Twenty-Five Years of Successful Application of Constraint Technologies at Siemens

Andreas Falkner, Gerhard Friedrich, Alois Haselböck, Gottfried Schenner, Herwig Schreiner

■ *The development of problem solvers for configuration tasks is one of the most successful and mature application areas of artificial intelligence. The provision of tailored products, services, and systems requires efficient engineering and design processes where configurators play a crucial role. Because one of the core competencies of Siemens is to provide such highly engineered and customized systems, ranging from solutions for medium-sized and small businesses up to huge industrial plants, the efficient implementation and maintenance of configurators are important goals for the success of many departments. For more than 25 years the application of constraint-based methods has proven to be a key technology in order to realize configurators at Siemens. This article summarizes the main aspects and insights we have gained looking back over this period. In particular, we highlight the main technology factors regarding knowledge representation, reasoning, and integration that were important for our achievement. Finally we describe selected key application areas where the business success vitally depends on the high productivity of configuration processes.*

In the 1960s and 1970s, predictions that AI technologies would solve many problems, such as automated reasoning, machine learning, vision, natural language understanding, robotics, or planning, within the next few decades were quite optimistic. To cite just one of the main proponents of those years, Marvin Minsky (1967) wrote that "Within a generation […] the problem of creating 'artificial intelligence' will substantially be solved. Although this general goal still needs to be fulfilled, certain areas of AI have reached a mature application status."

One of the AI technologies that successfully found its way into industrial applications is constraint satisfaction (Rossi, van Beek, and Walsh 2006). Constraint satisfaction is a form of automated reasoning where the problem is modeled by a set of variables and constraints. A solution is a variable assignment — each variable gets a value from its associated domain — where all constraints are satisfied. Constraints are logical or relational expressions restricting the values that may be assigned to the variables. Due to the strict and simple definition of constraint-satisfaction problems (CSPs), many highly efficient and robust constraint reasoning techniques, for example, constraint propagation (Bessiere 2006), complete search (van Beek 2006), and local search heuristics (Hoos and Tsang 2006), have been developed.

Configuration of technical and commercial products and

```
int : multimedia = 1;
int : scientific = 2;
int : internet = 3;
var 1 .. 3: usage ;
var 1 .. 16: number_cpus ;
constraint (usage == multimedia) –> (number_cpus >= 4);
```

Listing 1. MiniZinc Example.

services is one of the fields where constraint satisfaction evolved to become the main representation and reasoning technique (Faltings and Freuder 1998). Simply speaking, configuration is the selection and combination of parts from a component catalog, together with setting parameters of these parts such that all physical, technical, commercial, and strategic constraints are fulfilled.

Siemens AG, a German-based multinational electronics and electrical engineering company, has used constraint-satisfaction technologies in productive configuration systems successfully for about 25 years. In this article, we describe how constraint techniques can be embedded and used in industrial configurators, and how the complex structure and requirements of large-scale technical systems made it necessary to extend the standard theory of constraint satisfaction both in modeling expressiveness and reasoning capabilities. We describe examples of configurators, based on constraint satisfaction, that have been successfully put to productive use for many years.

A configurator, as is the case for most engineering and industrial software systems, has three different aspects: (1) Knowledge representation to provide a language to describe the domain of interest, along with data persistence and maintenance capabilities (see the section on Knowledge Representation). (2) Reasoning methods to provide functions for automatic problem solving and user assistance functions, for example, by filtering invalid options in the current situation (see the section on Reasoning). (3) An infrastructure to integrate those modeling and reasoning components into a software application. Important aspects of integration are user interface, input/output interfaces, connectors to enterprise software (product lifecycle management, enterprise resource planning, customer relationship management), a web environment, and so on (see the section on Integration).

In the Applications section, we briefly describe some applications whose backbone is constraint-satisfaction technology and that are or were in productive use at Siemens. Finally, the Summary concludes this article.

Knowledge Representation

Constraint technologies have become the main approach used to model and solve configuration problems. There are many different kinds of configuration problems, but one simple aspect to categorize them is whether a problem is static or dynamic.

In a static configuration problem the task is to select and combine components from a predefined and relatively small set of parts (for example, hundreds of pieces). For each part it has to be decided whether it shall be used in a concrete configuration without violating any predefined restrictions concerning the combination of parts. Examples of such problems are web-based shops for consumer products (like bicycles, computers, cars), feature models (Kang et al. 1990), or product line engineering systems (Clements and Northrop 2001). Those systems can be very elegantly modeled by standard constraint-satisfaction techniques.

An example of a constraint-satisfaction language is MiniZinc, which is a powerful language for specifying static problems in a very concise way (Nethercote et al. 2007). The code clip in listing 1 shows an example from a PC configuration: if the usage type of a computer is multimedia, the PC must be at least a quad-core.

From the knowledge representation point of view, languages like MiniZinc are well suited for representing combinatorial problems of manageable scales. If the problem gets large and structurally complex, object-oriented formalisms are a better way of representing the product taxonomy and dependencies in a natural way. Another aspect is the static versus the dynamic property of a problem. Consider the following realistic extensions to our PC example problem: An order contains not only a single PC but a whole

set of PCs, each with individual properties, and for each CPU of each PC, not only the number of CPUs but several properties (such as manufacturer, performance) may be specified. Additionally, global constraints refer to the whole set of PCs, for example, a suitable LAN router must be configured depending on the number of Internet and multimedia PCs. The formulation of such dynamic problems using static constraint languages imposes the representation of all largest possible problem instances based on a fixed set of variables and constraints resulting in excessive memory consumption. Moreover, the language constructs require a nontrivial transformation of the conceptual model into a CSP representation. This mapping is difficult to comprehend for software engineers who are trained to specify problems by object-oriented means. In particular, arrays and matrices of variables are used, which do not reflect the structural aspects of the problem.

In a dynamic configuration problem the set of different parts to be combined in a solution is not known beforehand. Static problems are mainly of combinatorial nature, while dynamic problems must additionally decide on creating new configuration objects and adding them to the current problem. Examples of such dynamic domains are many industrial fields such as railway safety systems, telecommunication systems, or power stations. Knowledge representation of such systems as object databases with integrity constraints have a long history (Hull and King 1987; Beneventano et al. 1998). There have been several attempts to extend the standard constraint theory to such dynamic, conceptual representation schemes, for example, conditional constraint satisfaction (Mittal and Falkenhainer 1990), dynamic constraint satisfaction (Dechter and Dechter 1988), and option types in the constraint language MiniZinc (Mears et al. 2014). Although these extensions provide mechanisms for a dynamic selection of variables that are active in a constraint network, they are still based on a static set of potential variables.

A more powerful extension is generative constraint satisfaction GCSP (Fleischanderl et al. 1998), where constraints are integrated into an object-oriented environment. The object-oriented paradigm is nowadays well-established in most nontrivial configurator implementations based on programming languages such as C++, C#, or Java.

GCSP can be seamlessly integrated into an object-oriented environment. Variables and constraints are defined on the class hierarchy level. Each time an instance of a class (that is, a configuration object) is created or destroyed, its variable and constraint instances are created or destroyed, too. The result is a constraint network that grows and shrinks along with the object network. By adding new variables induced by configuration object creation, domains of existing variables may be extended, too, because associations are represented by variables and their domains are therefore sets of configuration objects. Furthermore, domain values previously found to be consistent/inconsistent may change their consistency state because of variable insertion. Analogous cases apply to object deletions. A dependency network induced by the set of constraints identifies those variable domains whose values must be reevaluated due to variable insertion/deletion.

The GCSP code for our small PC configuration example is shown in listing 2. It is a straightforward implementation of the UML diagram shown in figure 1. A configuration consists of a set of PCs and each PC consists of 1 to 16 CPUs. The task of configuration is to create the necessary amount of PC instances along with their CPU instances and determine their properties.

An instance of class Configuration consists of an arbitrary number of PC instances, each of which is connected to several (min. 1, max. 16) CPU instances. The constraint code for requiring at least 4 CPUs for a multimedia PC looks very similar to the MiniZinc code above, with the main difference that the constraint is defined on the class level. Each time a PC is instantiated, a new constraint instance is created and added to the dynamic constraint network.

At Siemens, various GCSP-based configurators have been built and are in productive use. Examples are configurators for railway interlocking and safety systems, telecommunication switching systems, or gas compressors (see the Applications section). The knowledge bases and solutions in these domains are huge (hundreds of thousands of components), and it has become evident that it is not possible to build such systems in an efficient and clean way without the structuring capabilities of object-oriented knowledge representation and the declarative aspect of constraint satisfaction. Experience has shown that declarativity supports the maintenance of such systems very well, especially if they are long-lasting (for example, railway systems have a life span of about 30 years), because data and logic are cleanly separated and general, domain-independent reasoning methods are used for solving.

In rule-based systems, as mainly used for expert systems in the past (McDermott 1982), maintenance often becomes a nightmare, because procedural, forward-chaining rules such as seen in many commercial systems are directed and have many interdependencies (Soloway, Bachant, and Jensen 1987). Nevertheless, for certain kinds of problems where reasoning definitely is directed (for example, the wiring is determined by the arrangement of the modules in a rack, and not vice versa), production rules are a good choice. Knowledge engineers can write production rules in a very intuitive way and rule execution is fast. For such cases, we have integrated a forward-chaining rule mechanism into our configuration framework in addition to constraints.

```
enumeration Usage {
   multimedia, scientific, internet}

class Configuration {
  aggreg pcs[1..*]: PC;
}

class PC {
  attr usage: Usage;
  assoc configuration[1]: Configuration oppositeOf pcs;
  aggreg cpus[1..16]: CPU;
  constraint usageMultimedia {
   assert (usage == Usage.multimedia) -> (cpus.size() >= 4);
  }
}

class CPU {
  attr manufacturer: String;
  assoc pc[1]: PC oppositeOf cpus;
```

Listing 2. GCSP Example.

Reasoning

Separation of knowledge representation and reasoning is the key factor in knowledge-based systems such as configurators. There are many different kinds of reasoning tasks in a complex system, and therefore the architecture of such systems must be able to integrate various solvers and solving technologies (Falkner et al. 2011). The reasoning methods can be classified along different perspective; the main classifications for configuration applications are detailed in the following paragraphs.

Satisfaction versus optimization. Satisfaction is the task of finding an arbitrary solution where all constraints are satisfied. In many configuration problems, this is not enough. Users are interested in solutions that minimize or maximize one or several optimization criteria (for example, find the least expensive configuration). Of course, optimization algorithms are computationally more challenging than pure satisfaction algorithms.

Find one or all solutions. Configuration problems in industry and infrastructure domains often get very large and complex, so that trying to find all solutions is a hard task due to the huge search space. Typically, many solutions will be very similar because of symmetry reasons. Usually it is sufficient to find one solution or only a few. Examples where an exhaustive search of the solution space is necessary are feature models in the domain of product line engineering, where classification figures such as variability (the ratio of valid solutions to feature combinations) or commonality (the percentage of solutions containing the same feature configuration) play an important role (Benavides, Segura, and Cortés 2010).

Propagation-guided search. The most prominent techniques are search algorithms based on backtracking, where values are assigned successively to variables, and in the case of a dead end, conflicts in the past are to be resolved (van Beek 2006). Consistency algorithms are look-ahead methods, where inconsistent domain values are filtered out and this knowledge is propagated to still unassigned variables (Bessiere 2006). Usually, a combination of backtracking and look-ahead techniques is applied, where in each backtracking cycle assigned values are propagated to filter the domains of still unassigned variables (Freuder and Mackworth 1994).

Complete versus incomplete techniques. Backtracking or full consistency algorithms are complete techniques, because they will theoretically always find a solution, if one exists. For large and complex problems, however, such algorithms may not be applicable due to the sheer size of the search space. Heuristic search methods have to be used instead. Such methods are either constructive, generating a solution step by step using heuristic decisions, or perturbative. Perturbative search starts with a heuristically generated complete configuration and successively repairs or improves it by local, often random decisions (Hoos and Tsang 2006). Local search methods use partial or complete candidate solutions as search states. The search process starts with an initial state and proceeds from one search state to a neighboring state exploiting limited local information. Constructive and perturbative methods can also be combined. Some of the main heuristic search algorithms are GSAT, tabu search, simulated annealing, genetic algorithms, and large neighborhood search.

Interactive versus batch techniques. In batch configuration problems, all user requirements are comprehensively specified at the beginning, and a solver finds a solution without any user interaction. One of our challenging examples of a batch configuration problem is the Partner Units Problem, where zones and sensors are assigned to communication units (Aschinger et al. 2011). However, most configuration systems in practice are interactive systems because users want to be guided in their decision processes: they are not able to specify all requirements in advance but need to see the effects of each decision on the alternatives for the remaining variables — which values are no longer allowed or which values are direct and indirect consequences, respectively. For many of our dynamic configuration problems, users are not able to anticipate all additionally required decisions based on some initial choices; for example, depending on the selection of a software module type for some hardware element, different parameters must be specified by the user. Such problems call for reasoning methods that assist the user to build a solu-

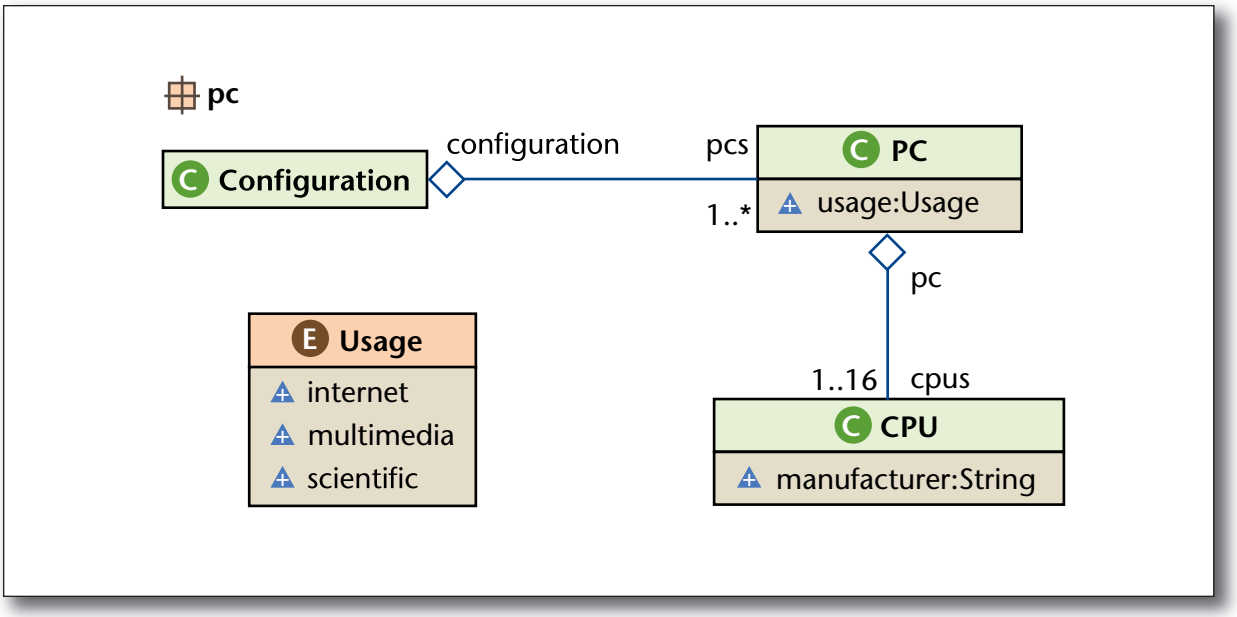

Figure 1. UML Model for the PC Example.

tion step by step rather than automatically derive one. Consistency algorithms and local search methods support these demands quite well. For example, the removal of items from a menu of currently possible options corresponds to filtering out values from a variable's domain by constraint propagation. Another reasoning task important for interactive configuration systems is to provide explanations for conflicts in configuration fragments. Falkner, Felfernig, and Haag (2011) describe some approaches for that.

Discrete versus continuous problems. Discrete problems have finite, discrete variable domains, like different module types or component properties with enumeration domains. All the aforementioned reasoning methods work on such discrete problems. In many configuration domains, for example, transportation systems (Ahuja, Möhring, and Zaroliagis 2009), continuous variables (such as real numbers) are also necessary. The most prominent techniques for such domains are linear programming, (mixed) integer programming (Papadimitriou and Steiglitz 1982), nonlinear programming (Bazaraa, Sherali, and Shetty 1993), convex optimization (Bertsekas 2009), and interval propagation (Hentenryck 1997).

Implementation

Nowadays, many different constraint solvers are available, both from academic research institutes and commercial vendors. There are indications that the CSP language MiniZinc (as briefly described in the Knowledge Representation section) is becoming a constraint language standard (Nethercote et al. 2007). The advantage of this standardization would be that more and more solvers will understand MiniZinc models, and application developers could choose the most suitable solver for a particular problem without changing the CSP knowledge representation. Examples of solvers that understand MiniZinc (or its "flattened" version FlatZinc) are G12, Gecode, ECLiPSe, and SICStus Prolog.

Recently, constraint-satisfaction search methods have been complemented by SAT solvers, which gained enormous potential due to new highly efficient inference algorithms like conflict learning or the parallel application and orchestration of different solving strategies (Biere et al. 2009, de Moura 2011).

But most of the existing solvers and algorithms are designed for standard, static constraint-satisfaction problems. Moving to a dynamic environment like generative constraint satisfaction (GCSP), algorithms must be adapted or new ones must be invented. This is challenging and far from being solved (Wallace and Freuder 1998). Modern logic programming methods like answer set programming, ASP (Leone et al. 2006; Brewka, Eiter, and Truszczynski 2011), allow the formulation of first-order theories. Using this approach, a compact knowledge representation is possible. However, current solving methods translate these first-order theories into propositional logic during the grounding phase, which results in unacceptable memory consumption for big problem instances. However, we have applied ASP successfully for hard midsized configuration problems (Aschinger et al. 2011) comprising hundreds of components.

An important alternative is the use of local or heuristic search techniques. These techniques swap

the "luxury" of being complete for being fast and returning acceptable solutions in most cases. One such method is iterative repair (Selman, Levesque, and Mitchell 1992) — violated constraints are repaired until, finally, a solution is found. Sophisticated (often domain-dependent) heuristics guide the search process so that the chance of finding an acceptable solution quickly is high.

Symmetry breaking is another approach developed in the context of constraint satisfaction and applies very well to the solving of configuration problems by systematic search. The basic idea is to prune those parts of the search tree that are symmetric to parts that have already been proven to be either consistent or inconsistent (Freuder 1991; Haselböck 1993; Crawford et al. 1996; Gent, Petrie, and Puget 2006).

Besides the aforementioned reasoning methods, there exist many other different solving techniques and tools, most of them specialized to tackle very specific problem types. Many real-world configuration problems are complex and heterogeneous in the sense that various forms of knowledge representations (for example, propositional logic, first-order logic, equations, and inequalities) should be employed for a compact problem specification. Therefore, there is an increasing need for an open architecture with the possibility to integrate such different reasoning methods and tools that are able to deal with a variety of different customer requirements.

In the configuration applications described in the Applications section, different solvers and solver technologies are used, often in a combined way. Some parts of an industrial configuration problem could be solved by mapping them to standard systems such as Choco, MiniZinc, or ASP, some were well represented by MILP (mixed integer linear programming) systems. In those cases, adapters from the formalism representing the object model and business logic to the formalism of the external solver have been implemented. For large-scale, dynamic problems, we mostly used a proprietary GCSP implementation — S'UPREME — which is a local search solver based on iterative repair. In turn, many techniques like arc consistency or forward checking developed in the context of constraint satisfaction have been adapted and used in S'UPREME. No adapters from the object model to the solver are necessary in that case, because S'UPREME works directly on that model.

Integration

AI technology, as described in the preceding section, is usually embedded in an infrastructure of different functions and services, for example, a client-server platform based on a database management system and an application server, a user interface, I/O functions for reading customer data and exporting data to other systems, and a test bed for verifying the generated results. Integration of a constraint solver is best supported if it is available as a middleware instead of a stand-alone modeling language. Some constraint systems provide APIs to main programming languages such as Java, C++, or C#. To name just a few: Choco (Java), Gecode (C++), Google CP Solver (Java, C++, .Net), IBM ILOG CP (C++), JaCoP (Java), OscaR (Scala).

An exemplary standardization initiative is the Java Constraint Programming API JSR 331. The goal is that different constraint system vendors provide their systems in terms of this standard API. This standardization supports the integration of constraint technology in a Java application, making it also possible to exchange the underlying solver without changing the application code.

Another integration aspect is characterized by the notion of system of systems. Often, more than one configuration system is necessary in the whole design, sales, and production process, or in the context of product portfolios where a concrete application consists of a set of interacting subproducts. Each subsystem may have its own individual and specialized configurator. They can be integrated either by loose coupling, exchanging partial configurations through interface files, or by working together on a common knowledge platform. Both methods have their strengths and weaknesses. In loosely coupled systems, the individual configurators can be developed independently from each other to a large extent, choosing the best technology for each task. The price is redundant data and expensive interfaces between the subsystems. On the other hand, monolithic systems based on a single knowledge base provide a simpler tool architecture, but have more restrictions on the technologies and frameworks used for implementation. A promising compromise is to keep the operational and organizational independence of the individual systems, but to offer appropriate modeling constructs for collaborative modeling, for automated data exchange between the different CSP models, and for preventing redundancy, inconsistency and ambiguity in shared data (Dhungana, Falkner, and Haselböck 2013).

Applications

In the late 1980s, no robust academic or commercial constraint systems were available ready to be used in industrial applications. Research mainly focused on developing efficient search and consistency algorithms. In the 1990s, the first constraint logic programming systems emerged, mainly based on Prolog (for example, ECLiPSe, SICStus Prolog, CHIP, B-Prolog), and practical application of constraint programming, like circuit verification, scheduling, or resource allocation, arose (Wallace 1996). In the middle of the 1990s, the ILOG Solver was one of the first CSP

solvers delivered as a C++ library. It is now part of the IBM ILOG CPLEX Studio (CP Optimizer). In the first decade of 2000, constraint systems like Choco, JaCoP, Gecode, Minion, or Google CP Solver (to name just a few) turned up and became more and more mature.

In the last years, the following trends have become apparent: (1) Constraint systems are available as libraries (for example, as a Java library like Choco) and can read constraint models specified in quasi-standard languages like MiniZinc or the Java Constraint Programming API JSR 331. This supports an easy and flexible integration into a standard application infrastructure. (2) On the basic level, often SAT solvers are used for search. An interesting variant of logic programming systems is Answer Set Programming (Gelfond 2008), for example, DLV, Smodels, or the tool suite Potassco. ASP combines deductive database systems, nonmonotonic reasoning, and constraint programming. (3) Constraint systems become more and more expressive by providing and combining solvers for different theories. Beyond discrete, finite domains, such theories support integers, real numbers, sets, lists, and various other data structures. The underlying logical framework is SMT (satisfiability modulo theories), for example, Z3 (de Moura and Bjørner 2008). (4) Constraint-based systems become platforms integrating various solvers, each specialized for a particular problem domain. An example is the system Numberjack (Hebrard, O'Mahony, and O'Sullivan 2010), which integrates mixed-integer programming solvers, SAT solvers and constraint solvers. (5) Constraint solving is used in interactive environments such as sales configurators and guided selling tools. A prominent example is Tacton's CPQ (Configure Price Quote) product TCsite, which competes well with precompiled approaches such as BDD-based Configit's virtual tabulation method.

In this section we describe several configurators developed at Siemens in the last 25 years that employ constraints for specifying technically correct configurations. Different domains and problems have different emphases. Sometimes finding a valid solution is sufficient, sometimes the focus lies on one or several optimization criteria. But all these applications have in common that they are complex, large, and of dynamic nature, and that constraints play a central role in representing integrity, consistency, and behavioral models of the systems.

Due to the fact that generative constraint-satisfaction problems need special representation and solving techniques, we developed a proprietary constraint framework S'UPREME, which is used for configuration of large-scale technical products within Siemens.

Configurator for Railway Interlocking Systems for Austria (1989 to Date)

At the end of the 1980s we began to implement the first large-scale configuration system that used constraint techniques. The task of this system was to configure hardware and software of railway control centers for hundreds of railway stations in Austria (see figure 2).

In railway signaling, an interlocking system is an arrangement of signals and other equipment that prevents conflicting movements of trains in a network of tracks, switches, and crossings. It ensures that trains receive clear signals only if their defined route is safe to be used. Modern interlocking systems use software programs running on special-purpose control hardware such as shown in figure 3. Siemens supplies a big variety of systems and components tailored to the requirements of different countries and companies.

Configured systems comprised more than 50.000 components and several megabytes of parameter data for safety-critical systems. Customer and requirements data (station and line topology, signal tables) are partly imported into the system through interface files and partly manually specified through a graphical UI. Constraints check the integrity and plausibility of these data. Outputs are manifold and include hardware assembling sheets in AutoCAD, cable layout plans, element connection diagrams (in PDF format), software parameter files (in XML and other text formats), configured user interfaces, bills of materials (for example, in MS-Excel), checklists for system tests (for example, in MS-Word). The configurator, written in C++, used classes and associations as knowledge-representation concepts. The generation of instances was mainly realized by rule-executions and extended by constraint features. Constraints were implemented in C++ and processed by consistency and filtering functions. Their task was to inform the user continuously and in detail about inconsistencies between user-set values of variables and to warn him/her whenever a rule calculated a different value for a variable after that variable had been changed by the user. However, those constraints were not used for solution finding (search), just for checking.

This configurator has been in productive use for more than 25 years. It is still maintained and continuously adapted to the newest railway technology.

Configurator for Telecommunication Switches (1994–2003)

Moving from rules to constraints was a significant milestone in reducing configurator development and maintenance costs.

For the domain of digital electronic telecommunication switches, we used the first implementation of generative constraint satisfaction, written in Smalltalk. Some of the interesting features of this configurator were as follows: (1) It provided both constraint filtering and backtrack search functionality. Backtrack search was enhanced by look-ahead techniques and domain-specific variable ordering

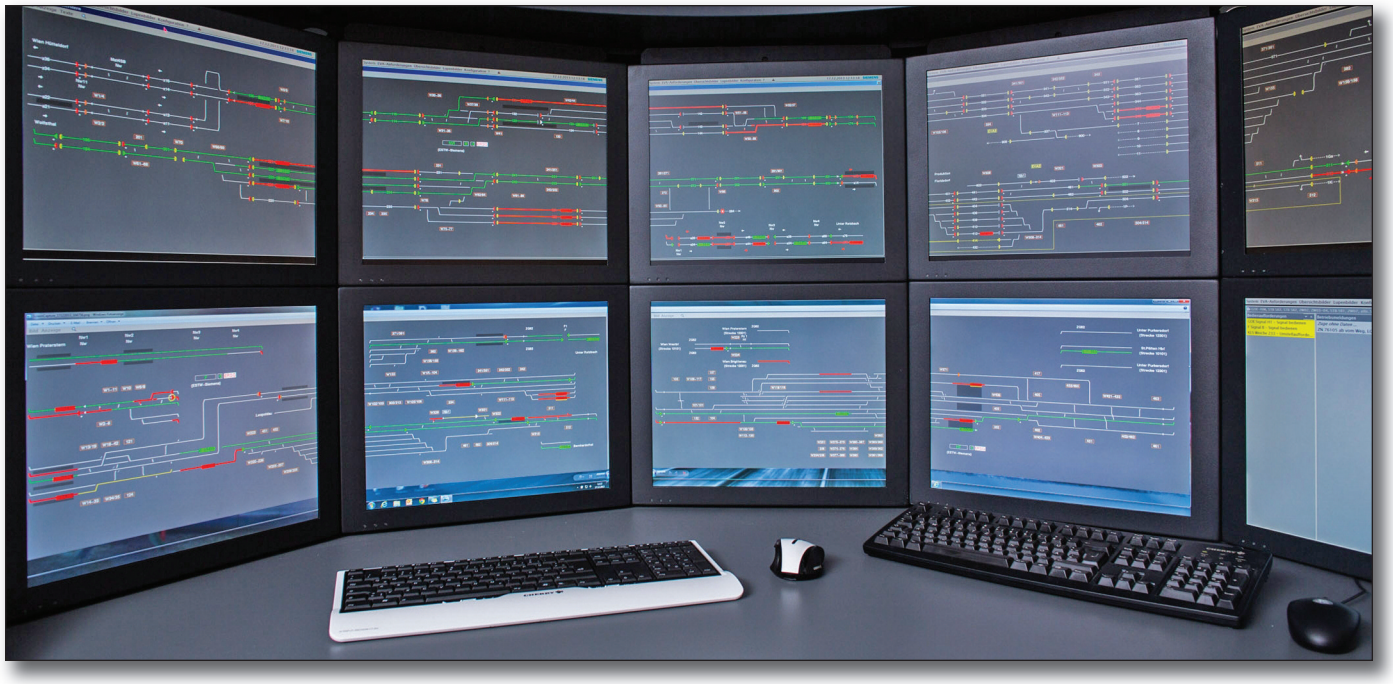

Figure 2. A Workplace in a Railway Control Center (© Siemens AG Österreich 2016).

Figure 3. Railway Interlocking System Hardware.

© Siemens AG Österreich 2016.

heuristics to cope with the large size of the products that were configured (Fleischanderl et al. 1998). (2) The concept of soft constraints (Meseguer, Rossi, and Schiex 2006) was used by associating a strength factor to each constraint. If no solution could be found fulfilling all constraints, constraints with a low strength factor were relaxed. It turned out that such soft constraints implied a high additional complexity to the reasoning, so we dropped it in the next version of S'UPREME, replacing it by allowing the user (or at least a class of expert users) to manually mark constraints permitted to be violated. Such violation permissions are documented in a report as part of the output and are therefore traceable for each final product. (3) A small part of the system was still rule based, especially the cable layout functions. Constraints are undirected, which is usually a big advantage over rules, but in this case undirectedness was not necessary, because cables were always layouted when everything else was already determined. So rules showed a winning performance in this case.

The configurator demonstrated the power of constraints for reducing maintenance costs and was in productive use for about 10 years until Siemens withdrew from the telecom business. In particular, the maintenance costs were reduced dramatically. Comparable "conventional" systems needed about 15 percent of the development costs for maintenance per year. By applying constraint methods we could reduce this to 2 percent.

Configurator Family for Railway Interlocking Systems (2000 to Date)

Encouraged by the success of our constraint-based configurator for telecommunication switches, we applied its techniques again to the railway automation domain, but this time for an interlocking technology that is used worldwide in various countries and projects.

Around the year 2000, we redesigned our genera-

tive constraints framework COCOS and ported it from Smalltalk to Java. Based on S'UPREME, we developed a tool family that enables project and sales engineers to perform a detailed technical configuration of electronic interlocking systems (including software, hardware, user interfaces, communication equipment). The size and complexity of the task is very high. Typical configurations comprise 10,000 elements, 50,000 attributes/associations, and 100,000 active constraints. In order to make the configuration process efficient, many of the necessary decisions are supported by our tool through initial values, computation functions, domain filtering, and solving. In a sequence of manual and automated steps, only several hundreds of explicit decisions need to be made by the user to achieve a complete configuration for average-sized systems.

The configurators are used in more than 10 countries for different railway companies. While most of the structures and requirements are the same for all of them, they differ in some — sometimes many — details. For instance, types of available hardware elements are different in different countries, constraints vary how to connect components, and special user interface views or import/export functions are needed. To cope with that situation, the architecture of our configurators consists of three layers: (1) A domain-independent kernel, providing mechanisms for knowledge base representation, constraint-based reasoning, and generic user interfaces. (2) A base layer, containing all domain-specific knowledge that is common to all configurator variants. (3) The variant layers, representing the specialities of different countries.

The layered architecture facilitates reuse considerably: the variant layers do not need to express the same again and again — thus the code size of a typical variant is approximately 1/10 of the whole code (a small value, considering that railway companies in different countries have quite different requirements).

The GCSP approach integrates OO and CSP paradigms: Although the knowledge base is modeled in an object-oriented way, the attributes and associations of the model can be viewed as variables of a constraint-satisfaction problem. For each attribute, one variable is generated. For each association two variables (one for each role of the association) are generated. For each variable in the knowledge base, additional information can be provided (initial values, possible values, cardinality of associations). From that, built-in constraints are derived.

The reasoning is based on constraints — built-in and special expressions such as logical expressions or expressions about association cardinalities. Every constraint is either specified in a domain-specific language (Configuration Specification Language) and then automatically transformed to Java, or directly implemented in Java, allowing arbitrary Java statements to be used additionally, provided the value (true, false, undefined) of the constraint only depends on the variables used in the current configuration. An example is shown in Listing 3.

The solving mechanism is repair based. Violated constraints indicate parts of the configuration that need to be repaired. The user can repair a constraint violation manually, by choosing from a list of possible repair steps. Alternatively, the solver can be started to solve the constraints according to the solver's current heuristics. A valid configuration is found, if all user requirements are fulfilled and there are no violated constraints left. The same mechanism is used for reconfiguration (for example, after knowledge base changes). Violated constraints indicate that parts of the system need to be modified. These violations can either be repaired manually, giving the user full control over the process, or automatically, with all solvers generally trying to minimize changes in the configuration.

Rules — which are a kind of domain-specific propagators — are used if the inference always works in only one direction. They are more efficient than constraints because they do not need search. The S'UPREME rule mechanism can be used in parallel with constraints. A rule is written in a procedural, object-oriented style. Its result must solely depend on the used variables; that is, when the values of the variables stay the same, the rule must compute the same output values.

The configuration of a large system consists of different subtasks (for example, hardware configuration, software configuration). To allow the user to concentrate on the task at hand, the whole problem space can be partitioned into clusters that can be activated and processed separately. Figure 4 shows the configurator's user interface with such clusters grouped as phases and steps in the upper left window.

Interactive Configurator for ETCS (2002 to Date)

ETCS (European Train Control System) is a European standard for train control. It mainly consists of transmitters (called *balises*) installed on the tracks, which send telegrams to the passing train providing movement authorities and track-side profile data (locally allowed maximum speeds, gradients). In the level 2 variant of ETCS, trains are controlled by a radio center. Balises are used for train position calibration (see figure 5).

Such systems tend to get very large (the number of variables and constraints involved is typically of magnitude 10^5) and are highly dynamic: each station has a different topology and consists of a different number of different types of signals, tracks, and branches.

We used generative constraint framework S'UPREME to realize the ETCS configurator. Working

```
enumeration SignalType {main, protection, shunting}

class Signal {
  attr type: SignalType ;
}

class TrackSegment {
  assoc signals [0..*]:  Signal;

  // at most one main signal is allowed on a track segment
  constraint onlyOneMainSignal {
     assert signals.select(s|s.type == SignalType.main).size <= 1;
  }
}
```

Listing 3. GCSP Constraint Example.

with that ETCS configurator is mainly an interactive task: the user creates the route topology with all necessary profile data to find the optimal number and locations of balises and to generate their telegrams.

Constraint violations guide the user to build a consistent and complete system. Constraint propagation is employed to filter inconsistent options. Typically, constraints have to navigate through the topology to find locations and distances. For example:

> The distance between two neighboring balises must not exceed 2.000 meters.

Such constraints are either used to tell the user that an additional balise must be created if the distance between two neighboring balises is too high, or to automatically place the balises at the correct distance. When all topology and profile data are consistently configured, the telegrams for the balises can be derived deterministically. Due to performance reasons, we use forward-chaining rules instead of constraints to represent the derivation of balise telegrams.

Our ETCS configurator is in productive use for about 20 different countries. Each country has slightly different operation regulations, so we have about 20 different variants of the configurator. The object-oriented layout of the knowledge base supports code stability and low maintenance costs.

Web-Based Configurators for Reciprocating Compressors and Valves (2010 to Date)

A gas compressor consists of various parts of different construction types and materials, like cylinders, pistons, valves, and a driving system (see figure 6). The goal is to find compressors and valves that fulfill all user requirements: for example, find a configuration for a given gas mix and operating conditions with a discharge temperature below a specified limit and appropriate cylinder sizes for a required volume flow.

The validity of a solution is described by two kinds of constraints: compatibility constraints specify which parts of which sizes fit together (for example, valves of type V1 are not compatible with compressors of type C1). In contrast, physical constraints calculate the thermodynamic behavior of the gas composition in the compressor chambers and restrict the selection of parts and materials. A typical example of a physical constraint is the relationship between eigenfrequency of a valve plate and rotation speed in a compression chamber:

$$\frac{eigenfreq_{plate}}{speed_{rotation}} > c$$

$$eigenfreq_{plate} = \sqrt{\frac{(N \times k_{valves}) + k_{plate}}{mass_{plate}}}$$

N is the number of valves for the compression chamber, k_{valves} is the spring constant of the valve springs, k_{plate} is the spring constant of the valve plate, and $mass_{plate}$ is the mass of the plate. $speed_{rotation}$ is the rotation speed of the driving system. c is a constant.

If this inequality is not satisfied for a concrete compressor and valve constellation, that constellation is not valid and is therefore ruled out.

Depending on the input requirements (like the required discharge pressure), a compressor consists of different stages, and each stage of several cylinders. An interesting aspect of this problem is that we divided it into subproblems (roughly speaking, a subproblem corresponds to a stage) and solved the whole problem in two phases: (1) find potential solutions for each stage, and (2) find valid combinations of stage solutions. We use a backtrack algorithm to find valid solutions for the stages. However, not only the physical validity of a solution is expected, but also optimality according to several criteria, like costs, maintenance effort, gas compatibility, and strategic preferences. So the search tree is fully examined in a backtrack search manner, using constraint propagation and ordering techniques for early pruning of dead-end branches or branches with low fitness functions. The number of different constellations is of magnitude 10^7, run time ranges from less than a second for simple compressors up to one minute. In the second phase, consistency propagation guarantees that only stage configurations that are consistent among each other are combined to a solution.

The properties of each cylinder (like type, diameter, valves) correspond to the variables of a CSP; their domains are the result of the preceding backtrack search. There are many restrictions determining which domain values of the different stages may be combined in a final solution. We used a constraint-propagation algorithm to achieve strong n-consistency (Bessiere 2006) where n is the number of variables, making these cross-stage restrictions explicit. The computation of strong n-consistency is expensive, but in our case the number of subproblems /

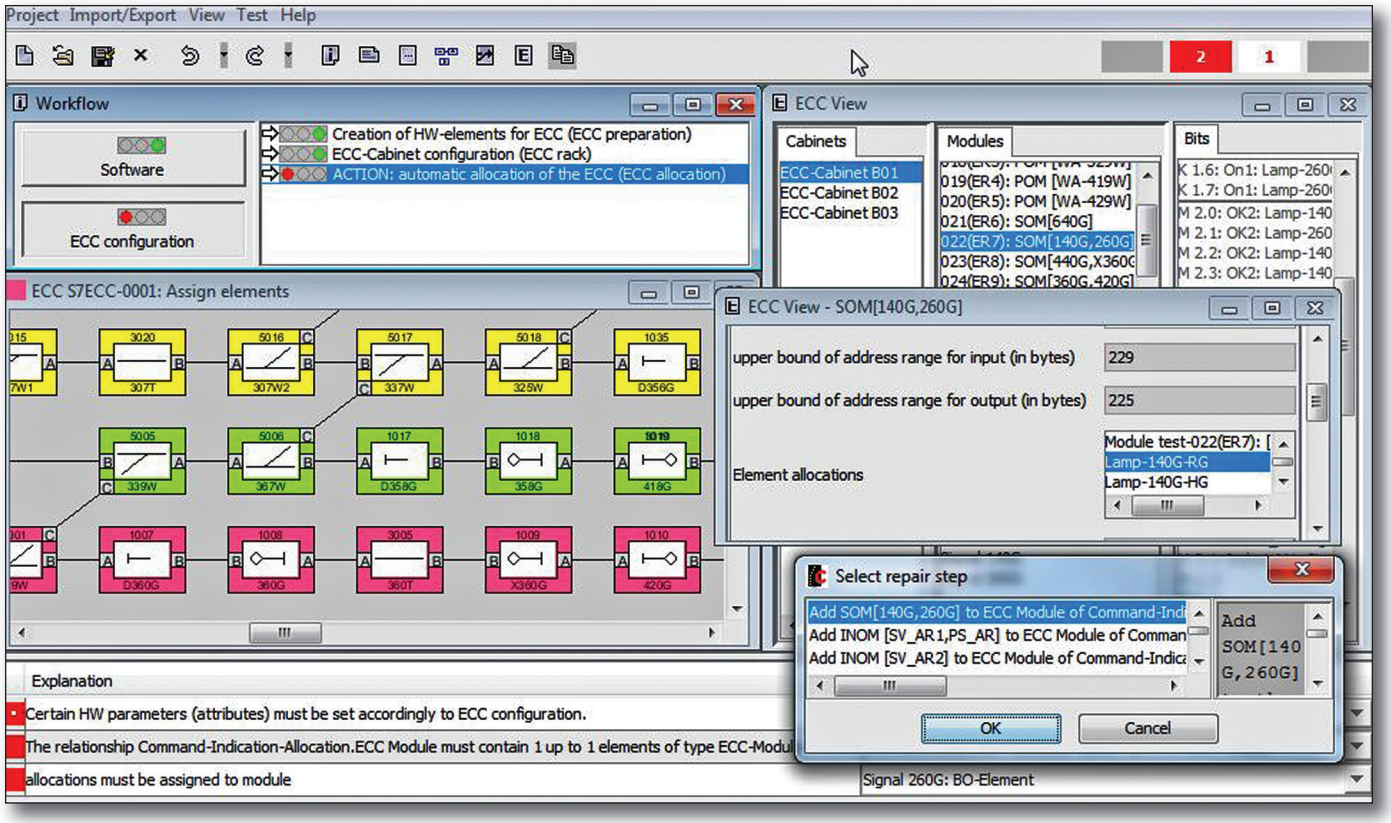

Figure 4. A Configurator for Railway Interlocking Systems.

stages are quite limited (< 10). Strong n-consistency has the remarkable property that a solution can be found without search (if one exists). Hence, when the user interactively selects one special cylinder, it is computationally efficient to remove all mismatching cylinders of the other stages from the selection lists on the GUI. As a result, consequent application of constraint methods helps to find the best solution within a given design space and to avoid erroneous manual trials.

Summary

Within Siemens, constraint technologies have been successfully used for solving configuration problems for more than 25 years. By the application of constraints we were able to significantly reduce not only the development effort but also maintenance costs compared to procedural or rule-based systems. In cases where we could compare similar configuration systems, approximately 80 percent of the maintenance costs and more than 60 percent of the development costs for the knowledge representation and reasoning tasks were saved. There were three crucial success factors.

First, the combination of object-oriented and constraint technologies. Object-oriented modeling provides an expressive language and a natural structuring of the domain knowledge. Constraints support a clean and declarative formulation of the different requirements and restrictions.

Second was the application of various reasoning services and methods, which were best suited for each concrete problem. The most important methods we used were: (1) Constraint checking to give feedback to the user about the current consistency state of the configuration. (2) Constraint propagation to filter out invalid choices. (3) k-consistency propagation to provide positive and negative implicants (for small problems). (4) Backtrack algorithms to implement a complete search (for small and medium-size problems). (5) Iterative repair algorithms to perform a search for large-scale problems. (6) Optimization search to find the best or good-enough solutions. (7) Reconfiguration technologies to reconcile legacy configurations.

Third was the capability to deal with large dynamic configuration problems, where the number of components in a solution is not known beforehand. In this case the standard theory of constraint satisfaction had to be combined with expressions that define instantiations of classes, resulting in generative constraint satisfaction.

Although many real-world problems have been

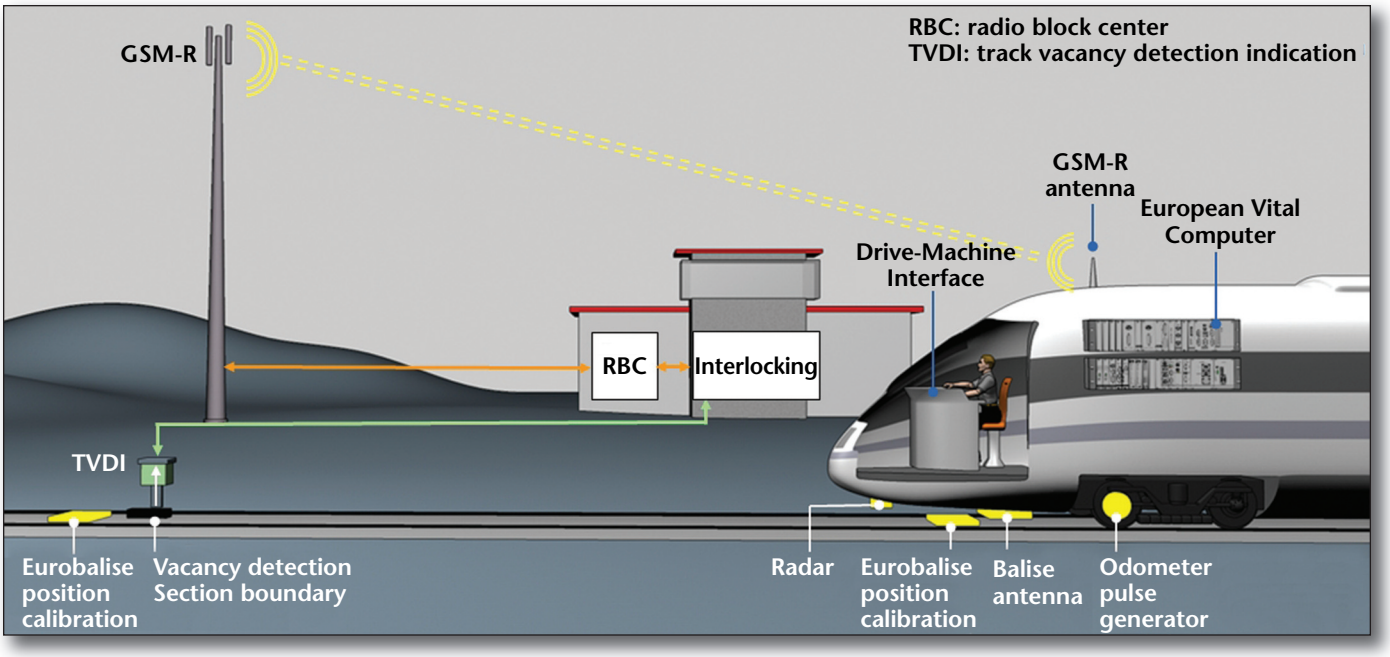

Figure 5. ETCS, a European Train Control Standard.

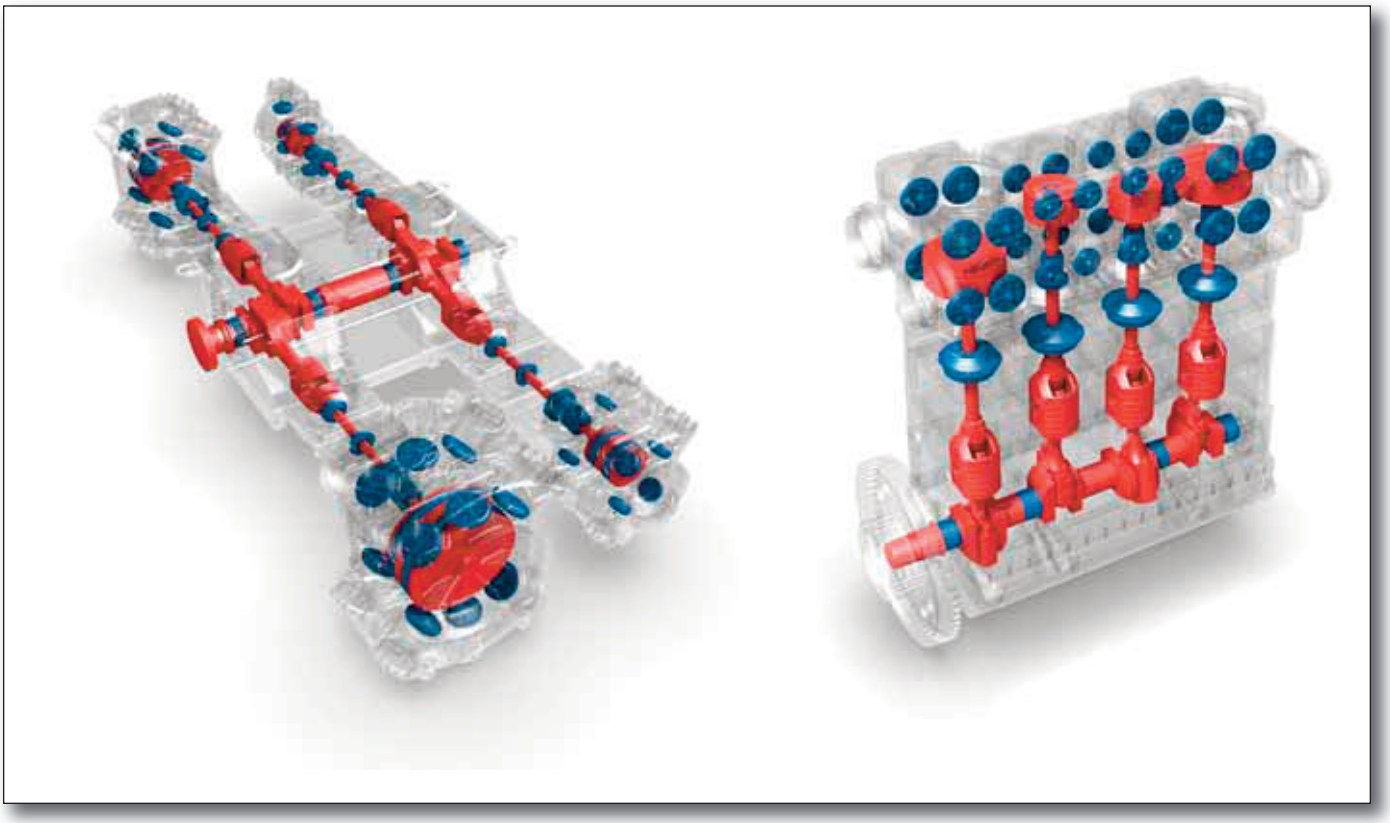

Figure 6. 3D View of Process Gas Compressors and Compressor Valves.

successfully solved with constraint technologies, we are still far from providing out-of-the-box solutions for complex, dynamic configuration problems. One of the main open tasks is to design general and robust methods for reasoning about the existence of objects in an object-oriented environment (that is, expressing existential quantification over complex structures) and to provide these methods in off-the-shelf configuration frameworks.

References

Ahuja, R. K.; Möhring, R. H.; and Zaroliagis, C. D., eds. 2009. *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems,* vol. 5868 of Lecture Notes in Computer Science. Berlin: Springer.

Aschinger, M.; Drescher, C.; Friedrich, G.; Gottlob, G.; Jeavons, P.; Ryabokon, A.; and Thorstensen, E. 2011. Optimization Methods for the Partner Units Problem. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems:* 8th International Conference Proceedings (CPAIOR 2011), vol. 6697 of Lecture Notes in Computer Science, ed. T. Achterberg and J. C. Beck, 4–19. Berlin: Springer. dx.doi.org/10.1007/978-3-642-21311-3_4

Bazaraa, M. S.; Sherali, H. D.; and Shetty, C. M. 1993. *Nonlinear Programming: Theory and Algorithms* (2nd ed.). New York: Wiley.

Benavides, D.; Segura, S.; and Cortés, A. R. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35(6): 615–636. dx.doi.org/10.1016/j.is.2010.01.001

Beneventano, D.; Bergamaschi, S.; Lodi, S.; and Sartori, C. 1998. Consistency Checking in Complex Object Database Schemata with Integrity Constraints. *IEEE Transactions on Knowledge and Data Engineering* 10(4): 576–598. dx.doi.org/10.1109/69.706058

Bertsekas, D. P. 2009. *Convex Optimization Theory.* Belmont, MA: Athena Scientific.

Bessiere, C. 2006. Constraint Propagation. *Handbook of Constraint Programming,* Foundations of Artificial Intelligence, vol. 2, ed. F. Rossi, P. van Beek, and T. Walsh, 29–83. Amsterdam: Elsevier. dx.doi.org/10.1016/S1574-6526(06)80007-6

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability,* vol. 185 of Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer Set Programming at a Glance. *Communications of the ACM* 54(12): 92–103. dx.doi.org/10.1145/2043174.2043195

Clements, P. C., and Northrop, L. M. 2001. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Boston, MA: Addison-Wesley.

Crawford, J. M.; Ginsberg, M. L.; Luks, E. M.; and Roy, A. 1996. Symmetry-Breaking Predicates for Search Problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning* (KR'96), 148–159. San Francisco, CA: Morgan Kaufmann Publishers.

de Moura, L. M. 2011. Orchestrating Satisfiability Engines. In *Principles and Practice of Constraint Programming* (CP 2011) 17th International Conference Proceedings, vol. 6876 of Lecture Notes in Computer Science, ed. J. H. Lee. Berlin: Springer.

de Moura, L. M., and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems,* 14th International Conference (TACAS 2008), vol, 4963 of Lecture Notes in Computer Science, ed. C. R. Ramakrishnan and J. Rehof, 337–340. Berlin: Springer. dx.doi.org/10.1007/978-3-642-23786-7_1

Dechter, R., and Dechter, A. 1988. Belief Maintenance in Dynamic Constraint Networks. In *Proceedings of the 7th National Conference on Artificial Intelligence,* 37–42. Menlo Park, CA: AAAI Press / The MIT Press.

Dhungana, D.; Falkner, A. A.; and Haselböck, A. 2013. Generation of Conjoint Domain Models for System-of-Systems. In *Proceedings of Generative Programming: Concepts and Experiences* (GPCE'13), ed. J. Järvi and C. Kästner, C., 159–168. New York: Association for Computing Machinery. dx.doi.org/10.1145/2517208.2517224

Falkner, A. A.; Felfernig, A.; and Haag, A. 2011. Recommendation Technologies for Configurable Products. *AI Magazine* 32(3): 99–108.

Falkner, A. A.; Haselböck, A.; Schenner, G.; and Schreiner, H. 2011. Modeling and Solving Technical Product Configuration Problems. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing* (AI EDAM) 25(2): 115–129. dx.doi.org/10.1017/S0890060410000570

Faltings, B., and Freuder, E. C. 1998. Guest Editors' Introduction: Configuration. *IEEE Intelligent Systems* 13(4): 32–33. dx.doi.org/10.1109/MIS.1998.708430

Fleischanderl, G.; Friedrich, G.; Haselböck, A.; Schreiner, H.; and Stumptner, M. 1998. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems* 13(4): 59–68. dx.doi.org/10.1109/5254.708434

Freuder, E. C. 1991. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of the 9th National Conference on Artificial Intelligence,* vol. 1, 227–233. Menlo Park, CA: AAAI Press / The MIT Press.

Freuder, E. C., and Mackworth, A. K., eds. 1994. *Constraint-Based Reasoning*. Cambridge, MA: The MIT Press.

Gelfond, M. 2008. Answer Sets. In *Handbook of Knowledge Representation,* vol. 3 of Foundations of Artificial Intelligence, ed. F. van Harmelen, V. Lifschitz, and B. W. Porter, 285–316. Amsterdam: Elsevier.

Gent, I. P.; Petrie, K. E.; and Puget, J. 2006. Symmetry in Constraint Programming. In *Handbook of Constraint Programming,* Foundations of Artificial Intelligence vol. 2, ed. F. Rossi, P. van Beek, and T. Walsh, 329–376. Amsterdam: Elsevier. dx.doi.org/10.1016/S1574-6526(06)80014-3

Haselböck, A. 1993. Exploiting Interchangeabilities in Constraint-Satisfaction Problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence,* ed. R. Bajcsy, 282–289. San Francisco: Morgan Kaufmann Publishers.

Hebrard, E.; O'Mahony, E.; and O'Sullivan, B. 2010. Constraint Programming and Combinatorial Optimisation in Numberjack. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems,* 7th International Conference (CPAIOR 2010) Proceedings, vol. 6140 of Lecture Notes in Computer Science, ed. A. Lodi, M. Milano, and P. Toth, 181–185. Berlin: Springer. dx.doi.org/10.1007/978-3-642-13520-0_22

Hentenryck, P. V. 1997. Numerica: A Modeling Language for Global Optimization. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (IJCAI 97), 1642–1650. San Francisco: Morgan Kaufmann.

Hoos, H. H., and Tsang, E. P. K. 2006. Local Search Methods.

In *Handbook of Constraint Programming,* Foundations of Artificial Intelligence vol. 2, ed. F. Rossi, P. van Beek, and T. Walsh, 135–167. Amsterdam: Elsevier. dx.doi.org/10.1016/S1574-6526(06)80009-X

Hull, R., and King, R. 1987. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys* 19(3): 201–260. dx.doi.org/10.1145/45072.45073

Kang, K.; Cohen, S.; Hess, J.; Novak, W.; and Peterson, S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* 7(3): 499–562. dx.doi.org/10.1145/1149114.114911

McDermott, J. P. 1982. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence* 19(1): 39–88. dx.doi.org/10.1016/0004-3702(82)90021-2

Mears, C.; Schutt, A.; Stuckey, P. J.; Tack, G.; Marriott, K.; and Wallace, M. 2014. Modelling with Option Types in MiniZinc. In *Integration of AI and OR Techniques in Constraint Programming:* 11th International Conference (CPAIOR 2014) Proceedings, vol. 8451 of Lecture Notes in Computer Science, ed. H. Simonis, 88–103. Berlin: Springer. dx.doi.org/10.1007/978-3-319-07046-9_7

Meseguer, P.; Rossi, F.; and Schiex, T. 2006. Soft Constraints. In *Handbook of Constraint Programming,* Foundations of Artificial Intelligence vol. 2, ed. F. Rossi, P. van Beek, and T. Walsh, 281–328. Amsterdam: Elsevier. dx.doi.org/10.1016/ S1574-6526(06)80013-1

Minsky, M. L. 1967. *Computation: Finite and Infinite Machines*. Upper Saddle River, NJ: Prentice-Hall, Inc.

Mittal, S., and Falkenhainer, B. 1990. Dynamic Constraint Satisfaction Problems. In *Proceedings of the 8th National Conference on Artificial Intelligence,* 25–32. Menlo Park, CA: AAAI Press / The MIT Press.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming* (CP 2007), 13th International Conference, vol. 4741 of Lecture Notes in Computer Science, ed. C. Bessiere, 529–543. Berlin: Springer.

Papadimitriou, C. H., and Steiglitz, K. 1982. *Combinatorial Optimization: Algorithms and Complexity.* New York: Prentice-Hall.

Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming,* Foundations of Artificial Intelligence vol. 2. Amsterdam: Elsevier North-Holland Inc.

Selman, B.; Levesque, H. J.; and Mitchell, D. G. 1992. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence,* 440–446. Menlo Park, CA: AAAI Press / The MIT Press.

Soloway, E.; Bachant, J.; and Jensen, K. 1987. Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base. In *Proceedings of the 6th National Conference on Artificial Intelligence,* 824–829. San Mateo, CA: Morgan Kaufmann.

van Beek, P. 2006. Backtracking Search Algorithms. In *Handbook of Constraint Programming,* Foundations of Artificial Intelligence vol. 2, ed. F. Rossi, P. van Beek, and T. Walsh, 85–134. Amsterdam: Elsevier. dx.doi.org/10.1016/S1574-6526(06)80008-

Wallace, M. 1996. Practical Applications of Constraint Programming. *Constraints* 1(1/2): 139–168. dx.doi.org/10.1007/ BF00143881

Wallace, R. J., and Freuder, E. C. 1998. Stable Solutions for Dynamic Constraint Satisfaction Problems. In P*rinciples and Practice of Constraint Programming* (CP98), 4th International Conference Proceedings, vol. 1520 of Lecture Notes in Computer Science, ed. M. J. Maher and J. Puget, 447–461. Berlin: Springer. dx.doi.org/10.1007/3-540-49481-2_32

Andreas Falkner holds an MS and a Ph.D. degree in computer science from the Vienna University of Technology. Since 1992 he has been developing product configurators for complex technical systems in various domains at Siemens AG Österreich. At present, he is a senior research scientist at Siemens' Corporate Technology, Research Group Configuration Technologies, and senior key expert for product configuration and mass customization.

Gerhard Friedrich is a full professor of computer science at the Alpen-Adria-University Klagenfurt, Austria. He is the dean of the Faculty of Technical Sciences and directs the research group on Intelligent Systems and Business Informatics. Before his academic career, he was the head of the Department for Configuration and Diagnosis Systems at Siemens AG Austria. His research interests include configuration, planning, and diagnosis as well as knowledge representation, acquisition, maintenance, and reasoning. Gerhard Friedrich received a Ph.D. and an MS in informatics from Vienna University of Technology, Austria. In 2012 he became a fellow of the European Association for Artificial Intelligence.

Alois Haselböck is a member of the research staff at Siemens AG Österreich, Corporate Technology, and is a senior key expert on deductive and constraint-based reasoning in the Research Group Configuration Technologies. He received an MS and a Ph.D. degree in computer science from the Vienna University of Technology. His research interests comprise knowledge representation and solving techniques for constraint-satisfaction systems where he has contributed fundamental findings in the field of generative constraint satisfaction.

Gottfried Schenner is a senior research scientist at Siemens' Corporate Technology, Research Group Configuration Technologies. He received his MS degree in computer science from the Vienna University of Technology. His research interests include constraint-based product configuration and the application of AI technologies (CSP, ASP) in industrial software development projects.

Herwig Schreiner is the head of the Research Group Configuration Technologies within Siemens Corporate Technology in Vienna. He holds an MS degree in computer science from Vienna Technical University and an IPMA senior project manager certificate. He has more than 20 years experience in technologies around configuring complex systems as well as management of large-scale software projects. His research interest focuses on knowledge-based systems and configuration technologies, particularly on constraint-based reasoning in static and dynamic environments.