Back-to-Back Testing a Soft Constraint Model for a Smart Exhibition Space*

Alexander Schiendorfer[1], Benedikt Eberhardinger[1], Michael Wißner[2], Wolfgang Reif[1], and Elisabeth André[2]

[1] Institute for Software & Systems Engineering, University of Augsburg, Germany
{schiendorfer, eberhardinger, reif}@isse.de
[2] Human Centered Mutimedia, University of Augsburg, Germany
{wissner, andre}@hcm-lab.de

Abstract Obtaining a constraint model faithful to real world requirements is hard and requires considerable expertise. It is even harder to obtain an efficiently solvable constraint model as a result of several reformulation and refinement steps. Most often, tool support is hardly available to guide the modeler. Similar problems are faced in the field of software testing, where a program's correctness with respect to a specification has to be confirmed. There, a test suite consists of a set of test inputs and expected outputs. Using a folklore technique in assessing the quality of a model (assigning a fixed known set of values to a problem's variables) and testing whether it is identified correctly as solution or non-solution, we can apply many (automated) software testing techniques to improve correct modeling and reformulation of constraint programs. With little effort, many of the existing techniques can then be applied to constraint programs. We present our findings with a real-world problem embedded in a smart exhibition space, present coverage criteria designed for constraint programs, and illustrate model faults that could be found using very simple testing techniques.

1 Constraint Model Engineering

Constraint programming (CP) is a prominent tool to solve hard combinatorial problems and gave rise to a number of highly efficient algorithms encapsulated, e.g., in propagators. It is particularly well-explored for a set of well-defined benchmark problems[3] and traditionally focused on the efficient solution of formally sound constraint models. Due to the inherent complexity of the problems to be solved, deep knowledge about these algorithms guides the modeling process – making it hard for non-experts. However, in practical applications such as autonomous systems, a constraint solver does not work in isolation – it is rather integrated as a decision making component in a larger software system. In such situations, assuring the quality of the constraint model (in terms of correctness with respect to a specification – not yet solver efficiency) therefore amounts to a software engineering task. Moreover, *initially* there is no concise formal presentation

* This research is partly sponsored by the German Research Foundation (DFG) in the project "OC-Trust" (FOR 1085) as well as by the research project *Testing self-organizing, adaptive Systems (TeSOS)*.

[3] see, e.g., CSPlib

of the problem to be solved. Instead, the constraint model is crafted from requirements written in natural language that are inherently incomplete. It is subject to varying interpretations that emerge during the naturally error-prone implementation. Put differently, at the beginning of a modeling project, the concrete properties and requirements of the decision-making component are not known in their entirety. A specific example is given at the end of Sect. 3.1. Correctness not only has to be ensured during the initial, basic modeling, but especially when advanced modeling and reformulation techniques such as symmetry breaking, redundant constraints, reification, meta-constraints, or channeling constraints are added to improve the solver performance. For a CP-layperson that may have difficulties understanding a basic model, it becomes even more difficult to judge whether an optimized model (still) adheres to the specification.

A similar task is faced in software testing, when a program as well as a test model are crafted from a set of requirements by *different* developers – called *back-to-back* testing [28]. The main virtues of this method are a structured process to automatically detect differences in the understanding of requirements by means of deviating interpretations necessary for implementation. Such deviations can be used for debugging and discussing whether the model or the implementation are correct and need refinement.

We suggest that, with lightweight methods borrowed and adapted from software testing, considerable effects can be achieved when it comes to modeling and testing constraint programs. This form of quality assurance increases users' trust in the obtained models and is indispensable for practical usage of constraint solvers in decision making components. In this paper, we discuss a prototypical implementation of concepts known in classical software engineering that were adapted to constraint models. It remains open for future work how well existing testing tools can be integrated into constraint modeling suites.

In a nutshell, our approach relies on the interplay of a constraint modeling expert (called *constraint engineer*) with a testing expert (called *test engineer*) that work together to properly formalize informal requirements. The constraint engineer is responsible for an efficiently solvable and correct constraint model. The test engineer has to write a *result checker* (similar to a constraint checker in the context of global constraints) in a conventional programming language such as Java as a part of the usual *scaffolding code* that has to be developed along with a test suite. Upon presenting a test solution to the solver and the result checker, we can at least identify deviating opinions of, e.g., whether a constraint holds or not. For the constraint engineer, this decision involves selecting the right type of constraint(s) to match the requirement. This feedback leads to modifications of either the result checker or the constraint model – both need to be reflected in the requirements specification. Clearly, testing cannot be exhaustive due to the exponential number of assignments. To judge the *adequacy* of a test suite (i.e., example solutions and non-solutions), we define coverage criteria tailored to constraint programs such as, e.g., "Has there been a positive (satisfying) and negative (violating) example for every constraint?". Using the result checker, test case generation with respect to these coverage criteria can be automated using established search-based software engineering techniques. Hence, our paper offers as contributions:

(i) a practical modeling example (Sect. 3) that is still sufficiently small to make its analysis tractable,

(ii) a software setup that connects constraint modeling to a variety of classical software testing techniques, illustrated by automated test case generation with search-based testing (Sect. 4.1), and
(iii) some coverage metrics designed for constraint programs (Sect. 4.2).

All aspects are discussed with respect to our case study (see Sect. 3), a real-world problem that is embedded in the context of a smart exhibition space consisting of multiple adaptive displays which have to decide which content to show to multiple users based on preference, spatial, and contextual constraints. The source code of our prototypical implementation is, of course, available online at `http://git.io/vqTgN` [21].

2 Related Work

Recently, the need for more support (especially for non-experts) during the modeling phase has been recognized [4,10]. One solution is to try to have a system learn or acquire the constraints properly describing an instance set of solutions and non-solutions [5]. A model learned from data may still show errors that need to be found by, e.g., testing. These algorithms can be integrated with our approach by replacing the constraint engineer with a learning system and have the test engineer investigate the learned model.

Also, [10] highlights the integration of software engineering into constraint programming and optimization by using imperative programming languages that import constraint techniques as libraries rather than purely relying on a modeling language such as the optimization programming language (OPL) [27] or Zinc [3]. Then, classical and constraint-specific testing techniques (such as those our approach hopes to provide) are adequate.

The relationship between constraint programming and software testing is heavily asymmetric: in the direction of "constraints for testing", the field of *constraint-based testing* is rather well-explored [7,12], i.e., using a constraint solver to generate test inputs that satisfy a set of constraints obtained by exploration through the control flow graph. Widely used instances are Microsoft Pex [26], KLEE [6], or CUTE [24] that combine symbolic with dynamic execution to identify constraints on test input.

The other direction, however, is less intensively studied. While software testing tools are designed for imperative programs relying on control or data flow graphs, developers of constraint programs that exhibit different characteristics (e.g., dealing with sets of solutions and non-solutions) are left with little support. A first step to resolve this issue was discussed in [17], leading to the development of the CPTEST tool [18] that can automatically analyze OPL models. It is intended to detect inconsistencies in different OPL models in a refinement relation. This refinement is defined as solution set inclusion. Assignments violating the refinement (e.g., being a solution to one model and not to the other) are found automatically. However, the tool is limited to an already formal (satisfiable) constraint model that serves as perfect oracle. By contrast, our approach aims at the process from informal requirements to a correct constraint model with respect to the requirements. Quality of a model in our setting focuses on "model correctness", whereas quality in [18] focuses on "model efficiency" in terms of solver performance. Hence, a first OPL model can be derived using our approach and then further analyzed with CPTEST. Also, [18] considers optimality criteria only in the form

of a cost function mapping to a totally ordered set whereas we design techniques for the more structured but also more general framework of *soft constraints* [19]. We particularly address formalisms that have a notion of "allowed violation" of constraints such as, e.g., partial constraint satisfaction [11] or constraint relationships [16].

3 Case Study: A Smart Exhibition Space

Consider the following setting: public displays are distributed in a large exhibition space such as a museum or a shopping mall. The displays can detect users as they approach them, and present hopefully interesting content (categorized into topics) without the need for explicit manual interaction [29]. While the users walk freely through the exhibition space, the displays can track them and are thus able to continue the content presentation for specific users across multiple displays. A major motivation stems from the fact that contents can be selected based on a learned model of users' preferences instead of being selected at random. We call such an adaptive system a *smart exhibition space*. If, however, multiple users approach the same display, the decision making component needs to resolve conflicts due to varying preferences or different previously seen contents. For instance, a requirement is to maintain a "flow of topic" to avoid users having to switch their focus when they change displays.

Users have move around in the exhibition since some contents are only available at some displays. This is motivated by, e.g., "Egyptian History" only being shown by a display standing next to a well-preserved mummy. The contents to be displayed are structured by a precedence relation to describe a logical order. Also, they may include different levels of detail, catering to both firmly interested and "browsing" users. And lastly, it should be avoided to bore users by showing them contents they already know. Prioritizing these goals is certainly not straightforward and requires domain-specific insight. A development process should support the design activities of constraint modeling and preference assignment in parallel, without having one impede the other.

Our approach follows classical software engineering. We define a conceptual *domain* model that is formal enough to make the intended semantics of goals more precise yet abstract enough to allow for communication with other software engineers and CP-laypeople. Such a methodology allows to trace back functional constraint tests (as those proposed by our method in Sect. 4) to their respective requirements.

Based on the common domain model, the constraint engineer implements the constraint model under test (say, in OPL) and the test engineer programs the result checker (say, in Java). Here, the domain model will be defined using the language of set and graph theory (and can be mapped to UML). We first establish needed nomenclature:

Frame defines a single content to be shown to a user that is to be selected by the display; a frame may belong to several topics, e.g., *robotics introduction*, *robotics advanced* ... with some prerequisite constraint, e.g., *introduction* has to be seen before *robotics advanced*

Topic a coherent set of frames, e.g., *robotics*, *verification*, *algorithms*

User is detected and stored, for example by a smart camera system; depending on their previous interactions with the displays, preferences for topics as well as favorite knowledge (i.e., level of detail) are recorded; also the seen frames are stored.

Group represents a group of users at a display

In addition to the prerequisite-related constraints that will be described in Sect. 3.1, the problem also needs to consider users' favorite topics and preferred knowledge level. The problem we will consider now is when multiple groups appear at a single display at (roughly) the same time and the single next frame is to be decided. There are at least three interesting directions the resulting constraint model can be extended to:

Multi-Steps Instead of a single following frame, the system could try to design a good sequence of frames by anticipating the users' path through the exhibition.

Multi-Frames To resolve conflicts differently, split-screen view could be employed to show more frames at one display (typing the decision variable as "set of frames").

Multi-Displays Multiple displays could also coordinate their decisions to, e.g., make sure that one display does not show the only frame remaining at another display.

Within this framework, the problem discussed in this work is *single-display*, *single-frame*, and *single-step*. However, as we shall see, we can already reveal modeling faults in the simple instance. Once the conceptual and constraint model are relatively stable, the extension in the other dimensions can be pursued.

3.1 A Conceptual Model

We write the finite set of frames as Frames, topics as Topics, groups as Groups, and users as Users. The prerequisite relations we want to consider include specific sequences (e.g., take f_1 then f_2 then $f_3, \ldots$) as well as some prerequisite knowledge that has to be obtained in *any* order (e.g., see all frames from the introductory section before proceeding to the advanced content). To express both requirements, prerequisite dependencies between frames are modeled by a directed graph $(G, \rightarrow_G)$ with $G \subseteq \mathit{Frames}$, where an edge $g \rightarrow_G h$ (with $g, h \in G$, $\rightarrow_G \ \subseteq G \times G$) indicates that frame g needs to be seen before h. More specifically, a *transition* from g to h is *feasible* for a user u if u saw g. Edges may also be labeled with *guard* conditions[4] that impose constraints on additional frames required to be seen before "taking that edge". This is represented by a mapping $\varphi : \rightarrow_G \rightarrow 2^G$.
Let $\mathit{Seen}_u \subseteq \mathit{Frames}$ be the set of already seen frames for user u. An edge $(g, h) \in (\rightarrow_G)$ is then feasible iff:

$$\text{isFeasible}_u(g, h) :\Leftrightarrow \{g\} \cup \varphi(g, h) \subseteq \mathit{Seen}_u \tag{1}$$

If the next frame should only be reachable *directly* from the last seen frame, we can additionally require $g = \mathit{Last}_u$, where $\mathit{Last}_u \in \mathit{Frames}$ represents the last seen frame of user u.

$$\text{isFeasible}_u(g, h) :\Leftrightarrow \{g\} \cup \varphi(g, h) \subseteq \mathit{Seen}_u \wedge g = \mathit{Last}_u \tag{2}$$

In fact, whether requiring g to be the current frame or not may be ambiguous in the original requirements and has to be discussed when formalizing it properly.

[4] inspired by guarded commands

A *trace* of length k through a graph $(G, \rightarrow_G)$ for user u is given by $\langle g_1, g_2, \ldots, g_k \rangle$ where $g_i \rightarrow_G g_{i+1}$ and $\mathrm{isFeasible}_u(g_i, g_{i+1})$. A trace is maximal, if there is no feasible edge $g_k \rightarrow_G h$. The task of the constraint satisfaction problem is then to select the next frame $f \in \mathit{Frames}$ to be displayed to the groups in front of the display as well as to select a feasible edge for each user individually that reaches frame f:

$$\forall u \in \mathit{Users} : \exists (g_u, f) \in \rightarrow_G : \mathrm{isFeasible}_u(g_u, f) \tag{3}$$

Since $\rightarrow_G$ is a finite set, the selected edge for each user can be modeled (in some appropriate viewpoint) as a decision variable taking edges identified by ids as values.

We now consider concrete prerequisite examples shown in Fig. 1. The guards drawn for each edge e show the set $\varphi(e)$. We first present the scenarios a designer should impose and then the graphs representing feasible traces.

Example 1. Graphs B and C are interpretations of graph A, in the sense that B captures the paths in A as valid traces. It encompasses as maximal traces $\{\langle 1, 2, 4 \rangle, \langle 1, 3, 4 \rangle\}$, thus 2 or 3 can be seen as optional contents both leading to content 4. This might, e.g., be useful if both represent essentially the same content required for 4 but one might be more elaborate or suited for a particular user group (or, think of 2 and 3 as being different language versions leading to a pure image on content 4). For B, we do not need additional guard predecessors. It becomes apparent why we need a Last-variable if we want to establish exclusiveness between 2 and 3. To see this, consider the state after the (non-maximal) trace $\langle 1, 2 \rangle$: Seen would then be $\{1, 2\}$ and $\mathit{Last} = 2$. According to (1), both $1 \rightarrow_G 3$ and $2 \rightarrow_G 4$ are feasible and 3 could be selected as the next node. It is a matter of design if subtraces to be extended must end at the *last* seen node or *any* previously seen frame and whether to use (1) or (2) as feasibility predicate. If such peculiarities are not immediately clear from the requirements, our back-to-back testing approach may help to reveal them by detecting inconsistencies between the result checker and the implemented OPL model.

Graph C shows another possible relationship: We have a common starting frame followed by two contents that *both* must be visited but their order is irrelevant, and we conclude with a final frame. This case is particularly interesting for resolving conflicts among several groups because it introduces flexibility in the frame sequences. However, if, e.g., two groups with histories $\langle 1, 3 \rangle$ and $\langle 1, 2 \rangle$ arrive simultaneously, the solver has to show either 3 or 2 before 4, dissatisfying at least one group. We annotate edges with additional predecessor contents that should have been seen before continuing. Thus, to take the transition $2 \rightarrow_G 4$, 3 has to have been seen already – analogously for $3 \rightarrow_G 4$ with 2. Consequently, all maximal feasible traces are $\{\langle 1, 2, 3, 4 \rangle, \langle 1, 3, 2, 4 \rangle\}$. For this simple example, the specification is very straightforward, but if we were to write down all orderings explicitly for n contents, we would have to consider $n!$ sequences. On the other hand, if explicit orderings are relevant ("assure 2 is always be seen before 3"), we can always write them down more fine-grained than with the set-based notation.

To illustrate possible deviating interpretations of informal requirements, consider the following example: Assume a content ("sensor fusion") belongs to two topics, say "robotics" and "autonomous systems". How should one interpret "no robotics-contents should be allowed at a display"? Does it mean that "sensor fusion" is disqualified because it is *also* a robotics-content or is it alright because it is also connected to one

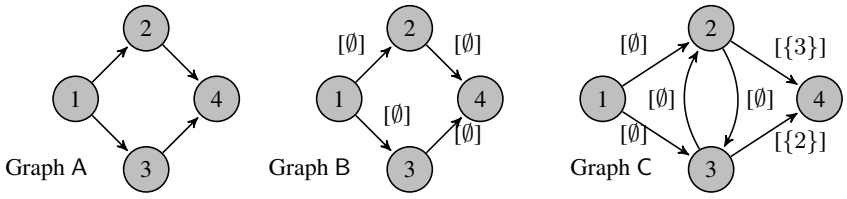

Figure 1. Three content structures possible to describe using the proposed content graph

non-robotics topic? Either way, upon judging an assignment of "sensor fusion" to the next frame f, different views on the satisfaction of this constraint could emerge.

3.2 An ILOG CP Implementation

We can implement the presented model in ILOG CP since it supports structured types (sets, arrays, ...) reasonably well. Nonetheless, some modeling effort goes into encoding sets for the solver to have constraints extractable. Also, the clean distinction of model code and instance data present in ILOG CP (also offered by MiniZinc) turns out to be beneficial for separation of concerns in our case study (constraint model, content data, and preference data). The model file itself is designed for human readability (even if the data are eventually generated from object graphs as instances of the domain model) to follow the principle of visibility in software quality [20].

We will now present some modeling peculiarities that may arise when using high-level modeling concepts such as sets or graphs. We intentionally restrict our attention to some points as the full OPL model can be found online [21]. Defining types for labeled edges is straightforward:

```
tuple Edge {
   int pred; // in minFrame .. maxFrame
   int succ; // in minFrame .. maxFrame
   {int} setPreds;
};
range edgeId = 1..3;
Edge edges[edgeId] = [<1,2, {}>, <1,3, {2}>, <1,4, {3}>];
```

Here, `setPreds` directly corresponds to $\varphi(e)$ for an edge e. As indicated before, we need to select a suitable edge for each user as well as a common next selected frame.

```
dvar int nextFrame in frameRange;
dvar int nextEdge[users] in edgeId;
```

In order to do so, we need to encode the predicates (1) and (2) in terms of constraints. No built-in functionality for a $\subseteq$ constraint is provided, so we need to construct suitable data structures in ILOG CP.[5] The main idea is to build the set $\psi(e) = \varphi(e) \cap Seen_u$ for each edge e and compare the cardinalities $|\psi(e)|$ and $|\varphi(e)|$. The set of seen frames $Seen_u$ is represented by `userData[u].seenFrames`:

[5] Indeed, for other solvers such as GECODE [23], a subset constraint on sets is already included – nonetheless we believe this modeling restriction by language features to be quite representative for practical scenarios.

```
int edgePredecessorSum[e in edgeId][u in users] =
  card(edges[e].setPreds inter userData[u].seenFrames) ;
int predCount[e in edgeId] = card(edges[e].setPreds);
int edgePredecessorValid[e in edgeId][u in users] =
  (predCount[e] == edgePredecessorSum[e][u]);
```

Based on these auxiliary structures we can formalize (1) and (2). More specifically, for (1), we write

```
int isFeasible[e in edgeId][u in users] =
  ((edgePredecessorValid[e][u] == true) &&
   (edges[e].pred in userData[u].seenFrames));
```

and for (2), we write

```
int isFeasible[e in edgeId][u in users] =
  ((edgePredecessorValid[e][u] == true) &&
   (edges[e].pred == userData[u].last));
   // assuming userData[u]. last in userData[u]. seenFrames as invariant
```

Among others, the actual constraints then amount to:

```
int isSuccessor[e in edgeId][f in frameRange] =
  (edges[e].succ == f); // has to be extractable
subject to {
  forall(u in users) { // not seen constraint
     !(nextFrame in userData[u].seenFrames) ;
  }
  forall(u in users) {
    // channels next frame and next edges
    isSuccessor[nextEdge[u]][nextFrame] == true
     && isFeasible[nextEdge[u]][u] == true;
  }
}
```

Note that we cannot simply write

```
edges[nextEdge[u]].succ not in userData[u].seenFrames;
```

since this expression is not extractable as a constraint but we can fortunately model it the way described. More modeling efforts are necessary when extending to multi-steps, multi-displays, or multi-frames.

However, the constraints described this way do not necessarily have to hold. In fact, there may be circumstances where a frame has to be shown but not for every user an appropriate edge can be found. Thus, we introduce softness by reification using penalties for every violated constraint and rewriting every (soft) constraint ϕ as

$$\phi' : (\phi \wedge p_\phi = 0) \vee (\neg\phi \wedge p_\phi = w_\phi) \tag{4}$$

where $(p_\phi)_{\phi\in\Phi}$ is a "penalty" vector indexed by the set of soft constraints Φ and w_ϕ is the weight to be issued if ϕ is violated. This encoding makes the problem an instance of the valued CSP framework [22], more specifically it is a weighted CSP. If we let $w_\phi = c$ for one constant value $c > 0$ (say, $c = 1$) and all ϕ alike, we obtain a Max-CSP. The objective is to reduce the total soft constraint violation, i.e., $\text{minimize} \sum_{\phi\in\Phi} p_\phi$.

To keep the weights easily parametrizable during the "calibration" of the overall system , we use the data and model separation of ILOG and write the weights to a

distinct file. In addition to the weights, we can further separate concerns by using dedicated data files. The content structure including topics, frames, and their dependencies are stored in a *static data file*. The *dynamic data file* contains information that depends on a momentary situation, i.e., a meet-up at a display including information about users and groups along with their seen frames. This separation into readable data files caters to the needs of the different stakeholders involved in the project, i.e., curators for the static content and software engineers (responsible for the camera-based recognition) for the dynamic aspects. As long as the static content data structures and soft constraints remain identical, the constraint engineer can add reformulation techniques and improve the model in terms of solver efficiency.

From these examples, it can be seen that structured information may require auxiliary decision variables and data structures. Especially if the constraints require such encoding instead of in-language availability (e.g., a subset constraint), testing methods are needed to judge the obtained OPL constraint model.

4 A Reference Test Setup

Our approach revolves around the concept of back-to-back testing, i.e., independently creating two versions of a software system based on a common specification. We suggest to map this principle to constraint programs by having the constraint engineer craft an OPL model (as sketched in the previous section) and a test engineer crafting a result checker with conventional imperative programs that identify whether a proposed solution is indeed one. More precisely, since we work in the soft constraint framework, both the result checker and the OPL model need to report the soft constraints an assignment violates. If there are inconsistencies, we can identify them and give a verbose message to both constraint and test engineer. The information about violated constraints issued by the solver is enabled by means of reification instead of a mere "unsolvable" message commonly reported when hard constraints are violated. Nevertheless, our approach is targeted at both classical and soft constraint problems.

We think that implementing a result checker in a conventional programming language is beneficial since there we already have a vast array of testing and verification techniques. If we are confident that the result checking methods are correct for a variety of test inputs, we can automatically test the constraint model. Moreover, presumably more software engineers are familiar with imperative programming languages than with optimization modeling languages (cf. [10]). As a side benefit of test development efforts, the result checker can be used for runtime verification to assure that *if* a solution is submitted by the solver, it is correct. One may even formally verify the result checker for more confidence [9]. There are several tasks for the result checker:

- An assignment is a solution (w.r.t. hard constraints) *iff* it is a solution according to the result checker.
- The set of violated soft constraints equals the set of reported violated soft constraints (and consequently the set of satisfied soft constraints match).
- The reported penalty equals the penalty calculated by the result checker.

As of now, we assume at least an agreement over the set of (informal) constraints to be modeled and check their correspondence one by one based on the common informal

Algorithm 1 Test Suite Execution

Require: $M = (X, D, C, \Phi)$ is a model under test, $T \subseteq [X \to D]$ is a test suite

1: **procedure** EXECUTE-TESTS(M, T)
2: **for all** $t \in T$ **do**
3: $(t, isSol, \Phi_{\text{vio}}, \Phi_{\text{sat}}) = \text{RESULT-CHECK}(t)$
4: $M' \leftarrow \text{FIX-SOLUTION}(M, t)$ $\triangleright \forall (x, d) \in t$: add "$x = d''$ to C
5: $(t, isSol', \Phi'_{\text{vio}}, \Phi'_{\text{sat}}) = \text{SOLVE}(M')$
6: $\text{ASSERTEQUALS}((isSol, \Phi_{\text{vio}}, \Phi_{\text{sat}}), (isSol', \Phi'_{\text{vio}}, \Phi'_{\text{sat}}))$

specification. However, the implementation of each constraint depends on the variables available in the chosen viewpoint where a mapping between corresponding solutions in different viewpoints is needed.

4.1 From Software Testing to Constraint Program Testing

In software testing, a test suite generally consists of a set of test inputs and expected outputs, and, of course, a program under test, possibly represented by a control flow graph (CFG). Using a folklore technique in assessing the quality of a model, viz. assigning a fixed known set of values to a problem's variables and checking whether it is identified correctly as solution or non-solution [15], we can apply (automated) software testing techniques to facilitate modeling and reformulation of constraint problems.

In analogy to classical software, test inputs then correspond to variable assignments and expected outputs are basically the (soft) constraints expected to be violated or satisfied. In the classical framework, a single constraint violation invalidates an assignment. With soft constraints, several of them may evaluate to a suboptimal value. Formally, for a soft CSP (X, D, C, Φ), a test case $(t, isSol, \Phi_{\text{vio}}, \Phi_{\text{sat}})$ consists of an assignment $t \in [X \to D]$, a Boolean flag $isSol$ indicating whether t is a solution w.r.t. C, and the sets of expected violated and satisfied soft constraints, Φ_{vio} and Φ_{sat}. The result checker can be used to calculate the expected results, thus serving as a test oracle.

Alg. 1 shows the basic testing procedure that consists of executing a set of test case assignments $T \subseteq [X \to D]$ (i.e., inducing a *test suite* of test cases[6]) both on the result checker and on a constraint solver using the model under test. The assertion in Line 6 of Alg. 1 allows one to precisely report model differences to a user, e.g., "Constraint ϕ should be violated according to the result checker but is in fact reported as satisfied by the constraint model".

A set of test cases could stem from a previously used model employing the result checker or be crafted manually. A rather obvious question asks when a test suite T is *adequate* to judge the correctness of the model M. Clearly, testing all $|D|^{|X|}$ assignments against the result checker is infeasible. To address similar questions in software testing, *coverage criteria* are employed: "The test suite is adequate if each line of code has been executed at least once" (*node coverage*). We borrow this idea and adapt it to constraint programs in Sect. 4.2. A simple coverage criterion might state: "For each

[6] Technically, we refer both to a set of assignments $T \subseteq [X \to D]$ and its induced test cases $T' = \{(t, isSol, \Phi_{\text{vio}}, \Phi_{\text{sat}}) \mid t \in T\}$ as test suites.

(soft) constraint we have at least one test case in T violating it". This can help to get more insight about the modeled boundary between solutions and non-solutions. Having defined suitable coverage criteria, we can generate test inputs satisfying them, which lead to detected model faults in our case study in Sect. 5. It is important to note that M is in fact a model instance in terms of ILOG CP, i.e., besides the decision variables and constraints themselves, the static content data and dynamic user data is incorporated into M. All data can be seen as parameters for the model under test. As a consequence, a test case for M needs to specify the content and user data as well, which can become considerable effort.

By contrast, in CPTEST [18], the parameters were restricted to choosing a scalar value k for the Golomb ruler problem. Our case study depends on more structured and numerous parameters, i.e., content graphs, users with their states (seen frames, preferred topics) and groups. It becomes more desirable to have these parameter sets generated. For instance, to detect a fault with the *knowledgeLevelFits*-constraint (a frame should exceed a user's preferred knowledge level), we need at least a graph with a content that supersedes a given user's preferred knowledge level.

4.2 Coverage Criteria for Constraint Programs

As mentioned before, to identify if each constraint is modeled correctly, we should investigate at least one solution that violates it and one that satisfies it. This idea roughly corresponds to node coverage. Formally, for a set of constraints C (similarly for soft constraints Φ) and a test suite T, we define *violation coverage* as the proportion of constraints that are violated by at least one $t \in T$:

$$\mathsf{vioCov}(C,T) := \frac{|\{\exists t \in T : t \not\models c \mid c \in C\}|}{|C|} \tag{5}$$

We proceed analogously for satCov for the coverage of satisfied constraints. Achieving $1 = \mathsf{vioCov}(C,T) = \mathsf{satCov}(C,T)$ is a minimal requirement to detect major flaws such as a constraint always evaluating to true (for all assignments, not solutions, as there *implied* constraints should always hold). Using the criteria satCov and vioCov, we can give a simple hill-climbing algorithm (see Alg. 2) to generate test suites that is also implemented for the case study problem in Sect. 5 where test assignments are simply frames to be selected. This technique is inspired by methods from search-based software engineering [13]. The implementation only needs the result checker instead of the actual constraint model to generate test cases.

Line 6 allows for a convex combination of coverage (increase) and cardinality of the test suite (decrease), specified by α. We would like to note that for the coverage criteria vioCov and satCov, one could, e.g., consider the sets of violated constraints by each $p \in P$ as *admissible* subsets of C and searching for a set of subsets of C of minimal size that covers C – hence solving a set covering problem [14].

Clearly, the quality (adequacy) of test suites is decisive for finding faults within the constraint model under test. In addition to the simple criteria vioCov and satCov that we implemented for our prototype, we envision stricter coverage criteria that can be imported directly from software testing but also the need for new ones specifically

Algorithm 2 Test Suite Generation using Simple Hill-Climbing

Require: $M = (X, D, C, \Phi)$ is a soft constraint problem
Require: $P \subseteq [X \to D]$ is a set ("pool") of test assignments (e.g., selected at random)
Ensure: return value T maximizes coverage (best-effort)

```
1: function CALCULATE-FITNESS(C, P')                                   ▷ P' ⊆ P
2:     T' = ∅
3:     for all p ∈ P' do
4:         t' = (p, isSol, Φ_vio, Φ_sat) = RESULT-CHECK(p)
5:         T' ← T' ∪ {t'}
6:     return α · (vioCov(C, T') + satCov(C, T')) − (1 − α) · |T'|
7: procedure GENERATE-TEST-SUITE(M, P)
8:     current ← RANDOMSUBSET(P)
9:     while not converged do
10:        next ← arg max_{n∈NEIGHBORHOOD(current,P)} CALCULATE-FITNESS(C, n)
11:        if CALCULATE-FITNESS(C, next) > CALCULATE-FITNESS(C, current) then
12:            current ← next
13:    return RESULT-CHECK(current)        ▷ lift RESULT-CHECK() to sets, cf. Lines 3 to 5
```

designed for constraint programs. In Alg. 2, these coverages must be incorporated in the fitness function in Line 6. We formulate these coverage criteria as requirements that need to be fulfilled by a test suite T and adapt established notions from logical coverage [1] to constraint programming.

Constraint Coverage (CC): In analogy to node coverage, **CC** combines vioCov and satCov into one test requirement: For each constraint $c \in C$, there has to be at least one test $t \in T$ such that c evaluates to *true*, and c evaluates to *false*.

Applied to a simple constraint $\mathsf{alldiff}(X = \{a, b, c\})$, the test suite $T = \{t_1, t_2\}$ with $t_1 = (a \to 1, b \to 1, c \to 2)$ and $t_2 = (a \to 1, b \to 2, c \to 3)$ satisfies **CC**. However, **CC** might be too weak if the considered constraint is in fact a logical expression composed of simpler constraints (a meta-constraint) [2]. Then, the individual parts of the constraints are not considered in particular by **CC**. If we intend to cover each individual part too, we consider our second coverage criteria saying:

Meta-Constraint Coverage (MCC): For each sub-constraint $s \in S_c$ of a meta-constraint c, there has to be at least one test such that s evaluates to *true*, and s evaluates to *false*. Consider the constraint $\mathsf{m} := (a \geq b \vee a \leq c) \wedge d$ with integer variables $\{a, b, c\}$ and a Boolean variable d. The following test suite $T' = \{t'_1, t'_2\}$ satisfies **MCC**: $t'_1 = (a \to 1, b \to 2, c \to -1, d \to \text{true})$ and $t'_2 = (a \to 1, b \to 1, c \to 2, d \to \text{false})$.

Alas, t'_1 and t'_2 do not satisfy **CC**; there is no subsumption relation between these two criteria. From a testing perspective, we would like to have a coverage criterion that tests constraints as well as their individual parts sufficiently. In order to achieve that, we try all combinations of truth values for the individual parts of the constraints.

Combinatorial Meta-Constraint Coverage (CoMCC): For each meta-constraint $c \in C$, there has to be at least one test for each combination of truth values to the sub-constraints S_c.

A meta-constraint c with n sub-constraints consequently leads to 2^n possible truth evaluations. Considering our example m, we already need 8 different test cases. This

Table 1. Truth table for fulfilling **APC** for the constraint $(a \geq b \vee a \leq c) \wedge d$. Test cases 7 and 8 are required additionally for **CoMCC**.

	$a \geq b$	$a \leq c$	d	$(a \geq b \vee a \leq c) \wedge d$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	F	F
6	F	F	T	F
7	F	T	T	T
8	F	F	F	F

makes **CoMCC** impractical for meta-constraints with more than a few sub-constraints. What we would like to have is a criterion that captures the effect of each sub-constraint, but does so in a reasonable number of tests. These observations lead to a collection of test criteria based on the notion of making sub-constraints of the meta-constraint "active", i.e., we check whether changing the truth value of a sub-constraint affects the meta-constraint's truth value, leaving all other sub-constraints unaffected.

To define the coverage criterion properly, we introduce the notion of *determination*: a sub-constraint s *determines* a meta-constraint c if flipping the truth value of s changes the truth value of c. Based on that definition, we consider the following coverage:

Active Part Coverage (APC): For each $c \in C$ and each sub-constraint $s \in S_c$, find an allocation to $s' \neq s \in S_c$ such that s determines c. **APC** requires that each s then evaluates to *true* and *false* in at least one test. As shown in Tab. 1, test cases 1 to 6 satisfy **APC** (and thus **CC** and **MCC**) whereas test cases 7 and from 8 would be needed to achieve **CoMCC**.

Apart from constraint coverages, one could also examine *domain coverage*, possibly with partitioning variables' domains to similar sets (e.g., if $D_x = \{0, \ldots, 100\}$, then $[0, 5]$ is "very small", $[30, 50]$ could be "medium", and $[90, 100]$ could be "high"). To find representative test cases while avoiding combinatorial overshoot, tools such as pairwise testing [25] might be useful.

5 Success Stories

We present a few representative "success stories" that illustrate how the testing process associated with the constraint modeling facilitated the development of the ILOG CP model presented in Sect. 3.2. The outputs can be re-enacted using the prototype [21]. A rather straightforward constraint implements the requirement `contentNotSeen`. The selected frame should not belong to the set of seen frames of a user:

```
forall(g in groups) {
  forall(userId in members[g]) {
    ( (nextFrame in userData[userId].seenFrames) &&
       userPenalties["contentNotSeen"][userId] == 0) ||
    (!(nextFrame in userData[userId].seenFrames) &&
       userPenalties["contentNotSeen"][userId] ==
         penalty_contentNotSeen);
```

```
  }
}
```

A rather simple checker method investigates a returned solution

```
private boolean contentNew(User u, Frame selectedFrame) {
  return !u.getSeenFrames().contains(selectedFrame);
}
```

and helps to diagnose that the OPL model is indeed *wrong*. Apparently, the constraint engineer mixed up the constraint and negated constraint along with their reified penalty setting. The result check is able to detect if the soft constraint is actually violated in a violated solution.

```
Exception in thread "main" java.lang.AssertionError:
We disagree on soft constraint contentNotSeen for user u1.
It should not be violated.
```

Another subtlety in the specification of another requirement could be found using the checker methods. More precisely, the question is how to interpret the requirement "there is a valid edge (with all additionally required frames already seen) leading to the next selected frame" if the selected frame has no incoming edges, i.e., is the *first* frame in a content structure. In our experiment, the statement was handled differently by the test engineer and the constraint engineer.

- The test engineer looked for a feasible edge leading to the selected frame. The check method returns true only if such an edge exists. Consequently, if the first frame (without incoming edges) is picked, the corresponding soft constraint is violated.
- The constraint engineer resolved this underspecification differently; an implicit edge from a "virtual" always seen frame is added to every node, making the soft constraint satisfied, if the first frame is picked.

It is indeed a matter of the requirements to decide which interpretation holds, but the discrepancy is again detectable, *iff* the first frame is indeed selected.

```
Exception in thread "main" java.lang.AssertionError:
We disagree on soft constraint predecessorsOkay for
user u1. It should be violated.
```

In this particular case, the constraint engineer's interpretation was adopted, leading to a modified check method.

The coverage criteria vioCov and satCov introduced in Sect. 4.2 helped to reveal a flaw introduced during the debugging of the model under test. Initially, a simple approach to testing the model was to simulate a few steps, i.e., let two groups meet at a display, have the solver select a frame, add the frame to the sets of seen frames, and repeat. This simulated trace was analyzed by the result checker and no fault was revealed. However, when analyzing the obtained solutions using coverage criteria, we noticed:

```
All soft constraints : "frameFitsTopic" "knowledgeFits"
  "contentNotSeen" "userInterested" "predecessorsOkay"
Violated S. constraints: "contentNotSeen" "frameFitsTopic"
Satisfied S. constraints: "contentNotSeen" "predecessorsOkay"
   "userInterested" "frameFitsTopic" "knowledgeFits"
Coverage violated : 40.0 %
Coverage satisfied: 100.0 %
```

Many soft constraints were not violated in any reported solution, making it unclear if they were modeled correctly in OPL (trivially, having them evaluate to true regardless of the assignment would not be detected). We therefore implemented Alg. 2 to find a test suite with constraint coverage (**CC**) and minimal cardinality of test cases, i.e., sets of frames to be selected.

```
Found a test suite with fitness: 1998.0
[Text-To-Speech;10, User Trust Model;4]
```

In fact, a test suite consisting of two frames to select was sufficient to provide 100% constraint coverage. When executing the set of test cases sequentially, we could in fact reveal the error in OPL. During debugging, the constraint engineer implemented the `knowledgeFits` constraint as `knowledgeLevel[nextFrame] == 1` which should have been:

```
knowledgeLevel[nextFrame] <=  userData[userId].knowledgeLevel
```

As it turned out, a solution that shows a higher knowledge level than 1 was required to reveal this flaw (which happened to be frame 10, "Text-to-Speech"), suggesting that test input data generation may be important for more elaborate constraint models.

6 Conclusions and Future Work

Upon introducing our case study, a smart exhibition space, we presented a pragmatic prototype for testing constraint models based on a back-to-back approach using a dedicated result checker. Relying on a double implementation of requirements, we hope to find inconsistencies between a constraint model and a result checker early on. We showed that, using straightforward coverage criteria and a simple hill-climbing algorithm inspired by search-based software engineering, we could automatically generate a test suite that revealed faults in the model.

We believe that the presented setup paves the way for more software testing techniques to help in practical constraint modeling. Our results give indication that even simple techniques can augment the quality assurance for modeling and reformulation tremendously and should be considered another "tool in the box".

Regarding concrete software testing techniques we deem useful, we want to close with some inspirations for future work:

Regression Testing When only refining an existing OPL model, a set of test cases with expected results obtained on the original model (e.g., violated (soft) constraints) can be compared to the refined model.

Mutation Coverage Classical test suites are declared adequate if they detect intentionally added small faults in a program such as writing $x < 5$ instead of $x \leq 5$. Similar ideas can be transferred to constraint test suites.

Capture-and-Replay If a constraint modeler promises to keep the solution space identical when only improving propagation etc., we want to check by simple simulations (user trajectories in our case study) if the model returns the same results.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
2. Bacchus, F., Walsh, T.: Propagating Logical Combinations of Constraints. In: Proc. 19th Int. Joint Conf. Artificial Intelligence (IJCAI'2005) (2005)
3. de la Banda, M.G., Marriott, K., Rafeh, R., Wallace, M.: The Modelling Language Zinc. In: Proc. 12th Int. Conf. Principles and Practice of Constraint Programming (CP'2006), pp. 700–705. Lect. Notes Comp. Sci. 4204, Springer (2006)
4. de la Banda, M.G., Stuckey, P.J., Van Hentenryck, P., Wallace, M.: The Future of Optimization Technology. Constraints 19(2), 126–138 (2014)
5. Bessiere, C., Coletta, R., O'Sullivan, B., Paulin, M., et al.: Query-Driven Constraint Acquisition. In: Proc. 20th Int. Joint Conf. Artificial Intelligence (IJCAI'2007). pp. 50–55 (2007)
6. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI. vol. 8, pp. 209–224 (2008)
7. DeMillo, R.A., Offutt, A.J.: Constraint-based Automatic Test Data Generation. IEEE Trans. Software Engineering 17(9), 900–910 (1991)
8. Eberhardinger, B., Steghofer, J.P., Nafz, F., Reif, W.: Model-driven Synthesis of Monitoring Infrastructure for Reliable Adaptive Multi-agent Systems. In: Proc. 24th Int. Symp. Software Reliability Engineering (ISSRE'2013). pp. 21–30. IEEE (2013)
9. Fischer, P., Nafz, F., Seebach, H., Reif, W.: Ensuring Correct Self-Reconfiguration in Safety-Critical Applications by Verified Result Checking. In: Proc. 2011 Wsh. Organic Computing (OC'2011). pp. 3–12. ACM, New York, NY, USA (2011)
10. Francis, K., Brand, S., Stuckey, P.J.: Optimisation Modelling for Software Developers. In: Proc. 18th Int. Conf. Principles and Practice of Constraint Programming (CP'2012). pp. 274–289. Springer (2012)
11. Freuder, E.C., Wallace, R.J.: Partial Constraint Satisfaction. Artif. Intell. 58(1–3), 21–70 (1992)
12. Gotlieb, A., Botella, B., Rueher, M.: Automatic Test Data Generation using Constraint Solving Techniques. In: ACM SIGSOFT Software Engineering Notes. vol. 23, pp. 53–62. ACM (1998)
13. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based Software Engineering: Trends, Techniques and Applications. ACM Comput. Surv. 45(1), 11:1–11:61 (Dec 2012)
14. Hoffman, K., Padberg, M.: Set Covering, Packing and Partitioning Problems. In: Floudas, C., Pardalos, P. (eds.) Encyclopedia of Optimization, pp. 2348–2352. Springer US (2001)
15. IBM: IBM Knowledge Center – Testing with a known solution (2015), `http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.2/ilog.odms.cpo.help/CP_Optimizer/User_manual/topics/debug_testing.html`, [Online; accessed 7-July-2015]
16. Knapp, A., Schiendorfer, A., Reif, W.: Quality over Quantity in Soft Constraints. In: Proc. 26th Int. Conf. Tools with Artificial Intelligence (ICTAI'2014). pp. 453–460 (2014)
17. Lazaar, N., Gotlieb, A., Lebbah, Y.: On Testing Constraint Programs. In: Proc. 16th Int. Conf. Principles and Practice of Constraint Programming (CP'2010), pp. 330–344. Springer (2010)
18. Lazaar, N., Gotlieb, A., Lebbah, Y.: A CP framework for testing CP. Constraints 17(2), 123–147 (2012)
19. Meseguer, P., Rossi, F., Schiex, T.: Soft Constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 9. Elsevier (2006)
20. Pezzè, M., Young, M.: Software Testing and Analysis: Process, Principles, and Techniques. Wiley (2008)
21. Schiendorfer, A., Eberhardinger, B., Wißner, M., Reif, W., André, E.: Testing-smart-exhibition-space. GitHub repository (2015), `https://github.com/Alexander-Schiendorfer/testing-smart-exhibition-space`

22. Schiex, T., Fargier, H., Verfaillie, G.: Valued Constraint Satisfaction Problems: Hard and Easy Problems. In: Proc. 14th Int. Joint Conf. Artificial Intelligence (IJCAI'95), Vol. 1. pp. 631–639. Morgan Kaufmann (1995)
23. Schulte, C., Lagerkvist, M., Tack, G.: GECODE. Software download and online material at the website: http://www. gecode. org (2006)
24. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C, vol. 30. ACM (2005)
25. Tai, K.C., Lie, Y.: A Test Generation Strategy for Pairwise Testing. Trans. Softw. Eng. (1), 109–111 (2002)
26. Tillmann, N., De Halleux, J.: Pex–White Box Test Generation for .Net. In: Tests and Proofs, pp. 134–153. Springer (2008)
27. Van Hentenryck, P.: The OPL Optimization Programming Language. Mit Press (1999)
28. Vouk, M.: Back-to-back testing. Information and Software technology 32(1), 34–45 (1990)
29. Wißner, M., Hammer, S., Kurdyukova, E., André, E.: Trust-based Decision-making for the Adaptation of Public Displays in Changing Social Contexts. Journal of Trust Management 1(1), 6 (2014)