Generating Configuration Models from Requirements to Assist in Product Management – Dependency Engine and its Performance Assessment

Configuration workshop 2018





Juha Tiihonen Iivo Raitahila Mikko Raatikainen Alexander Felfernig Tomi Männistö

Introduction: background

- H2020 OpenReq: Intelligent Recommendation Decision Technologies for Community-Driven Requirements Engineering
- Requirement management systems (RMS) applied in requirements engineering
- Issue tracker systems becoming increasingly popular
  - especially large-scale, globally distributed open source projects
  - tens of thousands requirements, bugs and other interdependent items
- RMSs provide primarily with support for individual requirements
  - Individual dependencies, even advanced constraints, can be expressed

Introduction: motivation

- Requirement dependencies are one of the key concerns
  - e.g.: requirements prioritization, release planning

→ When deciding what to do or not to do, need to deal with

  - different requirement options and alternatives and their properties
  - constraints / dependencies
  - consequences
- Holistic analysis over all requirements respecting the dependencies and properties is not well supported

→Issue trackers are suboptimal e.g. for product or release management

Idea: Dependency Engine

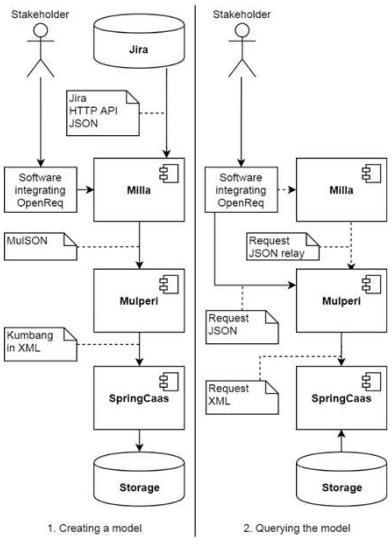
- Proof-of-concept Dependency Engine for holistically managing requirements as a single model.
- Automatically map requirements and their existing individual dependencies into the Kumbang variant of feature models
  - enables utilization of existing research on feature analysis
- A feature model is further mapped into a constraint satisfaction problem (CSP)
- The user can experiment with different configurations of requirements
  - While the system maintains the consistency of dependencies and resource constraints

Research questions

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

Architecture and workflows of the Dependency Engine

- Workflows
  - Creating a model from requirements data
  - Making queries against the model
- REST-type services
  - Java Spring framework
- Choco 4.0 Open-Source Java library for Constraint Programming
- Potential bottlenecks
  - network and external system
  - requirement model generation
  - feature model generation
  - feature model to CSP
  - CSP solving



Data: DataSet *JiraData from Qt Company*

- Qt provides a framework for cross-platform development
- Qt's Jira contains Issues and Bugs that can be considered as requirements
  - dependencies
  - attributes with constant values, such as priority and status.
- Qt's Jira contains 18 different projects
  - The biggest, QT-BUG contained 66,709 issues (April 2018)
- 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
  - Too small for real challenges, except to note that Choco default search strategy was not suitable for our purposes

Data: *SynData1* and *SynData2 datasets*

- *SynData1*
  - 450 models with permutations of the
  - # requirements (from 100 to 2000),
  - a 'requires' dependency 0% to 75% of requirements
  - an optional subfeature with one allowed feature 0% to 75% of requirements
  - 0 to 5 attributes
    - each attribute has two possible values, e.g., 10 and 20
- *SynData2*
  - 60 test cases like above, but
  - no subfeatures
  - 1 or 2 attributes with a fixed random value from 1 to 100

SynData2: example requirement

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

Tests: Initial trials and initial search strategy

- Complete a configuration of requirements with a minimal number of additional requirements with pareto-optimizer
  - *JiraData*

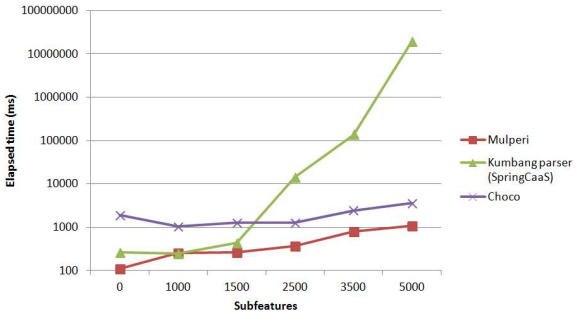
  ⇒ Choco default search strategy not good enough

  ⇒ Adopted minDomLBSearch

Strategy	Optional features	Mandatory features	Attributes	Solutions	Time
default	14	0	0	60	130 to 300 ms
default	20	0	0	1046	11600 to 11900 ms (unacceptable)
minDomLBSearch	14	0	0	1	120 to 170 ms
minDomLBSearch	20	0	0	1	150 to 190 ms
minDomLBSearch	235	0	2 per feature	1	160 to 200 ms
minDomLBSearch	118	117	2 per feature	1	400 to 650 ms

Tests: model generation

- *SynData1*
- Kumbang parser exhibits significant fatigue with over 1500 requirements
- Otherwise almost linear behavior



Tests: Requirement configuration (1)

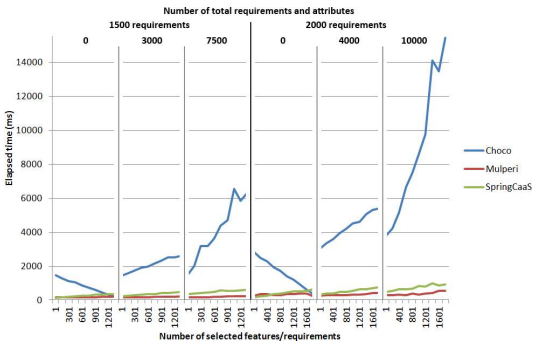
- *SynData1*
- Optimized auto-completion
  - find a (close to) minimum configuration with user-selected features (1% - 91%)
- Increasing number of dependencies eases Choco's inference burden

Table 7. Minimum, maximum and median test cases of the configuration phase, $SynData1$ dataset

Requirements	Dependencies	Subfeatures	Attributes	Requirements in request	Mulperi (ms)	SpringCaaS (ms)	Choco (ms)	Total (ms)
100	10	0	0	1	9	10	4	23
100	10	20	200	91	10	21	8	39
100	0	0	500	1	27	54	14	95
500	0	100	0	451	34	79	19	132
500	100	200	0	101	33	61	71	165
500	0	0	2500	201	34	266	549	849
750	0	150	0	601	60	122	55	237
750	0	150	1500	376	63	156	344	563
750	375	0	3750	601	70	273	1614	1957
1000	750	1000	0	1	129	133	126	388
1000	500	0	2000	401	90	252	777	1119
1000	0	0	0	1	1344	414	1788	3546
1500	0	0	0	1351	186	363	179	728
1500	0	0	3000	751	185	364	2159	2708
1500	300	0	7500	1351	263	715	9087	10065
2000	0	0	0	1801	237	619	334	1190
2000	200	0	4000	801	297	515	4445	5257
2000	1000	0	10000	1801	573	1056	16464	18093

Tests: Requirement configuration (2)

- The more user-selected requirements, the slower
- Non-linear growth in terms of Choco time when problem size grows



Test: Optimised release configuration under resource constraint

- Global sum (resource) constraints that reflect different scenarios of release planning
- *SynData2*
  - 100, 500, 750, 1500, 2000 features
  - Varying number of user requirements
- 60s, 10s and 3s timeouts
  - effect of allowed time on the solvability
  - get an impression on the quality of solutions

Constraint#	Constraint
0	$\sum attribute1 > 1000$
1	$\sum attribute1 = 1000$
2	$\sum attribute1 < 1000$
3	$\sum attribute1 > 1000 \wedge \sum attribute1 < 2000$
4	$\sum attribute1 > 1000 \wedge \sum attribute2 < 2000$

Test: Optimise for minimum number of requirements under constraints

Const-raint#	#attri-butes	Optimization goal
0, 1, 3	1	Simulate achieving desired utility with a minimal number of requirements to implement. Minimizes the number of requirements.
2	1	(Autocomplete, redundant for *SynData2*.)
4	2	Minimize the number of requirements to implement under constraints of minimum utility and maximum effort.

Table 8. Minimization of the number of features. Results of 60 second timeout compared with 10 and 3 second timeouts and the custom algorithm. Lower number of features in a solution is better. $Test$: the type of the testcases, $\#a$: the number of attributes in the test cases. $\overline{N}$: the average number of features in the minimal solution found with the 60s timeout. $N_{=_{10}}$: the number of test cases where 10s timeout search finds the same number of features than the 60s version. $N_{>_{10}}$: the number of test cases where 10s timeout search includes a larger number of features than the 60s version. $\overline{\Delta_{N_{10}}}$: the average number of additional features included in a solution found with 10s timeout when compared to the 60s search. $\overline{\Delta N_{10}}(\%)$: the average percentage of additional included features found by the 10s version. $N_{=_3}$, $N_{>_3}$, $\overline{\Delta_{N_3}}$, $\overline{\Delta N_3}(\%)$: 3 second timeout versions analogously as 10s. The corresponding figures of the custom algorithm are presented similarly: $N_{=_c}$, $N_{<_c}$, $N_{>_c}$, $\overline{\Delta_{N_c}}(\%)$. Note that $N_{<_c}$ is the number of cases where the custom algorithm finds a better solution. $SynData2$ dataset.

$Test$	$\#a$	$\overline{N}$	$N_{=_{10}}$	$N_{>_{10}}$	$\overline{\Delta_{N_{10}}}$	$\overline{\Delta N_{10}}(\%)$	$N_{=_3}$	$N_{>_3}$	$\overline{\Delta_{N_3}}$	$\overline{\Delta N_3}(\%)$	$N_{=_c}$	$N_{<_c}$	$N_{>_c}$	$\overline{\Delta_{N_c}}(\%)$
0	1	14.3	14	11	0.48	3.38%	7	8	0.60	4.92%	4	7	19	2573.33%
1	1	14.3	14	11	0.52	3.63%	7	8	0.67	4.92%	6	4	20	106.67%
2	1	1.0	25	0	0.00	0.00%	15	0	0.00	0.00%	30	0	0	0.00%
3	1	14.3	13	12	0.52	3.65%	7	8	0.73	4.92%	4	7	19	620.00%
4	2	14.4	17	8	0.32	2.26%	6	9	0.67	4.40%	0	0	30	1456.67%

Test: optimize for maximum sum (utility)

Constraint#	#attributes	Optimization goal
0, 1, 2, 3	2	Simulate maximisation of utility under resource constraint: Maximize sum of attribute2

Table 9. Maximization of sum of attribute 2 (e.g. utility). Results of 60 second timeout compared with 10 and 3 second timeouts. The custom algorithm is excluded. Higher sum of attribute 2 ($a2$) is better. $Test$: the type of the testcases, $\#a$: the number of attributes in the test cases. $\overline{N_{60}}$: the average number of features in a solution found with the 60s timeout. $\overline{a1_{60}}$, $\overline{a2_{60}}$:average value of attribute 1 / attribute 2 in solutions identified with 60s timeout, respectively. $N_{10,a2,<}$ and $N_{10,a2,=}$: the number of test cases where 10s timeout search finds a lower / same same sum of attribute 2 than the 60s version, respectively. $\overline{\Delta N_{10}(\%)}$: the average difference (percentage) between number of included features between 60s and 10s timeout versions. $\overline{\Delta a2_{10}(\%)}$: the average difference (percentage) between sum of attribute 2 of included features between 60s and 10s timeout versions. 3 second timeouts are analogous, $SynData2$.

$Test$	$\#a$	$\overline{N_{60}}$	$\overline{a1_{60}}$	$\overline{a2_{60}}$	$N_{10,a2,<}$	$N_{10,a2,=}$	$\overline{\Delta N_{10}(\%)}$	$\overline{\Delta a2_{10}(\%)}$	$N_{3,a2,<}$	$N_{3,a2,=}$	$\overline{\Delta N_3(\%)}$	$\overline{\Delta a2_3(\%)}$
0	2	976	48979	49255	0	25	0.0%	0.0%	0	15	0.0%	0.0%
1	2	33.2	1000	1821	24	1	-2.1%	-4.1%	15	0	-2.9%	-5.9%
2	2	33.5	998	1831	21	4	-3.4%	-4.6%	13	2	-5.7%	-6.5%
3	2	53.6	1998	2860	23	2	-2.1%	-3.2%	15	0	-3.6%	-4.5%

Test: Optimised release configuration under resource constraint

- The optimization task is computationally intensive.
  - Difficult for the solver: is this an optimal solution?

  → Solving practically always ends with a timeout
- Largest test cases (2000 features) and varying numbers of requirements are solvable with 60s timeout
- 10s timeout handles all cases except 2000 features (100 to 1500).
- 3s timeout is only applicable to cases with 100, 500 and 750 features
- When a solution is found, the versions with a lower timeout value remain almost as good as solutions obtained with 60s timeout.
- Search strategy *bestBound(minDomLBSearch())* improved performance

Conclusions

- Solutions without optimization are easy for solvers such as Choco
- Optimization: search strategy matching the problem crucial
- Surprise: "black-box" activityBasedSearch and Choco default domOverWDeg do not provide satisfactory performance
- The prototype engine easily scales to around 2000 requirements
  - even with optimization is desired
- Seems that the approach can scale into managing the requirements of large software projects, even for interactive use.
- Very large software projects, such as QT-BUG remain challenging
  - potential to constrain the problem size
  - separate bugs and real requirements, ignore historical non-active data, ignore requirements without dependencies
- 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

Thank you for your attention!

Questions?