be included in our feature model by the addition of a subfeature `Media` to `Mobile Phone` with the optional `Media` subfeatures `Camera` and `MPX` (see Benavides et al., 2010).

**Mobile Phone Feature Model: Constraints**

Constraints of a feature model represent restrictions that have to hold between the features. Constraint types that are typically supported by feature model languages are the following (Kang et al., 1990):

• **Requires:** A requires constraint between two features (`f_1` requires `f_2`) denotes the fact that if feature `f_1` is included in the configuration, feature `f_2` must be included as well. For example, if a `Camera` is included in a configuration, a high-resolution `Screen` must be included as well.

• **Excludes:** An excludes constraint between two features (`f_1` excludes `f_2`) denotes the fact that both `f_1` and `f_2` must not be part of the same configuration. For example, `Basic Screen` must not be combined with `GPS`.

For further discussions of feature model representations refer to Benavides et al. (2010).

**Mobile Phone Configuration (Solution)**

A configuration (solution) for a configuration task defined by a feature model and a corresponding set of customer requirements is represented by a list of features and an indication for each feature whether it is *active* or *inactive*. An example mobile phone configuration was already determined by the static constraint-based approach discussed in Subsection 6.2.1.
### Table 6.1 Semantics of feature models defined in terms of constraints (propositional logic).

Features are represented by Boolean variables.

<table>
<thead>
<tr>
<th>Relationship/Constraint</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>mandatory(P,C)</td>
<td>P ↔ C</td>
</tr>
<tr>
<td>optional(P,C)</td>
<td>C → P</td>
</tr>
<tr>
<td>or (P, C₁, C₂, ..., Cₙ)</td>
<td>P ↔ (C₁ ∨ C₂ ∨ ... ∨ Cₙ)</td>
</tr>
<tr>
<td>alternative (P, C₁, C₂, ..., Cₙ)</td>
<td>(C₁ ↔ (¬C₂ ∧ ... ∧ ¬Cₙ ∧ P)) ∧ (C₂ ↔ (¬C₁ ∧ ¬C₃ ∧ ... ∧ ¬Cₙ ∧ P)) ∧ ... ∧ (Cₙ ↔ (¬C₁ ∧ ... ∧ ¬Cₙ₋₁ ∧ P))</td>
</tr>
<tr>
<td>requires(P,C)</td>
<td>P → C</td>
</tr>
<tr>
<td>excludes(P,C)</td>
<td>¬P ∨ ¬C</td>
</tr>
</tbody>
</table>

### Semantics of Feature Models

The semantics of feature models can be defined using the static constraint representation introduced in Subsection 6.2.1. If we want to express the optional relationship between Mobile Phones and GPS, we can express this relationship in terms of the constraint GPS → Mobile Phones. If our goal is to translate a feature model into a static CSP representation, the rules depicted in Table 6.1 can be applied. In this definition, P represents the higher-level feature and C a lower-level feature that has a relationship to P.

Solution search for feature models can be implemented on the basic algorithms that are able to solve static constraint satisfaction problems (SCSPs) or satisfiability (SAT) problems.

### 6.3.2 UML Configuration Models

We now introduce a UML-based configuration knowledge representation\(^{13}\) that can also be used in complex domains where size and structure of configurations are unknown beforehand. In this context, we introduce a configuration model of personal computers (PCs), which is used as a working example in this book. It consists of two major parts: (1) The basic structure of a PC is represented in terms of a partonomy with associated generalization hierarchies (see Figure 6.7) and (2) additional constraints that restrict the combination of individual components. Some basic constraints (gc₁, gc₂, gc₃) are directly integrated in the configuration model of Figure 6.7; the remaining ones (prc₁, prc₂, resc₁, resc₂, crc₁, crc₂, crc₃, crc₄, compc₁, compc₂, compc₃) are introduced textually in Table 6.2.\(^{14}\) The modeling facilities introduced in the following can be seen as major concepts needed for the design of a configuration model (Falkner and Haselböck, 2013; Felfernig, 2007; Felfernig et al., 2000a, 2001, 2003; Soininen et al., 1998). For the representation of configuration models in UML we rely on the notation of class diagrams; no further notations such as component diagrams or sequence diagrams are needed in this context.

---

\(^{13}\) Unified Modeling Language (Booch et al., 2005).

\(^{14}\) For a detailed discussion of how to represent these constraints using the UML constraint language OCL (Warmer and Kleppe, 2003) refer to Felfernig (2007).
The major motivation for applying UML notations for configuration modeling is that the language is a de facto software engineering industry standard and thus more easily accessible for the stakeholders in a development project. Furthermore, it can be directly translated into predicate logic (Falkner and Haselböck, 2013; Felfernig, 2007; Felfernig et al., 2000a, 2003). For a discussion of a UML-based configurator application development process refer to Friedrich et al. (2014)\textsuperscript{15}.

**Computer Configuration Model in UML: Structure**

In a configuration model (see Figure 6.7), the structure of a configurable product is defined on the basis of the modeling facilities *component types* (*concepts* or *classes*), *associations with multiplicities*, and *generalizations*. Note that existing commercial configuration environments do not directly support UML-based representations but typically include similar modeling facilities that allow the representation of partonomies, generalization hierarchies, and constraints.

- **Component types**: A component type has a unique name and is characterized by a set of attributes. Attributes are defined on the basis of datatypes (the datatype of each attribute is defined in [datatype],

---

\textsuperscript{15}Chapter 11.
Table 6.2  Constraints related to the configuration model shown in Figure 6.7.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gc1</td>
<td>CPUs of type CPUS are incompatible with motherboards of type MBDiamond.</td>
</tr>
<tr>
<td>gc2</td>
<td>CPUs of type CPUD are incompatible with motherboards of type MBSilver.</td>
</tr>
<tr>
<td>gc3</td>
<td>Each OS of type OSAAlpha requires a CPU of type CPUD.</td>
</tr>
<tr>
<td>prc1</td>
<td>The price of one harddisk unit (HDUnit) is determined by the prices of the</td>
</tr>
<tr>
<td></td>
<td>disks (HDisk) and controllers (HDController) part of the HDUnit.</td>
</tr>
<tr>
<td>prc2</td>
<td>The price of one personal computer (PC) is determined by the prices of the</td>
</tr>
<tr>
<td></td>
<td>HDUnits, the motherboard (MB), the CPUs, the operating system (OS), the</td>
</tr>
<tr>
<td></td>
<td>Screens, additional applications (Applications), and the Internet connection</td>
</tr>
<tr>
<td></td>
<td>(InternetConn).</td>
</tr>
<tr>
<td>resc1</td>
<td>The computer price must be less or equal to the maxprice defined by the</td>
</tr>
<tr>
<td></td>
<td>customer.</td>
</tr>
<tr>
<td>resc2</td>
<td>The consumed hdcapacity (consumed by instances of component type OS and</td>
</tr>
<tr>
<td></td>
<td>Applications) must be less or equal to the produced capacity (produced by</td>
</tr>
<tr>
<td></td>
<td>instances of type HDisk).</td>
</tr>
<tr>
<td>crc1</td>
<td>Intended internet usage or multimedia usage (attribute of PC) requires the</td>
</tr>
<tr>
<td></td>
<td>existence of an Internet connection (InternetConn).</td>
</tr>
<tr>
<td>crc2</td>
<td>Scientific usage requires CPUs of type CPUD.</td>
</tr>
<tr>
<td>crc3</td>
<td>The required energy efficiency (attribute efficiency of PC) must be supported</td>
</tr>
<tr>
<td></td>
<td>by the MB.</td>
</tr>
<tr>
<td>crc4</td>
<td>The required energy efficiency (attribute efficiency of PC) must be supported</td>
</tr>
<tr>
<td></td>
<td>by the included Screens.</td>
</tr>
<tr>
<td>compc1</td>
<td>The price of an efficiency A Screen is 200 units.</td>
</tr>
<tr>
<td>compc2</td>
<td>The price of an efficiency B Screen is 150 units.</td>
</tr>
<tr>
<td>compc3</td>
<td>The price of an efficiency C Screen is 100 units.</td>
</tr>
</tbody>
</table>

which can denote a constant, an enumeration, or a range). For example, maxprice[0..2500] specifies an integer range attribute of the component type PC. In the examples in this book, attributes are single-valued; that is, no attribute has more than one value.

- **Associations and Multiplicities:** The part-of modeling facility is used to describe part-of associations between component types. In its simplest form, these associations are assumed to be of type composite (not shared); this means that no instance (component) of a component type can be part of more than one instance (whole component). For example, each CPU is part of exactly one MB (motherboard) and each MB consists of one or two CPUs. Note that we apply multiplicities to further describe associations between component types. Other examples of multiplicities are the following: each PC (personal computer) consists of one or more Applications (no upper limit defined here) and each Application is part of exactly one PC. Each harddisk (HDisk) has exactly one DiskPort and each DiskPort is associated with one HDisk (within the same HDUnit). Furthermore, each DiskPort
is connected with a ControllerPort. Note that additional types of associations are included in the individual book chapters where needed.

- **Generalizations**: This modeling facility relates two or more component types through a subset relation. The generalization relationship between subtypes and supertype (or the inverse specialization relationship between supertype and subtypes) can be characterized as **disjoint** and **complete**. Disjointness means that each instance of a component type X can be assigned to only one of the subtypes of X. For example, each CPU is either of type CPUS or CPUD but not both. Completeness means that each instance is assigned to one of the leaf nodes of the generalization hierarchy. Furthermore, generalization hierarchies in the configuration context typically do not allow multiple inheritance. Again, further modeling facilities with different semantics are introduced in the other chapters of this book where needed. Note that for reasons of simplicity no definition of specific application types is included in our example; it is assumed that each instance of type Application has the same required hdcapacity (200) and the same price, which is 50. In a complete model of a personal computer additional subtypes would be included or defined as part of a corresponding component catalog.

### Computer Configuration Model in UML: Constraints

Constraints represent restrictions that have to hold between attributes and/or component types. Constraints frequently used in configuration models are now explained on the basis of our PC configuration model (see Table 6.2).\(^\text{16}\)

We differentiate between the following types of constraints: (1) Constraints that are directly integrated into the configuration model depicted in Figure 6.7 are denoted as **graphical constraints** \((gc_i \in GC)\). (2) Constraints regarding the pricing of a configurable product can be defined as part of the configuration model. In many real-world scenarios pricing constraints (policies) are not part of the configuration model but rather implemented as a separate pricing component. We denote the set of **pricing constraints** as \(PRC = \bigcup prc_i\). (3) Constraints restricting the production and consumption of resources are denoted as **resource constraints** \((resc_i \in RESC)\). (4) Constraints that define in which context which additional components have to be part of a configuration (solution) are denoted as **requires constraints** \((crc_i \in CRC)\). Requires constraints \(\{crc1–crc3\}\) are defined in addition to the ones already contained in \(GC\). (5) Restrictions regarding the way in which different components (instances of component types) can be combined with each other are defined on the basis of **compatibility constraints** \((compc_i \in COMPC)\). The compatibility constraints \(\{comp1–comp3\}\) are defined in addition to the ones already contained in \(GC\). The graphical constraint \(gc1\) describes an **incompatibility**, which is the complement of a compatibility. Usabilities are used in cases where the number of allowed combinations of components (or attribute values) is low. Alternatively, if the number of incompatibilities is low, incompatibility constraints are used. Quite often, the term **compatibility constraint** denotes both compatibilities and incompatibilities. All the constraints of the configuration model \(C_{KB} = GC \cup PRC \cup RESC \cup CRC \cup COMPC\)\(^\text{17}\) together with a set of customer requirements are major elements of a **configuration task** that can be solved by a configurator (configuration system).

---

\(^{16}\)Formalizations of these constraints are introduced in Subsection 6.4.1.

\(^{17}\)Note that for simplicity we take into account only these specific types of constraints. However, industrial configuration environments typically allow the definition of further constraints that are not restricted to the categorization used here.
**Computer Configuration (Solution)**

A configuration (solution to a configuration task) can be represented as a UML instance diagram; an example of such a diagram is depicted in Figure 6.8. The diagram depicts concrete components (instances of the classes depicted in the configuration model of Figure 6.7).

---

**6.4 Logic-Based Knowledge Representation**

**6.4.1 First Order Logic (FOL)**

The semantics of UML modeling facilities is explained on the basis of the configuration model depicted in Figure 6.9. This is a fragment of our PC configuration model of Figure 6.7, which includes the **component types** \{PC, MB, MBDiamond, MBSilver, CPU, CPUS, CPUD, OS, OSA, OSP\} and the **constraints** \{gc1, gc2, gc3, prc2′, resc1\} (see Table 6.3).\(^{18}\) In order to define the semantics of UML-based configuration models, we will apply a decidable subset of first-order logic (FOL). This is in the line of research dedicated to the formalization of component-oriented representations (see, e.g., Falkner and Haselböck, 2013; Felfernig, 2007; Felfernig et al., 2000a, 2003; Schröder et al., 1996).

**Configuration Task and Solution**

We will now introduce a predicate logic-based definition of a **configuration task (problem)** and its **solution**, which is used in the remaining sections of this chapter. This definition is based on Felfernig et al. (2004, 2003). From now on we concentrate on a generalized (component-oriented) knowledge
representation based on component types and generic constraints (similar to the concepts also provided by generative constraint satisfaction approaches; Mailharro, 1998; Stumptner et al., 1998).

**Definition (Configuration Task).** A configuration task can be defined as a triple \((C_{KB}, REQ, CLANG)\) where \(C_{KB}\) and \(REQ\) are sets of logical sentences and \(CLANG\) is a set of predicate symbols. \(C_{KB}\) represents the configuration model and \(REQ\) specifies particular system (customer) requirements.
A configuration $CONF$ is described by a set of positive ground literals whose predicate symbols are in $CLANG$.

**Definition (Configuration (Solution)).** Given a configuration task $(C_{KB}, REQ, CLANG)$, a configuration $CONF$ is consistent if and only if $C_{KB} \cup REQ \cup CONF$ is satisfiable. A configuration is valid if it is consistent and a set of completeness axioms related to $CONF$ is fulfilled. $CONF$ together with the corresponding completeness axioms is denoted as $\hat{CONF} = CONF \cup AX$.

In order to be able to guarantee the completeness property, we have to introduce an additional formula for each of the predicate symbols in $CLANG$—the so-called completeness axioms ($AX$; Felfernig et al., 2004). These axioms help to deduce the negation of all possible instances of literals whose predicate symbols are in $CLANG$ with the exception of the positive ground literals contained in $CONF$; that is, all instances of predicates not explicitly described in $CONF$ are negated (for an example see Table 6.6). This process is also known as Clark Completion (see Russel and Norvig, 2003).

**Semantics of UML Configuration Models**

The first step in formalizing a configuration model is to define the needed component types, attributes, and relationships in terms of FOL unary and binary predicates (see Table 6.4).

For each attribute, we have to restrict the allowed attribute values in the context of a certain class. For example, the price of a MB can have a value between 0 and 350 whereas the price of a CPU is restricted to the value of 100. Thereafter, we define the part-of relationships between component types; in Table 6.4 this is exemplified on the basis of the part-of relationship between MB and CPU. The semantics of generalization (isa) hierarchies is assumed to be disjoint and complete. For example, each MB is either an MDBDiamond or an MBSilver. At the same time, a motherboard cannot be both an MDBDiamond and an MBSilver. All rules for translating UML configuration models into a corresponding logic-based representation can be found in Felfernig (2007), and Felfernig et al. (2000a, 2003).

In order to specify a configuration task we also need to define a set of customer requirements ($REQ$). These are represented as facts that have to be taken into account by the configurator (see Table 6.5).

**Computer Configuration (Solution) in FOL**

A configuration can be interpreted as an assignment to all the decision points that are specified in the configuration model. Such an assignment is complete if all decision points are instantiated; it is consistent if all the constraints in $C_{KB} \cup REQ$ are fulfilled. Finally, the assignment is valid if it is consistent and complete. A solution to the configuration task defined by our example is shown in Table 6.6.

### 6.4.2 Logic-Based Configuration

We used FOL to specify the semantics of modeling concepts that can be used for the development of a configuration model. In this section, we introduce logic-based approaches, which can be applied to solve a configuration task to calculate a solution (configuration).

**Answer Set Programming**

*Answer Set Programming (ASP)* emerged in the late 1990s as an approach to the declarative modeling and solving of computational problems. The paradigm is a result of intensive research in the areas of
Table 6.4  Example formalizations of the model \((C_{KB})\) depicted in Figure 6.9. \textit{getcpus} denotes a collection operator (Felfernig et al., 2000a) that collects all \textit{cpus} connected with motherboard \(Y\). For reasons of readability we limit the example to attribute range restrictions (e.g., \(PC(\text{efficiency})\)).

<table>
<thead>
<tr>
<th>Modeling Facility</th>
<th>Example in FOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component types</td>
<td>{PC/1, MB/1, MBDiamond/1, MBSilver/1, CPU/1, CPUS/1, CPUD/1, OS/1, OSA/1, OSB/1} \subseteq CLANG</td>
</tr>
<tr>
<td>Attributes</td>
<td>{\text{efficiency/2, price/2, maxprice/2, clockrate/2, hdcapacity/2}} \subseteq CLANG</td>
</tr>
<tr>
<td>Relationships</td>
<td>{pc-of-mb/2, mb-of-pc/2, mb-of-cpu/2, cpu-of-mb/2, pc-of-os/2, os-of-pc/2} \subseteq CLANG</td>
</tr>
<tr>
<td>PC (efficiency)</td>
<td>(\forall X : PC(X) \rightarrow \exists A_x : \text{efficiency}(X, A_x) \wedge A_x = A \lor A_x = B \lor A_x = C. \subseteq C_{KB})</td>
</tr>
<tr>
<td>MB (efficiency)</td>
<td>(\forall X : MB(X) \rightarrow \exists A_x : \text{efficiency}(X, A_x) \wedge A_x = A \lor A_x = B \lor A_x = C. \subseteq C_{KB})</td>
</tr>
<tr>
<td>MB (price)</td>
<td>(\forall X : MB(X) \rightarrow \exists A_x : \text{price}(X, A_x) \wedge A_x \geq 0 \land A_x \leq 350. \subseteq C_{KB})</td>
</tr>
<tr>
<td>CPUS (price)</td>
<td>(\forall X : CPUS(X) \rightarrow \exists A_x : \text{price}(X, A_x) \wedge A_x = 100. \subseteq C_{KB})</td>
</tr>
<tr>
<td>part-of(PC,MB)</td>
<td>(\forall X : PC(X) \rightarrow \exists Y : MB(Y) \wedge \text{pc-of-mb}(X, Y). \subseteq C_{KB})</td>
</tr>
<tr>
<td>part-of(PC,OS)</td>
<td>(\forall X : PC(X) \rightarrow \exists Y : OS(Y) \wedge \text{os-of-pc}(X, Y). \subseteq C_{KB})</td>
</tr>
<tr>
<td>part-of(MB,CPU)</td>
<td>(\forall X : MB(X) \rightarrow \exists Y : CPU(Y) \wedge \text{mb-of-cpu}(X, Y). \subseteq C_{KB})</td>
</tr>
<tr>
<td>isa(MB)</td>
<td>(\forall X : MB(X) \leftrightarrow MBDiamond(X) \lor MBSilver(X). \subseteq C_{KB})</td>
</tr>
<tr>
<td>isa(CPU)</td>
<td>(\forall X : CPU(X) \leftrightarrow CPUD(X) \lor CPUS(X). \subseteq C_{KB})</td>
</tr>
<tr>
<td>isa(OS)</td>
<td>(\forall X : OS(X) \leftrightarrow OSA/1 \lor OSB/1. \subseteq C_{KB})</td>
</tr>
<tr>
<td>gc1</td>
<td>(\forall X, Y : \text{mb-of-cpu}(X, Y) \wedge MBDiamond(X) \wedge CPUS(Y) \rightarrow \text{false.} \subseteq C_{KB})</td>
</tr>
<tr>
<td>gc2</td>
<td>(\forall X, Y : \text{mb-of-cpu}(X, Y) \wedge MBSilver(X) \wedge CPUD(Y) \rightarrow \text{false.} \subseteq C_{KB})</td>
</tr>
<tr>
<td>gc3</td>
<td>(\forall X, Y : PC(X) \wedge OSA/1 \wedge \text{pc-of-os}(X, Y) \rightarrow \exists U : MB(U) \wedge CPUD(V) \wedge \text{mb-of-cpu}(U, V). \subseteq C_{KB})</td>
</tr>
<tr>
<td>proc2</td>
<td>(\forall X : PC(X) \wedge \text{price}(X, PCP) \wedge \text{pc-of-mb}(X, Y) \wedge \text{pc-of-os}(X, Z) \wedge \text{getcpus}(Y, CPUS) \rightarrow PCP = \sum_{c \in {Y, Z} \cup CPUS, \text{price}(c, PCP)} P. \subseteq C_{KB})</td>
</tr>
<tr>
<td>resc1</td>
<td>(\forall X : PC(X) \wedge \text{price}(X, PCP) \wedge \text{maxprice}(X, PCMP) \rightarrow PCP \leq PCMP. \subseteq C_{KB})</td>
</tr>
</tbody>
</table>
Table 6.5 Example representation of user (customer) requirements ($REQ$).

<table>
<thead>
<tr>
<th>Modeling Facility</th>
<th>Example in FOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>User requirements</td>
<td>{PC(p_1), MB_Diamond(mb_1)} $\subseteq$ $REQ$</td>
</tr>
</tbody>
</table>

Table 6.6 Example configuration (CONF) with completeness axioms (AX).

<table>
<thead>
<tr>
<th>Modeling Facility</th>
<th>Example in FOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components (CONF)</td>
<td>{PC(p_1), MB_Diamond(mb_1), CPU_D(cpu_1), OS_ALPHA(os_1)} $\subseteq$ CONF</td>
</tr>
<tr>
<td>Attributes (CONF)</td>
<td>{efficiency(p_1, A), price(p_1, 600), efficiency(mb_1, A), price(mb_1, 350), clockrate(cpu_1, fast), price(cpu_1, 150), hdcapacity(os_1, 6), price(os_1, 100)} $\subseteq$ CONF</td>
</tr>
<tr>
<td>Associations (CONF)</td>
<td>{pc-of-mb(p_1, mb_1), mb-of-pc(mb_1, p_1)} $\subseteq$ CONF</td>
</tr>
<tr>
<td>Components (AX)</td>
<td>{PC(X)$\rightarrow$X = p_1, MB_Diamond(X)$\rightarrow$X = mb_1, \ldots}} $\subseteq$ AX</td>
</tr>
<tr>
<td>Attributes (AX)</td>
<td>{efficiency(X, Y)$\rightarrow$(X = p_1$\wedge$Y = A)$\lor$ \ldots}} $\subseteq$ AX</td>
</tr>
<tr>
<td>Associations (AX)</td>
<td>{pc-of-mb(X, Y)$\rightarrow$(X = p_1$\wedge$Y = mb_1), \ldots}} $\subseteq$ AX</td>
</tr>
</tbody>
</table>

Knowledge representation, deductive databases, and logic programming (Brewka et al., 2011). ASP is a subset of FOL interpreted under stable-model semantics (Gelfond and Lifschitz, 1988). It is worth mentioning that configuration was one of the first real-world applications of ASP (see Soininen et al., 2001; Tiihonen et al., 2003, 2013). ASP programs are represented as a finite set of rules $a \leftarrow b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_k$ where $a$ (head of the rule), $b_i$ and $c_j$ (body of the rule) are classical FOL atoms. In a rule, the atom $a$ in the head is true if all literals of the body are true; that is, there is a derivation for each positive literal $b_1, \ldots, b_m$ whereas none of the negative literals $\neg c_1, \ldots, \neg c_k$ can be derived.

A rule with an empty head (placeholder for false) is denoted as integrity constraint. For example, $\leftarrow$ mb_diamond(X), cpus(Y), mb-of-cpus(X, Y) denotes the incompatibility between the component types MB_Diamond and CPUs (see Figure 6.9). A rule with an empty body is denoted as fact; for example, mbsilver_domain(1..2) denotes the two facts mbsilver_domain(1) and mbsilver_domain(2). Finally, weight constraints allow the definition of choices. For example, 1[cpu(X) : cpu_domain(X)]2 denotes the fact that between 1 and 2 CPUs can be included in a configuration (cpu_domain describes the available CPU instances). The determination of configurations for ASP programs is based on two steps: (1) first order variables of the program are systematically replaced by the elements of the Herbrand universe (this process is also denoted as grounding (e.g., Gebser et al., 2009)) and (2) the resulting (variable-free) program is then fed into the ASP solver that determines stable models.
(answer sets), minimal models of the ASP program (Giunchiglia et al., 2006). The ASP representation of
the configuration model depicted in Figure 6.9 is shown in Table 6.7. For reasons of readability we omit-
ted the specification of attribute domains on all levels of the generalization hierarchy. For example, the
specification of the possible values of the clockrate attribute (component type cpu) can be implemented
as follows: \( \{ \text{clockrate}(C,A) : \text{clockrate}\_\text{domain}(A) \} \) \( \langle - \text{cpu}(C). \text{clockrate}\_\text{domain}(\text{slow;medium;fast}) \).
ASP representations of customer requirements and a configuration result are shown in Table 6.8.

### Table 6.7 Example formalizations of the PC configuration model shown in Figure 6.9 defined in clingo (see potassco.sourceforge.net).

<table>
<thead>
<tr>
<th>Concept</th>
<th>ASP Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain generalizations</td>
<td>mb_domain(X) :- mbsilver_domain(X). mb_domain(X) :- mbdiamond_domain(X). cpud_domain(X) :- cpus_domain(X). os_domain(X) :- osalpha_domain(X). os_domain(X) :- osbeta_domain(X).</td>
</tr>
<tr>
<td>Component types</td>
<td>mbsilver(X) :- component(X), mbsilver_domain(X). mbdiamond(X) :- component(X), mbdiamond_domain(X). mb(X) :- component(X), mb_domain(X). cpus(X) :- component(X), cpus_domain(X). cpud(X) :- component(X), cpud_domain(X). cpu(X) :- component(X), cpu_domain(X). osalpha(X) :- component(X), osalpha_domain(X). osbeta(X) :- component(X), osbeta_domain(X). os(X) :- component(X), os_domain(X).</td>
</tr>
<tr>
<td>Attribute domains</td>
<td>efficiency(X,b) :- mbsilver(X). efficiency(X,a) :- mbdiamond(X). price(X,250) :- mbsilver(X). price(X,350) :- mbdiamond(X). clockrate(X,medium) :- cpus(X). clockrate(X,fast) :- cpus(X). price(X,100) :- cpud(X). price(X,150) :- cpud(X). hdcapacity(X,6) :- osalpha(X). hdcapacity(X,13) :- osbeta(X). price(X,100) :- osalpha(X). price(X,500) :- osbeta(X).</td>
</tr>
<tr>
<td>Generalization hierarchies</td>
<td>mb(X) :- mbsilver(X). mb(X) :- mbdiamond(X). cpu(X) :- cpus(X). cpu(X) :- cpud(X). os(X) :- osalpha(X). os(X) :- osbeta(X).</td>
</tr>
<tr>
<td>Part of relationships</td>
<td>1 { \text{mb_of_cpu}(X,Y) : \text{cpu_domain}(Y) } 2 :- mb(X). 1 { \text{pc_of_mb}(X,Y) : \text{mb_domain}(Y) } 1 :- pc(X). 1 { \text{pc_of_os}(X,Y) : \text{os_domain}(Y) } 1 :- pc(X).</td>
</tr>
<tr>
<td>Symmetry properties</td>
<td>cpu_of_mb(Y,X) :- mb_of_cpu(X,Y). mb_of_pc(Y,X) :- mb_of_cpu(X,Y). os_of_pc(X,Y) :- pc_of_os(Y,X).</td>
</tr>
<tr>
<td>Component generation</td>
<td>component(Y) :- mb_of_cpu(X,Y). component(Y) :- pc_of_mb(Y,X). component(Y) :- pc_of_os(Y,X).</td>
</tr>
<tr>
<td>gc1</td>
<td>:- mbdiamond(X), cpus(Y), mb_of_cpu(X,Y).</td>
</tr>
<tr>
<td>gc2</td>
<td>:- mbsilver(X), cpud(Y), mb_of_cpu(X,Y).</td>
</tr>
<tr>
<td>gc3</td>
<td>:- osalpha(X), not cpud((C)), pc_of_os(P,X), pc_of_mb(P,M), mb_of_cpu(M,C).</td>
</tr>
<tr>
<td>proc2'</td>
<td>pcprice(A,P) :- P = #sum [price(C,PR) = PR], pc(A).</td>
</tr>
<tr>
<td>resc1</td>
<td>:- pc(X), maxprice(X,P), pcprice(X,Q), Q &gt; P.</td>
</tr>
</tbody>
</table>
Table 6.8 Example configuration result calculated by clingo (see potassco.sourceforge.net).

<table>
<thead>
<tr>
<th>Concept</th>
<th>ASP Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer requirements</td>
<td>pc(1). maxprice(1,2500). efficiency(1,b).</td>
</tr>
<tr>
<td>Configuration (result)</td>
<td>pc(1), maxprice(1,2500), efficiency(1,b), pc_of_mb(1,2), pc_of_os(1,9), pcprice(1,850), mbsilver(2), mb(2), efficiency(2,b), price(2,250), mb_of_cpu(2,7), mb_of_pc(2,1), osbeta(9), os(9), hdcapacity(9,13), price(9,500), os_of_pc(9,1), cpus(7), cpu(7), clockrate(7,medium), price(7,100), cpu_of_mb(7,2)</td>
</tr>
</tbody>
</table>

**Description Logics**

*Description Logics* (DL; Baader et al., 2003) is a popular knowledge representation formalism in the context of the Semantic Web. DL can be used for configuration knowledge representation, especially for the design of component type hierarchies (ontologies) and for coherence analysis. A Description Logic knowledge base consists of two major elements: the *TBox* introduces concepts, relationships, and constraints of the (product) domain, the *ABox* contains assertions about individuals. Thus, the TBox can be used to represent the configuration model and the ABox the resulting configuration. Description Logics have been successfully applied for the development of configuration systems—for a detailed related discussion refer to Hotz et al. (2006) and McGuinness and Wright (1998). Felfernig et al. (2003) have analyzed the applicability of Description Logics for the representation of configuration knowledge. DL can be used to represent configuration knowledge, however, some specific constraint types are supported in a restricted fashion. For example, no aggregation functions are provided and constraints on complex connection structures (via ports) cannot be defined due to the lower (but decidable) expressiveness of many DL dialects. Furthermore, because there is no standard inference method that provides automatic instantiation, the DL-Reasoner has to be included in an architecture that creates individuals from outside. This is needed because of the incremental aspects of configuration processes (see Buchheit et al., 1995; Hotz and Neumann, 2005; Stumptner, 1997). Beside this analysis, Felfernig et al. (2003) also introduce a formal definition of a configuration problem and its solution in Description Logic and show the equivalence of this definition with a predicate logic-based definition.

**Hybrid Configuration**

*Hybrid configuration* exploits several representation and reasoning mechanisms (Stumptner, 1997). Typically, there is an ontology-like (DL-based) representation (Baader et al., 2003) for representing components, their compositional relations, and attributes. Additionally, constraints are used for representing n-ary relations between components and for computing and inferring attribute values. The configuration process combines the related inference mechanisms, ontology inference services, and constraint propagation in a cyclic manner. Additionally, procedural knowledge (Hotz et al., 2006; McDermott, 1982) can be used for describing the order of configuration steps of the configuration.
<table>
<thead>
<tr>
<th>id</th>
<th>Criteria description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Standard graphical modeling concepts?</td>
</tr>
<tr>
<td>C2</td>
<td>Component-oriented modeling?</td>
</tr>
<tr>
<td>C3</td>
<td>Automated consistency maintenance?</td>
</tr>
<tr>
<td>C4</td>
<td>Modularization concepts available?</td>
</tr>
<tr>
<td>C5</td>
<td>Support of easy knowledge base evolution and maintenance?</td>
</tr>
<tr>
<td>C6</td>
<td>Model-based knowledge representation?</td>
</tr>
<tr>
<td>C7</td>
<td>Efficient reasoning?</td>
</tr>
<tr>
<td>C8</td>
<td>Able to solve generative problem settings?</td>
</tr>
<tr>
<td>C9</td>
<td>Able to provide explanations?</td>
</tr>
</tbody>
</table>

A detailed description of a configuration environment based on a hybrid configuration approach is given in Hotz and Günter (2014).19

### 6.5 Comparison of Knowledge Representations

We now compare the configuration knowledge representations that have been discussed within the scope of this chapter (RBS = rule-based systems, SCSP = static constraint satisfaction, DCSP = dynamic constraint satisfaction, GCSP = generative constraint satisfaction, SAT = SAT solving, FM = feature models, UML = UML configuration models, DL = Description Logics, ASP = Answer Set Programming, HB = hybrid configuration). We will first provide an overview of the relevant criteria (see Table 6.9) and then discuss the evaluation results (see Table 6.10).

#### 6.5.1 Evaluation Criteria

The evaluation criteria are summarized in Table 6.9. The first evaluation criterion (C1) is related to the availability of standard graphical modeling concepts that help (1) to improve the understandability of configuration knowledge and (2) to increase the efficiency of configuration knowledge base development and maintenance processes. Criterion C2 is related to the question whether the knowledge representation formalism supports configuration knowledge design on the basis of concepts near to real-world entities (i.e., component types, attributes, relationships, generalization hierarchies, relevant constraint types, etc.). C3 raises the question whether mechanisms for automated consistency maintenance are available; for example, if a new constraint (or concept) is added to the knowledge base, it should be immediately clear, whether inconsistencies are introduced or not. Criterion C4 is related to the availability of modularization and structuring concepts for knowledge bases; for example, in which way is it possible to organize (group) the constraints inside a knowledge base.

19Chapter 24.
6.5 Comparison of Knowledge Representations

Table 6.10 Evaluation of representations (RBS = rule-based systems, SCSP = static CSP, DCSP = dynamic CSP, GCSP = generative CSP, SAT = Sat Solving, FM = feature models, UML = UML configuration models, DL = Description Logics, ASP = Answer Set Programming, HB = hybrid configuration). √ = good support, ≈ = some support, and – = no support.

<table>
<thead>
<tr>
<th>id</th>
<th>RBS</th>
<th>SCSP</th>
<th>DCSP</th>
<th>GCSP</th>
<th>SAT</th>
<th>FM</th>
<th>UML</th>
<th>DL</th>
<th>ASP</th>
<th>HB</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>√</td>
<td>√</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>√</td>
<td>–</td>
<td>–</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>C3</td>
<td>–</td>
<td>≈</td>
<td>≈</td>
<td>≈</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>≈</td>
<td>≈</td>
</tr>
<tr>
<td>C4</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>√</td>
<td>–</td>
<td>–</td>
<td>≈</td>
<td>√</td>
<td>≈</td>
<td>≈</td>
</tr>
<tr>
<td>C5</td>
<td>–</td>
<td>≈</td>
<td>≈</td>
<td>≈</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>≈</td>
<td>≈</td>
</tr>
<tr>
<td>C6</td>
<td>–</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C7</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>–</td>
<td>√</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C8</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>√</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>≈</td>
</tr>
<tr>
<td>C9</td>
<td>≈</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

C5 evaluates the quality of support of knowledge base evolution and maintenance, for example, in terms of intelligent navigation support in complex knowledge spaces. C6 evaluates whether the underlying knowledge representation is model-based. C7 poses the question whether the type of knowledge representation allows efficient reasoning. C8 evaluates whether a generative configuration scenario can be implemented; that is, whether components and related constraints can be generated on-demand (see Subsection 6.2.2). C9 is related to the question whether the generation of explanations can be supported.

6.5.2 Comparison of Knowledge Representations

On the basis of the evaluation criteria summarized in Table 6.9 we are able to discuss the advantages and disadvantages of the knowledge representation formalisms presented in this chapter (see Table 6.10).

These formalisms are evaluated on the basis of criteria C1 – C9 on a rating scale of (√ = good support, ≈ = some support, and – = no support). Standard graphical knowledge representations (C1) are primarily provided by feature models (Benavides et al., 2010) and component-oriented modeling approaches such as the Unified Modeling Language (Felfernig, 2007; Felfernig et al., 2000a). Component-oriented modeling (C2) is supported by GCSPs (Stumptner et al., 1998) and modeling approaches in the line of UML, Description Logics (Baader et al., 2003), Answer Set Programming (ASP; Brewka et al., 2011), and Hybrid Configuration (HB; Stumptner, 1997). Automated consistency maintenance (C3) can easily be implemented on top of most of the given knowledge representations (Felfernig et al., 2004, 2012). FM and UML artifacts have to be connected with a corresponding reasoning support. Intelligent modularization (C4) in GCSP, UML, DL, and HB is supported in terms of a component (class) dependent definition of domain constraints. With the exception of RBS all representations allow an arbitrary ordering (organization) of the constraints in the knowledge base without changing its underlying semantics; RBS
often also dispose of concepts for structured rule representation. Graphical UML modeling supports the inclusion of advanced modularization concepts such as packaging and diagram contextualization (Felfernig et al., 2000b; Felfernig and Zanker, 2000). Techniques for easy knowledge base evolution and maintenance (C5) are not primarily depending on the knowledge representation itself but more on additional meta-functionalities that support intuitive knowledge navigation and adaptation; nearly all representation formalisms can be equipped with such a functionality (Felfernig et al., 2013b). RBS have serious maintenance problems due to the dependence of product domain and problem solving knowledge. With the exception of RBS, all other representations follow a model-based paradigm (C6) that supports a clear separation of domain and problem-solving knowledge. RBS and constraint satisfaction algorithms have proven to support efficient problem solving on an industrial scale. For this reason, these approaches are evaluated better with regard to performance (C7). GCSP allows the determination of solutions for generative configuration problems; ASP and HB are also able to provide such functionalities (C8). With the exception of FM and UML, all approaches provide basic support for explanations (as to why a configuration could (not) be found). Dependencies between domain and problem solving knowledge in RBS and encodings in SAT (see Subsection 6.2.1) make these approaches less accessible for explanation techniques.

6.6 Conclusion

In this chapter we provided an overview of different configuration knowledge representation and reasoning techniques. These techniques were discussed on the basis of a timeline ranging from the rule-based configuration system R1/XCON implemented in the late 1970’s via different types of constraint representations and constraint reasoning approaches through to advanced knowledge representation and reasoning approaches that are applied nowadays for the definition and solution of configuration tasks.

Acknowledgments

The work presented in this chapter was partially funded by the Austrian research promotion agency (grant numbers: 827587 and 825071).

References


