Progress report 1 for 2017-Team-Up-7 INTiMALS

Company Name: RainCode Labs
Project reference: 2017-TEAM-UP-7
"INTiMALS: Intelligent Modernisation Assistance for Legacy Software"
Reporting Period: Report No. 1 for the period 01/01/2018 to 01/10/2018

General status of the project at M10

The following table summarizes the project's overall progress with respect to the entire 3-year work plan.

WP Name	Expected Status	Expected %	Actual Status	Actual %	Comments
WP1 A shared language-parametric meta-model for software repositories and software patterns	2 deliverables complete out of 6 in total	2/6 = 33%	2 out of 6 deliverables complete	33%	• Deliverable **P1.1** complete. • Deliverable **P1.2** complete.
WP2 Frequent subgraph mining algorithms for uncovering software patterns	1 deliverable complete, 1 deliverable first version out of 6 in total	1,5/6 = 25%	1 deliverable complete, 1 deliverable first version	25%	• Deliverable **P2.1** delivered. • First version of deliverable **R2.2** complete, based on jointly-authored **academic paper** submitted to the NIER track of the IEEE ICSE conference.
WP3 Towards a pro-active modernisation assistant through matching algorithms for mined patterns	1 deliverable complete out of 3 in total	1/3 = 33%	1 deliverable complete	33%	• Deliverable **P3.1** complete.
WP0 Evaluation of developed technology on legacy systems at Raincode Labs	1 deliverable complete, 1 deliverable first version out of 6 in total	1,5/6 = 25%	1 deliverable complete, 1 deliverable delayed	1/6=17%	• Deliverable **P0.1** complete. WP2 results required additional effort of grammar extraction for Cobol. • A first version of deliverable **R0.2** was due in M7, and its final version in M12. We expect to deliver the

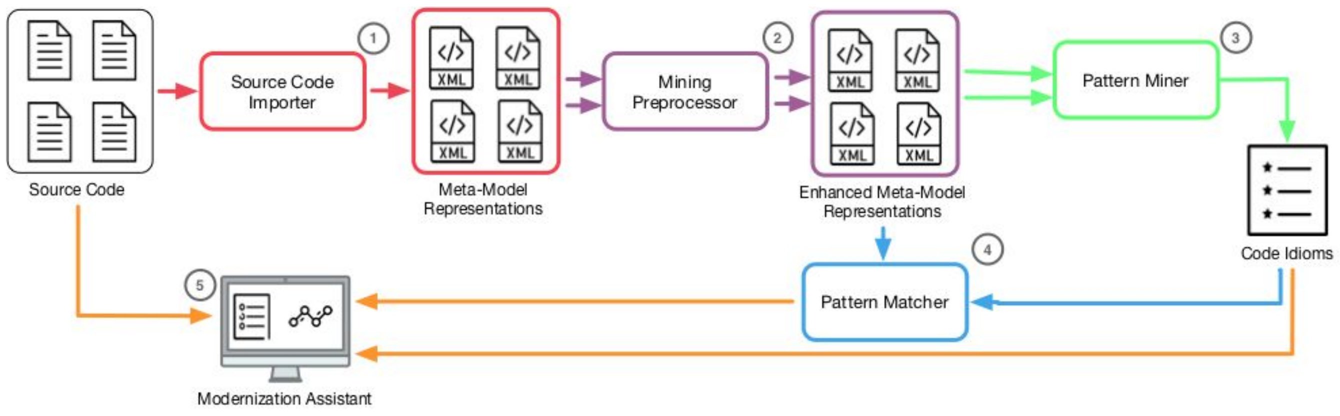
					final version of R0.2 in M14.

Main achievements of the period

For completeness' sake, we briefly repeat the project's overall objective from the project proposal:

The project's overall objective is to develop an *intelligent modernisation assistant* for legacy software systems. The assistant pro-actively recommends engineers source code *modernisation actions* by comparing their ongoing development efforts with insights gained by treating software systems as data, in particular their source code and development history. The assistant will draw its intelligence from continuously mining for previously-unknown patterns both in the current state of the system's source code and structure (e.g., so-called programming idioms, coding conventions, library usage protocols) and in past and ongoing changes made to this source code (e.g., so-called systematic edits, repetitive changes, ...). The modernisation recommendations made by the assistant will appear increasingly informed as it refines or uncovers more previously-unknown patterns in the source code and version repositories it mines. The success of the modernisation assistant —its apparent learning ability— hinges on the quality of the pattern mining algorithms it incorporates.

The activities towards realizing this intelligent modernisation assistant have almost entirely progressed according to plan. Each successive year of the project focuses on an increasingly difficult kind of patterns the assistant draws its intelligence from: idioms and coding conventions mined for in syntactic information (Y1), usage patterns mined for in semantic information (Y2), and edit patterns mined for in historical information (Y3).



In what follows, we briefly describe the architecture of the INTiMALS assistant at M10 of Y1. As depicted in the Figure above, the assistant is structured as a pipeline comprising five main components:

1. **Source Code Importer.** A first challenge for our modernisation assistant is to accommodate multiple (legacy) programming languages. Indeed, it would not be economical if a new version of had to be re-implemented for every language or even language dialect it is applied to. To address this issue within our framework, the meta-model of our assistant defines a language-agnostic abstract syntax tree format (AST) for source code. The purpose of the source code importers is thus to transform programs in a given language to their representation in this format.
2. **Mining Preprocessor**. Before they are passed to the pattern miner, the ASTs may be preprocessed in order to enhance the mining process. Different preprocessing steps may be applied, depending on what is being mined for. For example, when considering naming conventions as part of the mining, one preprocessor can split identifiers into a subtree based on camelcase or based on underscores.

Another example would be mining at a granularity of procedure-level entities and hence first removing elements at finer granularities like statements or (module-level) variable declarations.

3. **Pattern Miner.** The pattern miner is responsible for extracting idiomatic code patterns, taking the preprocessed ASTs as input. We are currently exploring the use of frequent graph mining algorithms, though other mining algorithms may be tried in the future. The most popular frequent graph mining algorithms are developed for trees and undirected graphs, although standard algorithms produce a (too) large amount of patterns (as discussed in section V). Thus, an important component of our pattern miner is the definition of the heuristics and constraints used during the mining process, so as to avoid discovering redundant or useless patterns. We are currently exploring what heuristics work best for different kinds of idioms, and how to represent these heuristics in an idiom- and language-agnostic way.
4. **Pattern Matcher**. The pattern matcher is responsible for finding all AST subtrees that match the patterns extracted by the miner. While these ASTs are already known to the pattern miner, we may want to apply post-processing steps to the patterns that are found, e.g., to further generalise them such that the patterns are more widely applicable. The pattern matcher is then needed to find matches of these modified patterns. Another application of the pattern matcher is that, when a pattern was mined in one project, the pattern matcher can now match this pattern against any other project. The tool is designed to be language-parametric and is based on code templates. A template is a concrete snippet of source code, in which some parts can be replaced by wildcards or meta-variables. It is also possible to attach so- called “directives” to parts of the snippet, which can affect the semantics of the pattern to match in various ways.
5. **Modernisation Assistant**. The modernisation assistant provides a GUI that allows to inspect all patterns uncovered by the pattern miner, and their matches, both as text and as graphs. The engineer is presented a list of patterns with their pattern size, support, confidence, and type of root AST node. A specific pattern can be selected for inspection showing an overview of pattern matches in the source code as well as concrete source code snippets highlighted according to the structure of the pattern.

The modular architecture of our framework is key to achieving our research objectives. For example, given a new programming language we mainly need to provide a new **Source Code Importer**. However, it is likely that we also need to define or configure a **Preprocessor** specific to the kind of idioms we want to mine for in that language, and that we need to adapt the heuristics and constraints used by the **Pattern Miner**. But the general pipeline and algorithms would remain the same. Similarly, if we would like to explore alternative or more advanced pattern mining algorithms, in a language-agnostic way, this could be done mostly by replacing the **Pattern Miner**.

We now report on the current state of the implementation of our framework, some preliminary results, as well as some of the challenges we have faced:

1. **Source Code Importer.** We currently have importers for COBOL and Java. The former is pragmatic custom code that is able to process the entire NIST COBOL 85 compliance test suite as well as the code for a variety of industrial legacy systems. The latter uses the Eclipse Java meta-model and is able to successfully produce ASTs for all source code in QUAATLAS: a refined subset of the Qualitas Corpus of Java programs. The importers also produce a description of the grammar of the language that is used by the miner. Again, the Java importer uses the Eclipse Java meta-model to produce this grammar, whereas for the COBOL importer this is custom code.
2. **Mining Preprocessor**. For the moment, we have only implemented a preprocessing component that is able to split the identifiers contained in a node into a subtree based on camelcase or the dash/underscore convention. When using that preprocessor, instead of considering identifiers as

similar only when they are equal, identifiers can be matched at a finer-grained level based on the similar keywords they contain.

3. **Pattern Miner.** Our pattern miner implements theFreqT frequent subtree mining algorithm. Although we have found that pure FreqT can indeed be used for mining idiomatic code patterns, it does have some limitations such as being highly time consuming and generating a large amount of patterns as well as redundant patterns. To tackle these problems, we have been exploring various customisations of the FreqT algorithm. Most significantly, we have worked on using the grammar itself as a source of input to drive the pattern search. As a result, we have managed to reduce the execution time of FreqT significantly, and to limit the number of discovered patterns. Although we have not completed a full empirical study yet, many of the discovered patterns seem to correspond to relevant code idioms.
   To achieve these results, we had to use a variety of heuristics and constraints. However, selecting the appropriate constraints to apply is not a trivial task since it seems to depend partly on the language and on the kinds of patterns one wants to find. Even though those constraints can easily be configured for other languages and other kinds of patterns, it is less obvious how to choose the appropriate constraints for legacy languages that are less well-known, or when we do not know upfront what kind of patterns we are looking for. A particular challenge of our current research therefore remains how to efficiently search for and evaluate interesting and surprising patterns.
4. **Pattern Matcher.** Currently, our pattern matcher is able to match precise syntactic patterns. In the future, we plan to support anomaly detection including the on-demand detection of partial matches for a given mined pattern. To facilitate inspection by a software engineer, the pattern matching algorithm should also quantify its results by indicating the extent to which a partial match corresponds to a given pattern.
5. **Modernisation Assistant.** Based on the output of the miner and pattern matcher, the modernisation assistant is able to visualise patterns, matches and their corresponding source code. Despite its seemingly summarising role, it was useful from very early on in the project to explore mining results and let human users interpret them. It has consequently been a driving force in customising the miner and matcher to provide results that are more straightforwardly interpretable by a modernisation engineer. For example, we found that since patterns are subtrees with "holes", it is important for highlighted source code to show which part of the source code corresponds to the subtree and which part of that code corresponds to a "hole". Hence, the pattern matcher should include this information in each match.

The following table provides a detailed overview of the status of all corresponding prototypes (deliverables starting with P) and reports (deliverables starting with R) scheduled for the entire duration of the project. Deliverables in bold were due by M10 according to the work plan.

ID	Description	Due	Status
P1.1	Prototype implementation of the meta-model and an accompanying importer for syntactic information	**M03**	complete
P1.2	Prototype implementation of its representation for coding conventions & idioms	**M03**	complete
P1.3	Extension of meta-model and importer with support for semantic information	M15	
P1.4	Prototype implementation of its representation for usage patterns	M15	
P1.5	Extension of meta-model and importer with support for change information	M27	
P1.6	Prototype implementation of its representation for edit patterns	M27	
P2.1	Frequent subgraph mining system for discovering patterns in attributed graphs	**M06**	complete
R2.2	Report on its application to mining for coding conventions & idioms in syntactic information	**M06** **M12**	M06 version complete
P2.3	Adaptation of the frequent subgraph mining algorithm to support usage patterns	M18	

R2.4	Report on its application to mining for usage patterns in semantic information	M24	
P2.5	Adaptation of the frequent subgraph mining algorithm to support edit patterns	M30	
R2.6	Report on its application to mining for edit patterns in change information	M36	
P3.1	Prototype offering assistance for program comprehension in the form of a browser for mined patterns	**M09**	complete
P3.2	Prototype offering assistance for anomaly detection in the form of a browser for partial matches of mined patterns	M21	
P3.3	Prototype offering assistance for program modernisation in the form of edit recommendations that complete a partial match for a mined pattern	M33	
P0.1	Prototype of legacy software importer (syntactic information) for the INTiMALS meta-model	**M04**	complete
R0.2	Report on the evaluation of INTiMALS' support for mining coding conventions & idioms in legacy systems	**M07-M12**	delayed to M14
P0.3	Extension (semantic information) of the legacy software importer for the INTiMALS meta-model	M16	
R0.4	Report on the evaluation of INTiMALS' support for mining usage patterns	M24	
P0.5	Extension (version information) of the legacy software importer for the INTiMALS meta-model	M28	
R0.6	Report on the evaluation of INTiMALS' support for edit patterns in UC3	M36	

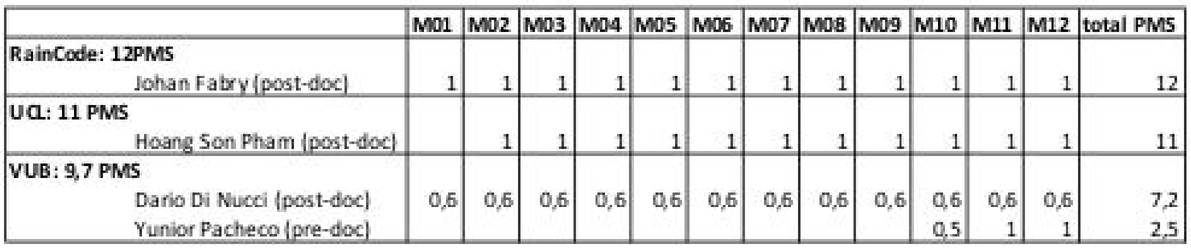
Problems encountered :

Deliverables delayed

Overall, the project has progressed almost entirely according to plan. Only deliverable **R0.2** has incurred a minor delay that will not impact the execution of the project. This deliverable is to report on the in-vivo evaluation by RainCode engineers of the modernization assistance provided by the INTiMALS assistant. A first version of this report was due in **M07**, and the final version in **M12**. We expect to deliver the final version in **M14**. The delay is due to an innovation in the pattern mining algorithm developed for deliverable **P2.1**. To reduce its search space, the algorithm takes the grammar of the programming language for which it is mining as additional input. The scope of the legacy software importer had to be increased correspondingly to include producing a grammar for Cobol in the format expected by the mining algorithm, which was not planned for.

Staffing

We also experienced some minor delays in staffing, as illustrated by the following table.



	M01	M02	M03	M04	M05	M06	M07	M08	M09	M10	M11	M12	total PMS
RainCode: 12PMS													
Johan Fabry (post-doc)	1	1	1	1	1	1	1	1	1	1	1	1	12
UCL: 11 PMS													
Hoang Son Pham (post-doc)		1	1	1	1	1	1	1	1	1	1	1	11
VUB: 9,7 PMS													
Dario Di Nucci (post-doc)	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	0,6	7,2
Yunior Pacheco (pre-doc)										0,5	1	1	2,5

However, these delays have not affected the overall progress with respect to the work plan:

1. The RainCode engineer, a PhD graduate with expertise in software renovation and visualisation, was able to start as planned on January 1st.
2. The UCL researcher, a PhD graduate with expertise in pattern mining algorithms, started on February 1st.

3. VUB was able to staff a 0.6 FTE part-time researcher, a PhD graduate with expertise in machine learning applications in software engineering, from January 1st onward. for the last three months of Y1, VUB has complemented this senior part-time researcher with a more junior full-time researcher, a PhD student with expertise in mining usage patterns. In addition, the PI has stepped in pro-bono to ensure the delivery of all due prototypes.

Overall vision of results and prospects for the future (To be completed only at the end of the project)

Implementation of the business plan and valorisation

Johan Fabry, the Raincode Engineer, presented an informal demonstration of our toolchain and the current version of the pattern browser (deliverable P3.1) during SCAM 2018, the 18th IEEE International Working Conference on Source Code Analysis and Manipulation, held end of September. Note that Johan was also program co-chair of the engineering track at this conference. Furthermore, attending different conferences has allowed the divulgation of Raincode's participation in the project to the research community.

Evolution of the company

Overall the company is having a normal positive evolution. Worth mentioning in particular is that Raincode Labs recently opened a new office in Bangalore, India, thus expanding their operations into one of the most important markets in the world.