# Using a feature model configurator for release planning

Mikko Raatikainen
Juha Tiihonen
Tomi Männistö
University of Helsinki
Helsinki, Finland
firstname.lastname@helsinki.fi

Alexander Felfernig
Martin Stettinger
Ralph Samer
Graz University of Technology
Graz, Austria
firstname.lastname@ist.tugraz.at

## ABSTRACT

The requirements for a system have many dependencies that can be expressed in the individual requirements managed in an issue tracker or a requirements management system. However, managing the entire body of requirements taking into account all complex dependencies is not well supported. We describe how a feature model based configurator can be used as a tool to help manage requirements data. Data transfer and constructing the needed requirements model can be carried out automatically by relying on a model generator. We implemented a prototype tool for requirements and release management that utilizes a knowledge-based configurator.

## CCS CONCEPTS

• **Software and its engineering → Requirements analysis**; **Software product lines**;

## KEYWORDS

Release management, feature modeling, requirements engineering.

**ACM Reference Format:**
Mikko Raatikainen, Juha Tiihonen, Tomi Männistö, Alexander Felfernig, Martin Stettinger, and Ralph Samer. 2018. Using a feature model configurator for release planning. In *22nd International Systems and Software Product Line Conference - Volume B (SPLC '18), September 10–14, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3236405.3236411

## 1 INTRODUCTION

Various kinds of *requirement management systems* (RMS) are being applied in requirements engineering [11]. Traditional RMS examples include Doors and Polarion. More recently, different issue trackers have become increasingly popular, especially in large-scale, globally distributed open source projects. For example, Linux kernel uses Bugzilla, Homebrew uses Github tracker and Qt uses Jira.

Especially issue trackers primarily support individual requirements throughout various requirements engineering activities, such as requirements documentation, analysis, and management including tracking the status of a requirement. *Dependencies* such as *requires* can be expressed in an individual requirement. Advanced analyses over all requirements taking into account the dependencies and properties of the requirements are not well supported. This, despite the fact that dependencies are one of the key concerns that need to be taken into account in requirements prioritization [1, 10, 22] and release planning [2, 21]. In fact, dependencies have not been at the core of the tool support for the requirement engineering activity. Rather, the focus is on a single requirement, its properties, and its life cycle.

Although requirements engineers and product management, especially for a single system, are not directly concerned with software variability management, the context and problems are very similar with the ones that variability management [13, 21] aims to solve. Variability management can address concerns, such as how to take into account the properties of and dependencies between numerous requirements especially in incremental development when several consecutive releases as an incrementally building sequence of product variants are planned. More precisely, variability management aims to capture knowledge of what configurations of systems can be constructed so that the resulting system is meaningful and meets users' needs. Thus, one essential part of variability management is to capture the dependencies of the system. For instance, if one desired requirement consists of other requirements, or requires another requirement, all need to be selected as a consequence of selecting the desired requirement. In particular, 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]. Consequently, variability models can also be seen as what is to be realized – it is not only variability in terms of selecting features but also the meaningful implementation order.

This paper describes how a configurator as a tool can be used to help in requirements engineering and product management. We addresses specifically release management related to requirements during the development life cycle and over the entire body of requirements. The specific concerns addressed are the management and analysis of dependencies between requirements and their properties. We describe how the requirements of a system under development can be automatically represented using the concepts of a variability model in order to utilize the existing research on variability management. We use a feature model dialect as the concrete variability model. Finally, we describe a prototype service-based system that realizes the concepts and can be used either by integrating to Jira or by dedicated prototype release planner.

## 2 CONCEPTUAL BASIS

In general, our approach is based on using requirements data stored in a issue tracker (or RMS) and using the constructs of a feature model to represent this data. For clarity, we use the term *requirement model* rather than a *feature model* of the resulting model of requirements.

The concept of a *requirement* is in general somewhat ambiguous and we adopt a relatively general notion of a requirement: We consider each requirement to be a single, unique entity for a system. A requirement is further characterized as roadmappable, meaning that a requirement is an entity that is decided to be implemented, e.g., in a specific release, or disregarded. For clarity, we presume that each requirement has a separate unique identification (ID), and name. The name (or content) describes the requirement in human-understandable form, often textually.

A requirement is seen as an instance of a *requirement type* that can define a set of named *properties* as property-value pairs for the requirements of a system. Archetypical properties include priority, planned release, effort and assignee. A requirement has at least implicitly a status as a property that indicates whether the requirement has been implemented. However, the actual property types are context-dependent [18].

It is possible to structure individual requirements hierarchically. A more general requirement can consist of a set of more detailed requirements that we refer to as a *part-of* hierarchy. As concrete examples, an *epic* can consist of a set of *user stories*, or a *user requirement* can be refined to a set of *technical requirements*. A requirement can also have *dependencies* as relationships beyond the part-of hierarchies to other requirements, such as a requirement *requires* or *excludes* another requirement. Industrial studies emphasize the importance of dependencies [15, 24] and taxonomies have been proposed of dependencies [7, 9, 17, 25]. Finally, the dependencies are not only between requirements per se, but also between the properties of requirements. For example, each requirement specifies its planned release as a property value, such as the release is "3", and an dependency can specify that the releases must not be the same for two specific requirements.

The notion of a *feature model*, similarly as a requirement, is not unambiguous. A *feature* of a feature model is defined, e.g., as a characteristic of a system that is visible to the end user [14], or a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among product variants [8]. A feature model is often represented as a graphical feature diagram, but textual feature models also exist. A feature 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, e.g., their semantics, such as whether the tree constitutes a part-of or an is-a structure [23]. Despite this, feature models have also been provided with various formalizations [3, 8, 20] including a mapping to constraint programming [5, 6].

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 constraints. As variability is resolved, i.e., a product is derived or configured, the result is a *configuration*. Variability is resolved—often stepwise—by making a set of decisions for variability resolving that we refer to as configuration *selections*. For instance, an optional feature is selected to be included, or one of the 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, although there can be still additional selections that can be made or changed, such as adding an optional feature. After making a selection or a set of selections, *consequences* are typically deduced in order to maintain the configuration consistent or to detect early an inconsistency. For example, when a selected feature requires another feature, the consequence is that both features become selected.

In order to make the connection between requirements and a feature model, we construct a requirements model by using the basic concepts of a feature model, such as sub-feature relations, and constraints. We make a correspondence between requirements and features at the representational level. Although requirements and features are similar and the correspondence is quite intuitive, we are not claiming or making a general similarity-mapping between the concepts of a requirement and feature—in one system requirements and features can actually be the same, in another system not. Rather, we aim at using the concepts of a feature model to represent a set of individual requirements including their dependencies in a requirements model so that the analyses of the model become possible with the analyses developed for feature models.

More specifically, we make each requirement correspond to exactly one feature in a feature model. The properties of a requirement correspond respectively to the attributes of a feature. Different requirement types, such as epics and user stories, can be represented by the respective concept of feature types. In order to make such a mapping, we are required to have a feature model dialect that is conceptually relatively expressive supporting, e.g., subtyping and attributes.

The part-of (or consist of) relationships of requirements constitute the tree hierarchy, thus being alike to subfeature-relationships. In order to construct the tree of requirements, a generic root is defined that, in practice, is the project itself, which has then all requirements underneath either directly, or through multiple and transitive part-of relationships. Other dependencies among requirements correspond to the cross-branch constraints of a feature model.

In practice, all requirements do not state their dependencies. The situation is similar as in a feature model in which especially all cross-branch dependencies are rarely stated to the extent that automated analysis and configuring is always reliable. However, as requirements often have descriptive text, we have also experimented with extracting missing dependencies automatically using natural language processing (NLP). The extracted dependencies are marked with a *proposed* status to allow humans to accept or reject them. Neverthless, rather than being a formal presentation, a requirements model always remains somewhat incomplete in which analyses remain approximations.

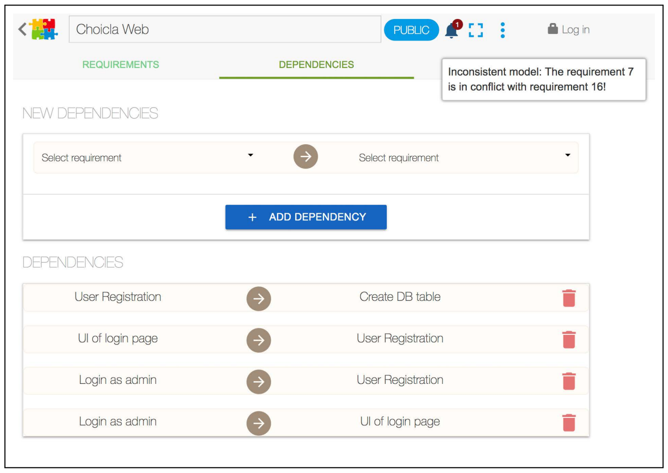
## 3 APPLYING A CONFIGURATOR FOR REQUIREMENTS AND RELEASE PLANNING

The resulting requirements model utilizing feature model representation technologies enables applying configurator technologies for requirements. A specific application scenario is a release planner [19], in which requirements are assigned to a set of releases. A release, similarly as a requirement, can have certain properties such as length bound by start and end dates, and an amount of available resources. The requirements are then assigned to ordered releases so that attributes of the releases are taken into account, in addition to the properties and dependencies of requirements.

First, the overall correctness and characteristics of the requirements per se can be assessed. Although requirement can express dependencies, it is not guaranteed that all dependencies are correct. For example, an analysis of dependencies is carried out to find circular hierarchies or mutually conflicting dependencies. The model is analyzed for meaningfulness, such as whether there exists a configuration and whether the cumulative effort exceeds the available effort. In addition, other analyses especially originating from the feature model analysis (cf. [4]) can be carried out. For example, dead requirements, which cannot be selected because of dependencies, can be identified. In practice, dead requirements are an indicator of incorrect dependencies between requirements. Dead requirements can also be a result of selecting one of mutually exclusive requirements, such as for different implementation technologies, and other requirements that are not selected become over time obsolete dead requirements, and can be removed or archived.

Second, the resulting requirements model can be used for analysis of and experimenting with different kinds of development scenarios and options. Basically, each configuration selection for requirements can result in consequences that need to be taken into account in development and product management. The selected requirement can require other requirements meaning that the required requirements need to be developed before or at the same time. Alternatively, there can be other dependencies, such as another requirement should not be developed at the same time or in the same release, because they represent functionality that has been decided to be introduced incrementally over time. It is also possible to assess the properties of requirements, such as the cumulative effort of the selected requirements in a consistent configuration. That is, requirements can form an interdependent group, which represents requirements for a set of features or modules of the system, and by combining the dependent requirements enables treating and analyzing them together. Thus, the analyses can be carried out for any individual requirement, a set of requirements to see the consequences.

Specifically, any planned release should represent a complete configuration in which the needed requirements are assigned to the specific or any earlier release. Consequently, release management becomes a configuration problem to find a sequence of consistent and complete configurations. A release can be analyzed for validity, such as no required requirements are left out to later releases, or the release does not contain requirements that are developed in the same release but should be developed in a different release. As the release is a property of a requirement, the total effort of a release can be analyzed as a cumulative effort of the individual requirements. We support limiting the maximum effort capacity of a release so that the cumulative effort of requirements assigned to the release must not exceed the capacity. Different options for development by assigning requirements differently to different releases can then be experimented with. Other similar analyses can be carried out, such



Inconsistent model: The requirement 7 is in conflict with requirement 16!

| | |
|---|---|
| User Registration | Create DB table |
| UI of login page | User Registration |
| Login as admin | User Registration |
| Login as admin | UI of login page |

**Figure 1: Inconsistent requirements model in OpenReq release planner user interface. The arrows are requires-dependencies.**

as the priorities of requirements can be analyzed in order that high priority requirements are included as early as possible, but if a low priority requirement is required by a high priority requirement, the low priority requirement should also be in an early release.

Finally, it is possible to propose repairs for inconsistent configurations. For example, if the available effort of a release is exceeded, low priority requirements can be suggested to be moved to later releases. In particular, model based diagnosis techniques [12] known from knowledge based configuration research can be applied for this purpose.

## 4 ARCHITECTURE AND DESIGN

We have implemented a proof of concept service-based system[1] using Java Spring for the above described concepts. The system consists of independent services that in practice operate in a choreographic manner combining the pipe-and-filter and layered architectural styles. The services following REST principles and collaborate through JSON message-based interfaces. An example of requirements datais shown in Figure 2 in Appendix.

The *Mulperi* service operates as a pipe-and-filter service that generates a requirements model (a feature model) from individual requirements utilizing the existing dependencies defined in them. Mulperi also operates as the controller for SpringCaaS when analysis tasks are performed.

The back-end logic realizing a feature model configurator is implemented in *Spring Configurator as a Service (SpringCaaS)*. SpringCaaS is based on extending an existing feature model based configurator [16]. SpringCaaS is a stand-alone service without any graphical user interface that facilitates feature model based configuring and a set of feature model analyses. In the basic scenario, SpringCaaS takes the requirements model expressed as a feature model as an input message and maps it to a Constraint Satisfaction Problem (CSP). Then after, different queries can be carried

[1]The services will be available as open source on OpenReq project webpage openreq.eu.

out including configuring selections that preserve consistency and deduce consequences.

The proof-of-concept design of the *OpenReq release planner* user interface is implemented as reactive web pages. The basic functionality is similar to an issue tracker or RMS to store requirements, their properties and dependencies. Figure 1 shows an example view of requirements and dependencies, which have been checked for consistency and an inconsistency is notified for a user. The individual requirements are transferred from OpenReq release planner to the above backend services using aforementioned JSON.

Additional services implement supporting requirements model management subtasks. The *Milla* service facilitates integration with different issue trackers or RMSs, and provides requirements in a uniform format to Mulperi. As an evaluation of the integration we have used Jira, in which Qt (www.qt.io) open source project stores its tens of thousands requirements and bug reports (bugreports.qt.io). For this integration, Milla fetches the requirements individually as a batch process using the *Mallikas* service as a persistent database cache before sending requirements to Mulperi. Integration to any other RMS can be done similarly either in Milla or by other similar service; the design is not specific for Jira per se. We have also experimented with ReqIF (www.omg.org/spec/ReqIF) as a general data interchange format used in many RMSs, such as Doors. Finally, the *Nikke* service is another standalone, experimental service that extracts missing dependencies from requirements data using different NLP algorithms.

## 5 CONCLUSIONS

We have described in this paper a tool demonstration that utilizes a feature model based configurator for requirements data. The tool consists of services that automatically generate the needed requirements model from requirements data in an issue issue tracker or requirements management system. The resulting requirements model can be used for a configurator as a service to help in handling the dependencies over the entire body of requirements, such as consistency of dependencies and consequences of selecting a set of requirements. Specifically, we focused on elaborating the scenario of release planning and management in which configuration technologies can help with the validity of requirements assignment to different releases.

We have now different basic services in place to demonstrate the technical feasibility. The experiments with performance indicate that the system is scalable for managing at least thousands requirements even on an office computer. The future work focuses especially on experimenting with and adapting to practically relevant scenarios and contexts.

## ACKNOWLEDGMENTS

This work is a part of OpenReq project that is funded by the European Union's Horizon 2020 Research and Innovation programme under grant agreement No 732463.

## REFERENCES

[1] Philip Achimugu, Ali Selamat, Roliana Ibrahim, and Mohd NazâĂŹri Mahrin. 2014. A systematic literature review of software requirements prioritization research. *Information and Software Technology* 56, 6 (2014), 568–585.
[2] David Ameller, Carles Farré, Xavier Franch, and Guillem Rufian. 2016. A Survey on Software Release Planning Models. In *International Conference Product-Focused Software Process Improvement*. 48–65.
[3] Timo Asikainen, Tomi Männistö, and Timo Soininen. 2007. Kumbang: A Domain Ontology for Modelling Variability in Software Product Families. *Advanced engineering informatics journal* 21, 1 (2007), 23–40.
[4] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.
[5] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Automated Reasoning on Feature Models. In *Conference on Advanced Information Systems Engineering*.
[6] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. 2005. Using Constraint Programming to Reason on Feature Models. In *17th International Conference on Software Engineering and Knowledge Engineering*.
[7] Pär Carlshamre, Kristian Sandahl, Mikael Lindvall, Björn Regnell, and Johan Natt och Dag. 2001. An industrial survey of requirements interdependencies in software product release planning. In *IEEE International Symposium on Requirements Engineering*. 84–91.
[8] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-Based Feature Models and Their Specialization. *Software process: Improvement and practice* 10, 1 (2005), 7–29.
[9] Åsa G. Dahlstedt and Anne Persson. 2005. *Engineering and Managing Software Requirements*. Springer, Chapter Requirements Interdependencies: State of the Art and Future Challenges, 95–116.
[10] Maya Daneva and Andrea Herrmann. 2008. Requirements prioritization based on benefit and cost prediction: A method classification framework. In *Euromicro Conference on Software Engineering and Advanced Applications*. 240–247.
[11] Juan M. Carrillo de Gea, Joaquin Nicolás, José L. Fernández Alemán, Ambrosio Toval, Christof Ebert, and Aurora Vizcaíno. 2012. Requirements engineering tools: Capabilities, survey and assessment. *Information and Software Technology* 54, 10 (2012), 1142 – 1157.
[12] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. 2012. An Efficient Diagnosis Algorithm for Inconsistent Constraint Sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)* 26, 1 (2012), 53–62.
[13] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2014. Variability in Software Systems — A Systematic Literature Review. *IEEE Transactions on Software Engineering* 40, 3 (2014), 282–306.
[14] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.
[15] Laura Lehtola, Marjo Kauppinen, and Sari Kujala. 2004. Requirements Prioritization Challenges in Practice. In *International Conference Product Focused Software Process Improvement*. 497–508.
[16] Varvana Myllärniemi, Mikko Ylikangas, Mikko Raatikainen, Jari Pääkkö, Tomi Männistö, and Timo Aaltonen. 2012. Configurator-as-a-service: tool support for deriving software architectures at runtime. In *Working IEEE / IFIP Conference on Software Architecture, Companion Volume*. 151–158.
[17] Klaus Pohl. 1996. *Process-centered requirements engineering*. Wiley.
[18] Norman Riegel and Joerg Doerr. 2015. A systematic literature review of requirements prioritization criteria. In *Working Conference on Requirements Engineering: Foundation for Software Quality*. 300–317.
[19] Gunther Ruhe and Moshood Omolade Saliu. 2005. The art and science of software release planning. *IEEE Software* 22, 6 (2005), 47–53.
[20] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. 2007. Generic semantics of feature diagrams. *Compututer Networks* 51, 2 (2007), 456–479.
[21] Mikael Svahnberg, Tony Gorschek, Robert Feldt, Richard Torkar, Saad Bin Saleem, and Muhammad Usman Shafique. 2010. A systematic review on strategic release planning models. *Information and Software Technology* 52, 3 (2010), 237 – 248.
[22] Rahul Thakurta. 2017. Understanding requirement prioritization artifacts: a systematic mapping study. *Requirements Engineering* 22, 4 (2017), 491–526.
[23] Juha Tiihonen, Mikko Raatikainen, Varvana Myllärniemi, and Tomi Männistö. 2016. Carrying Ideas from Knowledge-Based Configuration to Software Product Lines. In *International Conference on Software Reuse*. 55–62.
[24] Andreas Vogelsang and Steffen Fuhrmann. 2013. Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study. In *IEEE International Requirements Engineering Conference (RE)*. 267–272.
[25] He Zhang, Juan Li, Liming Zhu, Ross Jeffery, Yan Liu, Qing Wang, and Mingshu Li. 2014. Investigating dependencies in software requirements for change propagation analysis. *Information and Software Technology* 56, 1 (2014), 40–53.

## 6 APPENDIX

```
 {
    "project": {
        "id": "ABC",
        "name": "ABCExample",
        "specificRequirements": ["A", "B", "C"]
    },
    "requirements": [
      {
          "id": "A",
          "name": "Requirement a",
          "effort": 2
      },
      {
          "id": "B",
          "name": "Requirement b",
          "effort": 1
      },
      {
          "id": "C",
          "name": "Requirement c",
          "effort": 3
      }
    ],
    "releases": [
      {
          "id": 1,
          "capacity": 5,
          "requirements": ["A", "B", "C"]
      }
    ],
    "dependencies": [
      {
          "dependency_type": "requires",
          "from": "B",
          "to": "C"
      }
    ]
}
```

```
{
  "response": {
    "consistent": false,
    "diagnosis": [
      [
        {"requirement": "B"}
      ]
   ]
  }
}
```

**Figure 2: A simple example of requirements data as JSON that describes the requirements, the assignments of requirements to a release and a dependency (top). The requirements model is be checked for consistency and diagnosis for inconsistency caused by B exceeding the effort capacity of a release is proposed (bottom).**