Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2013), *27*, 37–49.

doi:10.1017/S0890060412000376

Efficient large-scale configuration via integer linear programming

INGO FEINERER

Institute of Information Systems, Vienna University of Technology, Vienna, Austria

(Received June 26, 2011; Accepted July 13, 2012)

Abstract

Configuration of large-scale applications in an engineering context requires a modeling environment that allows the design engineer to draft the configuration problem in a natural way and efficient methods that can process the modeled setting and scale with the number of components. Existing configuration methods in artificial intelligence typically perform quite well in certain subareas but are hard to use for general-purpose modeling without mathematical or logics background (the so-called knowledge acquisition bottleneck) and/or have scalability issues. As a remedy to this important issue both in theory and in practical applications, we use a standard modeling environment like the Unified Modeling Language that has been proposed by the configuration community as a suitable object-oriented formalism for configuration problems. We provide a translation of key concepts of class diagrams to inequalities and identify relevant configuration aspects and show how they are treated as an integer linear program. Solving an integer linear program can be done efficiently, and integer linear programming scales well to large configurations consisting of several thousands components and interactions. We conduct an empirical study in the context of package management for operating systems and for the Linux kernel configuration. We evaluate our methodology by a benchmark and obtain convincing results in support for using integer linear programming for configuration applications of realistic size and complexity.

Keywords: Configuration; Integer Linear Programming; Linux Kernel Configuration; Package Management; Unified Modeling Language

1. INTRODUCTION

The design and configuration of software and hardware systems has a long tradition as a key discipline in computer science and artificial intelligence (AI). A configuration (task) is "characterized through a set of components, a description of their properties, namely attributes and possible attribute values, connection points (ports), and constraints on legal configurations" (Felfernig et al., 2000, p. 450). The Alliance for Telecommunications Industry Solutions (2000) defines configuration as the arrangement of functional units according to their nature, number, and chief characteristics where functional units may encompass any form of component (e.g., integrated circuits, software programs, or building parts in a car). A configuration task means the process of setting up admissible arrangements under certain criteria of optimality (e.g., minimal number of components or preferences when connecting components).

Important criteria for the solution of a configuration task are the computation of the components to be included in the solution, the type of information on each of these components, the way the components are connected, and the values for the attributes of each component (Mailharro, 1998).

Existing methods and results in AI tackle a broad spectrum of applications. However, especially in an engineering context we are faced with unique challenges. The complexity and size of real-world systems is steadily increasing (e.g., the number of transistors in recent multicore processors or the number of product configurations in the automotive sector). As a consequence, there is a big demand for efficient techniques in configuration that scale well with large systems consisting of tens of thousands of components and even more interactions among them.

Tool support for assisting the engineer in designing such large systems is fundamental as manual handling and inspection becomes inherently complex. Moreover, the engineer has to deal with a diverse set of configuration tasks: checking the consistency of component systems, reasoning over system properties, finding minimal models, repairing inconsistent configurations due to modifications in the original design, or finding a viable layout for components under physical constraints. Numerous additional constraints like subclassing or

Reprint requests to: Ingo Feinerer, Institute of Information Systems, Vienna University of Technology, Favoritenstraße 9, A-1040 Wien, Austria. E-mail: ingo.feinerer@tuwien.ac.at

equational constraints on links between components might occur depending on the modeled scenario.

Proper handling of such tasks requires rigorous techniques, known as formal methods in computer science. Unfortunately, a main drawback of powerful formal methods is the so-called knowledge acquisition bottleneck because the design engineer must have a substantial mathematical background for understanding and applying a formal methodology correctly. Another drawback of such machinery is their high computational complexity (Stumptner, 1998; Falkner et al., 2011) on larger use cases.

Consequently, lightweight formal methods have been proposed (Jackson & Wing, 1996) to ease the implementation of formal techniques in real-world systems, to tackle the knowledge acquisition bottleneck, and to allow reasoning in a (semi)automatic way. Lightweight formal methods are characterized by "partiality and focused application" (Jackson & Wing, 1996). That means full formalization of complete systems with all properties is not tractable owing to the expressiveness of employed formalisms and their inherent complexity. Instead, we should focus on relevant properties and employ efficient methods to check them. In the end, the success of formal methods (lightweight or not) will also depend on whether the engineer is comfortable with using them in his design environment.

The Unified Modeling Language (UML; Object Management Group, 2011) is one of the most prominent modeling frameworks, and it has also been increasingly used for the design of component systems for configuration (Felfernig et al., 2000, 2002, 2003; Falkner et al., 2010). A range of techniques for formal reasoning on UML diagrams exists; however, basically all of them are computationally quite expensive because they use expressive formalisms (like first-order logic) for capturing the intended semantics of UML. This is especially relevant when dealing with numerical constraints that are very common in configuration (e.g., relating the number of components that are allowed to interact with each other).

Following the spirit of lightweight formal methods and combining them with widely used modeling frameworks like UML, we therefore recently proposed to use integer linear programming (ILP) by translating the most basic constructs (classes, associations, and multiplicities) of UML class diagrams to Diophantine inequalities (Feinerer, 2007; Feinerer & Salzer, 2007), that is, inequalities over variables with integer domain. Although addressing important basic aspects, several open issues have been identified that are highly relevant in real-world configuration applications (Falkner et al., 2010) but are not yet fully dealt with, including subclassing, partial configurations, and immediate feedback.

Thus, our primary objectives are lightweight formal methods for solving a configuration task supporting the full range of language elements necessary for modeling real-world applications, which are highly efficient and which scale well with large systems in the hardware and software domains. To achieve these goals, the main contributions of this paper are a unified theory using ILP principles that allows us to model real-world configuration scenarios and constraints and an experiment consisting of two large-scale applications with benchmarks to show the applicability of our approach. We see our approach as a further step toward exploring tractable cases in configuration (in contrast to the NP-hardness of configuration in general) and providing intuitive interfaces toward achieving this goal.

Solving an ILP corresponds to checking the consistency of a configuration (i.e., can the system of inequalities be solved at all) and to finding optimized models (because we will minimize a corresponding objective function). Partial configurations can be used as initial solutions in an ILP (the number of existing components corresponds to the variable values in the ILP), and immediate feedback is possible because the variables in the solution of the ILP correspond to components in a configuration. Note that we ignore attributes of components in our approach. Connecting the components computed by the ILP is not covered in this paper; however, an algorithm based on uniform distribution of links by Feinerer (2007) can be used for generating a topology (i.e., how components are linked) from an ILP solution.

Because the generation of an ILP is automatic for the main constructs of a given UML class diagram, we are able to provide an intuitive interface. UML class diagrams can be easily created and maintained with any standard compliant diagram editor. Customized plug-ins and tools can directly display inconsistencies detected by our underlying ILP methods. This gives us a visual representation that is easy to use and interpret for a broad range of users owing to UML standard compliance. Nonetheless, we have an exact syntax and semantics for formal reasoning under the hood by restricting the allowed language elements to a well-defined subset of UML class diagrams.

The two applications presented in this paper are configuration tasks in the context of package management for the Debian GNU/Linux operating system and for the Linux kernel. Both applications were chosen because their sources are publicly available, which allows for better reproducible research (compared to many examples that are limited in distribution because they come from an industrial setting with restrictive permissions); are large scale with thousands of components; and are of interest to a broad audience (e.g., package management is a core task in every Debian GNU/Linux installation worldwide).

The remainder of this paper is structured as follows. Section 2 demonstrates how central concepts necessary for configuration can be modeled with UML class diagrams or similar modeling frameworks. Section 3 shows how to translate such diagrams to an integer linear program and how to encode concept hierarchies, multiple links, and conflicts as linear constraints. This forms a unified theory that can be solved with any ILP solver. Section 4 describes both application settings and how they are modeled in our framework, and Section 5 shows a benchmark experiment and evaluates our ILP approach for both applications. Section 6 puts our work in context with related work, and Section 7 presents our conclusions.

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

2. MODELING SYSTEMS FOR CONFIGURATION

Before configuring technical systems, an engineer needs to have an exact specification of its underlying structure. The main work consists of specifying functional units and components and their interrelationships. Numerous additional constraints can be added once the fundamental interactions are clear. Modeling entities and their relationships has a long history in computer science, with entity relationship (Chen, 1976) diagrams and UML (Object Management Group, 2011) diagrams as the most commonly used formalisms nowadays. Because of their widespread use over the last decades and the availability of a multitude of (both commercial and open source) modeling tools, most engineers are familiar with such frameworks. Using such a formalism is therefore a natural and conservative starting point when modeling systems of interest as the basis for configuration (Felfernig et al., 2000).

2.1. General case

Consider the example depicted in Figure 1. It models two components, C and D. Each component C must have between between m_1 and m_2 partners of type D, and each component D must have between n_1 and n_2 partners of type C in turn. In UML terminology, C and D are classes, the relationship between them is denoted as an association, and the numerical intervals constraining the association are called multiplicities or cardinalities.

Such a diagram already captures our intuition about the desired semantics. However, there are several hidden intricacies.

2.2. Lower bound constraints

Without additional constraints, any formal reasoner will refrain from instantiating any objects at all when searching for minimal models. This is because UML diagrams do not implicitly enforce the existence of individual objects. This can be easily fixed with a constraint stating there needs to be at least one C object, for example, expressed in the Object Constraint Language (OCL; Object Management Group, 2012):

$$\text{context C inv: C.allInstances()}{->}\text{size()} > 0.$$

2.3. Multiple links

Next, it makes a difference whether we allow multiple links between the same objects or not. In some cases it is desirable, like multiple redundant network cables between two switches, whereas often it is not and an indication of a design error. UML allows us to change this behavior via the multiplicity attribute "(non)unique." The default is unique, which forbids (or ignores) multiple links between the same objects, and we stay with this convention when not otherwise explicitly stated.

Fig. 1. Two classes, C and D, a binary association between them, and multiplicities restricting the number of allowed links.

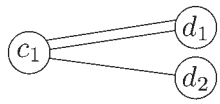

Fig. 2. A valid instance of Figure 1 for $n_1 = 1, n_2 = 2, m_1 = 3, m_2 = 4$, when the end of the association near D is marked as nonunique.

When mixing unique and nonunique on the same association, there are subtleties with the semantics of a valid instance of the UML class diagram. Consider Figure 1 and assume that $n_1 = 1$, $n_2 = 2$, $m_1 = 3$, and $m_2 = 4$ and that the end of the association near D is marked as nonunique. Then Figure 2 shows a valid instance with only two D objects (note that $m_1 = 3$). This is because c_1 has three different links to objects of class D and is not necessarily connected with three different objects, corresponding to the nonunique attribute. Consequently, we recommend avoiding mixed associations unless the designer is aware of the underlying semantics of such constructs. Instead, associations with symmetric attributes (i.e., either unique or nonunique on all association ends) can be used to model a broad class of constructs.

Besides the basics of managing functional units and their relationships, several additional features are necessary to allow an engineer to model realistic systems for configuration, like handling mutually exclusive components or conflict handling.

2.4. Subclassing

Subclassing relates similar components in a hierarchy, typically with a single superclass and multiple inherited subclasses. When modeling component systems, subclassing is primarily used to express disjunction in either the normal (logical or) or the exclusive (logical xor) form.

Consider Figure 3, which depicts the situation that two (sub)classes, C_1 and C_2, share a common structure and are thus related to the common superclass C via a generalization in UML. The idea behind this construct is that either C_1 or C_2 can be used when somebody asks for a component of type C.

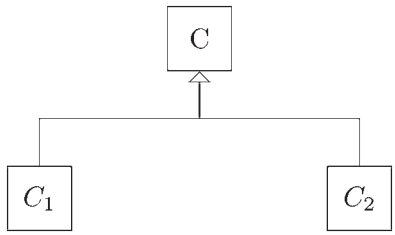

Fig. 3. Generalization with superclass C and two subclasses, C_1 and C_2.

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

If it is allowed that both C_1 and C_2 are instantiated at the same time, we can use the diagram as is. Otherwise, the additional UML constraint {xor} needs to be added (in the form of a dashed line between the associations connecting C_1 with C and C_2 with C).

2.5. Conflicts

Similarly to the constraint that only one out of multiple components can be chosen, some may be incompatible with others. For example, in an automotive scenario, we may not be allowed to mix tires of different types for safety reasons, or installing two different software libraries providing an overlapping set of files may destroy a working operating system setup.

A possible way to encode such a constraint is depicted in Figure 4. The key feature is the 0..0 multiplicity that states that each C object must have 0 links to objects of class D. This directly corresponds to incompatibility. Alternatively, separate constraints outside of the UML diagram may be used to avoid a blow up in the diagram if many incompatible cases are relevant to an application. For example, the OCL constraint

$$\begin{aligned}&\text{context C inv: C.allInstances()}{->}\text{size()} > 0\\ &\qquad\text{implies D.allInstances()}{->}\text{size()} = 0\end{aligned}$$

states that once there is at least one object of type C, then no object of type D is allowed to be instantiated.

3. ILP

In the previous section, we identified central concepts in configuration and how they can be modeled in an entity relationship inspired framework like UML. Now we will translate all constructs to linear inequalities forming a unified ILP theory. The semantics induced by the translation is compliant with the UML specification if not explicitly stated otherwise.

Our idea is based on some initial work by Lenzerini and Nobili (1990) and extensions by Feinerer and Salzer (2007) that translate a binary association to two linear inequalities adhering to the multiplicity constraints.

3.1. General case

A binary association as exemplified in Figure 1 is mapped to

$$\begin{aligned}m_1 \times |C| \leq n_2 \times |D|,\\ n_1 \times |D| \leq m_2 \times |C|,\end{aligned}$$

where $|C|$ and $|D|$ denote the number of C and D objects, respectively. The main observation for correctness of the above translation is that the number of links, l, between the two classes, C and D, is bounded by

$$\begin{aligned}m_1 \times |C| \leq l \leq m_2 \times |C|,\\ n_1 \times |D| \leq l \leq n_2 \times |D|.\end{aligned}$$

Combining these two inequalities results in the above translation.

Fig. 4. A binary association modeling a conflict between the two classes, C and D, via a 0..0 multiplicity.

3.2. Lower bound constraints

Enforcing that individual classes get instantiated at all enforces us to use lower bound constraints. For example, for Figure 1, a valid solution is $|C| = 0$ and $|D| = 0$ because all constraints are satisfied (which is easily verified by inserting $|C| = |D| = 0$ in the inequalities from the previous paragraph). However, normally this is not the desired semantics. Instead, we want to enforce that there is a least one component of type C that maps to the following simple ILP constraint

$$|C| \geq 1.$$

3.3. Multiple links

Inequalities from the general case hold in the presence of the nonunique UML attribute because their correctness relies on an argumentation of involved links. To ensure uniqueness among objects, we need additional constraints stating that links must end with mutually different partner objects. In Figure 1 the association has no explicit attribute markers and thus defaults to unique on both association ends. We obtain

$$\begin{aligned}|C| > 0 \rightarrow |D| \geq m_1,\\ |D| > 0 \rightarrow |C| \geq n_1.\end{aligned}$$

This ensures that if at least one C or D object gets instantiated, there are enough different (i.e., m_1 and n_1) D and C objects available for linking. Note that there is an implication involved in these formula that at first glance is not compatible with standard ILP. However, in our case, we can easily remedy this by solving our ILP and checking the preconditions afterward. Because all preconditions are of the form $|X| > 0$, this can be instantly checked from the ILP solution, and a new (augmented by the right-hand sides of above formulas) ILP is started with the old solution as its initial starting value. Because the inequalities presented so far form a solution space that can be monotonically processed by increasing variable values, this trick works without backtracking. As a result, the compound running time of the old and new ILP linearly corresponds to an ILP where it is known in advance whether the preconditions fire or not.

The inequalities presented so far can be efficiently solved by mapping them to a directed weighted graph. The variables form its nodes, and each inequality relating two variables induces an edge with a weight proportional to the coefficients of

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

the variables. Over such a weighted directed graph a variant of the Floyd-Warshall (Floyd, 1962) algorithm can be used to compute all shortest paths in the polynomial time of $O(V^3)$, where V stands for the total number of variables of all inequalities. These paths can be used for obtaining rational solutions that in turn can easily be translated into a (not necessarily minimal) integer solution. Note that this is not in contrast to the general NP-hardness of ILP. The reason is that we have no constraints of the form $m \times |C| + n \times |D| \leq k$ (with m, n, and k as positive integers) yet in our formulation. Note that the same strategy of building a graph also works for n-ary associations as long as all n ends of the association have the same uniqueness attribute.

Alternatively, any standard ILP solver can be used to obtain the minimal integer solution directly.

3.4. Subclassing

When modeling a generalization as depicted in Figure 3, we first have to decide on the semantics when there is a connection from C to another class, D. One interpretation is that C is a placeholder and D is connected to either C_1 or C_2. This means that C does not get instantiated at all (and as such $|C| = 0$). Alternatively, D can connect to C, and C is connected to either C_1 or C_2. Without loss of generality, we stick with the latter convention because we prefer to model associations explicitly (e.g., between D and C_1 with a separate association if there is a direct correspondence between them without using the generalization). The superclass C can be seen as a virtual coordinator that might ease the actual linking of components considerably. Superclasses can also easily be discarded if there is no need for them in the final solution (e.g., in the hardware domain where only subclasses are likely to be instantiated).

We observe the property that in Figure 3 each C can have zero or one C_1 partner objects, but a single C_1 (if it exists) must have exactly a single distinct C object. The same argument holds for C with C_2. The property is formalized to

$$\begin{aligned} 0 \times |C| &\leq 1 \times |C_1|, \\ 1 \times |C_1| &\leq 1 \times |C|, \\ 0 \times |C| &\leq 1 \times |C_2|, \\ 1 \times |C_2| &\leq 1 \times |C|, \end{aligned}$$

which simplifies to

$$|C_i| \leq |C|,$$

for $i = 1, 2$. This directly corresponds to our intuition that we need at least as many C coordinator objects such that there are no "useless" C_1 or C_2 objects that do not take part in the generalization.

Note that the sharing of a single C object by both an object of class C_1 and an object of class C_2 is allowed. This property models nondisjoint classification of individuals within a conceptual hierarchy, like a disjunction in classical propositional logic.

Thus far we have only ensured that there are enough "coordinators" of class C. This does not suffice because we have to provide enough subclasses (either C_1 or C_2) that can be used in the generalization for the superclass C. The following constraint models (summing up over all C_i and C objects) that we can use C_1, C_2, or both together as a subclass for C:

$$|C_1| + |C_2| \geq |C|.$$

If we restrict the setting to exclusive or ({xor} in UML terminology), then either C_1 or C_2 must exactly match with a single C. Consequently, we obtain

$$|C_1| + |C_2| = |C|,$$

stating that we need enough C_i for matching C objects but not more. This forbids object sharing and thus models disjunction in exclusive form.

We can easily generalize this setting to multiple subclasses C_i with $i > 2$ by considering the additional objects in the sums on the left-hand sides of the above (in)equalities.

3.5. Conflicts

If we model conflicts as outlined in Figure 4 and translate this directly via our standard approach, we obtain

$$\begin{aligned} 0 \times |C| &\leq n_2 \times |D|, \\ n_1 \times |D| &\leq 0 \times |C|. \end{aligned}$$

The first inequality is a tautology because we only allow nonnegative multiplicities (i.e., $n_2 \geq 0$) and nonnegative numbers of classes ($|D| \geq 0$), which thus can be removed without side effects.

The second inequality forbids any D object if $n_1 > 0$. This is semantically correct because when $n_1 = 0$ then there are isolated D objects allowed that need not be related to C objects at all. For the more typical case when a D object must a have connection to a C object (i.e., $n_1 > 0$), then instantiating a D object would mean we also need a C object. This in turn would violate our initial constraint that a C object must not be linked to a D object.

A subtle modification of the desired semantics could be that we want to allow D objects without connection to C objects, but as soon as there exists a single C object, we forbid Ds at all. This models a situation where we want to use parts of category C or D throughout our whole configuration but never allow them to occur together. Such a constraint can be expressed via

$$|C| > 0 \rightarrow |D| = 0.$$

To avoid the implication, we might be tempted to use

$$|C| \times |D| = 0,$$

which enforces that once we use parts of one category no objects of the other class are allowed. Unfortunately, this is a

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

nonlinear constraint that we will not use because we restrict our theory to linear integer programming for performance reasons.

Fortunately, we know that C is only included in a minimal solution if absolutely necessary (as otherwise minimality is violated). The same argument holds for D. Consequently, we know that a solution containing both C and D is invalid and can be instantly rejected. This observation allows us to handle the implication efficiently, with a similar argument as when dealing with the UML "unique" attribute for multiple links.

4. APPLICATIONS

This section describes two applications: one in the context of Debian package management and configuration and one in the Linux kernel configuration.

4.1. Debian package configuration

The Debian GNU/Linux operating system (http://www.debian.org) is one of the most prominent free and open source distributions in the Linux ecosystem. With its more than 29,000 packages, it provides a vast amount of available software out of the box that allows the user to set up individually customized package collections or even new distributions like Ubuntu (http://www.ubuntu.com). The variability (i.e., the number of possible combinations of interacting packages) is very high, and consequently a sophisticated packaging system (like the Advanced Packaging Tool available at http://packages.qa.debian.org/a/apt.html) managing all packages and their interdependencies in an installation is necessary.

Each Debian package typically has metadata providing the following information:

- *Package:* a unique name of the package.
- *Version:* a unique string identifying the version.
- *Depends:* a list of packages that must be installed for proper functionality. Alternatives are allowed such that, for example, only one of multiple packages must be installed providing some core functions.
- *Conflicts:* a list of packages that must not be installed at the same time.
- *Description:* a short (up to several lines) text description of the package and its main functionality.

The following excerpt shows an example for the Debian package emacs23-nox (Emacs without support for graphical environments), listing the individual metadata entries:

Package: emacs23-nox

Version: 23.3+1–1

Depends: emacs23-bin-common (= 23.3+1–1), libasound2 (> 1.0.18), libc6 (>= 2.3.6–6~), libdbus-1–3 (>= 1.1.1), libgpm2 (>= 1.20.4), libncurses5 (>= 5.5–5~)

Suggests: emacs23-common-non-dfsg

Conflicts: emacs23, emacs23-gtk, emacs23-lucid

Description: The GNU Emacs editor (without X support)
GNU Emacs is the extensible self-documenting text editor. This package contains a version of Emacs compiled without support for X.

4.1.1. ILP formulation

Given a Debian package, we extract its metadata and generate an integer linear program modeling the interrelationships between packages as outlined in the metadata:

- *Package* is mapped to a variable in the ILP and corresponds to a class in our UML representation. If and only if the package is part of the input specification by the user, we enforce a lower bound of 1 (assuming we consider package C at the moment):

$$|C| \geq 1.$$

In addition, we disallow multiple installation of the same package with the same version. That is, we need a Boolean range, either the package is installed or not:

$$|C| \geq 0,$$
$$|C| \leq 1.$$

- *Version* is used together with the package name if the package name alone is ambiguous. Versioned dependencies (i.e., when a dependency requires certain versions of a package) are not natively supported in our formulation.
- *Depends* means all package dependencies will introduce new variables (if not already present) of the Boolean range as modeled above. Note that we do not enforce a lower bound because otherwise this would include all packages irrelevant of their necessity (alternatives may allow for only one choice out of multiple packages).

Next the dependencies are represented by 0..1—1..1 associations in our UML class diagram, which directly correspond to following ILP (assuming that C depends on D):

$$|C| - |D| \leq 0,$$

which follows by inserting $n_1 = 0$ and $n_2 = m_1 = m_2 = 1$ in our general case inequalities. In cases where we deal with alternatives (e.g., assuming that C depends either on D or E whereas D and E may be both installed at the same time as long as no explicit conflict relation is stated), we obtain

$$|C| - |D| - |E| \leq 0.$$

Virtual packages (i.e., packages that are logical placeholders for packages with common functionality) could be modeled

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

using subclassing with the virtual package as superclass and the concrete packages as subclasses.

- *Conflicts* are expressed by the following inequality (assuming C conflicts with D):

$$|C| + |D| \leq 1.$$

This encoding works as we operate only on the Boolean range. No packages at all would be correct, also either C or D, but not both together. This models exactly our desired semantics in this setting.

Finally, we glue all of these inequalities together by an objective function including all classes, which gets minimized:

Minimize

$$|C| + |D| + |E|.$$

A main advantage of this objective function and ILP in general is that we can easily add coefficients (weights) that model the importance of the individual variables. For example, for package management we could use the package size (e.g., rounded to MB) as a weighting coefficient, which would mean that smaller packages get preferred over bigger ones.

For the example of the emacs23-nox package, this yields (only one or two inequalities of each category are shown for compact presentation):

Minimize

$$\text{emacs23-nox} + \text{emacs23-bin-common} + \text{libasound2} + \cdots$$

such that

$$\begin{gathered}\text{emacs23-nox} - \text{emacs23-bin-common} \leq 0,\\ \text{emacs23-nox} - \text{libasound2} \leq 0,\\ \text{emacs23-nox} + \text{emacs23} \leq 1,\\ \text{emacs23-nox} + \text{emacs23-gtk} \leq 1,\\ 0 \leq \text{emacs23-nox} \leq 1,\\ \text{emacs23-nox} \geq 1.\end{gathered}$$

The first and second side condition models the dependencies, the third and forth models the conflicts, the fifth states the Boolean range, and the sixth one enforces the selected package to be installed.

4.1.2. Implementation

For our experiment, we used the testing release of the Debian GNU/Linux operating system using its main (http://ftp.debian.org/debian/ testing main) and security (http://security.debian.org/ testing/updates main) repositories in June 2011.

We implemented the previously presented approach for Debian package management in two ways. We use a graphical user interface (GUI) tool called CLEWS (Niederbrucker & Sisel, 2011) capable of manipulating UML diagrams. It was developed as a prototype implementing our lightweight formal methods approach using ILP. We recently extended its functionality by using its application programming interface (API) in the context of software product configuration (Feinerer, 2011). Based on this initial work, we now consider Debian GNU/Linux package configuration and subsequently Linux kernel configuration as shown in this paper.

CLEWS is a pure Java program and provides a Java API. However, to extract the Debian GNU/Linux package information, we used a C++ library of the APT package management tool shipped with Debian. To match both ends, we used the Java Native Interface where we call Java routines for creating classes and associations in the underlying UML diagrams managed by CLEWS. The advantage of using CLEWS is that it allows us to graphically investigate the modeled scenario and rearrange associations and classes and enables a direct translation to an ILP.

Our second implementation avoids the intermediate step of an additional tool like CLEWS and directly translates the package dependency information into an ILP. Again we extract the Debian package metadata via the APT C++ library. Now we use the Mixed ILP solver lp_solve (http://lpsolve.sourceforge.net) and its C API to directly create an ILP.

4.2. Linux kernel configuration

The configuration of a Linux kernel is mainly done by choosing whether certain drivers are directly included in the kernel, are loaded when necessary as modules, or are excluded. This somehow resembles a tristate logic (yes, module, no) inherent to Linux kernel configuration. The whole set of dependencies (i.e., which drivers or modules depend on others, which conflict with others, etc.) is written down in a set of so-called Kconfig files. Several hundred of them exist, typically with a single one in each subdirectory of the kernel source code. Normally each subdirectory contains a specific driver or system architecture, and basically all Kconfig files could be merged to a single large one containing the whole metadata representing constraints of the kernel configuration.

In its full form, Kconfig files are quite complex because they allow different syntax for the same semantics. For example, dependencies might be included because of encapsulating menu entries or may be rewritten with conditional if statements. Conceptually, all these constructs can be broken down to a basic set of primitive operations as follows:

- *Config:* This defines a configuration option.
- *Tristate/string:* This defines the type of the configuration option. We are mainly interested in Boolean and/or tristate conditions.
- *Depends:* This is a set of dependent configuration options. It is noteworthy that dependencies can be combined with expressions like conjunction (&&), disjunction (||), or even negation (!). That is, conflicts can be implicitly encoded in the dependencies section.
- *Selects:* These are reverse dependencies. That is, the dependency is in the opposite direction and enforces the existence of a configuration option that is selected.

Besides these constructs, there are input prompts that are mainly useful for graphical configuration tools to decide which

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

options are shown to the user, default values if the user does not make a decision on a specific option, numerical ranges for input values, help texts, comments, and whole menus that allow hierarchies and encapsulation of config options.

The following excerpt shows an example of a Kconfig file for the configuration option X86_VSMP from the x86 architecture (arch/x86/Kconfig) subdirectory:

```
config X86_VSMP
bool "ScaleMP vSMP"
select PARAVIRT_GUEST
select PARAVIRT
depends on X86_64 && PCI
depends on X86_EXTENDED_PLATFORM
—help—
```

Support for ScaleMP vSMP systems. Say "Y" here if this kernel is supposed to run on these EM64T-based machines. Only choose this option if you have one of these machines.

4.2.1. ILP formulation

We translate primitive Kconfig entries as shown before to the following inequalities, which form the side conditions of the generated ILP:

- *Config:* Each configuration option corresponds to a class in our UML representation or a variable in the underlying ILP. We enforce a minimal bound of 1 for entries chosen by the end user (e.g., for a config option C):

$$|C| \geq 1.$$

  Clearly, the lower bound must not be used for automatically chosen options that are not enforced by direct user input.
- *Tristate/string:* To simplify our experiment, we only consider Boolean options. That is, we map tristate logic to Boolean by assigning *yes* and *module* to *yes* and *no* to *no*. This corresponds to the idea that we want to build a minimal kernel in the sense that it is complete (i.e., no dynamic loading of modules necessary) but it does not use more drivers than absolutely necessary for correct operation:

$$0 \leq |C| \leq 1.$$

  Tristate options could be implemented via subclassing, with a superclass representing a state for inclusion in the solution and two subclasses for *yes* and *module*.
- *Depends:* We mainly distinguish between the three cases: conjunction, disjunction, and negation. Normal dependencies and those formed by conjunction can be expressed in the same way as we did with Debian packages (assuming C depends on both D and E):

$$|C| - |D| \leq 0,$$
$$|C| - |E| \leq 0.$$

For disjunction, we reuse

$$C| - |D| - |E| \leq 0,$$

whereas negation maps to (assuming C conflicts with D)

$$|C| + |D| \leq 1.$$

The latter works because for the Linux kernel configuration it does not make sense to include the same driver multiple times.

Note that, in principle, arbitrary complex expressions (mainly due to the existence of subexpressions that may be layered) may exist. Consequently, expressions need to be flattened out first to fall into the above categories.

- *Selects:* Analogous to classical dependencies, we model reverse dependencies by

$$|D| \leq |C|,$$

  assuming that C selects D in this example. The correctness is immediate again by using $m_1 = 0$ and $n_1 = n_2 = m_2 = 1$ for the general case inequalities.

4.2.2. Implementation

For our experiment we used a vanilla Linux kernel version 2.6.39-rc4 (available at http://www.kernel.org). We start out with a Python parser for Linux Kernel config files called Kconfiglib (http://dl.dropbox.com/u/10406197/kconfiglib.html) version v3. Kconfiglib is capable of reading in all Kconfig files of a source tree, extracting their information, and bringing it in a format that can be more easily accessed in a streamlined way via Python. This allows us to restrict our attention to a basic set of primitive operators. Our implementation processes only the conjunctions reported by Kconfiglib after it has flattened out the constraints in order to uniformly access the Kernel configuration. We use Kconfiglib to create an intermediate XML format that can be easily accessed via C++. This way we can reuse parts of the implementation we already have and use for Debian package extraction. Similarly, we can create a UML class diagram both by interfacing CLEWS (Niederbrucker & Sisel, 2011) with its Java API via Java Native Interface and by directly creating an ILP in the lp_solve format.

5. EVALUATION

We evaluate the performance of our approach by a benchmark experiment that has two main objectives for investigation. First, we have to ensure that the generated solutions are reasonable. This means we have some sort of empirical evidence

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

of the applicability of our theory presented so far in practical use. Note that we do not claim our approach to be "complete" in the sense that we are able to model all settings in both application domains. Nonetheless, we argue that our approach works for a broad class of the most common constraints yielding reasonable solutions. Second, we are interested in the performance of our generated ILPs because we argue that the translation of configuration problems to ILPs is lightweight and effective for real-world examples.

For the Debian package configuration use case, we check the correctness by generating an ILP for input packages that are a small proper subset of actually installed packages on a real working Debian system. Consequently, the solution of the ILP will include a lot of dependencies that should still be a subset of the actual system, however. We compare this solution with the list of installed packages on the real system and check dependencies with a package management tool like *aptitude*.

The multitude of generated constraints correspond to conjunctions of dependencies and some simple dependencies selecting either one or the other dependency. Both virtual packages and versioned dependencies are ignored. A possible implementation for versioned dependencies is to create a symbolic class for each relevant combination of package and version number. For simple relations between versioned packages, this works sufficiently. For complex version dependencies, this approach generates an overhead in the number of necessary variables of the ILP. There is also a negative impact on the overall usability because these superfluous variables occur in the final solution reported back to the user.

For the Linux kernel configuration we start with a set of manually chosen symbols that resemble some typical components of a configuration for a x86 PC. Based on these input symbols, an ILP is generated modeling the dependencies as logical conjunctions. Its solution now includes a list of kernel symbols modeling its dependencies as reported by Kconfiglib. We checked its quality by comparing it to the results obtained when doing a classical manual kernel configuration (with standard tools like *make xconfig* in the kernel source tree). We note that our generated ILP does not fully comply with the solutions proposed by the Linux standard tools. This is expected because we concentrate on conjunctive dependencies and neglect other features like string handling or variable ranges. Still, manual comparison shows that the produced results look similar to the solutions by standard tools and capture the main features. That is, our solution is a reasonable starting point for configurations that get manually fine-tuned later on.

To evaluate the scalability and runtime performance of our generated ILPs, we start with a fixed number of input packages (in the Debian setting) or input kernel symbols (in the Linux kernel setting) and measure its runtime for solution. We use this as a base line. Note that all dependencies must be available to our tool and potentially be processed because we do not know in advance which will be relevant for the input packages/symbols. Now we stepwise increase the number of input packages/symbols. This means that likely a higher amount of dependencies needs to be processed and more inequalities will be generated.

Figure 5 and Figure 6 depict the runtimes of lp_solve in seconds for the generated integer linear programs. In Figure 5 we see a linear scaling behavior when increasing the number of packages to be processed as input by the user. The runtime is in the range of 10–20 s for amounts of packages that already go beyond any typical installation. Figure 6 shows the situation for the processing time of the ILP for kernel configuration. The total runtime is only a few seconds where we see almost a (sub)linear scaling behavior. We are not completely sure why there is a decrease in runtime in the range of 2000–3000 input symbols. A possible explanation could be that because the runtime is under 1 s, small differences in memory allocation might cause this effect. Note that each experiment was run several times to exclude unreproducible effects caused by singular events. Another explanation could be that once a certain amount of symbols is included in the solution set, most conjunctive dependencies are automatically fulfilled. This avoids a stepwise increase in the number of individual symbols, resulting in rather expensive rechecking of multiple dependency constraints.

The figures depict the mean runtimes averaged over all runs. The runs were conducted on low-end to middle-class PC hardware (two cores, 3 GB RAM), so the results should be even faster on more recent or dedicated hardware.

When using a GUI tool like CLEWS, the main constraining factors are the graphical components of the Java runtime. The actual translation of the UML diagram to inequalities is quite fast (a few seconds); however, displaying its results can take more time (tens of seconds). Nevertheless, for realistic examples consisting of hundreds of classes and associations between them (instead of several thousands), such GUI tools work reasonable well.

6. RELATED WORK

The representation of configuration problems in UML has some history. Felfernig et al. (2000) use it as domain-specific language for the construction of knowledge-based configuration systems. The UML constructs are then transformed to logical sentences in order to solve the configuration task. Later Felfernig et al. (2002, 2003) used UML and OCL for configuration knowledge base development and maintenance. Another framework following an object-oriented paradigm is described by Mailharro (1998). A configuration problem is considered both as a classification problem and as a constraint satisfaction problem (CSP). Stumptner et al. (1998) and Fleischanderl et al. (1998) extended the standard CSP model and work also with knowledge bases written in an object-oriented representation language. The presented papers cover relatively large fractions in the "configuration universe." A translation to logics or CSP seems natural. However, such an expressive power comes at the price of higher complexity such that "almost any practical applications of configurators crucially depend on the application of search

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

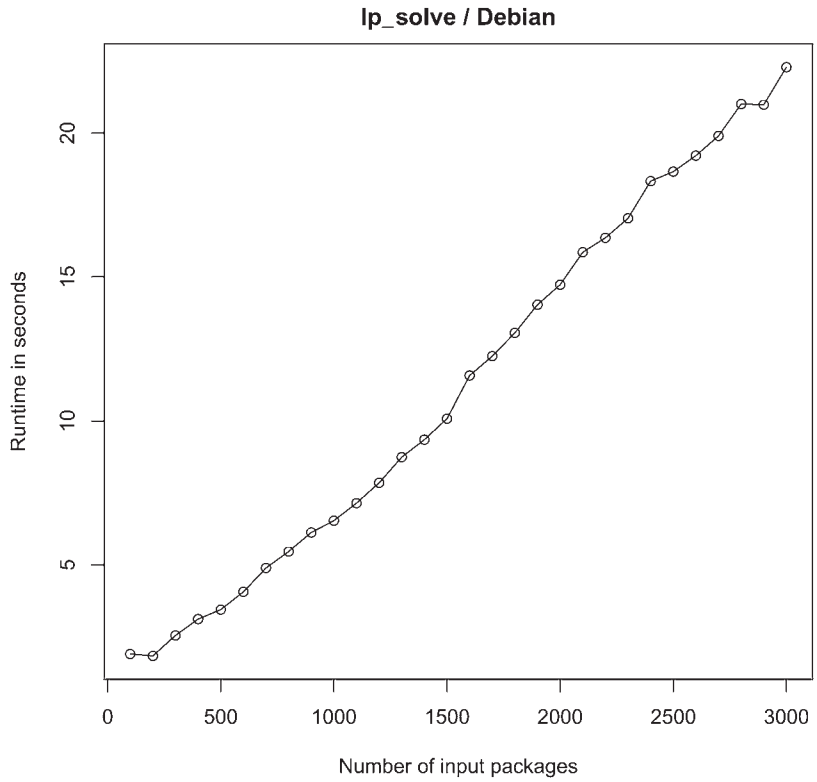

Fig. 5. Runtime of lp_solve in the Debian GNU/Linux setting.

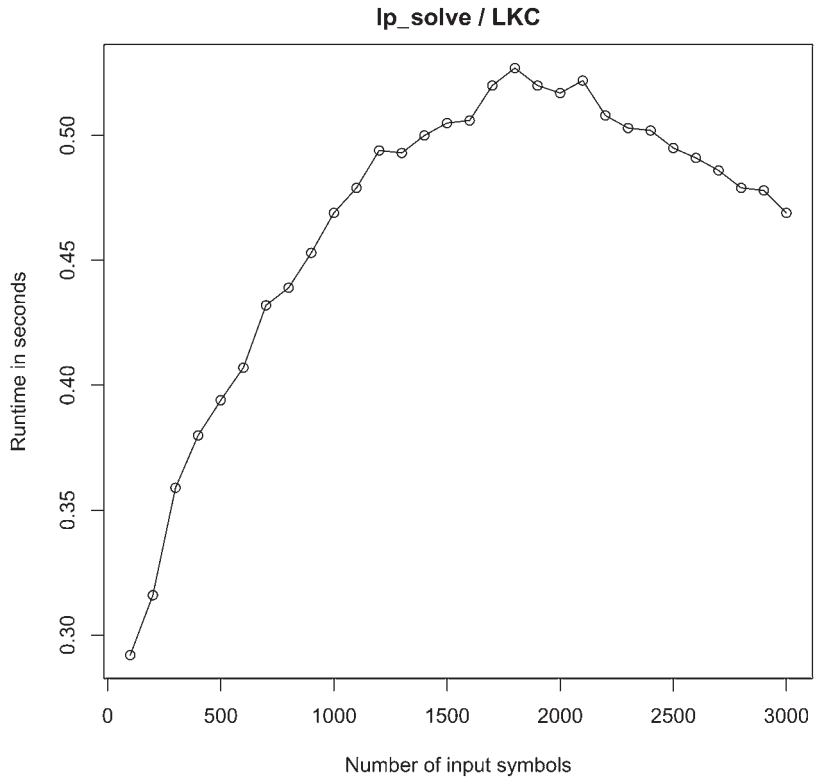

Fig. 6. Runtime of lp_solve for the Linux kernel configuration.

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

heuristics" (Felfernig et al., 2003, p. 44). In contrast, we identify our ILP approach, which works only with a limited set of design constructs, as more lightweight and driven for efficiency on standard configuration tasks.

Soininen et al. (1998) build a general ontology of product configuration covering connection-, resource-, structure-, and function-oriented approaches. Our approach can be seen in the spirit of the resource-based paradigm where "components . . . are modeled as abstract resources, and each technical entity is characterized by the type and amounts of resources it supplies, consumes, and uses" (Heinrich & Jüngst, 1991, p. 19).

Recently, there was a dedicated Special Issue on configuration in *AI EDAM* (Felfernig et al., 2011) with contributions on a variety of aspects of configuration. Several high-quality approaches have been presented that typically solve a specific problem very well.

Mayer et al. (2011) propose a configuration method for work processes. The key challenge is that paths and subprocesses in a work process cannot be handled adequately with existing methodology. As remedy, they extend constraint-based configuration methods to model certain aspects in UML.

Felfernig and Schubert (2011) deal with personalized diagnoses for inconsistent user requirements. They introduce an algorithm capable of giving detailed and informative feedback when inconsistent requirements are detected. This is in analogy to our ILP approach because we can identify inconsistent parts in UML class diagrams and give instant and useful feedback to the user (Feinerer, 2007). Our feature is even implemented in CLEWS (Niederbrucker & Sisel, 2011), which highlights inconsistent paths via red lines and allows the user to modify the design.

Nevertheless, Falkner et al. (2011) observe that although many individual approaches in AI achieve superior results in subfields of configuration, they are often hard to use for general-purpose modeling and reach their limits in scalability.

Consequently, lightweight approaches that scale well and are capable of modeling real-world applications as identified by Falkner et al. (2010) are necessary. We fill this gap by the unified ILP theory presented in this paper, significantly extending previous work on this topic (Feinerer & Salzer, 2007).

In the context of package management, there are promising results by Tucker et al. (2007) that deal with the install problem, minimum install problem, and uninstall problem. They encode the install problem and the uninstall problem utilizing SAT formulas, and the minimum install problem with SAT and pseudo-Boolean constraints to encode integer costs. The latter can be transformed to an ILP. Janota et al. (2012) tackle the software package upgradability problem by using an encoding based on weighted partial MaxSAT formulas, which they solve with weighted partial MaxSAT solvers and optimization pseudo-Boolean solvers. Abate et al. (2011) present a modular package manager, which could be used as a framework for our approach. There is also *mancoosi*, a European research project in the 7th Research Framework Programme of the European Commission, which deals with managing software complexity and package management. It should be noted that our ILP approach is not directly comparable to tailored approaches as presented by Tucker et al. (2007) and Janota et al. (2012). They are natively capable of handling more complex input cases, like dependencies on specific versions of packages, which are not dealt with in our experiment. Their solutions are more complete; however, their scalability depends heavily on the complexity and structure of the encoded dependencies.

For Linux kernel configuration, Sincero et al. (2007) propose to see the configuration problem as a software product line utilizing methodology of this field. Zengler and Küchlin (2010) encode the Linux kernel configuration as a SAT problem. In addition to our approach, they are capable of handling tristate variables, complex expressions on dependencies and selections, and menu blocks. Another difference is that they are mainly interested in finding sets of valid configurations and testing for satisfiability of a given configuration. For enumerating valid partial configurations when using SAT-solvers, Voronov et al. (2011) propose techniques employing binary decision diagrams. Complementary, our ILP approach has built-in support for minimizing variable values and therefore for finding minimal configurations.

Our approach of using UML as a design language and its mostly automatic translation to ILP aims at a broad range of configuration scenarios. The major focus lies in an intuitive interface and lightweight formal methods to handle standard configuration use cases in an efficient manner. Clearly, this implies that specialized solutions tailored for a specific application domain are more expressive and thus can provide better (in terms of completeness) solutions.

7. CONCLUSION

Due to the success of AI methods in configuration, we saw a steady increase in their usage for real-world applications for a broad range of domains. However, the complexity and size of the problems have been growing. One of the key challenges is to choose a matching technology that is expressive enough to model realistic scenarios but which can be handled efficiently. Only the latter allows us to scale the applied methodology to realistic problems consisting of thousands of components and even more interactions. In this paper, we proposed a unified theory based upon ILP. We showed how to model important concepts as identified in the literature via UML class diagrams and corresponding constraints, and how this terminology can be translated into an ILP. This idea is general enough to be applied to other object-oriented formalisms. ILP is known for its performance because certain subclasses are of polynomial time complexity and there is a wide range of excellent solvers available despite its NP-completeness in the general case. Within our formulation, polynomial algorithms exist as long we have no upper bounds for sums over two variables because this results in indeterminism whether one or the other variable needs to be increased, and backtracking if not successful.

Consequently, we argue that with our approach, a broad class of configuration problems can be modeled and we

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press

gain reasoning support via lightweight formal methods. The ILP formulation is especially suited for configuration problems with large conjunctive dependencies and few conflicts and occasional subclassing. The advantage over logics and SAT-based techniques is that minimizing an objective function is a primary objective that is central in configuration tasks. By contrast, in logics nontrivial techniques have to be employed for finite model reasoning and minimizing variables over integer ranges.

We evaluated our approach with two real-world large-scale examples: Debian GNU/Linux package management and the Linux kernel configuration. The aim of both applications was to show that our approach allows for an intuitive modeling of the configuration task when dealing with the core features of the respective domain. We expect our approach to work when scaled on larger problem instances. We argue that both aims are fulfilled. However, it should be clear that specific tools tailored for the application domain are typically better in terms of expressive power in order to cover all aspects of the configuration task. We observe this for both Debian package management and the Linux kernel configuration. In the end, it boils down to finding the right balance between needed features and the underlying expressive power of the constructs. For example, when handling tristate values via subclassing, there might be low overhead by the construction itself but the construct might be triggered very often; or when the application domain exhibits constraints mainly consisting of disjunctions instead of conjunctions; or how subclassing and hierarchies are implemented (for a discussion see, e.g., Maraee & Balaban, 2007; or Männistö et al., 2001).

ILP and SAT seem to be well suited for a diverse field of configuration tasks where each technique has its unique advantages. We think the next step toward more efficiency in configuration lies in the combination of various approaches into a hybrid framework to get the best from multiple worlds. For example, in the context of package management, a viable way to handle versioned dependencies is to use ILP as a building block that is called multiple times but with different subsets of variables. Because version conflicts are rare, this "outsourcing" of some constraints seems promising instead of forcing everything into a single large ILP.

As future work, we are working on identifying more complex constraints in large configuration applications given by our project partners at Siemens AG and how they can be modeled efficiently. We aim for highly efficient configuration tools with a wide range of formal methods support such that design engineers can model their settings in a natural way and get immediate feedback on design errors. Again a hybrid approach appears to be the most promising way toward achieving this goal.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive and insightful comments.

REFERENCES

Abate, P., Di Cosmo, R., Treinen, R., & Zacchiroli, S. (2011). MPM: a modular package manager. *Proc. 14th Int. ACM SIG-SOFT Symp. Component Based Software Engineering, CBSE-2011*.

Alliance for Telecommunications Industry Solutions. (2000). *ATIS telecom glossary 2000*. Accessed at http://www.atis.org

Chen, P.P.S. (1976). The entity–relationship model: toward a unified view of data. *ACM Transactions on Database Systems 1(1)*, 9–36.

Falkner, A., Feinerer, I., Salzer, G., & Schenner, G. (2010). Computing product configurations via UML and integer linear programming. *Journal of Mass Customisation 3(4)*, 351–367.

Falkner, A., Haselböck, A., Schenner, G., & Schreiner, H. (2011). Modeling and solving technical product configuration problems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25(2)*, 115–129.

Feinerer, I. (2007). *A formal treatment of UML class diagrams as an efficient method for configuration management* . PhD thesis. Vienna University of Technology.

Feinerer, I. (2011). Efficient configuration and verification of software product lines. *Proc. 15th Int. Software Product Line Conf, SPLC'11*. Association for Computing Machinery.

Feinerer, I., & Salzer, G. (2007). Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints. *Proc. 1st IEEE/IFIP Int. Symp. Theoretical Aspects of Software Engineering, TASE'07*. New York: IEEE.

Felfernig, A., Friedrich, G., & Jannach, D. (2000). UML as domain-specific language for the construction of knowledge-based configuration systems. *Journal of Software Engineering and Knowledge Engineering 10(4)*, 449–469.

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

Felfernig, A., Friedrich, G., Jannach, D., & Zanker, M. (2002). Configuration knowledge representation using UML/OCL. *Proc. 5th Int. Conf. the Unified Modeling Language, UML'02*. Berlin: Springer–Verlag.

Felfernig, A., & Schubert, M. (2011). Personalized diagnoses for inconsistent user requirements. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25(2)*, 175–183.

Felfernig, A., Stumptner, M., & Tiihonen, J. (2011). Special issue: configuration [Editorial]. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25(2)*, 113–114.

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

Floyd, R.W. (1962). Algorithm 97: shortest path. *Communications of the ACM 5(6)*, 345.

Heinrich, M., & Jüngst, E.W. (1991). A resource-based paradigm for the configuring of technical systems from modular components. *Proc. 7th IEEE Conf. Artificial Intelligence Applications*.

Jackson, D., & Wing, J.M. (1996). Lightweight formal methods. *IEEE Computer 29(4)*, 21–22.

Janota, M., Lynce, I., Manquinho, V.M., & Marques-Silva, J. (2012). Packup: tools for package upgradability solving. *Journal of Satisfiability, Boolean Modeling and Computation 8(1–2)*, 89–94.

Lenzerini, M., & Nobili, P. (1990). On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems 15(4)*, 453–461.

Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing 12(4)*, 383–397.

Männistö, T., Peltonen, H., Soininen, T., & Sulonen, R. (2001). Multiple abstraction levels in modelling product structures. *Data & Knowledge Engineering 36(1)*, 55–78.

Maraee, A., & Balaban, M. (2007). Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. *Proc. Model Driven Architecture—Foundations and Applications, 3rd European Conf., ECMDA-FA'07*.

Mayer, W., Stumptner, M., Killisperger, P., & Grossmann, G. (2011). A declarative framework for work process configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing 25(2)*, 143–162.

Niederbrucker, G., & Sisel, T. (2011). *CLEWS*. Accessed at http://www.logic.at/clews

Object Management Group. (2011). *Unified Modeling Language 2.4.1*. Accessed at http://www.omg.org

Object Management Group. (2012). *Object Constraint Language 2.3.1*. Accessed at http://www.omg.org

Sincero, J., Schirmeier, H., Schröder-Preikschat, W., & Spinczyk, O. (2007). Is the Linux kernel a software product line? *Proc. Int. Workshop on Open Source Software and Product Lines, SPLC-OSSPL'07*.

Soininen, T., Tiihonen, J., Männistö, T., & Sulonen, R. (1998). Towards a general ontology of configuration. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing 12(4)*, 357–372.

Stumptner, M., Friedrich, G., & Haselböck, A. (1998). Generative constraint-based configuration of large technical systems. *Artificial Intelligence in Engineering Design, Analysis and Manufacturing 12(4)*, 307–320.

Tucker, C., Shuffelton, D., Jhala, R., & Lerner, S. (2007). Opium: optimal package install/uninstall manager. *Proc. 29th Int. Conf. Software Engineering, ICSE'07*. Washington, DC: IEEE Computer Society.

Voronov, A., Åkesson, K., & Ekstedt, F. (2011). Enumeration of valid partial configurations. *Proc. IJCAI 2011 Workshop on Configuration*.

Zengler, C., & Küchlin, W. (2010). Encoding the Linux kernel configuration in propositional logic. *Proc. ECAI 2010 Workshop on Configuration Systems*.

Ingo Feinerer has been an Assistant Professor at the Vienna University of Technology since 2008. He received a PhD in computer science from the Vienna University of Technology and a PhD in business administration from the Vienna University of Economics and Business. In 2007 he was postdoctoral visiting scholar at the Computer Science Department of Carnegie Mellon University. His expertise and research interests comprise efficient methods for configuration, formal methods, text mining, and databases.

https://doi.org/10.1017/S0890060412000376 Published online by Cambridge University Press