Constraint Solver Requirements for Interactive Configuration

Andreas Falkner and Alois Haselböck and Gerfried Krames and Gottfried Schenner and Richard Taupe[1]

Abstract. Interactive configuration includes the user as an essential factor in the configuration process. The main two components of an interactive configurator are a user interface on the front-end and a knowledge representation and reasoning (KRR) framework on the back-end. In this paper we discuss important requirements for the underlying KRR system to support an interactive configuration process while focusing on classical constraint satisfaction as one of the most prominent KRR technologies for configuration problems. We evaluate several freely available constraint systems with respect to the identified requirements for interactive configuration and observe that many of those requirements are not well supported.

1 INTRODUCTION

Constraint satisfaction [18] is often used as an underlying reasoning system for product configuration problems [12]. Product configuration is usually an interactive task: Iteratively, the user makes a decision and the configurator computes the consequences of this decision. In his PhD thesis [6], David Ferrucci summarized the inherent interactivity of configurators as follows:

> *"Interactive configuration is a view of the configuration task which includes the user as an essential component of a dynamic process. The interactive configurator is designed to assist the user in an interactive and incremental exploration of the configuration space. It may guide or advise the user's decision making but it must communicate requirements or inconsistencies effected by the constraints in response to the user's choices. This feedback helps the user to refine the configuration space toward a satisfying solution.*
> *The key component of an interactive configurator is the constraint manager which is responsible for incrementally maintaining the relationship between choices and constraint violations. [. . .] The requirement to deliver meaningful and timely feedback imposes significant demands on the flexibility, efficiency and explainability of constraint management."*

Pieter van Hertum et al. studied in [9] how the knowledge base paradigm – the separation of concerns between information and problem solving – could hold in the context of interactive configuration. They identified a set of subtasks that overlaps well with the set of requirements we propose in this paper in Section 4: Acquiring information from the user, generating consistent values for a parameter, propagation of information, checking the consistency for a value, checking a configuration, autocompletion, explanation, and backtracking.

Matthieu Queva et al. describe in [17] requirements on interactive configurators mainly from the modelling perspective and call for high-level, expressive languages like UML or SysML. They also mention constraint modelling as an important aspect of configuration. From the viewpoint of constraint reasoning, Jeppe Madsen identified three fundamental interactivity operations in his master thesis [14]: add constraint, remove constraint, and restoration.

In this paper, we concretize those ideas and focus on applications where a tailor-made user interface (UI) or legacy system is enhanced with solving capabilities, i.e., by calling a general constraint solver as a component. An alternative would be to implement a special UI for an integrated configuration system such as a commercial CPQ tool or sales configurator suite, but that bears the risk of vendor-lock-in.

Figure 1 gives an overview of the addressed scenario: Users interact with the configurator to achieve their goals, thus putting challenges to user interface functionality. Those challenges pose reasoning interaction requirements on the API of the used constraint solver. The solver is an off-the-shelf product (open-source or commercial) and has a general API, partly dedicated to configuration problems. Together with a domain-specific product model (or knowledge base, KB), it forms the KRR component. This back-end is called by the control component which first hands over product model and user-set values for decision variables from the UI to the API and then sends the solver results back to the UI.

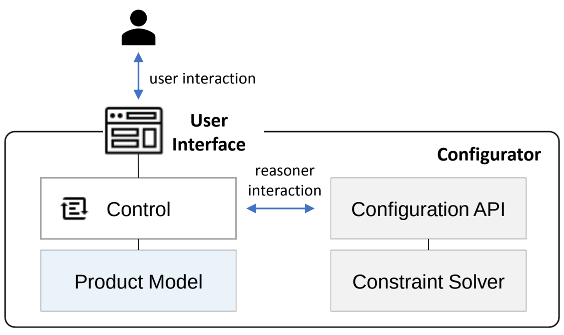

Figure 1. Components of an interactive configurator

The configurator shall help a user to configure a product according to his/her needs and in full compliance to the product model. The user expects the configurator to show all necessary decisions (variations) in a clear and well-arranged way, to highlight or preset the "best" alternative (value), to filter or grey-out infeasible values, to

[1] Siemens AG Österreich, Corporate Technology, Vienna, Austria
andreas.a.falkner@siemens.com, alois.haselboeck@siemens.com, gerfried.krames@siemens.com, gottfried.schenner@siemens.com, richard.taupe@siemens.com
Author names are ordered alphabetically.

recommend alternatives in case of conflicts, and to respond quickly (preferably instantaneously) to user inputs.

The remainder of this paper is organized as follows: In Section 2, an example for interactive configuration is introduced. In Section 3, we give a problem definition of interactive configuration. We define the requirements for an interactive configuration API in Section 4 and investigate in Section 5 how some typical constraint solvers satisfy these requirements. We conclude the paper in Section 6 with a summary and future work.

2 EXAMPLE

We show the challenges of interactive configuration in a small example for a configurable product with components that can occur multiple times (similar to generative constraint satisfaction [7] or cardinality-based feature modelling [3]).

A Metro train wagon has as configurable attributes the size (length in millimetres: 10000..20000) and the expected load (number of passengers: 50..200) which can be realized as seats or standing room. As components we consider only seats (max. 4 per meter of length) and handrails, and their number is configurable.

There is at most one handrail in a wagon (mandatory if there is standing room) and it has a configurable type: "standard" or "premium".

A single seat consumes standing room for 3 persons and has as configurable attributes the type ("standard", "premium", "special") and the color ("blue", "red", "white"). The type is constrained such that standard is not allowed to be mixed with premium (for seats and handrails). The color of all seats must be the same, except for special seats which have to be "red".

Users expect the following (static) default values: type = standard, color = blue. Furthermore, they prefer to use all available space (as defined by the length) for passengers (i.e., maximize the load factor).

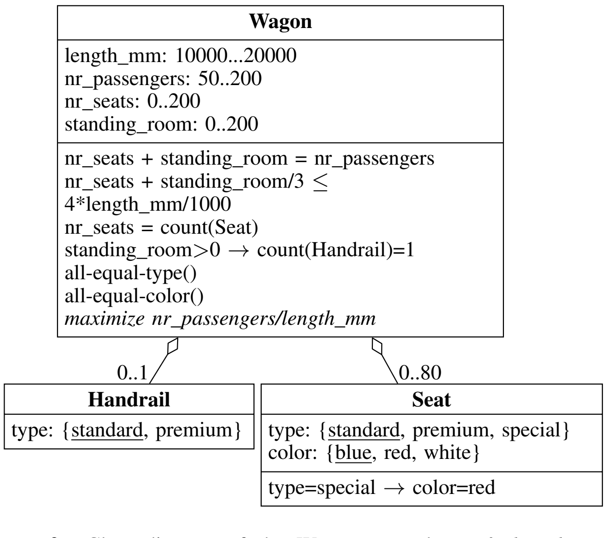
Wagon
length_mm: 10000...20000
nr_passengers: 50..200
nr_seats: 0..200
standing_room: 0..200
nr_seats + standing_room = nr_passengers
nr_seats + standing_room/3 ≤
4*length_mm/1000
nr_seats = count(Seat)
standing_room>0 → count(Handrail)=1
all-equal-type()
all-equal-color()
maximize nr_passengers/length_mm
0..1
0..80
Handrail
type: {standard, premium}
Seat
type: {standard, premium, special}
color: {blue, red, white}
type=special → color=red

Figure 2. Class diagram of the Wagon example. *Default values are underlined. Wagon.all-equal-type() stands for a constraint that all sub-parts must have the same type except for special. Wagon.all-equal-color() stands for a constraint that all associated seats (except if type=special) must have the same color.*

Figure 2 shows a UML class diagram for this sample specification, including pseudo code for all constraints. A modelling in a standard constraint solver is more verbose because it requires the mapping of the UML classes to arrays of variables and some other implementation decisions, e.g., special handling of the "dynamic" parts, i.e., number of seats dependent on the expected load and length – see the MiniZinc program in Listing 1.

Listing 1. MiniZinc program for the Wagon example

```
% Constants , Domains
int: min_length = 10000;
int: max_length = 20000;
int: max_seats = max_length*4 div 1000;
int: min_load = 50;
int: max_load = 200;

enum Color = { blue , red , white , noColor };
enum Type = { standard , premium , special , noType };

% Wagon
var min_length..max_length: length_mm;
var min_load..max_load: nr_passengers;
var 0..max_seats: nr_seats;
var 0..max_load: standing_room;
var 0..1: nr_handrails;

% Seats
array [1..max_seats] of var Color: seat_color;
array [1..max_seats] of var Type: seat_type;

% Handrail
var Type: handrail_type;

% Constrain numbers
constraint nr_seats + standing_room = nr_passengers;
constraint nr_seats + standing_room/3 <=
    ↪ length_mm*4/1000;

% Mandatory handrail for standing room with proper type
constraint standing_room > 0 -> nr_handrails = 1;
constraint handrail_type != special;
constraint nr_handrails = 0 <-> handrail_type = noType;
constraint nr_handrails > 0 -> forall (i in 1..nr_seats
    ↪ where seat_type[i] != special) (handrail_type =
    ↪ seat_type[i]);

% Same color and type for all seats but special
constraint forall (i in nr_seats+1..max_seats)
    ↪ (seat_color[i] = noColor);
constraint forall (i in nr_seats+1..max_seats)
    ↪ (seat_type[i] = noType);
constraint forall (i,j in 1..nr_seats where i<j)
    ↪ (seat_type[i] != special /\ seat_type[j] !=
    ↪ special -> seat_type[i] = seat_type[j]);
constraint forall (i,j in 1..nr_seats where i<j)
    ↪ (seat_type[i] != special /\ seat_type[j] !=
    ↪ special -> seat_color[i] = seat_color[j]);
constraint forall (i in 1..nr_seats) (seat_type[i] =
    ↪ special -> seat_color[i] = red);

% Use full length for passengers (avoid dead space)
solve maximize nr_passengers/length_mm; % load factor
```

3 PROBLEM DEFINITION

The main two parties in interactive configuration are the *user* and the *configurator*. The user's goal is to configure a product such that it is a valid and complete product variant that meets all his/her individual requirements. The configurator is a digital companion that supports the configuration process by deriving consequences of the user's choices and by assisting to avoid and resolve conflicts.

The simplest type of a user interaction is to set the value of a configuration parameter. This can be seen as answering a question that the configuration tool asked. Example: *For how many passengers should the wagon provide seats?* Of course, the user should also be able to withdraw his/her decision, which corresponds to unsetting a configuration parameter. The value of the parameter will then be either undefined or the default value or set by solving.

In most non-trivial configuration problems, a dynamic number of configuration objects plays an important role. Some authors of this paper have developed configurators for large industrial products for more than 25 years and always faced problems that could not be represented as simple, static lists of configuration parameters (see [4]). Therefore, another important type of user interaction is the creation and deletion of configuration objects. In the example problem in Section 2, *seats* are configuration objects whose number is not known beforehand, and where each seat can be configured individually (seat color and type). Typically, there are two different ways how the user manipulates the number of individual configuration objects in the user interface: either by creating them one by one, or by specifying the number of objects and letting the configuration tool create the individual objects. But in both cases, the result is a set of configuration objects whose number was not known beforehand and whose properties can be configured further. This is not the case for the representation of *standing_room* in our example. The user can set only the capacity as a number but no further individual properties. Thus, standing room need not to be modelled as configuration objects in our example.

Definition (User Interaction)**.** *The main types of user interactions in interactive configuration are: (i) create configuration object, (ii) delete configuration object, (iii) set/unset configuration object attribute, (iv) set/unset association between configuration objects.*

We are aware that various approaches to interactive configuration may define the list of main types of user interactions differently. For example, structure-based configuration considers the following types of user interaction: parametrization, decomposition, integration, and specialization [10]. Due to the limited scope of this paper, we focus on types of user interaction that are most relevant in our experience.

User interactions change the state of the configuration by making decisions. Solver interactions make implicit knowledge explicit to assist the user. For instance, the consequence of a user setting the attribute `type` of a seat to `special` is that the solver will automatically set its attribute `color` to `red` because of the according constraint – see, e.g., user action 5 in Figure 3. Another typical solver interaction is domain filtering. If, e.g., the `nr_passengers` was set to 160 as by user action 1 in Figure 3, then the lower bound of `length_mm` would change to 13334 because of the constraint $\texttt{nr\_seats + standing\_room/3} \leq \texttt{4*length\_mm/1000}$ (for the case that `nr_seats` is 0) and analogously, for the case that `length_mm` is 20000, the upper bound of `nr_seats` would be 40 (not more because the lower bound of `standing_room` must be 120 to achieve the 160 passengers). This knowledge was already implicitly contained in the configuration model of the problem, as described above, but the solver made its consequences explicit, thus creating a distinct benefit to the user.

Definition (Solver Interaction)**.** *A solver interaction is a set of consequences following a user interaction. Typical solver interactions are:*

- *Set or change the value of a variable not yet set by the user*
- *Remove or add a value to a variable domain*
- *Create or delete a configuration object because of resource demands by the user (such as the number of seats)*
- *Explain a conflict in variable values*
- *Propose alternative solutions to a conflict*
- *Automatically complete a partial configuration (autocompletion)*

Many of the solver interactions mentioned above must distinguish whether a configuration parameter has no value or a default value, or whether it has been set by the user or by the solver, because user-set values are typically not allowed to be overwritten by the solver.

With the current rise of data analytics and machine learning techniques and tools, additional types of solver interactions will become state-of-the-art in the future, like recommendation of input values learned from previous configuration sessions or the support of group decisions.

```
User action 1:
  Set nr_passengers = 160
Solver changes:
  domain(length_mm) = [13334,20000]
  domain(nr_seats) = [0,40]
  domain(standing_room) = [120,160]
  create handrail with
    type = standard

User action 2:
  Set nr_seats = 30
Solver changes:
  domain(length_mm) = [18334,20000]
  standing_room = 130
  create 30 seats with
    type = standard
    color = blue

User action 3:
  Set standing_room = 140
Solver proposes alternative conflict resolutions:
  1. nr_passengers = 170
  2. nr_seats = 20

User action 4:
  Unset nr_seats (accept proposal 2)
Solver changes:
  domain(length_mm) = [16667,20000]
  nr_seats = 20
  delete 10 seats

User action 5:
  Set first seat's type = special
Solver changes:
  first seat's color = red

User action 6:
  Autocomplete (with optimization)
Solver changes:
  length_mm = 16667 (maximize load factor)

User action 7:
  Set handrail's type = premium
Solver changes:
  for all seats except first, type = premium
```

Figure 3. Example of a configuration dialog for the configuration problem in Section 2

Definition (Interactive Configuration)**.** *An interactive configuration is an alternating sequence of user and solver interactions.*

An example of a typical configuration dialog between user and configurator is shown in Figure 3.

Having set up the definition of interactive configuration, we can sharpen our research question: *To what extent do existing solver frameworks support the different types of solver interactions in interactive configuration?* This question is relevant both for companies which sell configurable products/solutions and for solver tool providers (which often originate from academia). Product vendors want to use solvers which cover as many requirements as possible to avoid cumbersome workarounds or proprietary implementations. Tool providers are interested in this question to better focus on those features of their tools that customers really need. Presently, solvers seem to concentrate mainly on optimizing the performance of search for a solution. An indicator for this is the high number of performance challenges and competitions, such as the MiniZinc Challenge[2] or the international SAT competitions.[3]

Our contribution in this paper is the definition of the main requirements on an interactive configurator and a survey to which extent existing, non-commercial solvers fulfill those requirements. However, we do not claim to be exhaustive, neither in requirements nor in evaluated tools.

4 REQUIREMENTS

In this section we summarize the most important requirements on a configuration API for constraint solvers with the aim to facilitate interactive product configuration.

4.1 Basic interactions

The most basic interaction between the front-end (UI) and the configuration API is to set or unset a configuration parameter. Usually, the configuration domain is modelled in an object-oriented way, where configuration parameters correspond to attributes of configuration objects. Creation and deletion of configuration objects and relationships between them are also fundamental user actions.

Requirement (SetAttributeOrAssociation)**.** *The user can set or change an attribute of a configuration object (e.g., the length of a Wagon) or an association link between two configuration objects. The constraint solver model must be updated to be in sync with the front-end model.*

Requirement (CreateOrDeleteConfigurationObject)**.** *The user can create or delete configuration objects (e.g., create an instance of Handrail). The constraint solver model must be updated to be in sync with the front-end model.*

Our assumption to use a 3rd-party constraint solver as underlying reasoning system makes a transformation from the domain model to the solver model necessary. Most constraint solvers are optimized in dealing with flat (i.e., not object-oriented) integer-valued variable domains. This mapping from the object model to the constraint model is often cumbersome. Especially the encoding of objects and links between objects is not trivial (for instance see [20]).

Requirement (ModelTransformation)**.** *The configuration API should support mapping between a high-level, object-oriented domain model to the constraint model.*

Requirement (ExpressivenessOfConstraintLanguage)**.** *Constraints are the central tool for expressing dependencies between configuration objects. The language of the constraint solver must be rich enough to allow the formulation of all necessary integrity/consistency/resource constraints of the configuration domain.*

4.2 Preferred and default solutions

Default values are an essential means for enhanced usability, because the user is not forced to type in each parameter. In practice, default values are also a means of recommendation, guiding the user to the most common configuration. We distinguish *static* and *dynamic* (or *computed*) default values.

[2] `https://www.minizinc.org/challenge.html`
[3] `https://www.satcompetition.org/`

Requirement (StaticDefaults)**.** *Static default values are the most common form with built-in support by database systems and programming languages. See, e.g., solver changes after user actions 1 and 2 in Figure 3.*

Requirement (DynamicDefaults)**.** *Dynamic default values can be computed by almost arbitrary functions, which may use as input other variables of the same configuration (similar to the constraint of Seat in Figure 2) as well as variable values from historic configurations (e.g., most popular value).*

Requirement (UnsetVariable)**.** *Support Undo and Unset (i.e., set to UNDEF) of an earlier user decision, as, for example, in user action 4 in Figure 3. The user interface shall support overriding the default value as well as reverting to the default value. A "revert to default" capability is essential for good usability, guiding the users back to the paved path from which they got lost.*

Requirement (SoftConstraints)**.** *An essential property of default values is that they can be overridden by solvers without loss of data. If user-provided input values are regarded as constraints, then default values are soft constraints.*

For example, given that the default color of a seat is blue, but special seats are only available in red, then the solver overrides the default color selection with red for all special seats – see user action 5 in Figure 3. Usually this would not be perceived as data loss. On the other hand, if color blue has been chosen by a deliberate user action, and the seat type is changed to "special", then the configuration is inconsistent. In this case, the user must be notified and prompted for a corrective action as discussed in Section 4.5 – the color must not be changed without prior confirmation by the user.

4.3 Filtering

In an interactive configuration session, a user may be faced with many possible choices. For example, the number of configuration parameters for which the user can choose a value can be very high, and the number of possible values that can be chosen for a given parameter can also be very high. A user might consecutively set several parameters to values which at the end, possibly after several further steps of interaction between solver and user, do not allow a valid solution compatible to all those choices. The goal of *filtering* is to offer to the user as few alternatives as possible which do not lead to a valid solution.

Filtering can be used to grey-out values in the UI that are inconsistent with already user-set values (or communicate to the user in another proper way that they are infeasible) so that the user cannot choose them and end up in an invalid solution, e.g., the domains after user actions 1 and 2 in Figure 3.

Hiding those inconsistent values completely from the user, e.g., not showing values < 13334 for `length_mm` after user action 2 in Figure 3, is often not a good option as it reduces information and flexibility (i.e., the possibility to change decisions) of the user too much because the user would not know that he/she can set `length_mm` to 12000, for example (see Section 4.5 for more details).

Requirement (Filtering)**.** *The solver shall be able to filter invalid values from domains (up to a certain degree of consistency) and to return the current domain of a given variable on request.*

The challenge is to efficiently find all inconsistent values and show those for variables in the window that the user currently sees, even for complicated constraints. Besides the computational performance (the problem is NP-hard in general), it may also be difficult to present them to the user in a clear and understandable way – e.g., for an integer number variable show all even values greater than 100 except 2000.

Filtering, which is also found as propagation or constraint inference in the literature, is one of the techniques which constraint solving uses to tackle its underlying computational complexity. While search explores a solution space opened up by variables for which several possible value assignments exist, propagation deals with deterministic assignments that are forced by constraints. It may be interleaved with search or done as a preprocessing step [2, 19].

Constraint propagation algorithms usually assume the domains of the variables to be finite. They also assume constraints to be *binary*, i.e., to involve exactly two variables. However, all n-ary constraints can be transformed to binary ones [2, 19]. If a CSP (in which some domains may already have been tightened down) can be extended to a solution, it is called *globally consistent*. Since global consistency is very expensive to maintain, various forms of local consistency – mainly *arc consistency* – are used in practice [2, 19]. Another approach is to compile a CSP into a data structure that can maintain global consistency, such as an automaton [1] or a binary decision diagram (BDD) [15].

4.4 Solving

After having decided values for "important" variables, a user expects the system to automatically complete the partial solution (i.e., the user-set values) to a valid solution. As an alternative, the commercial configurator Tacton CPQ[4] offers a complete solution to the user all the time. In general, the following user actions shall be supported by the API of a constraint solver.

Requirement (ValidSolution)**.** *Computation of a valid solution for the current state of the UI, i.e., user-set values are to be treated as fixed. The implementation of this requirement can also be used for checking whether there is a valid solution or not.*

Requirement (OptimalSolution)**.** *Computation of an optimal solution, preferably with multiple optimization criteria.*

User action 6 in Figure 3 is an example for a simple objective function: maximizing the load factor minimizes the `length_mm` if the `nr_passengers` is already set.

Requirement (NextSolution)**.** *Computation of the next solution: In the case of Requirement (ValidSolution), this should be noticeably distinct from previous solutions so that the user gets the full bandwidth of potential solutions. In the case of Requirement (OptimalSolution), it will be equally good as the previous solutions or – for multiple optimization criteria – another instance in the Pareto front.*

ClaferMOO Visualizer[5] is an example for an optimizer which supports handling the Pareto front.

Requirement (IncrementalSolving)**.** *In an interactive configuration scenario, the user sets one variable after the other – without a predefined order. Performance might be increased if the solver supported incremental solving, i.e., continue with the latest state and not start from scratch after each user input.*

[4] `http://www.tacton.com/tacton-cpq/`
[5] `http://www.clafer.org/p/software.html`

4.5 Explanation

Unless the solver maintains global consistency after each user interaction (which is too expensive especially when fast response times are required, cf. Section 4.3), users can reach a dead end, i.e., a state where they have to revise a decision to be able to find a solution. It may also happen that a conflict is produced because the user deliberately sets a value that has already been filtered away by the solver, like in user action 3 in Figure 3.

The goal of explanation is to assist the user in this situation by suggesting previously made choices to be undone. For a good user experience it is important that these suggestions are understandable, i.e., violated constraints should be explained by descriptive prose.

Requirement (Explanation)**.** *The solver shall be able to explain in understandable terms why a current state cannot be extended to a valid solution.*

Usually we distinguish between background constraints and user constraints. Background constraints originate from the problem specification, cannot be changed and are assumed to be correct. Only user constraints, i.e., constraints originating from user interactions, can be part of explanations. A subset of user constraints is a *conflict* if the problem obtained by combining this subset with background constraints has no solution. There might be an exponential number of conflicts that explain the inconsistency of an over-constrained problem. For this reason, QuickXPlain [11] can be used to compute preferred explanations. Conflict diagnosis like QuickXPlain can be combined with a hitting-set algorithm to compute minimal *diagnoses*, i.e., minimal sets of faulty constraints. A diagnosis is a subset of user constraints so that the problem becomes consistent when this subset is removed from it [5].

While methods like QuickXPlain focus on conflicts, i.e., on what has gone wrong, *corrective explanations* have been proposed to focus on how to proceed in interactive solving towards a valid solution. A corrective explanation is more than a diagnosis in the sense that it does not just point out constraints (i.e., assignments of values to variables) that have to be retracted, but also proposes alternative assignments that guarantee to yield a solution [16].

Requirement (CorrectiveExplanation)**.** *The solver shall be able to suggest user actions suitable to correct a current state that cannot be extended to a valid solution.*

4.6 Integrability

Today's typical enterprise IT landscape is a system of systems with many internal and external interfaces. It must be possible to integrate a newly developed configurator into the existing infrastructure in order to be accepted and used. While the configurator will use other components such as a generic solver or a persistence service to fulfil its tasks, it will also itself be invoked by other components. In order to get widely adopted, a constraint solver should help companies to meet the challenges of integration, or at least it should not impose new ones.

Requirement (LanguageAPI)**.** *The constraint solver shall provide a well-documented API to a programming language that is widely accepted and used by companies, such as Java.*

Command-line interfaces to the solver are appreciated for testing, but using them for integration is a potential source of problems for integration and long-term maintenance. For this reason, a command-line interface alone is not deemed sufficient.

Requirement (TestSupport). *The constraint solver shall support debugging and automated testing.*

This can be implemented by compatibility with existing debuggers and testing tools of the host language as well as tool-specific facilities.

Requirement (MaintainanceSupport). *The constraint solver shall support long-term maintenance of the configurator.*

While the exact requirements cannot be anticipated, contributing factors could be:

- type-safe API language
- portable, standardized language for implementation and API
- use of standard and well-established tools

Requirement (NotificationAPI). *The constraint solver API shall support a notification concept, such as listeners or callbacks. Given the change of one input variable, which output variables change their values? What previously violated constraints are now fulfilled and vice versa?*

Requirement (ConstraintLanguage). *The constraint solver shall support a well-established language for constraint definition, such as MiniZinc or XCSP.*

This requirement is not a replacement but complements Requirement (LanguageAPI).

Requirement (LicenseCompatibility). *The license model shall be friendly in order to encourage industrial usage.*

Companies are reluctant to adopt open-source components with "sticky" licenses such as GPL, because they fear the legal risks imposed on the own intellectual property. A commercial license at a fair price is more likely to be accepted than a sticky open-source license. In general, industry-friendly licenses such as BSD and MIT style licenses will be appreciated.

Note that this also applies to the components depending on the solver: a dependency on a 3rd-party component with restrictive license conditions is very likely to inhibit the adoption of the solver. As a rule of thumb, all dependencies shall have the same or a more liberal license than the constraint solver component.

Requirement (MinimumDependencies). *The list of dependencies (libraries but also other resources) of the solver shall be as short as possible, because each added dependency is also regarded as an additional burden in terms of version management, license and security assessment, and long-term maintenance.*

In this section we concentrated on the integration of a constraint solver into a product configurator application via a configuration API. There are many other aspects for integration of configurators, e.g., interfaces to customer relation management (CRM), product data management (PDM), enterprise resource planning (ERP) systems and the like, or generation of quotations and other documents. Despite being considered more important by product managers and consuming much more effort in configurator solutions than product modeling itself, such aspects are out of scope of this work.

5 SURVEY OF EXISTING SOLVERS

In this chapter we will investigate how some constraint systems satisfy our proposed requirements for an interactive configuration API. To make our findings easily reproducible we have chosen freely available systems, each representing a different constraint solving paradigm: One offers a standardized constraint modelling language, one is propagation-based, one is a representative of constraint logic programming, and one is SAT-based. All systems allow the definition of classical static constraint problems with variables and constraints and provide basic solving capability (solving, optimization), but they differ in their support for interactive solving. We use a simplified version of the running example to illustrate the different ways to realize user inputs and domain filtering.

We are aware that there are integrated configuration systems on the market that fulfil the requirements posed in this paper at least partially. However, as already mentioned in Section 1, we focus on pure constraint systems that can be integrated into other applications without bearing the risk of vendor-lock-in.

5.1 MiniZinc

MiniZinc is a solver-independent constraint modelling language. The MiniZinc system is free and open-source. It includes various command line tools and the MiniZinc IDE for editing and solving MiniZinc models. For solving, the high-level MiniZinc models are compiled into FlatZinc, which is a low-level standard for defining constraint problems supported by most current constraint solvers. One reason for this is the annual MiniZinc challenge, which is the most popular solving competition for constraint solvers. As MiniZinc is a de-facto standard for representing constraint problems, it is a possible solution for a system satisfying Requirement (ConstraintLanguage).

Currently there is no way to directly access the domain filtering capabilities of a solver from MiniZinc. However, users can use solving to emulate domain filtering. For example, to compute the bounds of a variable domain the optimization statement of the original Listing 1 can be replaced with the following solve statements to determine the lower and upper bounds of a variable.

```
# determine lower bound
solve minimize standing_room;

# determine upper bound
solve maximize standing_room;
```

MiniZinc does not provide default reasoning out of the box, but it can be implemented with optimization or special heuristics all of which can be expressed in MiniZinc. Another option would be to use a system like MiniBrass[6] which adds soft constraints and preferences to the MiniZinc system.

To integrate MiniZinc into a software system, various approaches are possible. One such approach consists of manipulating the MiniZinc files programmatically and call the various tools (MiniZinc to FlatZinc compiler, solver executable) via system calls and standard I/O. A more efficient way for manipulating MiniZinc models is using libraries like JMiniZinc[7] or PyMzn.[8]

Another integration option would be to compile the MiniZinc file to FlatZinc and use the FlatZinc parser of the used solver. This has the benefit that solving can be controlled by the solver API, but the downside is that it is not always straightforward to map the low-level FlatZinc constraints and variables to that of the high-level MiniZinc model. This mapping is essential for interactive solving as the user interface must provide feedback in terms of the high-level constraints and variables.

[6] https://isse.uni-augsburg.de/software/minibrass
[7] https://github.com/siemens/JMiniZinc
[8] http://paolodragone.com/pymzn/

5.2 Choco

Choco is a well-established constraint library written in Java. It supports integer, boolean, set and real variables as well as basic constraint expressions and global constraints. Constraint problems are defined using Choco's Java API.

The following example shows how interactive configuration can be realized with Choco's API. A user input is simulated by posting a constraint containing the assignment selected by the user. The Choco constraint solver is based on constraint propagation [13]. Constraint propagation can not only be utilized during solving, but also to compute the current domains of the constraint variables. The example shows how variable domains are narrowed down by propagation after calling $Solver.propagate()$, i.e., Choco satisfies Requirement (Filtering).

```
// Initial domains:
// nr_seats = {0..80}
// standing_room = {0..200}
// nr_passengers = {50..200}
// post constraint
m.arithm(nr_seats,"+", standing_room, "=",
     ↪ nr_passengers).post();

// User input
m.arithm(nr_seats, "=", 40).post();

// Domain filtering
m.getSolver().propagate();
// Updated domains
// nr_seats = 40
// standing_room = {10..160}
// nr_passengers = {50..200}
```

If the propagation engine encounters an inconsistency, a `ContradictionException` is raised. Choco provides an explanation engine based on [22] for generating explanations for contradictions found during solving. Unfortunately, the explanation engine is by default not active during propagation. As an alternative a solver-independent algorithm like QuickXPlain can be adapted for Choco.

Choco does not support default values out of the box. One lightweight implementation of default values is to use search strategies similar to the approach in [8].

Choco is implemented in Java and therefore easy to integrate into an enterprise IT landscape. The source code of Choco is available on GitHub[9] and pre-built libraries are available for Maven.[10] As Choco is open source, missing features can be easily added to the current code base. On the downside the implementation of the features often requires detailed knowledge of the API and must be maintained when the API changes considerably (which has occurred in the past).

5.3 Picat

Picat is a logic-based multi-paradigm language well-suited for solving constraint problems. Most that will be said about Picat and interactive constraint solving will also apply to other constraint logic programming systems.

In Picat, constraint programming support is added through the *cp* module. After importing the *cp* module, constraint variables can be declared with `VAR::DOMAIN`. Constraints can be formulated with variable expressions (using special operators preceded with #), predicates and global constraints like `all_different` etc. The following shows the Picat implementation of the aforementioned example. In Picat every additional constraint expression triggers constraint propagation and the variable domains are adapted accordingly.

[9] `https://github.com/chocoteam/choco-solver`
[10] `https://search.maven.org/artifact/org.choco-solver/choco-solver/`

```
Picat> NR_SEATS::0..80,
  STANDING_ROOM::0..200,
  NR_PASSENGERS::50..200,
  NR_SEATS + STANDING_ROOM #= NR_PASSENGERS,
  NR_SEATS #= 40.
// output:
// NR_SEATS = 40
// STANDING_ROOM = DV_015c90_10..160
// NR_PASSENGERS = DV_015d28_50..200
```

Using the concept of action rules, it is possible to get notified of domain changes of constraint variables. Action rules have the form `Head, Cond, {Event} => Body`. Examples for events are:

- `ins(X)`, when a variable gets instantiated
- `bound(X)`, when the bounds of a variable change
- `dom(X,T)`, when a value T gets excluded from the domain of X

Therefore it is possible to trace all variable domains that have been effected by a user interaction, cf. Requirement (NotificationAPI).

Picat programs can be executed as shell scripts. Picat allows to define predicates by C functions. Unfortunately, it is currently not possible to call Picat from C, so integration must be done via standard I/O.

5.4 CP-SAT Solver

As an example of a constraint solver based on a non-constraint propagation paradigm we have chosen the SAT-based Google CP-SAT Solver, which is part of Google OR-Tools.[11]

As the CP-SAT Solver is SAT-based, it does not provide an API for accessing the current domain of variables. Therefore it is best treated as a black-box solver, i.e., by defining the constraint problem and solving it. Domain filtering can be simulated by calling the solver for a specific domain value or, as in the example below for bounded domains, calling minimize/maximize for a variable to compute its lower and upper bound. In the example the lower bound of the variable `standing_room` is computed. Of course this strategy only works for constraint problems where efficient solving is possible, otherwise the response time of the interactive system would deteriorate.

```
from ortools.sat.python import cp_model
model = cp_model.CpModel()
nr_seats = model.NewIntVar(0,80,"")
standing_room = model.NewIntVar(0,200,"")
nr_passengers = model.NewIntVar(50,200,"")

model.Add(sum([nr_seats,standing_room])==nr_passengers)
ui = model.NewIntVar(40,40,"")
model.Add(nr_seats == ui)

# Simulate domain filtering:
# Calling minimize for a variable
# finds the lower bound

model.Minimize(standing_room)
solver = cp_model.CpSolver()
status = solver.Solve(model)
print(solver.Value(standing_room))
# Output: 10
```

The CP-SAT solver is written in C++, but via SWIG[12] also an API in Java, C# and Python is provided, which makes it one of the few constraint solvers available in Python. Regarding Requirement (LanguageAPI) this provides a very good integrability for the basic functionality of the solver, although some special features can only be accessed through the C++ API.

[11] `https://github.com/google/or-tools/`
[12] `http://www.swig.org/`

5.5 Preliminary findings

This brief survey of currently available constraint solvers is by no means complete. Its main purpose was to illustrate the diversity of systems available for constraint solving and how each has particular strengths and weaknesses when it comes to satisfying the proposed requirements for interactive solving.

The focus of most constraint systems is on modelling and efficient solving. Only few solvers contain special features for interactive solving. Even in the case of solvers based on constraint propagation, the main purpose of constraint propagation is to assist the solver during search. Another indicator that features like domain filtering are sometimes less important than performance is the fact that Google recommends to use the new SAT-based constraint solver instead of the legacy constraint propagation based solver contained in Google's OR-Tools due to performance.[13] Only one system (Picat) provides a documented mechanism how to get notified about domain changes of constraints variables. Some solvers like Choco include an explanation engine. None of the solvers we are aware of supports default reasoning out of the box. All in all this supports our initial suspicion that interactive solving is a neglected aspect of constraint systems.

Of course, the decision which solver to use is complex and will not be driven by interactive features alone. Most of the interactive features can be implemented (although sometimes less efficiently) by treating the solver as a black box.

As can be seen even in our simple example, it is not trivial to translate a constraint problem from one solver to another. To avoid the risk of getting locked-in to a specific solver API, it might be better to use a solver-independent constraint modelling language like MiniZinc.

6 CONCLUSION

Real-world configuration problems are often interactive in nature, i.e., they include the user as an essential factor in the configuration process. To solve configuration problems, constraint satisfaction is often used as an underlying reasoning system. In this paper, we have made two contributions: First, we have proposed a set of requirements that a constraint solver should fulfil to support interactive configuration. Second, we have presented the results of a small survey covering a selected set of non-commercial off-the-shelf constraint systems.

Our findings show that interactive aspects of configuration are neglected by most constraint systems. They also show that the landscape of available system is highly diverse and that each solver has its own strengths and weaknesses when it comes to satisfying the requirements proposed in this work.

This is preliminary work. Neither our list of user and solver interactions nor our list of requirements nor our survey is exhaustive. We focused here on those requirements and tools that we experienced as most important in our daily work on industrial product configuration. Future work should extend this study to cover more requirements (like aspects of guided selling, prediction of default values, recommender features) and more constraint systems. Assessment of each constraint system shall be done in a more systematic and complete way and summarized in tables.

We invite the configuration and constraints communities to propose implementations of constraint systems that are suitable to support interactive configuration.

[13] `https://developers.google.com/optimization/cp/cp_solver`

REFERENCES

[1] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis, 'Consistency restoration and explanations in dynamic csps application to configuration', *Artif. Intell.*, **135**(1-2), 199–234, (2002).

[2] Christian Bessiere, 'Constraint propagation', In Rossi et al. [18], 29–83.

[3] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker, 'Formalizing cardinality-based feature models and their specialization', *Software Process: Improvement and Practice*, **10**(1), 7–29, (2005).

[4] Andreas A. Falkner, Gerhard Friedrich, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, 'Twenty-five years of successful application of constraint technologies at Siemens', *AI Magazine*, **37**(4), 67–80, (2016).

[5] Alexander Felfernig, Monika Schubert, and Christoph Zehentner, 'An efficient diagnosis algorithm for inconsistent constraint sets', *AI EDAM*, **26**(1), 53–62, (2012).

[6] David Angelo Ferrucci, *Interactive configuration: a logic programming-based approach*, Ph.D. dissertation, Rensselaer Polytechnic Institute Troy, NY, USA, 1994.

[7] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner, 'Configuring large systems using generative constraint satisfaction', *IEEE Intelligent Systems*, **13**(4), 59–68, (1998).

[8] Alois Haselböck and Gottfried Schenner, 'A heuristic, replay-based approach for reconfiguration', in *Proceedings of the 17th International Configuration Workshop, Vienna, Austria, September 10-11, 2015.*, eds., Juha Tiihonen, Andreas A. Falkner, and Tomas Axling, volume 1453 of *CEUR Workshop Proceedings*, pp. 73–80. CEUR-WS.org, (2015).

[9] Pieter Van Hertum, Ingmar Dasseville, Gerda Janssens, and Marc Denecker, 'The KB paradigm and its application to interactive configuration', *TPLP*, **17**(1), 91–117, (2017).

[10] Lothar Hotz, Thorsten Krebs, and Katharina Wolter, 'Combining software product lines and structure-based configuration – methods and experiences', in *Workshop on Software Variability Management for Product Derivation – Towards Tool Support*, (2014).

[11] Ulrich Junker, 'QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems', in *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, eds., Deborah L. McGuinness and George Ferguson, pp. 167–172. AAAI Press / The MIT Press, (2004).

[12] Ulrich Junker, 'Configuration', In Rossi et al. [18], 837–873.

[13] Narendra Jussien and Olivier Lhomme, 'Unifying search algorithms for CSP', Rapport technique, École des Mines de Nantes, (2002).

[14] Jeppe Nejsum Madsen, *Methods for Interactive Constraint Satisfaction*, Master's thesis, Department of Computer Science, University of Copenhagen, 2003.

[15] Andreas Hau Nørgaard, Morten Riiskjær Boysen, Rune Møller Jensen, and Peter Tiedemann, 'Combining binary decision diagrams and backtracking search for scalable backtrack-free interactive product configuration', In Stumptner and Albert [21], pp. 31–38.

[16] Barry O'Callaghan, Barry O'Sullivan, and Eugene C. Freuder, 'Generating corrective explanations for interactive constraint satisfaction', in *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, ed., Peter van Beek, volume 3709 of *Lecture Notes in Computer Science*, pp. 445–459. Springer, (2005).

[17] Matthieu Stéphane Benoit Queva, Christian W. Probst, and Per Vikkelsøe, 'Industrial requirements for interactive product configurators', In Stumptner and Albert [21], pp. 39–46.

[18] *Handbook of Constraint Programming*, eds., Francesca Rossi, Peter van Beek, and Toby Walsh, volume 2 of *Foundations of Artificial Intelligence*, Elsevier, 2006.

[19] Stuart J. Russell and Peter Norvig, *Artificial Intelligence - A Modern Approach (3. internat. ed.)*, Pearson Education, 2010.

[20] Gottfried Schenner and Richard Taupe, 'Encoding object-oriented models in MiniZinc', in *Fifteenth International Workshop on Constraint Modelling and Reformulation*, (2016).

[21] Markus Stumptner and Patrick Albert, eds. *Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09)*, 2009.

[22] Michael Veksler and Ofer Strichman, 'A proof-producing CSP solver', in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, eds., Maria Fox and David Poole. AAAI Press, (2010).