Final report BIG SOFTWARE Project MERIT
NWO project 628.008.002

Project partners: Prof.dr.ir. J.F. Groote, Eindhoven University of Technology, Eindhoven.
Prof.dr. J. Vinju, Centre for Mathematics and Computer Science, Amsterdam.
Eindhoven University of Technology, Eindhoven.
D.-J. Swagerman. Philips, Best.

Scientific goal/achievements

This research proposal focuses on the problem of re-engineering legacy software towards the model driven software engineering (MDSE) approach to software engineering. Within the area of Model Driven Software Engineering there is a tendency towards higher level, domain specific, descriptions of software which have a clear benefit in terms of production effort, maintainability, analysability and quality. Within the industry, government and service providers, large bodies of software exist, often written in a variety of languages, constructed over a period of many decades, having been influenced by many programming paradigms that were being developed, while the software was constructed and adapted.

The research question of this project is whether it is possible to migrate as efficiently as possible the large body of existing software into appropriate descriptions in domain specific languages. The research is applied on parts of software from the healthcare domain, provided by Philips Best, as Philips has substantial interest in a concise, high quality code base.

Within the project two PhD. students have been appointed, drs. R.T.A. (Rodin) Aarssen and O.Z.O. (Omar) al Duhaiby. Both students were partly present at Philips, Best, and partly at Eindhoven, and in case of Rodin Aarssen partly at the Mathematical Centre in Amsterdam. Unfortunately, the Covid-19 crisis reduced the presence at these locations, and discussions continued mainly electronically.

The project started out with on the one hand trying to learn state machine models from actual Philips software. The particular system chosen was software part of the control system of X-ray equipment. It was remarkably hard to start learning, because the interfaces to the software were not very explicitly defined. With great help within Philips, and strongly inspired by the test scripts, state machine learning could commence. Parts of the behaviour of the software could be learned and transformed into a state machine. There were two valuable lessons to be learned, summarised in [2]. The first one is that the behaviour of actual software is large, probably too large to be learned into a state machine. Probably more importantly, actual legacy software contains a lot of behaviour which was not intended or useful, and in particular most likely not used. For instance, the software under investigation could be activated a number of times, without ever being deactivated, but the number of activations would lead to observable differences in the behaviour. From this the interesting conclusion could be drawn that most likely most legacy software allows for non-intended, non-used behaviour on a massive scale. This behaviour can cause unintended side effects when the software is used in different ways, and such behaviour should not be learned and transformed into models of the software.

But parts of the software could be learned, and those parts provided insight in the behaviour that no other existing method can provide. For instance, the learning of the software revealed that the interface of some learned software components provide interface primitives, at a too low level, controlling the internals of the software, raising the question whether the designers used the right level of abstraction while designing the software components.

The extracted models were all partial. Analysing the behavioural properties of such models turned out to be easy, and in principle code could be generated from it. However, the essence of the problem is that the models were a too partial representation of the full behaviour for this to make sense.

Therefore, we changed the question to learn behavioural models in parts. The first question to be addressed was whether it is possible to recognize the module structure of software while learning where we cannot see how the components communicate. Unfortunately, this has been answered negatively [1]. The result is that if we observe behaviour modulo branching bisimulation, then a system can partition its observable actions over the components in any arbitrary way. If the observer cannot see how the components communicate internally, it cannot determine which actions belong to which component. Modulo divergence preserving branching bisimulation it is fundamentally impossible to distribute behaviour in an arbitrary way, and in principle some aspects of the distribution of actions over the internal structure can be observed. However, this is not practical as it requires to observe whether a system performs unbounded interal activity.

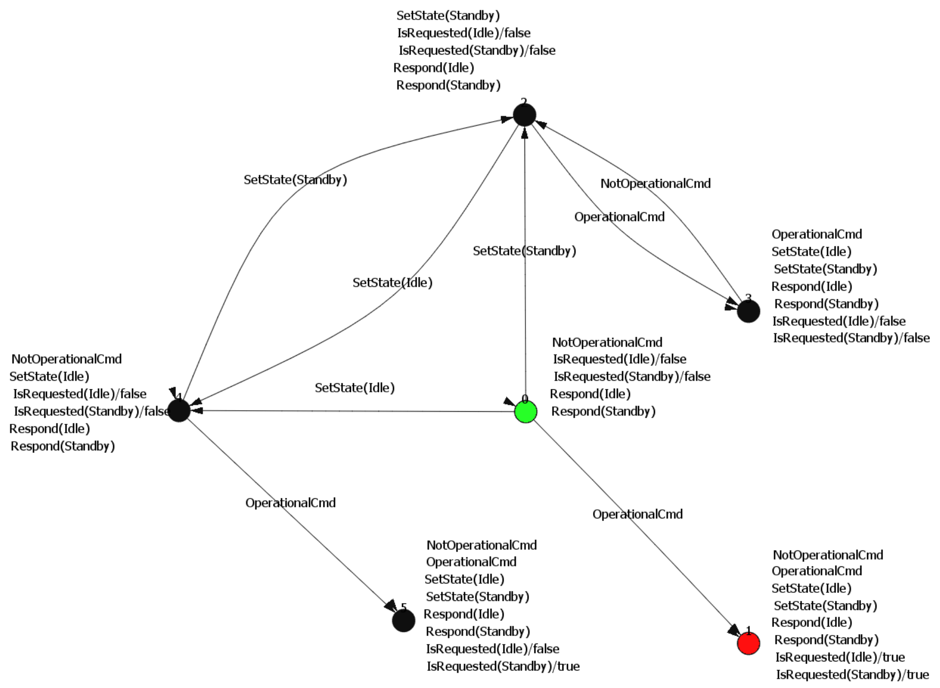
This led to a next question to learn larger systems. Suppose -- not unreasably -- that we know the internal structure of a system, and in particular, assume that we know that actions in different components do not interact with each other. Is it possible to learn the behaviour of the system. The answer to this is affirmative, but it is necessary to know the internal structure, as well as observe the communications between the components [3].

The question is whether it is possible to learn a system if the internal structure is not known, but we can observe the communications between the components of the system. This research is still ungoing.

The other part of the research focussed on static analysis. In order to be able to perform complete static analysis of software at Philips an extended C++ parser has been constructed that allows to parse the complete C++ including all extensions made available by the Microsoft Visual Studio compiler, as such extensions are heavily used within the software code at Philips. This was an extensive amount of work to lay the groundwork for further transformations.

Using this framework, information could be extracted out of the software at Philips, and transformation of software into other forms were possible. The question to be addressed is how to formulate the information extraction and transformation in an optimal way. Such meta programs can become very substantial, and if they become large, they tend to become error prone. It is most useful to specify meta programming with patterns matching the source code as much as possible. These patterns are then transformed into Raskal internal transformation patterns. This work is reported in [4] and [5]. Using this the flow of information through various software systems could be neatly visualised, allowing software designers within Philips to remove unnecessary buffer structures between components, without introducing software errors.

This work has been refined into Separator Syntax Trees, a formalism between concrete syntax and abstract syntax that allows to formulate transformation on source code, preserving many of the relevant aspects in the source code that tend not to be incorporated in abstract syntax trees [6]. For instance, maintaining comments in transformed programs, and leaving them at a human readable spot in transformed code is an essential aspect of practical legacy renewal, as such comments are essential for next generation programmers to find their way in the code.



A simple and partial learned automaton from X-ray devices at Philips.

Using these transformation techniques an attempt has been done to extract state machines out of the source code. At some places, the X-ray software contains explicitly formulated state machines. Unfortunately, they turned out to have been programmed and changed by various programmers in the rich and expressive language of C++, which means that although fragments of the state machines can be retrieved, it is impossible to retrieve the full machine, let alone constellations of cooperating state machines, that occur in the code.

Current work is still going on where the provided means are being used to remove COM interfaces from various components and replace them with plain C++ function calls. The transformations can be well formulated in the Separator Syntax Tree formalism. However, -- and these are the unpleasantries of actual software -- the software to be transformed does not use the COM interfaces according to prescription at all places, and routes information in ad hoc ways. Part of the transformation is to first replace such ad hoc component communication by a more structural approach.

This leads us to the answers we obtained regarding the research question:

(RQ) *Given a legacy software component(s), how can we automatically extract an abstract model of the software system's behaviour and incrementally transform that model to obtain a model with comparable behaviour and sufficient detail to enable code generation.*

The major conclusion is that it is certainly possible to extract information using source code analysis and automata learning from legacy software. However, legacy software is so intrinsically complex and versatile, being the result of decades of programming, that the intrinsic complexity of legacy software makes it very hard to extract nice and clean models out of it. The software is full of unintended behaviour and is written in a very wide range of programming styles, which is a much more formidable barrier to automatic transformation than foreseen.

It is useful to formulate the results of the project in terms of the workpackages.

Workpackage 1: *From static code to action-based models*. "The expected result of WP 1 is not only reusable fully automatic transformations from source code to complete databases of intermediate representations, but also new query formalisms and initial experiments with queries for extracting action-based models." We managed to transform full specifications into Raskal, ready for further transformation, and developed query formalisms to transform the results.

Workpackage 2: *From run-time behaviour to action-based models.* The purpose of the second workpackage was to obtain action-based models. Such models were obtained, but they quickly turned out to become not only very sizeable, but they revealed that the software did exhibit a substantial amount of behaviour that was not relevant and intended. Focus has been directed towards obtaining more usable models, for instance by learning larger behaviour in parts. As it stands we can conclude that the learned behaviour is very useful, and provides a unique way of getting insight in the software, but complete behavioural models are as yet completely out of reach. This can be compared by other approaches, where only very small systems were learned (software on smart cards) or where models were learned from software generated from models (ASML). The lesson is that legacy software is even more complicated than foreseen.

Work package 3: *Analysis and transformation of the action-based models.* The obtained behaviours action based models were all so small that they could easily be analysed and transformed, for instance using the mCRL2 toolset, which can deal with models orders of magnitude larger than those that can be obtained using state machine learning. However, as the state machines that were learned only fragmentarily represented the true behaviour in the legacy software, such analyses and transformations had limited value.

Work package 4: *From action-based models to industrial MDSE environment.* This part of the project has not been addressed as far as the learned action-based models are concerned as they insufficiently represent the behaviour of the system under study. But this was not the most challenging part of the proposal. We are capable of generating executable code from mCRL2 models. Using software transformation we are currently transforming large portions of code using Raskal, showing that this is actually possible.

Final words. The project shows that learning models out of legacy software is possible, but given the sheer size, irregularity and complexity of legacy software, learning complete systems is outside the realm of current techniques, and we expect that it may take a decade or more before learning legacy software is a viable alternative to current techniques. However, generated partial models give a unique view on the behaviour of programs, that would be hard to obtain elsewhere. Source code transformation representing the code in a Raskal model is far more viable, but also suffers from the irregular structure of legacy code. Once models are available, analysing them, as well as generating executable models is well within technological reach, certainly for the generated partial models.

The appointment of Rodin Aarssen has been prolonged with one year until September 2021 in order to finish his PhD thesis, which was delayed due to a long lasting and severe illness of prof. Vinju. Omar al Duhaiby will finish his appointment mid October 2020.

[1] Omar al Duhaiby and Jan Friso Groote. Distribution of behaviour into parallel communicating subsystems. In Jorge A. Pérez and Jurriaan Rot, editors, Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019, volume 300 of EPTCS, pages 54–68, 2019. Also appeared as arXiv: 1908.08213.
[2] Omar al Duhaiby, Arjan J. Mooij, Hans van Wezep, and Jan Friso Groote. Pitfalls in applying model learning to industrial legacy software. In Tiziana Margaria and Bernhard Steffen, editors, Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV, volume 11247 of Lecture Notes in Computer Science, pages 121–138. Springer, 2018.
[3] Omar al Duhaiby and Jan Friso Groote. Active Learning of Decomposable Systems. In 2020 IEEE/ACM 8th International Conference on Formal Methods in Software Engineering (FormaliSE), pages 1–10, Seoul, Republic of Korea, October 2020. ACM, New York, NY, USA. 2020.
[4] Rodin T.A. Aarssen, Jurgen J. Vinju, Tijs van der Storm. Concrete Syntax with Black Box Parsers. The Art, Science, and Engineering of Programming. 3(3): 15, 2019.
[5] Rodin T.A. Aarssen, Jurgen J. Vinju, Tijs van der Storm. Concrete Syntax with Black Box Parsers. Archiv: 1902.00543, 2019.
[6] Rodin T.A. Aarssen, Tijs van der Storm. High-fidelity metaprogramming with separator syntax trees. In: PEPM 2020: Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. Pages 27–37, 2020.