What makes the Difference? - Basic Characteristics of Configuration

Lothar Hotz
HITeC e.V., University of Hamburg, Germany
hotz@informatik.uni-hamburg.de

Abstract

This paper focuses on configuration as a process that iteratively applies commonly known reasoning techniques and creates an incrementally growing configuration description. This approach emphasizes the synthesis aspect of configuration, which continuously acquires requirements and computes their effects on a configuration in a cyclic way. We provide the definitions of needed ingredients as there are partial configuration, configuration decision, and reasoning for computing entailments of made configuration decisions. These ingredients are the basis for implementations of configuration systems that follow these approach.

1 Introduction

Configuration is the task of composing a valid system description from known component definitions (*configuration model*) and customer requirements. Typical configuration approaches map this task to reasoning techniques such as constraint solving [Sabin and Freuder, 1996; John, 2002], Description Logics [McGuinness, 2003], or Answer Set Programming [Soininen *et al.*, 2001]. Through this mapping, appropriate reasoners solve the configuration task by computing a solution of their respective logical reasoning problems.

These approaches make a fundamental assumption, i.e. that the requirements of the configuration task can be initially given, e.g., through a *a set of requirements* – see also [Sabin and Weigel, 1998] which call this approach *batch configuration*. But for many, not simple, configuration tasks this does not hold [Neumann, 1988; Stumptner *et al.*, 1998; Fleischanderl *et al.*, 1998]. This is due to the fact, that only during the configuration experience, when the configured product grows in front of the user's eye, the user realizes their own desires and needs [Simonson, 2003]. For example, during a sales conversation, if a requirement causes the need of a subsystem, previously not recognized, additional requirements come into account that are related to the subsystem. Thus, the requirements are not completely clear in the beginning but change during the configuration undertaking. Simple examples are web-based configurators for consumer products such as cars or electronic items that lead the user through a sequence of web-pages for successively acquiring requirements. Other examples are industrial configurators which firstly acquire features of a system and than configure system specific components, like it is described in [Haag, 1998; Ranze *et al.*, 2002; Hotz *et al.*, 2006] – see also [Sabin and Weigel, 1998] which call this approach *incremental configuration*.

These considerations lead to a configuration approach that combines reasoning techniques with the characteristic of a *configuration process*. Process-related subtasks are identification of next steps in the configuration process or managing partial configurations. Especially the retraction of decisions previously made by a user is a characteristic of configuration processes [Günter and Cunis, 1992; Hotz and Wolter, 2013].

Configuration approaches that take only a certain reasoning technology into account, such as configuration based on Description Logics [McGuinness, 2003] or pure constraint processing [Tsang, 1993], have to build an external architecture (or even user interface) around the reasoning kernel, which handles configuration process tasks. Such approaches consider a configuration process as a sequence of changing but fully defined configuration tasks. However, they leave the process-related subtasks to the external architecture and do not integrate them in a configuration system. These approaches lead to domain dependent non-declarative implemented configuration processes.

If a configuration system integrates reasoning facilities and process or procedural aspects, such as e.g. [Günter and Cunis, 1992; Stumptner *et al.*, 1998; Fleischanderl *et al.*, 1998; Günter and Hotz, 1999], the inherent dynamic aspects of configuration tasks can be solved in a general, domain independent way, based on the configuration model. This view is also supported by the early definitions or thoughts about configuration as a syntheses task, in opposite to an analysis task like diagnosis [Brown and Chandrasekaran, 1989; Cunis *et al.*, 1989; Günter and Cunis, 1992; Brown, 1996; Günter and Kühn, 1999].

Other approaches that focus on such aspect of configuration are generative constraints. In [Stumptner *et al.*, 1998] the incrementing configuration is modeled through specific variables that are activated if certain parts of a configured system come into play. This approach is similar to the one defined in the following, however, we do not map the generative notion to a certain reasoning technology (such as constraints in

[Stumptner *et al.*, 1998]), but we provide a framework around a reasoning technology that applies it on subsequently defined new configuration tasks. This is done by iteratively including requirement acquisition in the configuration process. Furthermore, we will not concentrate on a certain reasoning technique, but provide a general framework.

Thus, in this position paper, we elaborate on basic characteristics of the configuration process which lead to a complex mapping of the configuration task to reasoning techniques. Newly introduced operators allow a definition of the dynamics of the configuration process. These include definitions for components and restrictions as well as requirements as usual. Additionally, they provide notions for partial configurations and requirement variables. Furthermore, operators for computing requirement variables, acquiring requirements, and for the actual reasoning will lead to a comprehensive definition of the configuration problem.

Thus, we focus on the iterative characteristic of configuration, i.e. incrementing configuration with intermediate partial configurations (see Section 2). By taking this view, the paper furthermore tries to clarify the relationship of configuration to other reasoning techniques (see Section 3).

We follow ideas published in [Cunis *et al.*, 1989; Günter, 1995]. However, by introducing process related definitions and operators, a summary of the needed ingredients is established that shall give the basis for process-based configuration technologies.

2 General Definitions

In the following, through definitions that build on each other, we develop a definition of a complex configuration problem that take the iterative character of the configuration process into account. We develop a general definition of the complex configuration task that is independent of a certain knowledge representation, such as logic or constraint-based approaches.

First, we provide the definition of a configuration model. This model defines all possible configurations in a generic way. The model represents entities to be configured (such as hardware components, software modules, or services) and relations between them.

We here discuss a combination of entities (e.g., represented with component types or classes) and relations (e.g., represented with constraints). However, the operators defined in the following are independent of the representation. For example, if a pure constraint-based representation is initially used, appropriate operators should have to be developed for supporting the here considered complex configuration tasks.

The definitions are illustrated with an example of composing a menu with antipasti, main course, and dessert. While antipasti will always be selected if a hearty menu is desired, the selection of a dessert cannot be computed from constraints. A hearty menu is part of initial requirements, while the dessert is only selected after the main course, demonstrating a dynamic requirements acquisition.

Definition 1 (Configuration Model)**.** A configuration model $\mathcal{CM}$ is a generic description of entities of an application domain. $\mathcal{CM}$ is a tuple $\langle \Gamma, \Psi, \Phi \rangle$, where

- Γ is a set of attributed entity models $\mathcal{EM} \in \Gamma$ (e.g. classes or concepts) each representing a set of concrete entities to be configured. An entity model consists of named properties. Each property $\mathcal{P}$ is a binary relation that maps from $\mathcal{EM}$ to a *property domain* $\mathcal{PD}_\mathcal{P}$.
- Ψ is a set of property domains $\mathcal{PD}_\mathcal{P} \in \Psi$ of properties $\mathcal{P}$. A $\mathcal{PD}_\mathcal{P}$ might be a *structural property domain* (and $\mathcal{P}$ is called *structural property*) with a set of *structural property value*s, i.e. an entity model optionally combined with a cardinality. Or it might be a primitive datatype, such as a number, symbol, or string or a probably infinite set (e.g. for number ranges) of those, than $\mathcal{P}$ is called *data-type property*.
- Φ is a set of n-ary relations $\mathcal{ER} \in \Phi$ between properties of entity models.

Example 1. *An entity model with one data-type property, one structural property, and a n-ary relation representing the fact that a menu should consist in any case of a main course and might optionally has an antipasti and a dessert. Depending on the kind of taste, an antipasti is selected if hearty taste is desired:*

```
(Entity-Model name: Menu
  super-type: Aggregate
  data-properties: ((kindOfTaste {hearty light}))
  hasCourses:
   ((AntiPasti :min 0 :max 1)
    (MainCourse :min 1 :max 1)
    (Dessert :min 0 :max 1)))
(Entity-Model name: AntiPasti
  super-type: Part)
(Entity-Model name: MainCourse
  super-type: Part)
(Entity-Model name: Dessert
  super-type: Part)

(Constraint name: HeartyEqualsToAntiPasti
  Menu.kindOfTaste == hearty <=>
  Menu.hasCourses HAS (AntiPasti :min 1 :max 1))
```

For representing concrete components of an application domain, *entity instances* are used. First, we define a simple entity instance, later on we enhance this definition.

Definition 2 (Simple Entity Instance)**.** A $\mathcal{SEI}_{\mathcal{EM}}$ represents one concrete entity of the application domain. $\mathcal{SEI}_{\mathcal{EM}}$ belongs to an entity model $\mathcal{EM}$ and has all or some properties of $\mathcal{EM}$. The property values are constant values that belong to the respective property values of $\mathcal{EM}$.

For representing customer requirements about the desired configuration, configuration requirements are defined as follows.

Definition 3 (Simple Configuration Requirememts)**.** Simple configuration requirements $\mathcal{SR}$ are a set of simple entity instances with some properties filled with constant values.

Example 2. *Two simple entity instances representing the requirement "Hearty menu with a main course":*

```
(Entity-Instance name: menu-1
  entity-model: Menu
  kindOfTaste: {hearty}
  hasCourses:  {mainCourse-1})
(Entity-Instance name: mainCourse-1
  entity-model: MainCourse)
```

Typically, a configuration task is defined as follows:

Definition 4 (Simple Configuration Task)**.** A configuration task is defined through an entity model $\mathcal{EM}$ and configuration requirements $\mathcal{SR}$.

Definition 5 (Simple Configuration)**.** A configuration is defined through a set of completely filled configuration instances $\mathcal{CI}$.

A knowledge representation technique allows it to represent an configuration model and configuration requirements. Furthermore, it provides reasoning techniques that allow to solve the simple configuration problem.

Definition 6 (Simple Configuration Problem)**.** $\mathcal{CM} = \langle \Gamma, \Psi, \Phi \rangle$ be a configuration model. A *simple configuration problem* in $\mathcal{CM}$ is a tuple $\langle \mathcal{CM}, \mathcal{SR} \rangle$, where $\mathcal{SR}$ is a set of initial simple entity instances. A *solution* (a *configuration*) of the problem $\langle \mathcal{CM}, \mathcal{SR} \rangle$ is a set of simple entity instances that is consistent with $\mathcal{CM}$ and fulfills the configuration requirements $\mathcal{SR}$.

How consistency is concretely defined depends on the knowledge representation. However, in general for structural properties, consistency means that entity instances belong to $\mathcal{S}$ according to the cardinality descriptions of the structural property. Thus, structural relations emphasize a configuration being a collection of related entity instances, while n-ary relations related properties of those instances.

Example 3. *A resulting configuration, the constraint infers the need of an entity instance representing antipasti:*

```
(Entity-Instance name: menu-1
  entity-model: Menu
  kindOfTaste: {hearty}
  hasCourses:  {mainCourse-1 antiPasti-1})
(Entity-Instance name: mainCourse-1
  entity-model: MainCourse)
(Entity-Instance name: antiPasti-1
  entity-model: AntiPasti)
```

These definition provide the basis for configuration tasks: a configuration model, the customer requirements, and a knowledge representation that allows for creating a configuration that fulfills the customer requirements. This definition makes a basic assumption, i.e. that $\mathcal{SR}$, the set of particular customer requirements, are given. In simple, one step configuration problems, such as parameterization of technical systems, this is a reasonable assumption. In more complex applications, the requirements evolve during the configuration process. This observation leads to further definitions.

First, we enhance the definition of a simple entity instance by allowing not only constant values for properties but also subsets.

Definition 7 (Partial Property Domain)**.** Let $\mathcal{P}$ be a property of an entity instance $\mathcal{EI}$ of entity model $\mathcal{EM}$. And let $\mathcal{PD}_{\mathcal{P}}$ be the property value of $\mathcal{P}$ as defined in $\mathcal{CM}$. A partial property domain $\mathcal{PPD}_{\mathcal{P}}$ of $\mathcal{P}$ is a subset of $\mathcal{PD}_{\mathcal{P}}$, i.e. $\mathcal{PPD}_{\mathcal{P}} \subseteq \mathcal{PD}_{\mathcal{P}}$.

Please note that a specific partial property domain is a property domain with one value representing a constant value, e.g. a number of a data-type property or a single instance of a structural property.

Example 4. *Example for a partial property domain of a structural property with one entity instance and two open cardinality definitions:*

```
hasCourses: {mainCourse-1 (AntiPasti :min 0 :max 1)
                          (Dessert :min 0 :max 1)}
```

Definition 8 (Terminal Property Domain)**.** A terminal property domain $\mathcal{TPD}_{\mathcal{P}}$ of property $\mathcal{P}$ is a partial property domain that is marked with *terminal*. A constant value is automatically marked as terminal. Structural property values or sets may be marked through the heuristic operator (see below).

Example 5. *Example for a partial property domains of a data-type property indicated as terminal:*

```
kindOfTaste: {hearty [terminal]}
```

Definition 9 (Entity Instance)**.** An $\mathcal{EI}_{\mathcal{EM}}$ represents one concrete entity of the application domain. $\mathcal{EI}_{\mathcal{EM}}$ belongs to an entity model $\mathcal{EM}$ and has the same properties as $\mathcal{EM}$. The property values might be the same as defined for $\mathcal{EM}$ or subsets of those, they might be partial or terminal property domains. These *partially filled entity instances* represent uncertain knowledge about the concrete entity. A *completely filled entity instance* has a terminal property domain for *each* property.

Example 6. *One partially filled entity instance:*

```
(Entity-Instance name: menu-1
  entity-model: Menu
  kindOfTaste: hearty
  hasCourses:  {mainCourse-1 (AntiPasti :min 0 :max 1)
                             (Dessert :min 0 :max 1)})
```

Example 7. *One completely filled entity instance:*

```
(Entity-Instance name: menu-1
  entity-model: Menu
  kindOfTaste: {hearty [terminal]}
  hasCourses:  {mainCourse-1 antiPasti-1 [terminal]})
```

Definition 10 (Partial Configuration)**.** A partial configuration $\mathcal{PO}$ is a set of partially or completely filled entity instances.

Definition 11 (Configuration Requirememts)**.** Configuration requirements $\mathcal{R}$ are a set of instances with some property values set to terminal property domains.

Thus, configuration requirements are a specific kind of partial configuration namely one with instances whose properties do not have a terminal property domain for each property. We also call this partial configuration *initial partial configuration*.

Definition 12 (Final Configuration)**.** A final configuration $\mathcal{FC}$ is a set of completely filled entity instances. Furthermore, for each structural property of an instance in $\mathcal{FC}$, a related instances exists in $\mathcal{FC}$ according to the structural property value (i.e. the defined entity model and the cardinality).

Now, probably the main step in our definitions follows, i.e. the introduction of the definition of a variable. Typically, properties of components are considered as variables and the configuration task is to provide a value for these variables, i.e. for the properties of the components. In our definition, a variable stands for a decision that has to be made for gaining a final configuration. Thereby, each variable stands for one property of an entity instance that has to be determined. However, during the configuration process there might be several variables for one property, e.g. if a property value is reduced by the user in several steps.

Definition 13 (Variable)**.** A variable $\mathcal{V}$ represents one decision of setting the value of one certain property. One or more decisions have to be made for each property of every entity instance. The variable represents possible outcomes of the decision through its variable domain $\mathcal{V}_d$. A variable domain is a property value.

Example 8. *Variable representing the decision that antipasti and dessert shall be selected as courses of a menu:*

```
(Variable
 entity-instance: menu-1
 property: hasCourses
 property-value: {mainCourse-1
                  (AntiPasti :min 0 :max 1)
                  (Dessert :min 0 :max 1)})
```

Definition 14 (Reduced Variable)**.** A reduced variable $\mathcal{R}_V$ of a variable $\mathcal{V}$ with a domain $\mathcal{V}_d$ is a variable with a domain $\mathcal{R}_{V_d}$ with $\mathcal{R}_{V_d} \subset \mathcal{V}_d$. The reduced domain might also be a terminal property domain. For a structural property with a structural property value, the subset is a set of instances that are conform with the cardinality descriptions of the structural property value.

The reduced variable represents the result of a made decision.

Definition 15 (Heuristic Operator)**.** A heuristic operator $\mathcal{HO}$ is an operator that takes a variable $\mathcal{V}$ and computes a reduced variable $\mathcal{R}_\mathcal{V}$ (probably with a terminal property domain) by some heuristic method, thus: $\mathcal{HO}$: $\mathcal{V} \rightarrow \mathcal{R}_\mathcal{V}$.

The heuristic operator represents the method for gaining a reduced variable, e.g. the selection of a default value, the computation of a function computing a reduced domain for the variable, or *the acquisition of a value for that variable from the user*. Thus, the heuristic operator acquires subsequent requirements that come up during the configuration process. Furthermore, by reducing a structural property the heuristic operator incrementally expands the configuration, because new entity instances are created when reducing the structural property (see above). A new entity instance has the same properties and property values as its entity model.

Example 9. *Reduced variable created by a heuristic operator that asks the user, if a dessert is needed, answer was "yes":*

```
(Variable
 entity-instance: menu-1
 property: hasCourses
 property-value: {mainCourse-1
                  (AntiPasti :min 0 :max 1)
                  dessert-1)})
```

Example 9 represents the answer to the typically raised question after a meal "Would you like a dessert?", which is a simplistic example for a dynamic requirement acquisition.

Definition 16 (Entailment Operator)**.** An entailment operator $\mathcal{EO}$ is an operator that takes the configuration model $\mathcal{CM}$ (especially the defined n-ary relations), a partial configuration $\mathcal{PC}_i$, and one reduced variable $\mathcal{R}_\mathcal{V}$ and computes a new partial configuration $\mathcal{PC}_{i+1}$ which contains the value of the reduced variable and all entailments of this reduction, computed by some reasoning method, thus:
$\mathcal{EO}$: $\mathcal{CM}$, $\mathcal{PC}_i$, $\mathcal{R}_\mathcal{V} \rightarrow \mathcal{PC}_{i+1}$.
In the initial case, $\mathcal{R}_\mathcal{V}$ might be empty, i.e. $\mathcal{EO}$: $\mathcal{CM}$, $\mathcal{PC}_0$, $\rightarrow \mathcal{PC}_1$ computing the entailments of the values in $\mathcal{PC}_0$, i.e. the initial partial configuration.

The entailment operator represents the integration of one made decision into a partial configuration and the computation of the influences of this decision to the partial configuration. The influences are computed on the basis of the configuration model, especially the n-ary relations, which relate properties of entity models. An influence or entailment is a reduction of domains of some variables in $\mathcal{PC}_i$. Typical examples for an entailment operator are the solution of a constraint problem or applying Description Logic services.

Example 10. *Reduced variable created by an entailment operator that uses the constraint for deciding that an antipasti is needed if a hearty menu was selected:*

```
(Variable
 entity-instance: menu-1
 property: hasCourses
 property-value: {mainCourse-1
                  antiPasti-1
                  (Dessert :min 0 :max 1)})
```

Definition 17 (Open Issue Operator)**.** An open issue operator $\mathcal{OIO}$ is an operator that takes the configuration model $\mathcal{CM}$ and a partial configuration $\mathcal{PC}$ and computes new variables $\mathcal{V}_i$ that have no terminal property domains and are collected in the set $\mathcal{A}$, thus: $\mathcal{OIO}$: $\mathcal{CM}$, $\mathcal{PC} \rightarrow \mathcal{A}$, with $\mathcal{V}_i \in \mathcal{A}$.

From Example 6 the $\mathcal{OIO}$ computes the variable shown in Example 8 because of the partial property domain for property `hasCourses`.

Definition 18 (Select Operator)**.** A select operator $\mathcal{SO}$ is an operator that select one variable $\mathcal{V}$ out of the set $\mathcal{A}$ by some selection method, thus: $\mathcal{SO}$: $\mathcal{A} \rightarrow \mathcal{V}$.

Now, we define the actual configuration process and identify its main part, i.e. the *configuration cycle*. A configuration process starts with an initial partial configuration given through the requirements of a customer. By successively applying the above operators the final configuration will be created. This process is defined as follows:

Starting from the initial partial configuration $\mathcal{IP}$, $\mathcal{EO}$ computes the first entailments and, thus, $\mathcal{PC}_1$. Hereafter, the open issue operator $\mathcal{OIO}$ can be applied to $\mathcal{PC}_1$ for computing next variables with non-terminal property domains and builds $\mathcal{A}_1$. The operator $\mathcal{SO}$ selects a next variable to be decided $\mathcal{V}_1$. From here the heuristic operator $\mathcal{HO}$ reduces the variable's domain to $\mathcal{RV}_1$. The entailment operator uses the previous partial configuration ($\mathcal{EO}(\mathcal{PC}_1)$) for computing the next partial configuration $\mathcal{PC}_2$. This way the operators are successively applied until the final configuration $\mathcal{PC}_n$ is created and no more open issues can be identified (i.e. $\mathcal{OIO}$ computes an empty set). In total, we have:

$$\mathcal{IP}, \xrightarrow{\mathcal{EO}} \mathcal{PC}_1 \xrightarrow{\mathcal{OIO}} \mathcal{A}_1 \xrightarrow{\mathcal{SO}} \mathcal{V}_1 \xrightarrow{\mathcal{HO}} \mathcal{RV}_1$$
$$\xrightarrow{\mathcal{EO}(\mathcal{PC}_1)} \mathcal{PC}_2 \xrightarrow{\mathcal{OIO}} \mathcal{A}_2 \xrightarrow{\mathcal{SO}} \mathcal{V}_2 \xrightarrow{\mathcal{HO}} \mathcal{RV}_2$$
$$\xrightarrow{\mathcal{EO}(\mathcal{PC}_2)} \mathcal{PC}_3 \xrightarrow{\mathcal{OIO}} \mathcal{A}_3 \xrightarrow{\mathcal{SO}} \mathcal{V}_3 \xrightarrow{\mathcal{HO}} \mathcal{RV}_3$$
$$\ldots$$
$$\xrightarrow{\mathcal{EO}(\mathcal{PC}_{n-2})} \mathcal{PC}_{n-1} \xrightarrow{\mathcal{OIO}} \mathcal{A}_{n-1} \xrightarrow{\mathcal{SO}} \mathcal{V}_{n-1} \xrightarrow{\mathcal{HO}} \mathcal{RV}_{n-1}$$
$$\xrightarrow{\mathcal{EO}(\mathcal{PC}_{n-1})} \mathcal{PC}_n \xrightarrow{\mathcal{OIO}} \emptyset$$

Thus, the general configuration cycle is defined as follows:

Definition 19 (Configuration Cycle)**.** A configuration cycle $\mathcal{COC}$ is a sequence of operators $\mathcal{OIO}, \mathcal{SO}, \mathcal{HO}, \mathcal{EO}$ as follows:

$$\mathcal{PC}_i \xrightarrow{\mathcal{OIO}} \mathcal{A}_i \xrightarrow{\mathcal{SO}} \mathcal{V}_i \xrightarrow{\mathcal{HO}} \mathcal{RV}_i \xrightarrow{\mathcal{EO}(\mathcal{PC}_i)} \mathcal{PC}_{i+1}$$

Definition 20 (Complex Configuration Problem)**.** $\mathcal{CM} = \langle \Gamma, \Psi, \Phi \rangle$ be a configuration model. A *complex configuration problem* in $\mathcal{CM}$ is a tuple $\langle \mathcal{CM}, \mathcal{R}, \mathcal{HO}, \mathcal{EO}, \mathcal{SO}, \mathcal{OIO} \rangle$, where $\mathcal{R}$ is a set of initial entity instances and $\mathcal{HO}, \mathcal{EO}, \mathcal{SO}, \mathcal{OIO}$ the previously introduced operators. A *solution* of the problem $\langle \mathcal{CM}, \mathcal{R}, \mathcal{HO}, \mathcal{EO}, \mathcal{SO}, \mathcal{OIO} \rangle$ is a final configuration that was computed by applying the configuration cycle, that is consistent with $\mathcal{CM}$, and that fulfills the configuration requirements $\mathcal{R}$.

Like in the simple configuration problem definition, how consistency is concretely defined depends on the knowledge representation. Furthermore, the knowledge representation defines the entailment operator.

3 Discussion

The commonly used view on configuration is to specify a configuration model and customer requirements as a reasoning task of a reasoning system, such as a constraint system, and than solve the reasoning task and present the resulting configuration. This paper provides a view on configuration that basically iterates these two steps of defining requirements and reason about them, i.e.:

1. Start from an initial configuration including the requirements,
2. compute entailment of the requirements on the configuration (operator $\mathcal{EO}$),
3. create an agenda with not yet made decisions (operator $\mathcal{OIO}$),
4. select one decision (operator $\mathcal{SO}$),
5. make the decision (operator $\mathcal{HO}$), and
6. compute its entailments (operator $\mathcal{EO}$),
7. goto Step 3.

The computation of the entailments (Step 2 and Step 6) correspond to the solution of the reasoning task, e.g. by constraint processing. The construction of a reasoning task is included in the configuration process in the steps 3, 4, and 5.

Thus, the overall schema for configuration as seen in our approach provides an iterative application of commonly applied reasoning techniques for configuration. For solving a complex configuration problem, a configuration system or a technological approach should realize the operators defined above.

Of course, there exist variations of the here presented basic framework. For example, the proposed approach to represent requirements with instances might be enhanced to complex requirements that need further reasoning to compute them [Thäringen, 1995; Kopisch and Günter, 1992]. Or the selection of a decision may be enhanced to selecting multiple decisions or to let the user select next decisions from the agenda. Another extension is to include techniques for conflict resolution [Günter and Hotz, 1995; Felfernig and Schubert, 2010; Hotz and Wolter, 2013]. However, this paper provides the basic ingredients for solving a configuration task incrementally. Configuration tools which follow our approach are KONWERK [Günter and Hotz, 1999], engcon [Hollmann *et al.*, 2000], or Plakon [Cunis *et al.*, 1989].

4 Summary

This paper defines the necessary ingredients for a configuration process that iteratively generates a configuration. Besides the typically used reasoning techniques, the process additionally supplies steps for creating requirements on the fly and include them and their entailments in a growing configuration. Enhancements in future work will be the inclusion of operators for resolving conflicts that might occur during the configuration process.

References

[Brown and Chandrasekaran, 1989] D.C. Brown and B. Chandrasekaran. *Design Problem Solving - Knowledge Structures and Conrtol Strategies*. Research Notes in Artificial Intelligence Series. Pitman Publishing, London, 1989.

[Brown, 1996] D.C. Brown. Some Thoughts on Configuration Processes. AAAI 1996 Fall Symposium Workshop: Configuration FS-96-03, MIT, Cambridge, Massachusetss, USA, 1996.

[Cunis *et al.*, 1989] R. Cunis, A. Günter, I. Syska, H. Peters, and H. Bode. PLAKON - An Approach to Domain-Independent Construction. In *Proc. of Second Int. Conf. on Industrial and Engineering Applications of AI and Expert Systems IEA/AIE-89*, pages 866–874, June 6-9 1989.

[Felfernig and Schubert, 2010] A. Felfernig and M. Schubert. Diagnosing Inconsistent Requirements. In L. Hotz and A. Haselböck, editors, *Proc. of the Configuration Workshop on 19th European Conference on Artificial Intelligence (ECAI-2010)*, Lisbon, Portugal, August 2010.

[Fleischanderl *et al.*, 1998] Gerhard Fleischanderl, Gerhard E. Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, July/August 1998.

[Günter and Cunis, 1992] A. Günter and R. Cunis. Flexible Control in Expert Systems for Construction Tasks. *Journal Applied Intelligence*, 2(4):369–385, 1992.

[Günter and Hotz, 1995] A. Günter and L. Hotz. Auflösung von Konfigurationskonflikten mit Wissensbasiertem Backtracking und Reparaturanweisungen (Conflict Resolution with Knowledge-based Backtracking and Repair-Statement). In A. Günter, editor, *"Wissensbasiertes Konfigurieren"*, St. Augustin, 1995. Infix.

[Günter and Hotz, 1999] A. Günter and L. Hotz. KONWERK - A Domain Independent Configuration Tool. *Configuration Papers from the AAAI Workshop*, pages 10–19, July 1999.

[Günter and Kühn, 1999] A. Günter and C. Kühn. Knowledge-Based Configuration - Survey and Future Directions. In F. Puppe, editor, *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, Springer Lecture Notes in Artificial Intelligence 1570, Würzburg, March 3-5 1999.

[Günter, 1995] A. Günter. *Wissensbasiertes Konfigurieren (Knowledge-based Configuration)*. Infix, St. Augustin, 1995.

[Haag, 1998] A. Haag. Sales Configuration in Business Processes. *IEEE Intelligent Systems*, pages 78–85, July August 1998.

[Hollmann *et al.*, 2000] O. Hollmann, T. Wagner, and A. Günter. EngCon: A Flexible Domain-Independent Configuration Engine. In *Proc. ECAI-Workshop Configuration*, page 94 pp, Berlin, Germany, August 21-22 2000.

[Hotz and Wolter, 2013] Lothar Hotz and Katharina Wolter. Beyond Physical Product Configuration - Configuration in Unusual Domains. *AI Commun.*, 26(1):39–66, 2013.

[Hotz *et al.*, 2006] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, and J. MacGregor. *Configuration in Industrial Product Families - The ConIPF Methodology*. IOS Press, Berlin, 2006.

[John, 2002] U. John. *Konfiguration und Rekonfiguration mittels Constraint-basierter Modellierung (Configuration and Reconfiguration by Means of Constraint-Based Modeling)*. Infix, St. Augustin, 2002.

[Kopisch and Günter, 1992] M. Kopisch and A. Günter. Configuration of a Passenger Aircraft Cabin - based on Conceptual Hierarchy, Constraints and Flexible Control. In F. Belli and F.J. Radermacher, editors, *Proceedings of IEA/AIE*, Paderborn, 1992. Springer-Verlag.

[McGuinness, 2003] D. L. McGuinness. Configuration. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *Description Logic Handbook*, pages 397–413. Cambridge University Press, 2003.

[Neumann, 1988] B. Neumann. Configuration Expert Systems: A Case Study and Tutorial. In Bunke, editor, *Proc. 1988 SGAICO Conference on Artificial Intelligence in Manufacturing, Assembly, and Robotics*. Oldenbourg, Munich, 1988.

[Ranze *et al.*, 2002] K.C. Ranze, T. Scholz, T. Wagner, A. Günter, O. Herzog, O. Hollmann, C. Schlieder, and V. Arlt. A Structure-Based Configuration Tool: Drive Solution Designer DSD. *14. Conf. Innovative Applications of AI*, 2002.

[Sabin and Freuder, 1996] D. Sabin and E.C. Freuder. Configuration as Composite Constraint Satisfaction. In *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996.

[Sabin and Weigel, 1998] Daniel Sabin and Reiner Weigel. Product Configuration Frameworks - A Survey. *IEEE Intelligent Systems*, pages 42–49, 1998.

[Simonson, 2003] I. Simonson. Determinants of Customer's Responses to Customized Offers: Conceptual Framework and Research Propositions. *Stanford GSB Working Paper No. 1794*, 2003.

[Soininen *et al.*, 2001] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing Configuration Knowledge with Weight Constraint Rules. In Alessandro Provetti and Tran Cao Son, editors, *1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge*, pages 195–201, 2001.

[Stumptner *et al.*, 1998] M. Stumptner, G. Friedrich, and A. Haselböck. Generative Constraint-based Configuration of Large Technical Systems. *AI EDAM*, 12(04):307–320, 1998.

[Thäringen, 1995] M. Thäringen. Wissensbasierte Erfassung von Anforderungen (Knowledge-based Acquisition of Requirements). In A. Günter, editor, *Wissensbasiertes Konfigurieren*. Infix, 1995.

[Tsang, 1993] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, San Diego, New York, 1993.