Generating Configuration Models from Requirements to Assist in Product Management

Tiihonen, Juha Tapani

CEUR-WS.org
2018-09-23


http://hdl.handle.net/10138/298437

Downloaded from Helda, University of Helsinki institutional repository.

This is an electronic reprint of the original article.

This reprint may differ from the original in pagination and typographic detail.

Please cite the original version.
Generating Configuration Models
from Requirements to Assist in Product Management –
Dependency Engine and its Performance Assessment

Juha Tiihonen\textsuperscript{1} and Iivo Raitahila\textsuperscript{1} and Mikko Raatikainen\textsuperscript{1} and Alexander Felfernig\textsuperscript{2} and Tomi Männistö\textsuperscript{1}

Abstract. Requirements engineering is often, especially in the context of major open source software projects, performed with issue tracking systems such as Jira or Bugzilla. Issues include requirements expressed as bug reports, feature requests, etc. Such systems are at their best at managing individual requirements life-cycle. The management of dependencies between issues and holistic analysis of the whole product or a release plan is usually scantily supported. Feature modeling is an established way to represent dependencies between individual features, especially in the context of Software Product Lines — well-researched feature model analysis and configuration techniques exist. We developed a proof-of-concept dependency engine for holistically managing requirements. It is based on automatically mapping requirements and their dependencies into a variant of feature models, enabling utilization of existing research. The feature models are further mapped into a constraint satisfaction problem. The user can experiment with different configurations of requirements. The system maintains the consistency of dependencies and resource constraints. To evaluate the feasibility of the system, we measure the performance of the system both with some real and generated requirements. Despite some remaining performance issues, it seems that the approach can scale into managing the requirements of large software projects.

1 INTRODUCTION

There are various kinds of requirement management systems (RMS) applied in requirements engineering [10]. In particular, different issue tracker systems, in which requirements are captured as issues, are becoming increasingly popular, especially in large-scale, globally distributed open source projects, such as in the cases of Bugzilla for Linux Kernel, Github tracker for Homebrew, and Jira for Qt. An issue tracker can contain tens of thousands requirements, bugs and other items that are different ways interdependent from each other.

Issue tracker systems as RMSs provide primarily with support for individual requirements throughout various requirements engineering activities, such as requirements documentation, analysis, and management as well as tracking the status of a requirement over its life cycle. Even though individual dependencies, including more advanced constraints, can be expressed in the case of an individual requirement, more advanced analysis over all requirements of a system taking into account the dependencies and properties of the requirements is not well supported. For example, deciding a set of requirements to be implemented simultaneously might need to follow all dependencies transitively, which is not readily supported by the issue trackers. The issue trackers are not either necessarily optimal for the concerns of product or release management that need to deal with different requirement options, alternatives and constraints, as well as their dependency consequences when deciding what to do or not to do. However, dependencies in general are found to be one of the key concerns that need to be taken into account, e.g., in requirements prioritization [1, 9, 18] and release planning [2, 17]. In fact, the above concerns are not at the core of issue trackers’ support for the requirements engineering activity. Rather, issue trackers focus more on a single issue, its properties, and its life cycle. The situation is not necessarily specific only for issue trackers, but it exists also in other kinds of RMS.

In the context of a Horizon 2020 project called OpenReq, we developed a proof-of-concept Dependency Engine for holistically managing requirements as a single model. It is based on automatically mapping requirements and their existing isolated dependencies into the Kumbang [3] variant of feature models, enabling utilization existing research. A feature model is further mapped into a constraint satisfaction problem. The user can experiment with different configurations of requirements. The system maintains the consistency of dependencies and resource constraints.

This paper outlines the principle of the Dependency Engine and addresses its feasibility in terms of performance. We measure the performance of the system with both some real and generated requirements. Responsive performance is important for interactive usage, e.g., what-if analysis of requirements to include in a release. Furthermore, it is important that decisions are based on current information; either relatively fast model generation or a way to update models ‘on-the-fly’ are required.

The rest of the paper is organized as follows. Section 2 outlines the concept of a feature model. Section 3 presents the research questions, general idea of the Dependency Engine, applied data and tests. Section 4 presents the results, Section 5 provides analysis and discussion. Finally, Section 6 concludes.

2 BACKGROUND: FEATURE MODELING

The notion of a feature model, similarly as a requirement, is not ambiguous. A feature of a feature model is defined, e.g., as a characteristic of a system that is visible to the end user [12], or a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among product variants [8]. A feature

\footnotesize{\textsuperscript{1}University of Helsinki, Finland, first.last@helsinki.fi}\textsuperscript{2}Graz University of Technology, Austria, alexander.felfernig@ist.tugraz.at

CEUR-WS.org/Vol-12220/12_CONFWS18_paper_15.pdf
model is a model of features typically organized in a tree-structure. One feature is the root feature and all other features are then the subfeatures of the root or another feature. Additional relationships are expressed by cross-branch constraints of different types, such as requires or excludes. Feature model dialects are not always precise about their semantics, such as whether the tree constitutes a part-of or an is-a structure [19]. Despite this, feature models have also been provided with various formalizations [8, 16] including a mapping to constraint programming [5, 6].

Specifically, we apply the Kumbang feature model conceptualization [3] as the basis. It has a textual feature modeling language and it has been provided with formal semantics. Kumbang specifies sub-features as part-of relations and allows defining separate is-a hierarchies. Kumbang supports feature attributes and its constraint language can be used to express cross-branch relationships.

A feature model is a variability model roughly meaning that there are optional and alternative features to be selected, and attribute values to be set that are limited by predefined rules or constraints. When variability is resolved, i.e., a product is derived or configured, the result is a configuration. Variability is resolved by making configuration selections such as an optional feature is selected to be included, or one of alternatives is selected. A consistent configuration is a configuration in which a set of selections have been made, and none of the rules have been violated. A complete configuration is a consistent configuration in which all necessary selections are made.

Feature modeling has become a well-researched method to manage variability and has been provided with several different analyses to assist in system management [4].

3 METHODS AND DATA

We follow Design Science in the sense that the aim is to innovate a novel intervention and bring it into a new environment so that the results have value in the environment in practice [11]. Dependency engine is the artifact of the intervention and this paper focuses on its quality attributes. The specific research questions are:

- **RQ1:** Can the OpenReq Dependency Engine scale to real-world projects?
- **RQ2:** How can the performance of the Dependency Engine be improved?

### 3.1 Approach and architecture

To facilitate requirement management via a feature-based approach, we make each requirement correspond to exactly one feature. The properties of a requirement correspond to the attributes of a feature. The dependencies of individual requirements are mapped to hierarchies and constraints of a feature model. We currently rely only on the dependencies explicitly expressed in requirements although we will aim to extract missing dependencies with NLP technologies. In order to make such a mapping, we need a feature model dialect that is conceptually relatively expressive supporting feature typing and attributes. Kumbang was selected for this purpose.

The Dependency Engine currently consists of three stand-alone software components with specific responsibilities: **Milla**, **Mulperi** and **SpringCaaS**, see Figure 1. There are two different workflows: creating a model from requirements data and making subsequently queries against the model. These three components operate as REST-type services and are implemented using the Java Spring framework.

Milla is a front-end that is used to access requirement data via volatile or case-dependent interfaces. For example, it extracts requirements via the API of Jira. It outputs MulSON, a JSON based intermediate transfer format understood by Mulperi.

Mulperi converts from a small number of stable requirement input formats such as MulSON into the Kumbang feature modeling language. It can generate a number of queries to SpringCaaS.

SpringCaaS takes as input Kumbang feature models and converts them into a corresponding Constraint Satisfaction Problem (CSP). Choco Open-Source Java library for Constraint Programming [15] was selected because it is Java-based, popular, and has good performance and a permissive license. The Kumbang model and corresponding the data structures are saved for subsequent use.

### 3.2 Potential bottlenecks and related tests

#### Network and external system bottlenecks

Jira integration fetches requirements from the RMS one requirement at a time over network, which can potentially create performance bottlenecks. These bottlenecks are outside the scope of this paper.

#### Requirement model generation

Milla generates feature models from requirements data fetched from Jira. Effectively, relevant data, such as IDs, dependencies and the attributes that are needed in inference, are extracted and a MulSON presentation is generated.

#### Feature model generation

A requirement model expressed in MulSON is sent to Mulperi. Mulperi generates a feature model expressed in the Kumbang feature modeling language. Mulperi’s func-
tionality is largely based on data structure manipulation - JSON input and Kumbang output. The transformation is straightforward. Mulperi also saves the results into an in-memory database. This model is then sent to SpringCaaS in a single message.

**Feature model to CSP**  A feature model expressed in Kumbang is parsed. Kumbang syntax resembles many programming languages. Therefore parsing is potentially heavy.

Based on the data structures representing the feature model, a corresponding Constraint Satisfaction Problem (CSP) is generated. Basically, a set of Boolean CSP variables represents instances of individual feature types. Each of these is related to corresponding integer CSP variables that represent attribute values of these individual features. Enumerated strings are mapped to integers. Choco constraints are created based on the dependencies; the constraints can access the presence of a feature, and relationships between attribute values of features. The current implementation supports only binary relationships (requires, excludes).

In addition, it is possible to specify global resource (sum) constraints over specific attributes. For example, the sum of efforts of included features can be constrained. To facilitate this, the implementation reserves attribute value 0 to attribute values of features that are NOT in configuration.

**CSP solving**  The prime suspect for performance bottlenecks is solving the CSP representing a model of requirements. There are a number of tasks to accomplish based on a constructed model.

- check a configuration of features for consistency
- complete a configuration of features
- determine the consequences of feature selections

The selection of search strategy often has significant effect on solvability and quality of solutions [15].

3.3 Data

The performance evaluations are based both on real data from the Qt company and synthetic data.

3.3.1 Real requirements

Qt is a software development kit that consists of a software framework and its supporting tools that are targeted especially for cross-platform mobile application, graphical user interface, and embedded application development. All requirements and bugs of Qt are managed in the Qt’s Jira that is publicly accessible. Jira is a widely used issue tracker that can contain many issue types and has a lot of functionality for the management of issues. Issues and bugs can be considered as requirements and they have dependencies and attributes with constant values, such as priority and status. Thus, known requirements and their dependencies have already been identified and entered into Jira. Qt’s Jira contains 18 different projects and although some of the projects are quite small and discontinued, QT-BUG as the largest project contains currently (April 2018) 66,709 issues.

For empirical evaluation with real data, a set of issues was gathered from Qt’s Jira and processed through the whole pipeline. Only well-documented requirements having dependencies were selected to the dataset JiraData that contains 282 requirements.

3.3.2 Synthetic data

The synthetic datasets were created and run using automated scripts. SynData1 dataset contains a total of 450 models with permutations of the amounts of requirements (from 100 to 2000), a ‘requires’ dependency (from 0 to 75% of the requirements), an optional subfeature with one allowed feature (from 0 to 75% of the requirements) and a number of attributes (from 0 to 5 per requirement); each attribute has two possible values, e.g., 10 and 20.

A smaller dataset (60 test cases), SynData2, was used for optimization tests with sum constraints, see Section 3.4.5. SynData2 contains models with permutations of the amounts of requirements (from 100 to 2000), a ‘requires’ dependency (from 0 to 75% of the requirements), no further subfeatures and 1 or 2 attributes with a fixed random value from 1 to 100. An example of a SynData2 requirement in MULSON format:

```json
{
  "requirementId": "R4",
  "relationships": [
    {
      "targetId": "R25",
      "type": "requires"
    }
  ],
  "attributes": [
    {
      "name": "attribute1",
      "values": ["10", "20"],
      "defaultValue": "10"
    },
    {
      "name": "attribute2",
      "values": ["9"],
      "defaultValue": "22"
    }
  ],
  "name": "Synthetic requirement nro 4"
}
```

3.4 Empirical tests

3.4.1 Test setup

Measurements should be conducted when the software’s behaviour is typical[13]. Since there is currently no production environment, the tests are conducted on a development environment that closely resembles the possible production environment. Furthermore, we aim to perform tests that could correspond to real usage scenarios.

The test machine is an Intel Core i5-4300U 1.9GHz dual core laptop with 16GB of RAM and an SSD disk, running Ubuntu Linux 16.04 and a 64-bit version of Oracle Java 1.8.0. All software components except for Jira are run on the same machine.

The examined software components log execution times to files that are collected after each automated test run. A timeout was set to limit the solving of Choco in SpringCaaS.

Although SpringCaaS is a single component, we often report the execution time in two parts: Choco Solver and the rest of SpringCaaS. This is because often Choco’s solve operation takes the most time, but the initial tests showed that the Kumbang parser becomes a bottleneck in specific situations.

3.4.2 Initial trials and initial search strategy

Initial testing was performed with the goal to complete a configuration of requirements with a minimal number of additional requirements. The pareto optimizer of Choco was applied to provide alternative solutions. All features were included in the pareto front. By default, Choco uses the domOverWDeg search strategy for integer and Boolean variables [15]. Table 3 describes the search strategies

---

5 https://bugreports.qt.io
6 https://www.atlassian.com/software/jira
7 Pareto optimizer dynamically adds constraints: a solution must be strictly better than all previous solutions w.r.t. at least one objective variable [15].
applied. In our domain and way of modeling, the strategy effectively leads to selection of excessive number of features. This is contrary to the initial goal. As results in the beginning of Section 4 will show, an alternative search strategy was required to achieve satisfactory performance. The Search Strategy was changed to minDomLBSearch; it is used in the rest of the tests unless otherwise mentioned.

3.4.3 Requirement model generation

JiraData and SynData1 Datasets were applied to run the whole pipeline from gathered requirements to a Kumbang textual model and a serialized feature model. The process is illustrated at the left hand side of Figure 1. The different steps were timed.

In the case of SynData1 dataset, Milla was bypassed because the test cases were expressed directly in MULSON. Note that model generation includes finding a consistent configuration of features; this search is performed as a form of a model sanity check.

3.4.4 Requirement configuration

Autocompletion of configurations was performed with the JiraData dataset. A run was performed with a sum calculation constraint. Here, each requirement has a numerical priority attribute. The query instructed SpringCaaS to give a configuration where the sum of these priority attributes was greater than 100.

More substantially, requirement configuration was also performed with the SynData1 dataset to analyse the performance under varying number of requirements and their properties (attributes, dependencies), and user-specified requirements. This test applies optimization to find a (close to) minimum configuration that includes pre-selected features, if any. Effectively, the configuration of requirements is completed. This is presumably one of the computationally most intensive operations. The configuration phase is tested in ten iterations: first selecting only one requirement and then increasing the number of selected requirements by 10% so that the tenth iteration has 1 + 90% requirements selected.

3.4.5 Optimised release configuration under resource constraint

We performed a number of resource-constrained optimization tests. Here, we applied global sum (resource) constraints specified in Table 1 to constrain the allowed solutions. SynData2 Dataset contains test cases with 1 or 2 attributes per requirement (see Section 3.3.2). Effectively, the combination of number of attributes and the applied constraint correspond to usage scenarios presented in Table 2. Finally, we applied the bestBound(minDomLBSearch()) search strategy, after we had experimented with different alternatives, see Section 3.4.6 and corresponding results.

We run the tests with 60s, 10s and 3s timeout values to see the effect of allowed time on the solvability and to get an impression on the quality of solutions. In addition, we developed and experimented with a custom algorithm that (roughly) first ‘filled’ effort bounds with ‘big’ features and used ‘small’ ones to meet the bound.

3.4.6 Determining search strategy

We tested a set of different search strategies for performance, utilizing the 2000 requirement test cases of the SynData2 dataset. The experimented basic search strategies included activityBasedSearch, Choco default domOverWDog, and minDomLBSearch, see Table 3. These were augmented with bestBound, lastConflict or both; e.g., bestBound adds directs search based on the objective function and a strict consistency check.

<table>
<thead>
<tr>
<th>Constraint#</th>
<th>#attributes</th>
<th>Optimization goal</th>
</tr>
</thead>
</table>
| 0           | 1           | Simulate achieving desired utility with a minimal number of requirements to implement. Minimizes the number of requirements.
| 1           | 1           | Check the consistency of a given partial configuration with respect to maximum effort and complete the configuration with as few requirements as possible. Minimizes the number of requirements. In the case of SynData2, the test is redundant, only the root feature will be included.
| 3           | 1           | (Not relevant, 2 attributes in constraint, 1 in model) |
| 4           | 2           | Simulate maximisation of utility under resource constraint: Maximize sum of attributes. |
| 4           | 2           | Minimize the number of requirements to implement under constraints of minimum utility and maximum effort. |

Table 1. Resource constraints for optimization tests

Table 2. Constraints, number of attributes and usage scenarios

4 RESULTS

The results of the initial trials are in the two first rows of Table 4. The timeout and solution limits were disabled. The processing time was unacceptable, as reflected in the results.

4.1 Requirement model generation

The first two rows of Table 6 present the results of processing the JiraData dataset through the whole pipeline. Table 5 shows the results of processing the SynData1 dataset. A save operation includes finding a consistent non-optimized configuration of requirements.

Figure 2 presents cases with 1000 requirements. Each bar color corresponds to a test case with a specific number of dependencies (from 0 to 200) and subfeatures (from 200 to 1000). The elapsed time in Mulperi, SpringCaaS and Choco are shown for 0, 2000 and 5000 attributes, that is, 0, 2 or 5 attributes per feature, each with two possible values per requirement.

Figure 3 depicts a case with 1000 requirements and different number of subfeatures (a requirement can be a subfeature of many requirements). Please note the logarithmic scale. With 5000 subfeatures it took over five hours to parse the model.

Starting from (some) models with 1000 requirements, the serialization of the parsed Kumbang model failed due to a stack overflow error. It was necessary to increase the Java Virtual Machines stack size to one gigabyte to prevent out-of-memory errors.
### Table 4. Effect of search strategy with Pareto optimizer, JiraData dataset

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Optional features</th>
<th>Mandatory features</th>
<th>Attributes</th>
<th>Solutions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>60</td>
<td>130 to 300 ms</td>
</tr>
<tr>
<td>default</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1046</td>
</tr>
<tr>
<td>minDomLBSearch</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>120 to 130 ms</td>
</tr>
<tr>
<td>minDomLBSearch</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>150 to 190 ms</td>
</tr>
<tr>
<td>minDomLBSearch</td>
<td>235</td>
<td>0</td>
<td>2 per feature</td>
<td>1</td>
<td>160 to 200 ms</td>
</tr>
<tr>
<td>minDomLBSearch</td>
<td>118</td>
<td>117</td>
<td>2 per feature</td>
<td>1</td>
<td>400 to 630 ms</td>
</tr>
</tbody>
</table>

### Table 5. Minimum, maximum and median test cases of the save phase, SynData dataset

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Dependencies</th>
<th>Subfeatures</th>
<th>Attributes</th>
<th>Mulperi time (ms)</th>
<th>SpringCaaS time (ms)</th>
<th>Choco time (ms)</th>
<th>Total time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>8/5</td>
<td>200</td>
<td>0</td>
<td>85</td>
<td>158</td>
<td>25</td>
<td>241</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>500</td>
<td>100</td>
<td>504</td>
<td>247</td>
<td>495</td>
<td>1244</td>
</tr>
<tr>
<td>750</td>
<td>76/1</td>
<td>139</td>
<td>0</td>
<td>139</td>
<td>2,02</td>
<td>401</td>
<td>780</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1200</td>
<td>0</td>
<td>1200</td>
<td>1,71</td>
<td>2,239</td>
<td>3,950</td>
</tr>
<tr>
<td>1500</td>
<td>1125</td>
<td>600</td>
<td>0</td>
<td>684</td>
<td>509</td>
<td>2,099</td>
<td>3,702</td>
</tr>
<tr>
<td>2000</td>
<td>1500</td>
<td>1500</td>
<td>3000</td>
<td>3,000</td>
<td>3982</td>
<td>7,526</td>
<td>18,531</td>
</tr>
<tr>
<td>2500</td>
<td>2250</td>
<td>7500</td>
<td>21,747</td>
<td>21,747</td>
<td>17,38</td>
<td>38,12</td>
<td>57,575</td>
</tr>
<tr>
<td>3000</td>
<td>1000</td>
<td>800</td>
<td>0</td>
<td>841</td>
<td>386</td>
<td>4,781</td>
<td>5,608</td>
</tr>
<tr>
<td>2000</td>
<td>1500</td>
<td>4000</td>
<td>0</td>
<td>6,238</td>
<td>2018</td>
<td>4,043</td>
<td>8,301</td>
</tr>
</tbody>
</table>

### Table 3. Choco Search strategies

<table>
<thead>
<tr>
<th>Search strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activityBasedSearch</td>
<td>Search strategy for 'black-box' constraint solving. The idea of using the activity of variables during propagation to guide the search. A variable activity is incremented every time the propagation step filters its domain and is aged. [14]. Used parameters (GAMMA=0.999d, DELTA=0.2d, ALPHA=8, RESTART=1.1d, FORCE_SAMPLING=1) [15]</td>
</tr>
<tr>
<td>domOverWDeg</td>
<td>Choco default. Intuitively, it avoids some trapping by first instantiating variables involved in the constraints that have frequently participated in dead-end situations [7]. Slightly oversimplifying, the strategy attempts to solve hard parts of a CSP first, weighting constraints by their participation in dead-ends.</td>
</tr>
<tr>
<td>minDomLBSearch</td>
<td>Assigns the non-instantiated variable of smallest domain size to its lower bound [15]</td>
</tr>
<tr>
<td>bestBound</td>
<td>Search heuristic combined with a constraint performing strong consistency on the next decision variable and branching on the value with the best objective bound (for optimization) and branches on the lower bound for SAT problems [15]</td>
</tr>
<tr>
<td>lastConflict</td>
<td>&quot;Use the last conflict heuristic as a plugin to improve a former search heuristic. Should be set after specifying a search strategy.&quot; [15]</td>
</tr>
</tbody>
</table>

![Figure 2. Performance effect of attributes](image2.png)

![Figure 3. Kumbang parser’s fatigue](image3.png)
4.2 Requirement configuration

The results of the configuration task with the *JiraData* dataset are from row 3 onwards in Table 6. In case of the sum constraint (the last row), SpringCaaS was able to find 107 to 120 solutions before the timeout at 5s was reached.

Table 7 contains the minimum, maximum and median measurements of total execution times for varying numbers of requirements, dependencies, subfeatures, attributes and number of pre-selected requirements in the request.

Figure 4 shows the effect of the number of dependencies in case of 1500 requirements per test case, but with a varying number of require-dependencies.

Figure 5 shows the effect of the number of requirements and attributes in case of 1500 and 2000 requirements per test case.

4.3 Optimized configuration under resource constraint

Table 8 presents a summary of the results of test that minimize the number of features. Note that constraint #4 is from test cases with 2 attributes, the others apply to test cases that have one attribute that is constrained. Because all features are optional, tests with constraint #1 trivially contain only the root feature of the model.

Table 9 represents the results of optimizing via Maximization of the sum of attribute 2 (e.g. utility) under constraints on attribute1.

Test cases with 100, 500, 750, 1000, 1500 and 2000 requirements and varying numbers of requirements are solvable with 60s timeout. 10s handles all cases except 2000 requirements. 3s timeout is only applicable to cases with 100, 500 and 750 requirements.
A memory of 3 GB was required to complete the tests. The bestBound search strategy became feasible by applying the optimization to one variable or to the sums of attributes. A pareto front with all feature variables caused excessive memory consumption.

### 4.4 Determining search strategy

Table 10 compares search strategies with 2000 requirement test cases and minimization tasks. defaultSearch and activityBasedSearch fail in a number of cases with 60s timeout. minDomLBSearch can solve all these cases. Negative \( \Delta N \) indicates that the compared search strategy found better solutions (e.g. Total number of features was 18 less in the 30 tests).

Constraint #2 with 2 attributes is essentially unconstrained for big problems. Here, the optimal solution includes all features. Plain minDomLBSearch fails to 'notice' that. Both bestBound(minDomLBSearch()) and lastConflict(bestBound(minDomLBSearch())) help the solver to find the maximal solution. Of these, in terms of maximized result on attribute2, bestBound(minDomLBSearch()) is slightly better in 3 cases and lastConflict(bestBound(minDomLBSearch())) in one. Due to limitations of space, further details are omitted.

Earlier tests with all features in the pareto front prevented the usage of bestBound strategy due to increased memory consumption.

### Requirement model generation

The number of dependencies between the requirements seem to have no impact during the save phase. To avoid out-of-memory errors, Kumbang model read and write methods could be overridden with an implementation that suits better for the Kumbang data structure, or the serialization phase could be omitted altogether. On the other hand, optimized solving needs even more memory.

Increasing the number of attributes increases the processing time of each component steadily, see Figure 2. Increasing the amount of subfeatures increases the processing time of Mulperfi and Choco steadily as well, but when the amount of subfeatures is very large, the Kumbang parser slows down drastically; see Figure 3.

### Requirement configuration

The results in Section 3.4.4 suggest that a five second timeout would be sufficient for models with about 500 requirements or less. The configuration of all 1000 requirement models and most of the 1500 requirement models can be performed in less than five seconds.

The timeout value of the save phase could be set to be longer. Both timeout values could be controlled with parameters, for example if the user thinks that he/she can wait for a full minute for the processing to complete. During the configuration phase, the dependencies actually ease Choco’s inference burden. Figure 4 with 1500 requirements shows that when there are no dependencies, the preselected requirements in the configuration request speed up Choco linearly.

The increase in configuration request size adds processing overhead to SpringCaaS. Secondly, when the dependency rate gets higher, more requirements are included in the configuration early on, again helping Choco perform faster. The same is true for subfeatures: selecting requirements with subfeatures decreases processing time.

With attributes, the situation is the opposite. The more there are attributes and the more configuration request contains selected requirements, the more time it takes to select attributes, see Figure 5.

The optimization task is computationally intensive. It is difficult for the solver to determine if an optimal solution has been found. Therefore solving practically always ends with a timeout.

### Optimised release configuration under resource constraint

When a solution is found, the versions with a lower timeout value remain almost as good as solutions obtained with 60s timeout. The custom algorithm was expected to perform well in test case types 1 and 2. However, this seems not be the case. Out of 150 test cases, the algorithm finds better solutions than the 'normal' minimizer in 18 cases. In the clear majority of cases, it performs worse.
Determining search strategy  The best search strategy for our purposes is \( \text{bestBound}( \text{minDomLBSearch}() ) \) instead of plain \( \text{minDomLBSearch}() \), because it provides slightly better results in minimization tests and maximizes significantly better.

6 CONCLUSIONS

Solutions without optimization are easy to find; solvers such as Choco have an easy task with sparse dependencies. Still, at least for optimization, the selection of a search strategy matching the problem at hand remains crucial. It was surprising that the "black-box" activityBasedSearch[14] and Choco default domOverWDep[7] did not perform in a satisfactory way.

The prototype engine easily scales to around 2000 requirements, even when optimization is desired. Despite some remaining performance issues, it seems that the approach can scale into managing the requirements of large software projects, even for interactive use.

However, very large software projects, such as QT-BUG remain challenging. A more close examination of the Qt Jira is required, because it seems that performance can be managed in various ways. First, there are different types of issues such as bugs and requirements that do not need to be considered at the same time. Second, Qt has used Jira over a decade and there is a lot of historical data. The rate of new Jira issues seems to be up to 20 per a day. So, considering only issues created or modified within three years would significantly decrease the amount of data. Third, the exact nature of Qt data issues created or modified within three years would significantly decrease the amount of data. The prototype engine easily scales to around 2000 requirements, even for interactive use.

The concept of Dependency Engine is novel and it seems to be feasible for its intended use for providing holistic support for the management of dependencies, also in the context of large software projects.

ACKNOWLEDGEMENTS

This work has been funded by EU Horizon 2020 ICT-10-2016 grant No 732463. We thank the Qt Company for sharing the data.

REFERENCES


