Journal of Universal Computer Science, vol. 26, no. 3 (2020), 343-373
submitted: 1/10/19, accepted: 15/3/20, appeared: 28/3/20

Solver Requirements for Interactive Configuration[1]

Andreas Falkner, Alois Haselböck, Gerfried Krames
Gottfried Schenner, Herwig Schreiner, Richard Taupe
(Siemens AG Österreich, Corporate Technology, Vienna, Austria
{andreas.a.falkner, alois.haselboeck, gerfried.krames, gottfried.schenner,
herwig.schreiner, richard.taupe}@siemens.com)

Abstract: Interactive configuration includes the user as an essential factor in the configuration process. The two main components of an interactive configurator are a user interface at the front-end and a knowledge representation and reasoning (KRR) framework at the back-end. In this paper we discuss important requirements for the underlying KRR system to support an interactive configuration process. Representative of many reasoning systems and tools used for implementing product configurators, we selected MiniZinc, Choco, Potassco, Picat, CP-SAT solver, and Z3 for evaluation and reviewed them against the identified requirements. We observe that many of those requirements are not well supported by existing stand-alone solvers.

Key Words: interactive configuration, product configuration, constraint programming, knowledge representation and reasoning, user interface
Category: I.2.4, F.4.1, H.5.2

1 Introduction

With the significant rise of individualized products over the last few years, product configuration has strengthened its role as an important business process [Felfernig et al., 2014]. State-of-the-art configurators are tools that support this process using knowledge representation and reasoning (KRR) technologies such as constraint satisfaction [Junker, 2006]. Configuration is usually an interactive task, iteratively involving two parties: 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 needs and requirements. The configurator is a digital companion that supports the configuration process by deriving consequences of the user's choices and by helping to avoid or resolve conflicts. In this paper, we focus on configurators where a tailor-made user interface (UI) or legacy system is enhanced with solving capabilities, i.e. by calling a general solver as a component.[2]

[Fig. 1] gives an overview of the typical architecture and scenario: At the front-end, a **user** interacts with the **configurator** via a UI. The **solver** is avail-

[1] This is an extended version of [Falkner et al., 2019]. Author names are ordered alphabetically.

[2] We neglect the alternative of implementing a special UI for an integrated configuration system such as a commercial CPQ tool or sales configurator suite, as this bears high risk of vendor-lock-in.

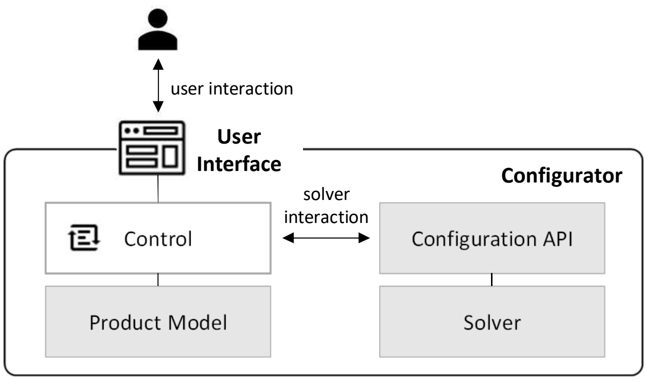

Figure 1: Components of an interactive configurator

able as a stand-alone library or program (open-source or commercial) and has a general API, and potentially an API extension, for solving configuration problems in particular. Together with a domain-specific **product model** (or knowledge base, KB), it forms the KRR component. This back-end is invoked 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's results back to the UI. While other back-end technologies, such as recommender systems, could facilitate interactive solving as well, in this article we focus on solvers for declarative knowledge bases.

The configurator helps the user to configure a product according to his/her needs and in full compliance with the product model. The user expects the configurator to show in a clear and concise way what decisions are necessary, to highlight or preset the "best" alternative (value), to filter or grey-out infeasible values, to recommend alternatives in case of conflicts, and to respond quickly (preferably instantaneously) to user inputs. Of course, the user should also be able to withdraw his/her decision, which corresponds to unsetting a configuration parameter.

In most non-trivial configuration problems, a dynamic number of configuration objects plays an important role [Falkner et al., 2016]. This requires a different kind of user interaction: the creation and deletion of configuration objects. 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 allowing the configuration tool to create the individual objects. However, in both cases the result is a set of configuration objects whose number was not known beforehand, and whose properties can be configured further.

User interactions are: (i) create configuration object, (ii) delete configuration object, (iii) set/unset configuration object attribute, (iv) set/unset association between configuration objects.

User interactions change the state of the configuration by making decisions. Solver interactions make implicit knowledge explicit to assist the user, e.g. via domain filtering or rules which set attributes depending on values of other attributes. Such knowledge is implicitly contained in the configuration model of the problem and the solver makes its consequences explicit, thus creating a distinct benefit to the user.

A **solver interaction** is a set of consequences following a user interaction. Typical solver interactions are: (i) set or change the value of a variable not yet set by the user, (ii) add or remove a value to/from a variable domain ("domain filtering"), (iii) create or delete a configuration object because of resource demands by the user (such as the number of seats), (iv) explain a conflict between values of several variables, (v) propose alternative solutions to a conflict, (vi) 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.[3]

An **interactive configuration** is an alternating sequence of user and solver interactions. In this paper, we formalize those interactions as requirements on a functional interface (Configuration API) for solvers and evaluate to which extent existing, stand-alone solvers fulfil those requirements and thus support interactive configuration. This work is relevant for all involved parties: Product vendors, i.e. companies which sell configurable products or solutions, want to use solvers which cover as many requirements as possible to avoid cumbersome workarounds or proprietary implementations. Configurator system companies may improve their offerings and internal architecture. Solver providers can focus on those features of their tools that configuration customers really need.

Being aware that there are many different scenarios for using configurators, we do not see our work, which may be somewhat biased by personal experiences and opinions, as a complete and fully methodical evaluation, but as the first step of an incremental process to evaluate and improve tooling for interactive product configuration that will hopefully be continued.

The remainder of this paper is organized as follows: [Section 2] discusses related work. In [Section 3], an example for interactive configuration is introduced by means of a model and an interactive process. We define the requirements for

[3] 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.

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 Related Work

In his PhD thesis, [Ferrucci, 1994] 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."*

[Hertum et al., 2017] studied 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 [Section 4] of this paper: 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.

[Queva et al., 2009] describe 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, [Madsen, 2003] identified three fundamental interactivity operations in his master thesis: add constraint, remove constraint, and restoration. Other approaches to interactive configuration 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 [Hotz et al., 2014].

In product configuration often the product model language and the language of the used solver differ. [Schneeweiss and Hofstedt, 2011] show how to map a product model based on feature models to a constraint model and discuss the challenges of implementing interactive features like retracting a user decision. [Janota, 2010] gives definitions for user actions like "completing a configuration", "making a choice for the user" for interactive configuration in combination with a SAT solver.

[Pleuss et al., 2011] focus on visualisation aspects of interactive product configuration in the context of product line engineering.

3 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 [Fleischanderl et al., 1998] or cardinality-based feature modelling [Czarnecki et al., 2005]). The example comprises the main configuration challenges that we experience in real-world projects in domains such as power generation or rail automation [Falkner et al., 2016].

3.1 Product Definition

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 numbers are 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 occupies the 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).

3.2 Product Models

[Fig. 2] shows a UML class diagram for this sample specification, including pseudo code for all constraints.

A model for a standard constraint solver is much 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 depending on the expected load and length. The MiniZinc program in [Listing 1] is a full implementation of the example in a standard constraint language which can be executed by most constraint solvers but requires modelling knowledge to be understood.

In the example problem presented in [Section 3], `seats` are configuration objects whose number is not known beforehand, and where each `seat` can be

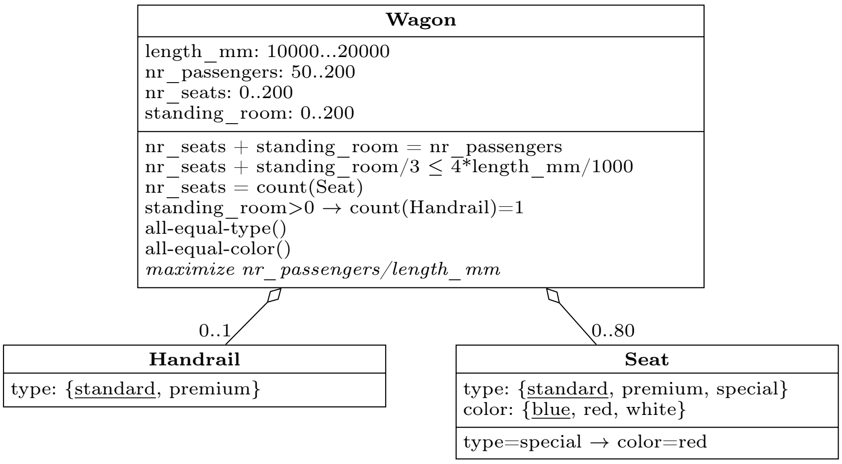

Figure 2: Class diagram of the Wagon example. Default values are underlined. `Wagon.all-equal-type()` stands for a constraint that all sub-parts must have the same `type` except for `special`. `Wagon.all-equal-color()` stands for a constraint that all associated `seats` (except if `type`=`special`) must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

configured individually (attributes `color` and `type`). 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 additional individual properties. Thus, standing room need not be modelled as configuration objects in our example.

3.3 User and Solver Interactions

An example of a typical configuration dialog between user and configurator is shown in [Fig. 3]. The solver responds to user actions and thus makes implicit knowledge explicit to assist the user, e.g. by domain filtering.

For instance, user action 1 in [Fig. 3] sets the `nr_passengers` to 160 and the solver changes the lower bound of `length_mm` from 10000 to 13334 because of the constraint `nr_seats + standing_room/3` $\leq$ `4*length_mm/1000` (for the case that `nr_seats` is 0). Analogously, for the case that `length_mm` is 20000, the upper bound of `nr_seats` must be not more than 40 in order to achieve the 160 passengers (and leads to a lower bound of 120 for `standing_room`).

In addition, the solver can set some values which are the only remaining valid alternatives, e.g. creating a handrail after user action 1 in [Fig. 3] because `standing_room/3` > 0 requires one. Similarly in user action 5, the user sets the attribute `type` of a seat to `special` and the solver automatically changes its attribute `color` to `red` because of the according constraint and because that `color` was not set explicitly by the user before.

Following user action 2, the solver creates the correct amount of seats and sets their attributes to the predefined default values.

```
% 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
```

Listing 1: MiniZinc program for the Wagon example

User action 3 causes a conflict and the solver proposes different ways to resolve it. The user chooses to take back an earlier decision by unsetting the corresponding parameter with user action 4 and the solver adjusts dependent values accordingly.

Autocompletion in user action 6 sets only `length_mm` to the minimal valid value (according to the optimization function). All the other parameters and necessary configuration objects were already set before.

```
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 last 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

4 Requirements

In this section we summarize the most important requirements on a functional configuration API for general solvers with the aim to facilitate interactive product configuration as defined in [Section 1]. Although they are only a systematic compilation of our personal opinions, they are based on experiences of configurator users in the domains of health care, production industry, power generation, and rail automation, and they were confirmed in discussions with the product configuration community, e.g., at the International Workshop on Configuration.

4.1 Modelling

Before using a general solver as underlying reasoning system for a configurator, a domain-specific product model must be defined. Many of the major commercial configurator frameworks offer some kind of object-based models, although none of them covers all aspects of object-orientation. A representation such as UML has advantages for specifying complex products: highly understandable, compact, easy to maintain.

However, most constraint and logic-based solvers are built to work on flat (i.e. not object-oriented) integer-valued variable domains. They concentrate on optimizing the performance of search for a solution – see the high number of performance challenges and competitions, such as the MiniZinc Challenge [Stuckey et al., 2014] or the international SAT competitions [Heule et al., 2018].

Therefore, a mapping from the object-oriented product model to the solver model is necessary, but this is often cumbersome. The encoding of objects and links between objects is especially tricky (for instance see [Schenner and Taupe, 2016]). Our requirement for solvers is not to support UML or object-oriented encodings directly, but that mapping from object-oriented models to the solver model is uncomplicated.

Requirement M1 (ModelTransformation). *Support mapping between a high-level, object-oriented product model and the solver-specific representation.*

Example from [Section 3]: Mapping from [Fig. 2] to [Listing 1].

Requirement M2 (ConfigurationObjects). *Allow the definition and specialization of classes (abstract types) for multiple configuration objects.*

Example from [Section 3]: `Wagon`, `Seat`, and `Handrail` are classes. For `Seat`, multiple objects are allowed. Specialization, i.e. inheritance, does not occur in the example.

Requirement M3 (AttributesAndAssociations). *Allow the definition of parameters (aka attributes, features, properties) of various types for each class as well as links (aka associations) with defined cardinalities (i.e. multiplicities) between configuration objects.*

Example from [Section 3]: Objects of type `Wagon` have 4 integer attributes, e.g. `length_mm`. `Handrail` has an enumeration attribute – `type`. The association between `Wagon` and `Seat` has a cardinality of exactly one on the `Wagon` side and allows up to 80 objects on the `Seat` side.

Requirement M4 (RulesAndConstraints). *Support constraints (or rules) to define dependencies between configuration objects and their parameters. The language of the solver must be rich enough to allow the formulation of all necessary integrity, consistency, and resource constraints of the configuration domain.*

Example from [Section 3]: The first two constraints of `Wagon` define dependencies between its integer attributes. The third one binds the attribute `nr_seats` to the cardinality of the association to `Seat`. The implication of the fourth constraint can be seen as a rule to create a handrail as soon as some standing room is configured.

Requirement M5 (StaticDefaults). *Allow the definition of a static default value (constant) for configuration parameters.*

Example from [Section 3]: The default value for `type` is `standard`, for `color` it is `blue`. See the solver changes after user actions 1 and 2 in [Fig. 3].

Requirement M6 (DynamicDefaults). *Allow the definition of dynamic default values – they can be computed by almost arbitrary functions which may use as input other variables of the same configuration as well as variable values from historic configurations (e.g. most popular value).*

Requirement M7 (SoftConstraints). *Support soft (aka weak) constraints which are not seen as hard requirements but as preferred alternatives. They can be used for setting dynamic default values and for optimization (set as many highly preferred values as possible).*

There are no soft constraints in the example in [Section 3], but see [Meseguer et al., 2006] for an overview.

4.2 Basic interactions

The most basic interactions between the front-end (UI) and the configuration API allow users to change the configuration (by setting/unsetting configuration parameters and creating/deleting configuration objects) – thus taking decisions towards their individualized product. All the time, the internal solver model must be kept synchronized with the front-end model.

Requirement B1 (CreateOrDeleteObject). *Allow the user to create or delete configuration objects.*

Example from [Section 3]: Manually create an instance of `Handrail`.

Requirement B2 (SetValue). *Allow the user to set or change the value of an attribute of a configuration object or an association link between two configuration objects. The solver model must be updated to be in sync with the front-end model.*

Example from [Section 3]: The user sets the number of passengers (attribute `nr_passengers` of a `Wagon`) as in user action 1 in [Fig. 3].

Requirement B3 (UndoDecision). *Support Undo and Unset (i.e. set to UNDEF) of an earlier user decision.*

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 path from which they strayed.

Example from [Section 3]: User action 4 in [Fig. 3].

Requirement B4 (Recommendations). *Recommend decisions to the user, e.g. by setting default values.*

Recommendation can be done by setting default values as defined in requirements **M5 (StaticDefaults)**, **M6 (DynamicDefaults)**, and **M7 (SoftConstraints)**, but also by using heuristics or a history of user actions. They are an essential means for enhanced usability, because the user is not forced to type in each parameter and the user is guided to the most common configuration. They can be overridden by user interaction or further solver decisions without loss of data.

Example from [Section 3]: Given that the default color of a seat is `blue` but `special` seats are only available in `red`, the solver overrides the default color selection with `red` for all `special` seats – see user action 5 in [Fig. 3]. Usually this would not be perceived as data loss. However, 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 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. The challenge is to find, as efficiently as possible, all consistent 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 all consistent values to the user in a clear and understandable way – e.g. for an integer variable show all even values greater than 100 except 2000.

Constraint solvers use two main approaches in interleaved steps: While search explores a solution space opened up by variables for which several possible value assignments exist, propagation (aka constraint inference) deals with deterministic assignments that are forced by constraints. If a constraint satisfaction problem (CSP), in which some domains may already have been reduced due to user actions, 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 [Bessiere, 2006, Russell and Norvig, 2010]. Another approach is to compile a CSP into a data structure that can maintain global consistency, such as an automaton [Amilhastre et al., 2002] or a binary decision diagram (BDD) [Nørgaard et al., 2009].

Filtering can be used to grey-out values in the UI that are inconsistent with values already user-set (or communicate the fact that they are infeasible to the user in another appropriate way) 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 [Fig. 3].[4]

Requirement F1 (CompleteFiltering). *Filter* ***all*** *invalid values from the domains (i.e. those which do not lead to a consistent solution) of all relevant parameters and return the current domain of a given variable on request.*

Example from [Section 3]: As an example see the definition of user interaction in [Section 3] and the configuration dialog in [Fig. 3] – e.g. the solver changes after user action 2.

Requirement F2 (FastFiltering). *Filter many of the invalid values from domains (e.g. up to a certain degree of consistency) and return the resulting domain of a given variable, suitable for interactive usage (e.g. answer within 100 msec).*

Requirement F3 (LocalFiltering). *Filter values from domains w.r.t. constraints in the local neighbourhood even if there are violated constraints in other parts of the system.*

4.4 Solving

After having set the values for "important" variables, the user expects the system to automatically complete the partial solution (i.e. the user-set values) to a valid solution.

[4] Hiding those inconsistent values completely from the user, e.g. not showing values < 13334 for `length_mm` after user action 2 in [Fig. 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 S1 (ExistsSolution). *Report whether there is at least one valid solution, given the values which the user set so far.*

Requirement **S2 (ValidSolution)** can be used to check whether there is a valid solution or not:

Requirement S2 (ValidSolution). *Compute a valid solution for the current state of the UI, i.e. user-set values are to be treated as fixed.*

As an alternative, the commercial configurator Tacton CPQ offers a complete solution to the user all the time [Orsvärn and Axling, 1999].

Requirement S3 (OptimalSolution). *Compute an optimal solution, preferably based upon multiple optimization criteria.*

Example from [Section 3]: User action 6 in [Fig. 3] is an example of a simple objective function: maximizing the load factor minimizes `length_mm` if `nr_passengers` is already set.

Requirement S4 (NextSolution). *Compute the next valid solution. In the case of Requirement S3, it must be at least as good as previous solutions or – for multiple optimization criteria – another instance in the Pareto front.*

For interactive configuration, we prefer the sequential computation of solutions over the computation of all solutions or the next N solutions for performance and utility reasons.

We prefer to show a user few but noticeably distinct alternatives instead of many similar solutions. In [Hebrard et al., 2005] and [Eiter et al., 2009], methods for finding diverse solution sets are described. ClaferMOO Visualizer is an example for an optimizer which supports the handling of the Pareto front [Murashkin et al., 2013].

Requirement S5 (IncrementalSolving). *In order to increase performance, support incremental solving, i.e. do not start from scratch after each user input but continue with the latest solver state.*

In an interactive configuration scenario, the user sets one variable after the other – without a predefined order. Typically, there is a sequence of decisions within one session.

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.

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 E1 (Explanation). *Explain in understandable terms why a current state cannot be extended to a valid solution.*

This can be achieved by model-based diagnosis (see chapter 7 of [Felfernig et al., 2014]) where the constraints from the problem specification are considered background knowledge and the user decisions are considered as additional constraints which can lead to conflicts. The first are assumed to be correct and only the latter can be part of explanations. A subset of user decisions 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 [Junker, 2004] can be used to compute preferred explanations.

Requirement E2 (CorrectiveExplanation). *Suggest user actions that correct a current state which cannot be extended to a valid solution.*

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 decisions so that the problem becomes consistent when this subset is removed from it [Felfernig et al., 2012]. A corrective explanation is more than a diagnosis in the sense that it does not just point out decisions (i.e. assignments of values to variables) that have to be retracted, but also proposes alternative assignments that guarantee to yield a solution [O'Callaghan et al., 2005].

Example from [Section 3]: After user action 3 in [Fig. 3], the solver suggests the only two reasonable (i.e. minimal) repair actions.

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. As the solver is invoked by or intermingled in the configurator, the solver provider must contribute to meet the challenges of integration, or at least it should not impose new ones.

In this section, we put our focus on the integration of a solver into a product configurator application via a configuration API. There are many other aspects of integrating 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 modelling itself, such aspects are out of scope here.

Requirement I1 (LanguageAPI). *Provide a well-documented API for a general-purpose programming language that is widely accepted and used in industry, e.g. Java or C#.*

Desirable properties of the chosen host language(s) comprise: (i) a standardized or de-facto industrial standard language, (ii) type-safety, (iii) portability, (iv) tool support (e.g. debugger, profiler).

Requirement I2 (NotificationAPI). *Support a notification concept, such as listeners or callbacks: After the change of an input variable, which output variables change their values and which constraints change their violation state?*

Requirement I3 (SpecificationLanguage). *Support a well-established, defacto standard language for model and problem specification (e.g. constraint definition), such as MiniZinc or XCSP.*

Command-line interfaces to the solver are appreciated for testing purposes, 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 sufficient.

Requirement I4 (TestSupport). *Support automated (regression) testing.*

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

Requirement I5 (DebugSupport). *Support finding bugs in configuration models.*

Requirement I6 (LicenseCompatibility). *Friendly license model, i.e. not hampering industrial usage – for the solver itself and all its third-party components.*

Companies are reluctant to adopt open-source components with "sticky" licenses such as GPL, because they fear the legal risks imposed on their 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 third-party component with restrictive license conditions is very likely to inhibit the adoption of the solver. All dependencies shall have a license as least as liberal as that of the main solver component.

Requirement I7 (MinimumDependencies). *Minimize the number of dependencies to third-party components.*

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, export and customs control, and long-term maintenance.

5 Survey of Existing Solvers

In this chapter we investigate how some selected solver systems satisfy our proposed requirements for an interactive configuration API. We selected the systems/tools for evaluation based on the following criteria: Does the system provide *basic solver functionality* suitable for the special needs of configuration applications? Is the system *prominent* in the sense of being known or used by a greater community? Is the system *alive* in the sense of further development, regular updates, and an active community? Is the system non-commercial? Is the system representative of a whole class of systems (e.g. constraint-based solvers, answer set solvers)?

Due to limited space, we had to restrict our investigation to the following 6 systems: **MiniZinc** [Section 5.1] is a prominent constraint system that mainly provides a front-end language and IDE, and that integrates third-party solvers (like Gecode) for solving. **Choco** [Section 5.2] is a well-known, classical, Java-based constraint system. A similar Java-based system is JaCoP [Kuchcinski and Szymanek, 2013]. **Potassco** [Section 5.3] with its integrated solver `clasp` is a typical representative of an answer set programming system. Another such tool is DLV [Alviano et al., 2017]. **Picat** [Section 5.4] is a representative of a constraint logic programming system. Another such system which is reasonably well-known is ECLiPSe [Apt and Wallace, 2007]. **CP-SAT Solver** [Section 5.5] is a part of the Google OR Toolkit. It is a typical constraint system that uses a SAT solver for solving and is quite successful in solver competitions. Finally, **Z3** [Section 5.6] is a prominent instance of an SMT solver.

We asked the authors/owners of the selected systems to review our ratings of their respective system and got some valuable comments which we integrated in this version of the article.

Some systems, though suitable for interactive configuration in principle, are not part of this study due to space restrictions, but are candidates for future work: feature Model systems like Clafer [Bak et al., 2016]; MILP solvers (mixed-integer linear programming) like lpsolve [Berkelaar, 2019]; constraint solvers based on MDD technology (multiple-valued decision diagram) like the commercial tool Configit [Møller et al., 2001]; planning tools like OptaPlanner [De Smet and open source contributors, 2006], another system quite prominent and potent because of its focus on planning and scheduling, but not on configuration.

The following sections contain a short description of each selected system and our ratings to what degree each requirement of [Section 4] is supported by the respective system. We use the following simple rating scale:

++ : The system natively supports the requirement or there is an easy, straightforward workaround.

+ : The system does not natively support the requirement, but there is a manageable workaround.

0 : The system does not natively support the requirement, and there is no workaround at all or a workaround would be disproportionately elaborate.

Even though we use this rating scale to quantify the extent to which various solvers meet our requirements, this evaluation is not about comparing tools against each other. Our aim is to investigate how the field of KRR solvers supports interactive configuration in general and how this situation could be improved.

5.1 MiniZinc

MiniZinc [Nethercote et al., 2007] 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. [Listing 1] shows a MiniZinc encoding of the example in [Section 3].

Typically, you run MiniZinc via its IDE. Various approaches allow to integrate MiniZinc into a software system. One such approach consists of manipulating the MiniZinc files programmatically and call various tools (MiniZinc to FlatZinc compiler, solver executable) via system calls and standard I/O. A more efficient way to manipulate MiniZinc models is using libraries like JMiniZinc,[5] PyMzn,[6] or iminizinc.[7] Another integration option is to compile the MiniZinc file

[5] `https://github.com/siemens/JMiniZinc`
[6] `http://paolodragone.com/pymzn/`
[7] `https://github.com/MiniZinc/iminizinc`

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 that it is not always straightforward to map the low-level FlatZinc constraints and variables back to those 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.

[Tab. 1] shows our requirements evaluation for the tool MiniZinc.

5.2 Choco-solver

Choco-solver [Prud'homme et al., 2017] 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 [Jussien and Lhomme, 2002]. 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-solver satisfies Requirement **F2 (FastFiltering)**.

```
// 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}
```

Choco is implemented in Java and therefore easy to integrate into an enterprise IT landscape. The source code of Choco is available on GitHub[8] and pre-built libraries are available for Maven.[9] As Choco is open source, missing features can easily be 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 if the API changes considerably (which has occurred in the past).

[Tab. 2] shows our requirements evaluation for the system Choco.

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

Requirement	Rating	Comment
M1 ModelTransformation	+	No native support of object-oriented models, but it is possible to encode objects using arrays of constraint variables.
M2 ConfigurationObjects	+	Cf. M1.
M3 AttrsAndAssocs	+	No native support of object attributes/associations as objects, but all basic data types supported, with a focus on integers.
M4 RulesAndConstraints	++	All standard constraint types and global constraints are supported. As a system intrinsically based on constraints, rules are not supported.
M5 StaticDefaults	0	Cannot be realized, because no default reasoning and no distinction between user-set and tool-/initial-set variables supported.
M6 DynamicDefaults	0	Cf. M5.
M7 SoftConstraints	+	Natively, only hard constraints are supported. By reification in combination with optimization, soft constraints could be simulated. Another option is MiniBrass [Schiendorfer et al., 2018], which adds soft constraints and preferences to MiniZinc.
B1 CreateOrDeleteObject	+	Incremental addition/removal of variables not supported; when config. objects are created/removed, the constraint model must be rebuilt.
B2 SetValue	++	Supported.
B3 UndoDecision	0	Not supported.
B4 Recommendations	0	Not supported.
F1 CompleteFiltering	+	Not natively supported; can be emulated by solver call for each variable value.
F2 FastFiltering	0	Currently there is no way to directly access the domain filtering capabilities of a solver from MiniZinc.
F3 LocalFiltering	0	In case of an inconsistent situation, MiniZinc solvers stop further propagation.
S1 ExistsSolution	++	Supported.
S2 ValidSolution	++	Supported.
S3 OptimalSolution	++	Supported.
S4 NextSolution	0	Not supported; only solving for 1, N or all solutions supported.
S5 IncrementalSolving	0	Not supported.
E1 Explanation	0	Not supported.
E2 CorrectiveExplanation	0	Not supported.
I1 LanguageAPI	+	No native API in a standard programming language, but several options to integrate in Java or Python.
I2 NotificationAPI	0	No event mechanism on solver decisions.
I3 SpecLanguage	++	MiniZinc and FlatZinc are de-facto standards for representing constraint problems.
I4 TestSupport	0	Not supported.
I5 DebugSupport	0	Not supported.
I6 LicenseCompatibility	++	Creative Commons Attribution-NoDerivatives 4.0 International License.
I7 MinimumDependencies	+	Beside some required libraries, at least one solver backend must be installed.

Table 1: Requirements evaluation table for MiniZinc.

Requirement	**Rating**	**Comment**
M1 ModelTransformation	+	No native support of object-oriented models, but it is possible to encode objects using arrays of constraint variables.
M2 ConfigurationObjects	+	Cf. M1.
M3 AttrsAndAssocs	+	No native support of object attributes/associations as objects, but all basic data types supported, with a focus on integers.
M4 RulesAndConstraints	++	All standard constraint types and global constraints are supported. As a system intrinsically based on constraints, rules are not supported.
M5 StaticDefaults	0	Cannot be realized, because no default reasoning and no distinction between user-set and tool-/initial-set variables supported.
M6 DynamicDefaults	0	Cf. M5.
M7 SoftConstraints	+	Natively, only hard constraints are supported. By reification in combination with optimization, soft constraints can be simulated.
B1 CreateOrDeleteObject	+	Incremental removal of variables not supported; when configuration objects are removed, the constraint model must be rebuilt.
B2 SetValue	++	Supported.
B3 UndoDecision	+	Not natively supported, but implementation can use DecisionPath and Decision objects (not documented in the user manual).
B4 Recommendations	0	Not supported.
F1 CompleteFiltering	+	Not natively supported; can be emulated by solver call for each variable value.
F2 FastFiltering	++	Supported.
F3 LocalFiltering	0	In case of an inconsistent situation, the Choco solver throws a contradiction exception on propagation.
S1 ExistsSolution	++	Supported.
S2 ValidSolution	++	Supported.
S3 OptimalSolution	++	Supported.
S4 NextSolution	++	Supported.
S5 IncrementalSolving	+	Supported through incremental model changes and solver re-starts.
E1 Explanation	+	Choco provides an explanation engine based on [Veksler and Strichman, 2010], but it is not active by default; some amount of programming is necessary to use it.
E2 CorrectiveExplanation	0	Not supported.
I1 LanguageAPI	++	Native Java API.
I2 NotificationAPI	0	A callback function (not documented in the user manual) could be used to trace solver actions, but no dedicated event mechanism is available.
I3 SpecLanguage	+	Java API, support of FlatZinc and XCSP as input language.
I4 TestSupport	+	No native support, but Choco benefits from Java IDEs' test support.
I5 DebugSupport	+	No native support, but Choco benefits from Java IDEs' debugging features.
I6 LicenseCompatibility	++	BSD license.
I7 MinimumDependencies	++	Apart from a few required libraries, Choco has a very small footprint.

Table 2: Requirements evaluation table for Choco.

5.3 Potassco

Potassco, the Potsdam Answer Set Solving Collection [Gebser et al., 2019a], includes clingo [Gebser et al., 2019b] as a popular and actively developed system for Answer Set Programming (ASP). ASP is declarative knowledge representation formalism in which problems are encoded as logic programs [Gelfond and Kahl, 2014, Baral, 2003, Gebser et al., 2012, Lifschitz, 2019]. This formalism is quite different from classical constraint satisfaction, but lends itself quite well to the representation of object-oriented models like in our example [Falkner et al., 2015, Schenner and Taupe, 2017].

Compilations of clingo can be used to solve answer-set programs stored in text files. Alternatively, the system provides APIs for C++ and Python. It is licensed under the MIT license.

The language of ASP easily allows to formulate integrity, consistency, and resource constraints as well as weak constraints [Gebser et al., 2015a, Calimeri et al., 2020], but does not include typical global constraints (e.g. alldifferent) known from constraint modelling languages like MiniZinc.

Defaults can very naturally be encoded in ASP by use of default negation. For example, the following rule states that the `type` of a seat is `standard` except if the user has set another type:

```
seat_type(S,standard) :- seat(S), not userset_seat_type(S,_).
```

However, this rule will make the problem unsatisfiable if a seat cannot be of the `standard` type because of constraints.

An alternative solution is to use the *asprin* extension for preferences [Gebser et al., 2019a, Brewka et al., 2015], which allows for encodings like the following:

```
default_seat_type(S) :- seat_type(S,standard).
#preference(defaults,more(cardinality)){default_seat_type(S) :
    ↪ seat(S)}.
#optimize(defaults).
```

Here, the answer set with the highest number of default-type seats is considered optimal.

[Tab. 3] shows our requirements evaluation for the Potassco toolset.

5.4 Picat

Picat [Zhou et al., 2015] 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

Requirement	Rating	Comment
M1 ModelTransformation	+	Mapping from object-oriented model to ASP must be done manually by coding. An alternative is to use an extension such as Clorm [Rajaratnam, 2019].
M2 ConfigurationObjects	+	Cf. M1.
M3 AttrsAndAssocs	++	Mapping from object-oriented model to ASP must be done manually by coding. Associations are facilitated by predicates and aggregates. Common data types are supported.
M4 RulesAndConstraints	++	Rich support for integrity, consistency, and resource constraints, but not for typical global constraints (e.g., alldifferent) known from constraint modelling languages.
M5 StaticDefaults	+	Can be encoded manually using soft constraints. An alternative is using the *asprin* extension for preferences [Gebser et al., 2019a, Brewka et al., 2015], which cannot be used together with optimization statements or soft constraints.
M6 DynamicDefaults	+	Cf. M5.
M7 SoftConstraints	++	Supported out of the box.
B1 CreateOrDeleteObject	+	Multi-shot solving (cf. S5) can be used to iteratively solve problems without having to start from scratch; previous information can be retracted by setting external atoms to false.
B2 SetValue	++	Supported.
B3 UndoDecision	+	Cf. B1.
B4 Recommendations	+	Recommendation features not directly supported, but can be implemented using weak constraints, optimization statements or domain-specific heuristics.
F1 CompleteFiltering	++	Supported through brave reasoning (for the whole problem at once).
F2 FastFiltering	0	Not supported without an elaborate encoding.
F3 LocalFiltering	0	Not supported without an elaborate encoding.
S1 ExistsSolution	++	Supported by actually computing a valid solution.
S2 ValidSolution	++	Solving for valid solutions supported.
S3 OptimalSolution	++	Definition of optimization criteria and solving for optimal solutions supported.
S4 NextSolution	+	No support apart from searching for all solutions and processing them as they appear.
S5 IncrementalSolving	++	Supported through multi-shot solving [Gebser et al., 2019b]. Furthermore, an interactive shell for clingo has been proposed [Gebser et al., 2015b].
E1 Explanation	0	Not natively supported, but an active area of research [Fandinno and Schulz, 2019].
E2 CorrectiveExplanation	0	Not natively supported, but an active area of research [Fandinno and Schulz, 2019].
I1 LanguageAPI	++	Python, C++, and C APIs available.
I2 NotificationAPI	0	Callbacks for intercepting models available, but not for more detailed solver actions, e.g. propagation.
I3 SpecLanguage	++	The language of ASP is standardized [Calimeri et al., 2020].
I4 TestSupport	+	Testing frameworks for implementation languages available, but no built-in constraint testing framework.
I5 DebugSupport	0	Not natively supported, but an active area of research [Dodaro et al., 2019].
I6 LicenseCompatibility	++	MIT license.
I7 MinimumDependencies	++	Potassco depends on few third-party components.

Table 3: Requirements evaluation table for Potassco.

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.

```
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 **I2 (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.

[Tab. 4] shows our requirements evaluation for the tool Picat.

5.5 CP-SAT Solver

As an example of a constraint solver based on a non-constraint propagation paradigm we have chosen Google CP-SAT Solver, which is part of Google OR-Tools and based on boolean satisfiability (SAT).[10]

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,"")
```

[10] https://github.com/google/or-tools/

```
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[11] also an API in Java, C# and Python is provided, which makes it one of the few constraint solvers available in Python. Regarding Requirement **I1 (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.

[Tab. 5] shows our requirements evaluation for the tool CP-SAT Solver.

5.6 Z3

Z3 [De Moura and Bjørner, 2008] is a satisfiability modulo theories (SMT) theorem prover by Microsoft Research. The main application of SMT solvers in the past has been software verification, but it can also be used as a constraint solver for product configuration.

As Z3 is not a classic constraint solver it does not provide domain filtering out of the box. But it has some convenient features like incremental reasoning using scopes. The following listing shows a simple example:

```
from z3 import *
s = Solver()
// add constraints
s.add(nr_seats + standingroom == nr_passengers);
s.add(nr_passengers == 100)
s.add(nr_seats == 20)
// define new scope
s.push()
s.add(standingroom == 10)
if s.check()==unsat:
  // in case of inconsistency
  // return to previous scope
  s.pop()
```

Z3 supports the SMTLIB2 standard [Barrett et al., 2010] and can thus produce unsatisfiable cores and check assertions. These features can be exploited to provide explanations for inconsistencies. The Z3 solver is written in C++ with additional language bindings for .Net, Java, Python, OCaml and has a rich API.

[Tab. 6] shows our requirements evaluation for the tool Z3.

[11] http://www.swig.org/

Requirement	Rating	Comment
M1 ModelTransformation	+	Must be implemented by encoding the object-oriented model as a constraint problem.
M2 ConfigurationObjects	+	Must be implemented.
M3 AttrsAndAssocs	+	Must be implemented.
M4 RulesAndConstraints	++	As Picat includes solver modules for cp, sat, mip and smt (via Z3) there is a rich set of modelling options with standard constraint programming and global constraints.
M5 StaticDefaults	0	Not supported.
M6 DynamicDefaults	0	Not supported.
M7 SoftConstraints	+	Implementation is possible, but will depend on the used solver module.
B1 CreateOrDeleteObject	+	Implementation possible and facilitated by Picat data structures.
B2 SetValue	++	Supported.
B3 UndoDecision	0	Not supported.
B4 Recommendations	0	Not supported.
F1 CompleteFiltering	+	Not natively supported; can be emulated by solver call for each variable value.
F2 FastFiltering	++	Supported by built-in constraint propagation and user-defined constraint propagators.
F3 LocalFiltering	+	Can be implemented with constraint propagators.
S1 ExistsSolution	++	Supported.
S2 ValidSolution	++	Supported.
S3 OptimalSolution	++	Supported.
S4 NextSolution	++	Supported.
S5 IncrementalSolving	+	Supported by constraint-propagation.
E1 Explanation	0	Not directly supported.
E2 CorrectiveExplanation	0	Not directly supported.
I1 LanguageAPI	0	Language binding for Java etc. planned, but currently not available.
I2 NotificationAPI	+	Supported via action rules.
I3 SpecLanguage	+	FlatZinc, XCSP (cp module), SMTLIB (smt module) etc. supported.
I4 TestSupport	0	No testing framework available.
I5 DebugSupport	++	Built-in debugger.
I6 LicenseCompatibility	++	Mozilla Public License 2.0.
I7 MinimumDependencies	++	Standalone GitHub repository. Implemented in C/C++.

Table 4: Requirements evaluation table for Picat.

Requirement	Rating	Comment
M1 ModelTransformation	+	Encoding of object-oriented model as constraint problem necessary.
M2 ConfigurationObjects	+	Encoding of object-oriented model as constraint problem necessary.
M3 AttrsAndAssocs	+	Must be implemented. Integer and boolean supported.
M4 RulesAndConstraints	++	Basic constraint expressions and global constraints supported. Some global constraints do not support reification.
M5 StaticDefaults	0	Not supported.
M6 DynamicDefaults	0	Not supported.
M7 SoftConstraints	0	With a considerable effort using reification and optimization.
B1 CreateOrDeleteObject	+	By rebuilding constraint model.
B2 SetValue	++	Supported.
B3 UndoDecision	0	Not supported.
B4 Recommendations	0	Not supported.
F1 CompleteFiltering	+	Must be implemented by calling solver for every domain value.
F2 FastFiltering	0	Not supported.
F3 LocalFiltering	0	Not supported.
S1 ExistsSolution	++	Supported.
S2 ValidSolution	++	Supported.
S3 OptimalSolution	++	Supported.
S4 NextSolution	++	Supported.
S5 IncrementalSolving	0	Not supported.
E1 Explanation	0	Not supported.
E2 CorrectiveExplanation	0	Not supported.
I1 LanguageAPI	++	Implemented in C++. Language bindings (via SWIG) for Python, Java and C#.
I2 NotificationAPI	0	Not supported.
I3 SpecLanguage	+	Up-to-date FlatZinc import available (Winner of 2019 MiniZinc competition).
I4 TestSupport	+	Testing frameworks for implementation languages available, but no built-in constraint testing framework.
I5 DebugSupport	+	Debugging of implementation languages. Source code of solver available.
I6 LicenseCompatibility	++	Apache License 2.0.
I7 MinimumDependencies	+	Integrated into or-tools framework.

Table 5: Requirements evaluation table for CP-SAT Solver.

Requirement	**Rating**	**Comment**
M1 ModelTransformation	+	Encoding of object-oriented model as constraint problem necessary.
M2 ConfigurationObjects	+	Encoding of object-oriented model as constraint problem necessary.
M3 AttrsAndAssocs	+	Mapping from attributes and associations to a flat constraint model must implemented.
M4 RulesAndConstraints	++	Z3 supports basic constraint expressions over booleans, integers, reals, uninterpreted functions, data structures and some global constraints (Distinct).
M5 StaticDefaults	+	Can be emulated with soft constraints.
M6 DynamicDefaults	+	Can be emulated with soft constraints.
M7 SoftConstraints	++	Supported by API add_soft(constraint).
B1 CreateOrDeleteObject	+	Incremental reasoning is supported by API.
B2 SetValue	++	Supported.
B3 UndoDecision	+	Can be implemented with scope functionality (push/pop).
B4 Recommendations	0	Not supported.
F1 CompleteFiltering	+	Can be achieved by invoking a consistency check for every possible domain value.
F2 FastFiltering	0	Not supported.
F3 LocalFiltering	0	Not supported.
S1 ExistsSolution	++	Supported.
S2 ValidSolution	++	Supported.
S3 OptimalSolution	++	Definition of optimization criteria and solving for optimal solutions supported.
S4 NextSolution	+	Supported (by asserting the negation of the current solution).
S5 IncrementalSolving	++	Supported.
E1 Explanation	+	Can be implemented with assertions and unsat cores.
E2 CorrectiveExplanation	0	Not supported.
I1 LanguageAPI	+	Implemented in C++ with additional language bindings (.NET, Java, Python...).
I2 NotificationAPI	0	Not available.
I3 SpecLanguage	++	Various import formats supported (SMTLIB, Datalog, Dimacs, ...).
I4 TestSupport	+	Only for implementation language (C++).
I5 DebugSupport	+	Only for implementation language (C++).
I6 LicenseCompatibility	++	MIT License.
I7 MinimumDependencies	+	Source code available on GitHub with standard C++ build process (Make).

Table 6: Requirements evaluation table for Z3.

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, knowledge-based reasoning systems such as constraint-based or logic-based solvers are often employed. In this paper, we have made two contributions: First, we have proposed a set of requirements that a reasoner should fulfil to support interactive configuration. Second, we have presented the results of a small survey covering a selected set of non-commercial stand-alone solvers.

The focus of most reasoners is on modelling and efficient solving. Only few solvers contain special features for interactivity. Even in the case of solvers based on constraint propagation, the main purpose of propagation is to assist the solver during search. Only very few systems provide notification mechanisms or an explanation engine. None of the solvers we are aware of support default reasoning out of the box.

Our findings show that interactive aspects of configuration are not well supported by most stand-alone solvers although it may be possible to find workarounds (which need profound tool know-how or are difficult to maintain or are less efficient). Our findings also show that the landscape of available systems is highly diverse and that each solver has its own strengths and weaknesses when it comes to satisfying the requirements proposed in this work.

We are aware that our work may be biased by our experiences and focus. Future work should take a more systematic approach to collect more opinions (e.g. a comprehensive user survey). Furthermore, this study should be extended to cover more requirements (like aspects of guided selling or prediction of default values) and more KRR systems. We invite the configuration and KRR communities to propose implementations of knowledge-based reasoners that are suited to support interactive configuration.

Acknowledgements

The authors are thankful to Charles Prud'homme, Max Ostrowski, Philipp Obermeier, Philipp Wanko, and Torsten Schaub for their comments on a previous version of this article.

References

[Alviano et al., 2017] Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P., and Zangari, J. (2017). The ASP system DLV2. In *LPNMR*, volume 10377 of *LNCS*, pages 215–221. Springer.

[Amilhastre et al., 2002] Amilhastre, J., Fargier, H., and Marquis, P. (2002). Consistency restoration and explanations in dynamic CSPs — application to configuration. *Artif. Intell.*, 135(1-2):199–234.

[Apt and Wallace, 2007] Apt, K. R. and Wallace, M. (2007). *Constraint logic programming using Eclipse*. Cambridge University Press.

[Bak et al., 2016] Bak, K., Diskin, Z., Antkiewicz, M., Czarnecki, K., and Wasowski, A. (2016). Clafer: unifying class and feature modeling. *Software and Systems Modeling*, 15(3):811–845.

[Baral, 2003] Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

[Barrett et al., 2010] Barrett, C., Stump, A., and Tinelli, C. (2010). The SMT-LIB Standard: Version 2.0. In Gupta, A. and Kroening, D., editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*.

[Berkelaar, 2019] Berkelaar, M. (2019). Package ‘lpSolve’. `https://cran.r-project.org/web/packages/lpSolve/lpSolve.pdf`.

[Bessiere, 2006] Bessiere, C. (2006). Constraint propagation. In *Handbook of Constraint Programming*, pages 29–83. Elsevier.

[Brewka et al., 2015] Brewka, G., Delgrande, J. P., Romero, J., and Schaub, T. (2015). asprin: Customizing answer set preferences without a headache. In *AAAI*, pages 1467–1474. AAAI Press.

[Calimeri et al., 2020] Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., and Schaub, T. (2020). ASP-Core-2 input language format. *TPLP*, 20(2):294–309.

[Czarnecki et al., 2005] Czarnecki, K., Helsen, S., and Eisenecker, U. W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29.

[De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[De Smet and open source contributors, 2006] De Smet, G. and open source contributors (2006). Optaplanner user guide. `https://www.optaplanner.org`.

[Dodaro et al., 2019] Dodaro, C., Gasteiger, P., Reale, K., Ricca, F., and Schekotihin, K. (2019). Debugging non-ground ASP programs: Technique and graphical tools. *TPLP*, 19(2):290–316.

[Eiter et al., 2009] Eiter, T., Erdem, E., Erdogan, H., and Fink, M. (2009). Finding similar or diverse solutions in answer set programming. In *ICLP*, volume 5649 of *Lecture Notes in Computer Science*, pages 342–356. Springer.

[Falkner et al., 2019] Falkner, A., Haselböck, A., Krames, G., Schenner, G., and Taupe, R. (2019). Constraint solver requirements for interactive configuration. In Hotz, L., Aldanondo, M., and Krebs, T., editors, *21th Configuration Workshop (ConfWS)*, number 2467 in CEUR Workshop Proceedings, pages 66–73, Aachen.

[Falkner et al., 2016] Falkner, A. A., Friedrich, G., Haselböck, A., Schenner, G., and Schreiner, H. (2016). Twenty-five years of successful application of constraint technologies at Siemens. *AI Magazine*, 37(4):67–80.

[Falkner et al., 2015] Falkner, A. A., Ryabokon, A., Schenner, G., and Schekotihin, K. (2015). OOASP: connecting object-oriented and logic programming. In *LPNMR*, volume 9345 of *Lecture Notes in Computer Science*, pages 332–345. Springer.

[Fandinno and Schulz, 2019] Fandinno, J. and Schulz, C. (2019). Answering the "why" in answer set programming - A survey of explanation approaches. *TPLP*, 19(2):114–203.

[Felfernig et al., 2014] Felfernig, A., Hotz, L., Bagley, C., and Tiihonen, J. (2014). *Knowledge-based Configuration: From Research to Business Cases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Felfernig et al., 2012] Felfernig, A., Schubert, M., and Zehentner, C. (2012). An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62.

[Ferrucci, 1994] Ferrucci, D. A. (1994). *Interactive configuration: a logic programming-based approach*. PhD thesis, Rensselaer Polytechnic Institute Troy, NY, USA.

[Fleischanderl et al., 1998] 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.

[Gebser et al., 2015a] Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., and Schaub, T. (2015a). Abstract gringo. *TPLP*, 15(4-5):449–463.

[Gebser et al., 2019a] Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., Thiele, S., and Wanko, P. (2019a). Potassco guide version 2.2.0. `https://github.com/potassco/guide/releases/tag/v2.2.0`.

[Gebser et al., 2012] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2012). *Answer Set Solving in Practice*. Morgan and Claypool Publishers.

[Gebser et al., 2019b] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2019b). Multi-shot ASP solving with clingo. *TPLP*, 19(1):27–82.

[Gebser et al., 2015b] Gebser, M., Obermeier, P., and Schaub, T. (2015b). Interactive answer set programming - preliminary report. *CoRR*, abs/1511.01261.

[Gelfond and Kahl, 2014] Gelfond, M. and Kahl, Y. (2014). *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, New York, NY, USA.

[Hebrard et al., 2005] Hebrard, E., Hnich, B., O'Sullivan, B., and Walsh, T. (2005). Finding diverse and similar solutions in constraint programming. In *AAAI*, pages 372–377. AAAI Press / The MIT Press.

[Hertum et al., 2017] Hertum, P. V., Dasseville, I., Janssens, G., and Denecker, M. (2017). The KB paradigm and its application to interactive configuration. *TPLP*, 17(1):91–117.

[Heule et al., 2018] Heule, M. J., Järvisalo, M. J., Suda, M., et al. (2018). Proceedings of SAT competition 2018. `http://hdl.handle.net/10138/237063`.

[Hotz et al., 2014] Hotz, L., Krebs, T., and Wolter, K. (2014). Combining software product lines and structure-based configuration – methods and experiences. In *Workshop on Software Variability Management for Product Derivation – Towards Tool Support*.

[Janota, 2010] Janota, M. (2010). *SAT solving in interactive configuration*. PhD thesis, University College Dublin.

[Junker, 2004] Junker, U. (2004). QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *AAAI*, pages 167–172. AAAI Press / The MIT Press.

[Junker, 2006] Junker, U. (2006). Configuration. In *Handbook of Constraint Programming*, pages 837–873. Elsevier.

[Jussien and Lhomme, 2002] Jussien, N. and Lhomme, O. (2002). Unifying search algorithms for CSP. Rapport technique, École des Mines de Nantes.

[Kuchcinski and Szymanek, 2013] Kuchcinski, K. and Szymanek, R. (2013). JaCoP – java constraint programming solver. In *CPSolvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th CP*.

[Lifschitz, 2019] Lifschitz, V. (2019). *Answer Set Programming*. Springer.

[Madsen, 2003] Madsen, J. N. (2003). Methods for Interactive Constraint Satisfaction. Master's thesis, Department of Computer Science, University of Copenhagen.

[Meseguer et al., 2006] Meseguer, P., Rossi, F., and Schiex, T. (2006). Constraint propagation. In *Handbook of Constraint Programming*, pages 281–328. Elsevier.

[Møller et al., 2001] Møller, J., Andersen, H. R., and Hulgaard, H. (2001). Product configuration over the internet. In *Proceedings of the 6th INFORMS Conference on Information Systems and Technology*.

[Murashkin et al., 2013] Murashkin, A., Antkiewicz, M., Rayside, D., and Czarnecki, K. (2013). Visualization and exploration of optimal variants in product line engineering. In *SPLC*, pages 111–115. ACM.

[Nethercote et al., 2007] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In *CP*, volume 4741 of *LNCS*, pages 529–543. Springer.

[Nørgaard et al., 2009] Nørgaard, A. H., Boysen, M. R., Jensen, R. M., and Tiedemann, P. (2009). Combining binary decision diagrams and backtracking search for scalable backtrack-free interactive product configuration. In [Stumptner and Albert, 2009], pages 31–38.

[O'Callaghan et al., 2005] O'Callaghan, B., O'Sullivan, B., and Freuder, E. C. (2005). Generating corrective explanations for interactive constraint satisfaction. In *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 445–459. Springer.

[Orsvärn and Axling, 1999] Orsvärn, K. and Axling, T. (1999). The Tacton view of configuration tasks and engines. In *Workshop on Configuration, Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 127–130.

[Pleuss et al., 2011] Pleuss, A., Rabiser, R., and Botterweck, G. (2011). Visualization techniques for application in interactive product configuration. In *SPLC Workshops*, page 22. ACM.

[Prud'homme et al., 2017] Prud'homme, C., Fages, J.-G., and Lorca, X. (2017). Choco documentation. `http://www.choco-solver.org`.

[Queva et al., 2009] Queva, M., Probst, C., and Vikkelsøe, P. (2009). Industrial requirements for interactive product configurators. In [Stumptner and Albert, 2009], pages 39–46.

[Rajaratnam, 2019] Rajaratnam, D. (2019). Clorm: An ORM API for clingo. `https://clorm.readthedocs.io/en/stable/`.

[Russell and Norvig, 2010] Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence – A Modern Approach*. Pearson Education, 3rd international edition.

[Schenner and Taupe, 2016] Schenner, G. and Taupe, R. (2016). Encoding object-oriented models in MiniZinc. In *Fifteenth International Workshop on Constraint Modelling and Reformulation*.

[Schenner and Taupe, 2017] Schenner, G. and Taupe, R. (2017). Techniques for solving large-scale product configuration problems with ASP. In Zhang, L. L. and Haag, A., editors, *Proceedings of the 19th International Configuration Workshop*, pages 12–19, La Défense, France.

[Schiendorfer et al., 2018] Schiendorfer, A., Knapp, A., Anders, G., and Reif, W. (2018). MiniBrass: Soft constraints for MiniZinc. *Constraints*, 23(4):403–450.

[Schneeweiss and Hofstedt, 2011] Schneeweiss, D. and Hofstedt, P. (2011). FdConfig: A constraint-based interactive product configurator. In *INAP/WLP*, volume 7773 of *Lecture Notes in Computer Science*, pages 239–255. Springer.

[Stuckey et al., 2014] Stuckey, P. J., Feydy, T., Schutt, A., Tack, G., and Fischer, J. (2014). The MiniZinc challenge 2008-2013. *AI Magazine*, 35(2):55–60.

[Stumptner and Albert, 2009] Stumptner, M. and Albert, P., editors (2009). *Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09)*.

[Veksler and Strichman, 2010] Veksler, M. and Strichman, O. (2010). A proof-producing CSP solver. In *AAAI*. AAAI Press.

[Zhou et al., 2015] Zhou, N., Kjellerstrand, H., and Fruhman, J. (2015). *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer.