A Research Agenda for Formal Methods in the Netherlands

Marieke Huisman

Wouter Swierstra

Eelco Visser (editors)

Technical Report UU-CS-2019-004
July 2019

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Contents

A Research Agenda for Formal Methods in the Netherlands

Marieke Huisman
University of Twente
m.huisman@utwente.nl

Eelco Visser
Technical University of Delft
e.visser@tud.nl

Abstract

On September 3 and 4, 2018, we organized a meeting on formal methods research in the Netherlands. Goal of the meeting was to create a Dutch formal methods community, to increase awareness of each other's activities, and to find common grounds for collaborations. All researchers working on formal methods in the Netherlands were invited to contribute a 2-page abstract with their vision on the future of formal methods research. This document bundles these visions.

1 Goal of the Workshop

Research on Formal methods in the Netherlands is doing well. Success stories on the use of mechanized theorem proving, proof assistants, model checking, high-level programming languages, program verification etc. are ample. Our techniques are becoming mature and have the potential to be applied in industry.

However, research on formal methods is not "done" yet: there are still many interesting challenges ahead, for example concerning scalability and usability of formal methods. To be able to continue along this path, we need to improve visibility and funding opportunities for formal methods research.

To achieve this, we need a strong formal methods research community, that actively collaborates to lobby for research in this area.

To build up this community, we organized a two day workshop in the Lorentz Center in Leiden on Monday 3, and Tuesday 4 September 2018. During this workshop, we brought together the key scientists in formal methods in The Netherlands.

The purpose of this meeting was to better understand our individual goals and challenges, and to discuss about the potential and ideas for research in formal methods for the next 10 years. Goal of the workshop was to achieve a better understanding of each other's research areas, and to find common challenges and grounds for collaborations, as such collaborations can further strengthen the Dutch formal methods research community. Furthermore, we also believe that being better informed about each other's research is a first step towards creating a community with active lobbying power.

A Research Agenda for Formal Methods in the Netherlands, September 3–4, Lorentz Center, Leiden, The Netherlands
2018.

2 Workshop Contents

The workshop was filled with short presentations about current and future research, with ample time for discussions. During the workshop, presentations were given by: Marieke Huisman (UT), Jan Friso Groote (TU/e), Jurriaan Hage (UU), Marko van Eekelen (OU), Eelco Visser (TUD), Robbert Krebbers (TUD), Sung-Shik Jongmans (OU), Jorge A. Perez (RUG), Frank S. de Boer (UL/CWI), Wouter Swierstra (UU), Sebastiaan Joosten (UT), Erik Poll (RU), Gabriele Keller (UU), Sebastian Erdweg (TUD), Herbert Bos (VU), Anton Wijs (TU/e), Anna-Lena Lamprecht (UU), Jeroen Keiren (OU), Michel Reniers (TU/e), and Herman Geuvers (RU).

3 This Document

All participants were asked to contribute a two page abstract with their vision for the future of formal methods research. This document bundles the visions of all participants, as well as the vision of some researchers in formal methods in Netherlands that were not able to join the meeting.

Formal Methods for Better Software in Computational Science

Anna-Lena Lamprecht
Universiteit Utrecht
The Netherlands
a.l.lamprecht@uu.nl

1 Introduction

Science across all domains is increasingly data-driven and computational, and thus the **correctness of research software** becomes increasingly critical for the validity of scientific results. Yet formal methods and software quality assurance in general have not received great attention in this context in the past.

Software engineering practices in science can differ widely from what is common in industry. In a recent review article [3], Heaton et al. report, for example, that scientific software developers generally do not use a formal software development methodolody, do not produce proper requirements specifications, and do not treat design as a distinct step in the development process. Furthermore, they find that the effectiveness of the testing practices currently used by scientific software developers is limited, that **testing is more complicated** for scientific development than traditional software development since the correct results are often unknown, and similarly that the lack of suitable test oracles or comparable software makes validating scientific software difficult. There are many ways that defects can enter software, but scientists often suspect that any problems in the result of software result from scientific theory rather than from implementation. Experimental validation is often impractical because developers lack the information they would prefer to use to validate the software.

Moreover, even true wet lab scientists are often not only end users of software developed by more or less professional software engineers. Since research topics are often extremely specific, software for the specific purpose is frequently not readily available and so scientists **need to develop software themselves**. In many cases this concerns just "glue code" to connect existing components and automate the processing of large numbers of data sets, but nevertheless correctness is important. Generally, however, scientists (many of them **self-taught programmers**) do not test systematically, let alone verify their programs formally. Code is often intended to be "Kleenex software" for one-time use, with the developer being its only user and publishing the results being the only purpose that matters, so there is no incentive for putting effort into further quality considerations. Of course, it happens frequently that the software is nevertheless used again later on, modified and extended, but never tested, validated, verified and released properly.

There are many possibilities how formal methods can help to address these issues. In the following I outline possible applications in the area of scientific workflows and in relation to dynamically typed languages like Python and R, which relate to own previous work and experiences with scientists from different domains.

2 Scientific Workflows

The concept of **scientific workflows** has become popular in the scientific community over the last years [1]. Workflows provide a systematic way of describing data-intensive computational processes, and provide an interface between domain specialists and computing infrastructures. Several workflow management systems exist that support scientists in their construction and execution, often making use of graphical representations of the workflow models. Historically grown and driven by practice, a variety of **workflow languages** exists, but slowly the community is progressing to standardized scientific workflow languages, with the Common Workflow Language (CWL) and the Workflow Description Language (WDL) currently enjoying great popularity.

Being inherently component-oriented and model-driven, scientific workflows are an almost natural target for the application of **model-based formal methods**, such as model checking and synthesis [4, 5]. **Model checking** can in this context act as a "spell checker", monitoring the workflow under development and alerting the user in case it violates certain constraints. These can be generic, rather technical issues like mismatching data types (e.g., a certain component is not able to work on the data format provided by its predecessor) and other static analyses, or more domain-specific, semantic constraints that express knowledge about the workflow's purpose or rules of best practice, like, for example, that all experimental data will eventually be stored in the project repository, that unexpected analysis results will always lead to an alert, or chargeable services will not be called before permission is given by the user.

Synthesis goes a step further: Instead of using constraints only to check if a workflow is correct, it starts with constraints and automatically assembles a workflow that is thus correct by construction. In addition to general constraints like those sketched above, this requires the user to express their intents about the workflow in a suitable way. Intents like "Take a BLAST result as input and finally produce a phylogenetic tree." or "Having a single (genetic) sequence, I want to find similar sequences and get information about their evolutionary relationship." can be expressed in formal logic and processed by a synthesis framework.

The application of model checking and synthesis as sketched above requires to tailor methods and frameworks to the specifics of the domain, making use of **domain-specific ontologies** that facilitate working with a controlled vocabulary at a level of abstraction that feels natural to the user. In particular the bioinformatics community has made a lot of progress in this regard in the last years, with the EDAM ontology providing a rich domain-specific vocabulary for the description of bioinformatics operations, data types and formats, from which tool annotation and constraint-driven workflow synthesis are straightforward next steps [6].

Future work in this area will in the first place have to address issues of **usability** (accessible specification languages, usable tools, integration into the scientists' software ecosystems, high-quality domain models) and **scalability** (mitigation of state explosion effects, scaling up ontological modelling and semantic annotation) in collaboration with the targeted application domains.

3 Python and R

Formally trained computer scientists are often irritated by the fact that Python and GNU R appear to be the **most popular** languages among data scientists of all disciplines. They are regarded easy to learn, but most importantly in practice, large amounts of libraries are available for these languages that readily provide frequently needed functionality. That is, they are often simply the fastest means to get data analyzed (and results published). Considerations about correctness, code quality and maintainability come second.

Python and R are **dynamically typed languages**, that is, variables are declared without a concrete type, and get assigned concrete types only at runtime. Such code is usually faster to write, but when a program gets more complex, runtime errors caused by mismatching data types are the norm and can be difficult to fix (especially by programmers without much experience). Even worse, it can happen that type mismatches remain undetected and lead to wrong results: Many libraries like those mentioned above transparently perform type casts to make programming simpler for the user, by "guessing" the correct type. Not suprisingly, sometimes this goes wrong, and suddenly the data has changed, leading to alternative results, but no runtime error or warning alerts the user. If the obtained values are really far away from the expected, the scientist might get suspicious, but when they remain in feasible ranges or even give a sensational result, they just go through to publication.

As of version 3.6 Python includes a **static type checker** to support programmers to catch common type-related errors already while coding. It remains to be seen, however, if this feature will be taken up by the scientific community. Likely programming in Python, R and similar languages would benefit greatly from IDEs that are aware of **domain-specific type systems** related to the domain models sketched above, or even **dependent types** [7], and based on that provide support to **diagnose and fix type errors** as described in [2].

4 Conclusion

In the development of scientific software more attention needs to be devoted to correctness and other aspects of software quality. I am convinced that formal methods can make a great contribution there, up to providing intuitive, user-level frameworks for correct-by-construction development of scientific software. In particular, this will require close collaboration with scientists from the targeted domains in order to adapt the general methods to their concrete needs and to capture and formalize the required domain knowledge adequately. The Netherlands has strong and internationally recognized communities in e-Science and Semantic Web research, so there is a lot of potential for joint efforts and impact in this area.

References

[1] M. Atkinson, S. Gesing, J. Montagnat, and I. Taylor. Scientific workflows: Past, present and future. *Future Generation Computer Systems*, 75:216 – 227, 2017.

[2] J. Hage. The usability of static type systems. In *A Research Agenda for Formal Methods in The Netherlands*, 2018.

[3] D. Heaton and J. C. Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207 – 219, 2015.

[4] A.-L. Lamprecht. *User-Level Workflow Design - A Bioinformatics Perspective*, volume 8311 of *Lecture Notes in Computer Science*. Springer, 2013.

[5] A.-L. Lamprecht, T. Margaria, and B. Steffen. Supporting Process Development in Bio-jETI by Model Checking and Synthesis. In *Semantic Web Applications and Tools for Life Sciences (SWAT4LS 2008). CEUR Workshop Proceedings*, volume 435, 2008.

[6] M. Palmblad, A.-L. Lamprecht, J. Ison, and V. SchwÃďmmle. Automated workflow composition in mass spectrometry-based proteomics. *Bioinformatics*, page bty646, 2018.

[7] W. Swierstra. Programming with dependent types. In *A Research Agenda for Formal Methods in The Netherlands*, 2018.

Scalable, Non-Intrusive Formal Verification via Modularisation and Parallelisation

Anton Wijs
A.J.Wijs@tue.nl
Eindhoven University of Technology, Eindhoven, The Netherlands

1 Introduction

The development of complex, concurrent software is error-prone and costly. Formal methods can play a key role in making the software development process more structured and transparent, but in order to do so, formal verification results should be produced efficiently and frequently during the development process. In my research, I focus on the application of *model checking* [1]. Model checking is a push-button technique to formally verify the functional correctness of hardware and software models. It is performed by systematically exploring the state space implied by the model. The main drawback of model checking is the state space explosion problem: a linear growth of the model tends to lead to an exponential growth of the state space. Although traditionally, this meant that computer memory was practically the only bottleneck, these days, with large amounts of memory at our disposal, the (lack of) scalability of the run time is often hindering our ability to reason about models in a reasonable amount of time. For this reason, I focus on two seemingly disconnected topics, which on closer inspection in fact strengthen each other, and are in my opinion key topics to focus on in order to make formal methods practically appealing for industrial use:

1. The *modularisation* of verification problems into subproblems, such that verification tasks do not need to be performed monolithically.
2. The *parallelisation* of verification algorithms to make the verification of (sub)problems as efficiently as possible.

2 Modularisation of model checking

To make the verification of (designs of) industrial-sized software practically feasible, I focus in my research on various approaches to break down the verification task into smaller subtasks that can each ideally be solved totally independently. The approaches that I focus on mostly are:

1. *Compositional model checking.* As many software designs consist themselves of multiple components or concurrent processes, these can be analysed individually and the results can gradually be combined, ideally leading to a result about the entire system by using fewer resources than the straightforward, monolithical model checking approach would need. In recent work, we have studied what the key structural characteristics are of software designs that determine whether compositional model checking can be applied successfully or not [2, 4]. We have also worked on incrementally searching for a counter-example to a given property by combining concrete parts of the model with overapproximations of the remainder of the model, and gradually making the overapproximation part more concrete [11].
2. *Model transformation verification.* In model driven software engineering, software designs are typically not constructed in one single design step. Instead, they usually result from a design process in which an initial abstract model is made more and more concrete via a number of steps (see Figure 1). These steps can be automated by means of *model transformations*, which makes the design process more systematic since the application of transformations is both repeatable and reversable. In my research, I focus on formalising such model transformations, and investigating to what extent functional properties of the transformations themselves, as opposed to the models they are applied on, can be determined. This has lead to a (in turn formally verified) technique to efficiently identify whether a transformation *preserves* given functional properties [3]. In other words, given *any* model that satisfies a property φ, application of such a property preserving transformation is guaranteed to produce a new model that also satisfies φ. Verifying the development workflow in this way is likely to be much more efficient than each time verifying a new model resulting from a transformation application, since the definition of the transformations tends to be very small, while the models they produce can be arbitrarily large. In the future, we plan to work on investigating *property maintenance*, i.e., we plan to consider the possibility of evolving the properties along with the models themselves.

In addition, I also work on the construction of suitable Domain Specific Languages to conveniently design software systems, and on verifying the final step in the model driven software engineering workflow (Figure 1), namely the construction of code based on a detailed design of the system, for instance see [13]. For this purpose, a collaboration with the team of prof. Marieke Huisman of the University of Twente has recently been initiated, to apply their code verification techniques to verify whether produced code adheres to the

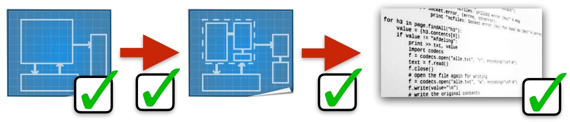


Figure 1. Verified development steps from initial model to code in a model-driven development workflow

specification given in the form of the software design that serves as input to the code generator.

3 Parallelisation of model checking

One way to improve the run time of model checking is by exploiting the computing power of modern parallel architectures. Graphics processing units (GPUs) have a lot of potential in this respect: they can run thousands of threads in parallel and can offer a speed-up of several orders of magnitude. GPUs tend to have much less memory than modern computer systems, but the current trend is that this amount doubles every few years. Hence, it is interesting to investigate to what extent model checking algorithms can be adapted to run on GPUs. In the last few years, GPUs have been successfully used for several model checking procedures, and in my work, I have studied the applicability of GPUs to perform on-the-fly model checking [6–8, 12], minimisation of state spaces [5], and structural analysis of state spaces [9, 10]. By carefully designing implementations of the algorithms, speedups over hundreds of times can be achieved compared to traditional, sequential computations. Practically, this can reduce computation times from days or weeks to mere seconds or minutes, which can suddenly make repeated application of these methods feasible and non-intrusive in the software development workflow. A prime example is the tool GPUExplore [7, 12], which is the first to completely perform on-the-fly model checking on the GPU.

The work mentioned above all focusses on *explicit-state* model checking, in which each possible state of the system design is handled individually. Recently, I started a project to investigate the potential for GPUs to accelerate *symbolic* model checking techniques, in particular the ones relying on SAT solving. Initial results are promising, in which we have demonstrated that GPUs can effectively impact SAT solving by employing them to simplify SAT problems before trying to solve them. Not only can such a simplification phase be performed much faster, the computational power of GPUs also allows to push the simplification further within a shorter amount of time, thereby achieving higher quality simplifications that positively impact the solving time.

References

[1] C. Baier and J.-P. Katoen. *Principles of model checking.* MIT Press, 2008.

[2] S. de Putter and A. Wijs. Compositional Model Checking is Lively. In *Proc. 14th International Conference on Formal Aspects of Component Software (FACS 2017)*, volume 10487 of *Lecture Notes in Computer Science*. Springer, 2017.

[3] S. de Putter and A. Wijs. A Formal Verification Technique for Behavioural Model-To-Model Transformations. *Formal Aspects of Computing*, 30(1):3–43, 2018.

[4] S. de Putter and A. Wijs. To Compose or Not to Compose, That is the Question: An Analysis of Compositional State Space Generation. In *Proc. 22nd International Symposium on Formal Methods (FM 2018)*, 2018 (accepted for publication).

[5] A. Wijs. GPU Accelerated Strong and Branching Bisimilarity Checking. In *Proc. 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, volume 9035 of *Lecture Notes in Computer Science*, pages 368–383. Springer, 2015.

[6] A. Wijs. BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In *CAV, Part II*, volume 9780 of *LNCS*, pages 472–493, 2016.

[7] A. Wijs and D. Bošnački. GPUexplore : Many-Core On-the-Fly State Space Exploration Using GPUs. In *TACAS*, volume 8413 of *LNCS*, pages 233–247. Springer, 2014.

[8] A. Wijs and D. Bošnački. Many-core on-the-fly model checking of safety properties using GPUs. *STTT*, 18(2):1–17, 2015.

[9] A. Wijs, J.-P. Katoen, and D. Bošnački. GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components. In *Proc. 26th International Conference on Computer Aided Verification (CAV 2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 310–326. Springer, 2014.

[10] A. Wijs, J.-P. Katoen, and D. Bošnački. Efficient GPU Algorithms for Parallel Decomposition of Graphs into Strongly Connected and Maximal End Components. *Formal Methods in System Design*, 48(3):274–300, 2016.

[11] A. Wijs and T. Neele. Compositional Model Checking with Incremental Counter-Example Construction. In *Proc. 29th International Conference on Computer-Aided Verification (CAV 2017), Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 570–590. Springer, 2017.

[12] A. Wijs, T. Neele, and D. Bošnački. GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking. In *FM*, volume 9995 of *LNCS*, pages 694–701. Springer, 2016.

[13] D. Zhang, D. Bošnački, M. van den Brand, C. Huizing, B. Jacobs, R. Kuiper, and A. Wijs. Verification of Atomicity Preservation and Deadlock Freedom of a Generic Shared Variable Mechanism Used in Model-To-Code Transformations. In *Proc. 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2016) - Revised Selected Papers*, volume 692 of *Communications in Computer and Information Sciences*, pages 249–273. Springer, 2017.

Tools, tools, tools

Arend Rensink
Department of Computer Science
University of Twente
Enschede, The Netherlands
arend.rensink@utwente.nl

Abstract

We argue that a major obstacle to the (more) widespread adoption of formal methods is the scarcity of adequate tool support. To change this, we need to rethink our values about what constitutes good research in formal methods, and move from a too-exclusive emphasis on papers to a new status quo where tool development and support is regarded as normal.

Keywords Tool support

1 Tooling is immature

What should we do to improve the adoption of formal methods in practice? We should package them for use by those who can benefit from them. This especially means: provide reliable tool support.

On the positive side: the past decade has seen a growing realisation of the importance of tooling in formal methods research. There is now a number of established competition-type events where tools with similar functionality are compared with one another. There is also some more attention toward making formal methods experiments reproducible, in the sense of making the underlying tool chain, the input data and the raw results available: several conferences have established procedures for submitting and reviewing the tool support underlying a submitted paper.

On the negative side, we are still plagued by the concept of “PhD-ware:” the prototype tools created in the course of a PhD project all too often stop being maintained after the candidate has received his degree — maybe even with good reason, if the prototype was never developed far enough to run more than the few samples it was used for. Our submission, reviewing and publication processes all revolve around *papers* — there typically is not even the possibility of submitting other electronic resources, be they data or code, though they may have been instrumental in achieving the results. This sends out the message that the ladders may be discarded once the heights have been scaled. I believe that is the wrong message, as those ladders (or probably a less rickety version of them) are essential to get formal methods to be adopted.

If, as the call for contributions for this event states, “success stories on the use of [name your favourite formal method] are ample,” and “our techniques are becoming mature and have the potential to be applied in industry,” then the next step is to proceed from stories to books and from potential to actual application. The availability of tools is a necessary requirement that has not yet been met.

2 Tooling is hard

It is easy to bring up arguments why we (as in “the academic community”) will never be able to produce tools with sufficient maturity and support to enable a breakthrough in their use in practice. The most common arguments of this kind are that developing tools is not the core business of researchers — indeed, that they may in fact not be very qualified to do so — and that “sufficient maturity” is an unattainable goal, especially where it concerns the guarantee of support for users that may come to rely on the tool in question. Such support requires that there is a person in charge of mainaining the tool, at least to the level of answering questions and (where necessary) fixing severe bugs. That contrasts with the composition of typical research groups in computer science, most particularly formal methods groups, where any budget to be had for personnel preferably goes to researchers — PhD candidates, postdocs, lecturers — and very little if any long-term lab support exists.

I think a more interesting and fruitful discussion can be engendered by turning the question around: let’s not ask ourselves why our tooling currently does *not* meet standards that make their adoption feasible, but instead how we can create a viable ecosphere in which there *is* room for meeting such standards in the future. This may very well require redefining or broadening and expectations about what constitutes leading research: to name just one thing, already alluded to above, our exclusive focus on papers (published in prestigious conferences and journals) as the end-all and be-all of academic research may have to be put in question.

3 Tooling is undervalued

This brings me to my final observation, which for me touches upon the root cause of our generally low tooling standards. In contrast to other technical disciplines, computer science (and again formal methods in particular) does not have a valued tradition of laboratory infrastructure as the foundation upon which new results are achieved. *The concept of a laboratory hardly exists.* Reasons may be thought of why this is so — the essence of our subject matter being in software rather than hardware comes to mind — but I believe this observation itself to be key, and worth thinking about. If only it were an accepted normality that formal research methods can only exist when closely coupled to a laboratory, properly staffed with technicians, where all research ideas are by default put to the rigour of being properly tool-supported, this would reflect a very different system of values according to which we judge research quality.

Such a system of values would have a price tag. Infrastructure is not cheap. Again, this is an accepted fact in other technical disciplines — why not in computer science? If it remains the case that there is only funding for pure research positions, and any mention of technical support is a detractor in the chances for acceptance of a project proposal, then our tooling will forever remain underdeveloped. Nothing comes for free.

Let us not forget to acknowledge that there *are* clear cases where the creation and maintenance of a tool or tool set has greatly contributed to the reputation and visibility of research groups. This goes to show that there is no contradiction between great research and great tool support, and such tools have demonstrably contributed to the acceptance of the methods thus made accessible to those who can benefit from them. However, such cases are currently the exception rather than the rule, and exist in spite of, rather than thanks to, what we primarily value as good research in formal methods. I propose to challenge that status quo.

Fast and Safe Linguistic Abstraction for the Masses

Eelco Visser
Delft University of Technology

Abstract Language workbenches support the high-level definition of (domain-specific) programming languages and the automatic derivation of implementations from such definitions. The mission of language workbench research is to increase the level of abstraction of language definitions and expand the range of tools that can be generated automatically from language definitions. In this note, I give an overview of research into language workbenches at TU Delft and the perspective of future research.

Linguistic Abstraction Software engineering is the act of encoding intent into programs. A mismatch between the concepts being encoded and the intent to be expressed leads to unnecessary software complexity. Linguistic abstraction is the process of turning software design patterns into language constructs in order to support expressing intent at the right level of abstraction [15]. A linguistic abstraction provides notation, static checking, and implementation strategies that take the understanding of the domain into account. For example, in the IceDust language for data modeling, a programmer directly expresses how derived values are computed from base values, without being concerned about implementation techniques for caching computed values and incrementally recomputing values upon data changes; an implementation of an efficient incremental re-computation strategy is generated automatically from an IceDust program [3–6]. While many DSLs exist, the process of developing DSLs should become more systematic and requires further industrial case studies.

Linguistic Abstraction for the Masses A domain-specific programming language can be beneficial for its programmers, but can be expensive to develop. Language workbenches are tools to reduce the cost of DSL development by generating language implementation from language definitions [1, 2]. Spoofax is a language workbench developed in the Programming Languages group at TU Delft [7, 16]. Spoofax has been applied to DSLs in industry (Oracle, Océ) and academia.

Linguistic Abstraction for Linguistic Abstraction Rather than implememting languages in a general purpose language, language workbenches provide domain-specific abstractions for language development, i.e. linguistic abstractions for linguistic abstraction. Spoofax is a test bed for the exploration of the design of high-level declarative meta-languages for various aspects of language definition. A recent example is the development of the Statix DSL for the specification of type systems [11, 12]. The DSL is based on a novel approach to the formalization of name binding in programming languages using *scope graphs* [8]. A scope graph is an abstraction of a program that represents its binding facts. This abstraction allows the definition of a language independent calculus to define name resolution in programs, giving rise to a range of language independent tools.

Fast Linguistic Abstraction Traditionally, language workbenches define *code generators* to realize the implementation of DSLs programs. The disadvantage of this approach is that the semantics of the language is defined through a translation relation in terms of a target language. We are investigating whether it is possible to define directly the semantics of languages and derive an implementation from such definitions. The DynSem meta-language is designed for the high-level declarative specification of the operational semantics of programming languages [13]. DynSem specifications can be executed directly using a meta-interpreter that interprets a DynSem specification with respect to an object program, which gives rise to two levels of interpretation and significant overhead. A promising direction of research is to use partial evaluation techniques to specialize the application of a meta-interpreter [14] to provide both a fast turn around time for language design experiments and at the same time get a fast run time for the language under specification.

Safe Linguistic Abstraction In addition to producing implementations, the proper design of a programming language requires proofs that a design satisfies properties such as type soundness and semantics preservation of transformations. Traditionally, proving such properties is separated from language implementation, leading to proofs for only subsets of a language and a divergence of specification and implementation. We are investigating meta languages that enable the automatic verification of such properties. A first proof of concept automates the verification of type soundness. A complication in type soundness proofs is the alignment of the binding of names statically and at run time. We have developed a systematic approach to formalizing this alignment by relating scopes in scope graphs to frames in frame heaps [9]. By encoding a scope graph based type system as an intrinsically typed abstract syntax signature, type checking the evaluation rules of a definition interpreter against such a signature entails type soundness of the language under definition [10].

Future Work In this abstract I have described several proof of concept components of a development environment for creating new (domain-specific) programming languages from declarative specifications that are safe by construction and that have fast runtimes. Fully realizing the promise of these proofs of concept requires scaling up the techniques to more advanced programming languages and large code bases.

References

[1] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches - conclusions from the language workbench challenge. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2013.

[2] Sebastian Erdweg, Tijs van der Storm, Markus VÃűlter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[3] Daco Harkes, Danny M. Groenewegen, and Eelco Visser. Icedust: Incremental and eventual computation of derived values in persistent object graphs. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[4] Daco Harkes and Eelco Visser. Unifying and generalizing relations in role-based data modeling and navigation. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västeras, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2014.

[5] Daco Harkes and Eelco Visser. Icedust 2: Derived bidirectional relations and calculation strategy composition. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[6] Daco C. Harkes, Elmer van Chastelet, and Eelco Visser. Migrating business logic to an incremental computing dsl: a case study. In David Pearce 0005, Tanja Mayerhofer, and Friedrich Steimann, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*, pages 83–96. ACM, 2018.

[7] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, SiobhÃąn Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[8] Pierre NÃľron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015.

[9] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[10] Casper Bach Poulsen, Arjen Rouvoet, Andrew P. Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018.

[11] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016.

[12] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018.

[13] Vlad A. Vergu, Pierre Néron, and Eelco Visser. Dynsem: A dsl for dynamic semantics specification. In Maribel FernÃąndez, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[14] Vlad A. Vergu and Eelco Visser. Specializing a meta-interpreter: Jit compilation of Dynsem specifications on the Graal vm. In Eli Tilevich and Hanspeter MÃűssenbÃűck, editors, *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*. ACM, 2018.

[15] Eelco Visser. Understanding software through linguistic abstraction. *Science of Computer Programming*, 97:11–16, 2015.

[16] Guido Wachsmuth, Gabriël Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014.

Formal Methods for Security?

Erik Poll
Digital Security group
Radboud University
erikpoll@cs.ru.nl

This position paper sketches some opportunities in applying formal methods for security, more specifically security of software.

A killer application for formal methods?

At first sight – and even at second sight – security looks like an interesting application area for formal methods.

One reason for this is that security flaws have a higher impact than more harmless bugs. This might justify or even require the extra effort to invest in using formal methods. Indeed, the highest levels of certification using the Common Criteria security evaluation standard require the use of formal methods.

Another reason is that some security problems are orthogonal to – or at least largely independent of – the functionality of a system. Indeed, security vulnerabilities can be seen as 'anti-functionality': functionality that is unintentionally provided, also to attackers, which should not be available at all. Given that writing complete functional specifications is hard, and totally unfeasible in most cases, concentrating on partial specs that ensure some generic safety properties problems might provide a better return of investment, also because such specs could be re-used across applications.

Unfortunately, things are not so simple. Security properties can be tricky to specify. Indeed, attackers can be very creative in finding and exploiting new loopholes. A common and natural way to specify security properties is in a 'negative' way, by saying that something – some type of attack – is not supposed to be possible. For example, a web application should not be vulnerable to SQL injection or XSS. List of these negative properties are useful in testing, as they suggest negative tests (i.e. test-cases which are supposed to fail, by triggering some error response), but they are not immediately helpful in construction. Moreover, these lists of negative properties are typically incomplete. They only address a limited set of known potential problems, not all potential problems.

Moreover, sometimes security problems arise because it turns out fundamental assumptions about programs can be broken by an attacker, invalidating the very abstractions that we use in formal methods to reason about programs. Classic examples here are fault attacks (for example the Rowhammer attack to flip some bits in DRAM memory) or information leaking through side-channels (for example the Spectre and Meltdown timing attacks on modern CPUs). Of course, our formal models could be refined to accomodate these low-level attacks, but at the (high) cost of extra complexity.

Security functionality $\neq$ secure functionality

When investigating at the security of software, it is natural to focus on the security functionality, i.e. the functionality specifically intended to provide some security guarantees, such as access control checks or security protocols like SSL/TLS. Such functionality is obviously security-critical.

However, while this may suggest rewarding aspects or components to investigate, it is dangerous to fall into the trap of thinking that such security functionality is the only or even the most important area to look for problems. Not only the security functionality has to be secure: *all* functionality needs to be, as security vulnerabilities can lurk in any line of code that can be triggered by input controlled by the attacker. The bulk of security bugs is not in code specifically aimed at achieving some security goal, but is in more mundane functionality, say the parsing of PDF files or the rendering of some graphical format, with Flash as the most notorious example.

LangSec: back to basics?

The paradigm of LangSec (language-theoretic security)[1] provides very good insights into the root causes of the overwhelming majority of security flaws, namely bugs in handling input, which typically boil down to bugs in the parsing and processing input languages and formats, rather than bugs in the application logic.

One problem is with the input languages themselves: they are typically overly complex, too expressive, and poorly – and informally – specified. To make matters worse, there are many of these languages, at every level of the network and software stack, and they can be combined or nested. A further problem here is that the code to process these languages is typically hand-coded, and not obtained using parser generation.

Ironically – or embarrassingly, for the computing science community – theories for formal language definitions and parser generation are some of the oldest and most established areas in formal methods. Still, somehow the whole world is still writing long prose documents to specify languages and protocols, and then hand-coding parsers – often in memory unsafe languages like C or C++, where the potential security

[1]See also http:\langsec.org, esp. [1], or [7] for a more recent entry point into the LangSec literature.

impact of flaws is the biggest (namely remote code execution). There is a huge opportunity here to provide better notations and tools to prevent all this misery. Or maybe these already exist, and we should do a better job in training people – incl. our students – on how to use them?

One step further from formal specs and associated code generation for parsers and pretty printers would be domain-specific programming languages to support different input formats and languages as first class citizens, as envisioned in Wyvern [8].

Security Testing & Model Extraction

The past decade has seen a lot of fruitful interaction between formal methods and testing, also in security testing. An interesting trend is the use of formal methods, notably symbolic/concolic execution, for security testing [6, 11] or even going one step further and actually develop exploits as in the angr tool [10]. If we cannot get developers to use formal methods, maybe we should concentrate efforts on getting security testers and hackers to use formal methods? (Of course, with more robust parser code that has generated from formal language specs, as we argue for above, it should be harder to find security flaws ...)

Test techniques can also provide a way to obtain formal specs from implementations. Given the difficulty of obtaining formal models this is an interesting direction of work. Existing fuzzers can already reverse-engineer input formats [2, 3], and state machine inference can be used to extract security-relevant behaviour from code [9].

All such formal techniques for security testing or model inference could be combined with machine-based learning or AI approaches, to improve results and/or the level of automation.

Practical information flow

Information flow properties are an interesting class of security properties. Information flow can be used to track potential leakage of confidential information or to track the flow of tainted input to places where such input may do damage. Research on information flow has a long history, dating back to the 1970s [4], and ad-hoc information analyses are implemented in code analysis for security flaws – aka Static Application Security Testing (SAST), by tools such as Coverity, Checkmarx, or Fortify, but flexible and practical approaches to express and enforce information flow for programs in popular programming languages (e.g. [5] for Java) are still rare and not commonly used.

New (and safer) programming languages, new opportunities?

One positive development for security in recent years has been the advent of new programming languages – Rust, D, Go, Swift, Nim, ... – where safety is very much a design goal. Some of these languages are specifically aimed for low-level programming and might become viable, widely-used, and safer alternatives for C/C++ and then reduce the prevalence of memory corruption problems.

The advent of these new languages is a double-edged sword. An advantage is that they are designed to be more amenable to formal analysis and have some security guarantees built in at the language level. A downside is that new languages require new tools. Building and maintaining good formal methods tools is a major bottleneck, so here the advent of new languages is bad news. For researchers these new languages represent new research opportunities. This may be good news, if this new research gets us further, or bad news, if this research is merely repeating and recycling the same old ideas without getting us further.

References

[1] 2013. LangSec: Recognition, Validation, and Compositional Correctness for Real World Security. (2013). USENIX Security BoF hand-out. Available from http://langsec.org/bof-handout.pdf.

[2] J. Caballero, H. Yin, Z. Liang, and D. Song. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *CCS'07*. ACM, 317–329.

[3] P.M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. 2009. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 110–125.

[4] D.E. Denning and P.J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513.

[5] W. Dietl, S. Dietzel, M.D. Ernst, K. Muşlu, and T.W. Schiller. 2011. Building and Using Pluggable Type-checkers. In *ICSE'11*. ACM, 681–690.

[6] P. Godefroid, M.Y. Levin, and D. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20.

[7] F. Momot, S. Bratus, S.M. Hallberg, and M.L. Patterson. 2016. The seven turrets of Babel: A taxonomy of LangSec errors and how to expunge them. In *Cybersecurity Development (SecDev)*. IEEE, 45–52.

[8] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. 2014. Safely composable type-specific languages. In *ECOOP'14 (LNCS)*, Vol. 8586. Springer, 105–130.

[9] E. Poll, J. de Ruiter, and A. Schubert. 2015. Protocol state machines and session languages: specification, implementation, and security flaws. In *Workshop on Language-Theoretic Security (LangSec'15), Symposium on Security and Privacy Workshops*. IEEE, 125 – 133.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK:(state of) the art of war: Offensive techniques in binary analysis. In *Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[11] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS'16*, Vol. 16. Internet Society, 1–16.

A Research Agenda for Formal Methods in the Netherlands

Frank de Boer
Leiden Institute of Advanced Computer Science/CWI
The Netherlands

Software libraries are the building blocks of millions of programs, and they run on the devices of billions of users every day. Therefore, their correctness is of the utmost importance. Recently, the use of formal methods led to the discovery of a major flaw in the design of TimSort, the default sorting library in many widely used programming languages, including Java and Python, and platforms like Android. An improved version of TimSort was proven correct with the state-of-the-art theorem prover KeY. This correctness proof convincingly illustrates the importance and potential of formal methods as a means of putting state-of-the-art software to the test and improving it.

A major future challenge is a systematic verification of mainstream libraries of the popular programming language Java. However, the TimSort case study revealed several challenging scientific problems which need to be addressed, e.g., automatic generation of high-level proof scripts, a novel proof methodology for the specification and verification of interfaces, and new co-inductive techniques for the incremental behavioral refifinement of an abstract model of the program.

Model Learning and Testing

(and some observations on formal methods research in the Netherlands)

Frits Vaandrager
Institute for Computing and Information Sciences
Radboud University
Nijmegen, the Netherlands
F.Vaandrager@cs.ru.nl

1 Formal Methods in The Netherlands: Where Are We?

Formal methods is often defined as the applied mathematics of computer system engineering. The Netherlands has a strong tradition in this area. This started with scientific giants such as Van Wijngaarden and Dijkstra, continued with e.g., De Bakker, De Roever, Rozenberg, Rem, Barendregt, Bergstra and Klop, and led to the strong formal methods groups that we see at Dutch universities today. In addition, some highly visible Dutch formal methods researchers are active abroad, e.g., Katoen, Holzmann, Van Glabbeek and Bloem.

Still, my impression is that Dutch formal methods research is not as influential and authoritative as it used to be. A number of factors may have contributed to this:

1. The field has matured and changed considerably over the years. Whereas, for instance, Bergstra & Klop could create major impact with papers on complete axiomatizations of process algebras, nowadays a nice theoretical idea is not enough. You also must show (or at least make it plausible) that an idea can be implemented and be effectively used to advance the state-of-the-art of computer system engineering. This is a different game, where rather than one or two brilliant theoreticians, you need a whole team/network of researchers, with different people focusing on theory, tools and applications. Despite notable exceptions, the Dutch formal methods community as a whole has not adapted fast enough to this new reality. Too often, I see formal methods papers in which the introduction refers to the importance of correct software, but the proposed methods have not been applied to real software yet, and there is not even a plausible scenario of how the results could be applied to real systems. Too often, also, I see colleagues write papers on questions and research approaches that are almost identical to the ones they explored in their thesis many years ago.
2. The funding situation has not been very helpful with on the one hand the personal grants that support individuals (rather than teams/networks), and on the other hands projects that require direct support from industry (and typically have a focus on short term applications). The funding does not have the scale and time horizon needed to solve the challenges that our field is facing.
3. Maybe we no longer succeed to attract the most brilliant and ambitious students. Somehow, cyber security, artificial intelligence and data science appear to have a more attractive proposition.

2 Formal Methods in The Netherlands: Where Do We Want To Go?

So what should we do to address the above problems? A first thing is to bring the community together and to discuss the problems. I am most grateful to Marieke and Eelco for their initiative to organize this workshop. I suggest a couple of actions:

1. **Research agenda.** We need to identify a couple of major research challenges where we believe Dutch FM researchers can really make a difference on industrial practice within say seven years, and outline how we want to do it. Only by working together we can create the desired impact. Different teams may work on each of these challenges, combining expertise ranging from pure theory all the way to practical application. (Below I will discuss the research challenge we work on.)
2. **Industrial support crucial.** In his contribution for this meeting, Joost-Pieter Katoen argues for “More Programs, Less Models”, and observes an international trend “from model-based to code-based analysis”. In my view, there are many good reasons why the Dutch high-tech industry is still pursuing a model-based approach (high level of abstraction, possibility to simulate cyber-physical systems and explore design alternatives before they are built (“digital twins”), automatic code generation, etc,..). The presence of a number of big companies that invest in model-based development offers excellent opportunities for Dutch formal methods research, as long as we are willing to face the complexity of industrial design, and try to help people in industry with the problems they face (e.g., dealing with complexity of models and with legacy software for which no models are available. Having said that, I do think (based on my own experience and feedback from students who did internships) that within industry there is much ignorance about modern software

analysis techniques, and also within the formal method community we could use more expertise on e.g., static analysis and software verification.

3. **Funding strategy/lobby.** Once a national formal methods agenda is there, we should just start working on it, irrespective of whether the agenda is supported by funding agencies. Simultaneously, we should try to get funding from a variety of sources asap. For this it would be helpful if the agenda gets some formal status and is officially recognized by VERSEN, IPN, as part of the new sector plan, and/or even as part of the Dutch national science agenda. We should definitely lobby for a Jacquard like program funded by NWO. In the discussion about funding it is *urgent* to arrive at a clear division of work / areas of expertise between different formal methods groups in the Netherlands, e.g., via a list of ten subfields where each university claims a leading role in at most two or three topics.
4. **Attracting talent.** Clearly, having a convincing research agenda with challenging problems and a vision on how to tackle these problems will attract talent. My suggestion would be to organize another meeting at some point to exchange ideas on how we can attract more talented students to our area.

3 Model Learning and Testing

Active automata learning (or model learning) aims to construct black-box state machine models of software and hardware systems by providing inputs and observing outputs. State machines are crucial for understanding the behavior of many software systems, such as network protocols and embedded control software, as they allow us to reason about communication errors and component compatibility. Model learning is emerging as a highly effective bug-finding technique [1]. It has been successfully used in several different application domains, including

- generating conformance test suites of software components, a.k.a. *learning-based testing*,
- finding mistakes in implementations of security-critical protocols,
- learning interfaces of classes in software libraries,
- checking that a legacy component and a refactored implementation have the same behavior.

There is certainly a large potential for application of model learning to many different aspects of software development, maintenance and refactoring, especially when it comes to handling legacy software. To realize this potential, two major challenges must be addressed: (1) currently, techniques do not scale well, and (2) they are not yet satisfactorily developed for richer classes of models.

One way to address these challenges is to augment model learning with white-box information extraction methods (e.g., symbolic execution, tain analysis, static analysis), which are able to obtain information about the system-under-learning at lower cost than black-box techniques. When dealing with computer-based systems, there is a spectrum of how much information we have about the code. For third party components that run on separate hardware, we may not have access to the code at all. Frequently we will have access to the executable, but not anymore to the original code. Or we may have access to the code, but not to adequate tools for analyzing it (this often happens with legacy components). If we can construct a good model of a component using black-box learning techniques, we do not need to worry about the code. However, in cases where black-box techniques do not work and/or the number of queries becomes too high, it makes sense to exploit information from the code during model learning.

Active automata learning is closely related to model-based testing, and both activities can be viewed as two sides of the same coin. Whereas automata learning aims at constructing hypothesis models from observations, model-based testing checks whether a system under test conforms to a given model. Model-based test tools play a crucial role within active automata learning, as a way to determine whether a learned model is correct or not. For this reason, the activities in Nijmegen on model learning and model-based testing are closely aligned, and inspire/challenge each other.

Within our group, Nils Jansen recently started as an assistant professor working at the intersection between formal verification, machine learning, and control theory, along the lines described in Sections 4 and 5 of Katoen's contribution.

Research objective. Our objective is to reach — within say seven years – the point where active automata learning and model based testing have become standard tools in the toolbox of the software engineer, equally mature as model checking is right now.

Collaboration. Our group e.g., collaborates closely with the groups of Bernhard Steffen and Falk Howar at the TU Dortmund on learning tools, with the group of Andreas Zeller from Saarland University on the use of taint analysis in learning, with TNO ESI on the model-based testing tool TorXakis, with the Digital Security Group in Nijmegen on case studies related to security, and with ASML and Philips Healthcare on case studies related to refactoring of legacy software. All these collaborations are vital for reaching our research objectives. We would be most interested to collaborate with other groups in the Netherlands, e.g., on the use of white-box analysis techniques in automata learning.

References

[1] F.W. Vaandrager. 2017. Model Learning. *CACM* 60, 2 (Feb. 2017), 86–95. https://doi.org/10.1145/2967606

Formal Methods for Systems Security

Wan Fokkink, Cristiano Giuffrida, Herbert Bos

Formal methods are being successfully applied in the realm of security, notably in the analysis of network protocols. However, security vulnerability of computer systems nowadays occur more and more at the hardware level. A poignant example is the recent string of cache side-channel attacks, which exploit minimal information leakage via caches. Or the sensational exploits of speculative execution, where microprocessors do work in advance, to prevent delay in case the work is actually needed later; if it turns out the work is not needed, changes are reverted and results are ignored. Notable recent attacks that target microprocessors are Meltdown, Spectre and TLBleed. And at the software level, software patches, which often aim to repair security leaks, tend to introduce new security vulnerabilities that go undetected by software developers.

Formal methods are of the essence to help in harnassing computer systems against such low-level attacks. For this, first of all formal models need to be developed that are at a sufficiently abstract level to obtain general correctness results, but at a sufficiently detailed level to analyze and guarantee security properties at the hardware level. Notably, the CacheAudit tool for static analysis of cache side channels developed by Boris Köpf and his co-workers is an interesting first step in this direction. The groundbreaking work of the team of Gernot Heiser at NICTA, on the microkernel seL4 that was verified using Isabelle/HOL, can also be considered a road map for achieving firm security guarantees at the system level through formal verification. The work has reached sufficiently mature levels to allow for adoption in real-workd systems. Such and similar work at MIT and other places on reliable and secure file systems, compilers and other components of low-level systems indicate both a growing trend in research in the space between systems security and formal methods and an increasing need for such research.

The security and formal methods groups in the department of computer science at the Vrije Universiteit Amsterdam are joining forces and work on formal methods for systems security. A first result of this collaboration is the master thesis of Kevin Maiutto from August 2018, in which the TLBleed attack has been analyzed using CacheAudit. The long-term plan for this effort is to generalize the current framework to several classes of side-channel attacks and model arbitrary defenses as constraints on the underlying formal model. More broadly, focus points for the coming years are formally reasoning over the properties of side channel attacks in the memory subsystem and guaranteeing the correctness of software patches.

Computer Assisted Mathematical Proofs: using the computer to verify computers

Herman Geuvers
Radboud Universiteit
The Netherlands

Mathematical proofs get more and more difficult and complex. At the same time more and more computer systems (software and hardware) can be verified rigidly using mathematical proof, so there is an increasing request for completely verified mathematical proofs.

The field of Computer Assisted Mathematical Proofs fills this gap by allowing users to create complete mathematical proofs, interactively with the computer, where the computer checks each small reasoning step. In this way we obtains the utmost guarantees of correctness.

In the talks we will discuss the present state of Computer Assisted Mathematical Proofs and how it works by some examples. We will discuss its limitations, which basically rest on the limitations of proof automation. It has recently become clear that Machine Learning provides methods that apply very well to speeding up proof automation. Machine Learning does not supersede standard techniques (from Automated Theorem Proving) but provides the ideal additional technique.

Formal modelling and analysis

The road to affordable, high quality (control) software

Jan Friso Groote, Bas Luttik, Julien Schmaltz, Erik de Vink, Wieger Wesselink, Tim Willemse

{J.F.Groote,S.P.Luttik,J.Schmaltz,E.P.d.Vink,J.W.Wesselink,T.A.C.Willemse}@tue.nl
Department of Mathematics and Computer Science
Eindhoven University of Technology

Abstract

The research interest of the Formal Systems Analysis (FSA) research group at Eindhoven University lies in the development of techniques to efficiently develop software that is guaranteed to work as expected. Our tools and methods are starting to find widespread use in the embedded systems industry. We identify the success factors and indicate what we believe are the most important research directions.

1 Introduction

Can we make software that works most of the time? The answer is yes, and it is all around us. But a trickier question is whether we can make software that *always* works as expected and that is still affordable. The answer to that is quite open, and it is one of the fundamental quests of computer science. In the Formal Systems Analysis group at Eindhoven University we believe that the solution to this quest can only be found through the use of mathematical analysis techniques assisted by appropriate methods and tools.

We can safely say that throughout the years we made good progress towards an answer. We have analysed the actual software of numerous large scale projects varying from train and bridge controllers [4, 9], control software for wafer steppers [10] and X-ray scanners [11] and even the detector control software of the CMS and ATLAS projects at CERN [8]. In almost all cases we could identify flaws, have them repaired and we could always show that the resulting software was performing according to given correctness requirements. This success led to the uptake of the mCRL2 toolset as the verification backend of the Verum toolset. Verum is currently a major supplier for tools to design reliable embedded software. Analysis of the use of Verum's methods show that they lead to a tenfold reduction of errors and a speedup of development of a factor three [11].

Below we identify the factors that made these projects successful and how they lead the way of further research. The successful methods need to be strengthened and they have to made available to larger groups of developers at an even further reduced cost.

1.1 Formal Model-Driven Engineering

All projects where we successfully applied model checking to increase the quality of the code were using some form of Domain Specific Language (DSL). All DSLs that we encountered essentially consisted of a finite automata language with inputs and outputs. Examples are safety programmable logical controllers (PLCs), finite automata at CERN and ASD and Dezyne at Verum [1].

The effectiveness of DSLs has three causes. Firstly, programs written in a DSL are much more concise than programs written in a general programming language. Secondly, DSLs restrict the expressivity of programmers, which means that programs written in DSLs are far more comprehensible by other programmers. Finally, DSLs are easier to analyse and verify by formal means, meaning that their quality can be made much higher than that of regular programs. This last aspect is probably the most important of these three. Formal verification is an effective way the let the software attain its desired quality, avoiding expensive testing and continuous redesign.

In all cases we wrote translators from DSLs to the most appropriate verification framework (mCRL2 [6] or SAT) to perform the verification. Formal verification of systems written in general purpose languages require manual translation to a formal setting. As such software is complex and often badly documented, its translation is generally unsatisfactory and inefficient, unsuitable to become part of an effective workflow.

Model-Driven engineering with behavioural analysis can be made more effective in the following ways:

- More effective means to transform programs written in a DSL into a verification framework. It is not efficient to build verification toolsets for each DSL from scratch. It should be possible to define the translations abstractly, after which the necessary transformation framework is automatically generated.
- An effective theory to denote and reason about the meaning of DSL programs. This theory should also allow to assess the correctness of the translations to verification formalisms.
- An understanding of the do's and don'ts within the design of DSLs such that they are suitable for their domains, allow verification and do not become too

expressive hampering understandability of programs written in it. As an example the DSLs ASD and Dezyne do not allow to write programs that manipulate input data, to avoid a state space explosion that could hamper verifiability.

1.2 Coordination Architecture

Another important observation behind most of the effective verifications is that the architecture of the system has a strong influence on verifiability [5]. At CERN we are speaking about systems of up to 60,000 cooperating components in a strict tree structure. At ASML we verified a system with 250 components with on average more than 1000 'rule cases' each, of which the coordination architecture could be transformed into a tree [10].

But it is unlikely that tree like software architectures would fit all applications. We need to identify which software structures are more amenable to verification, and which verification techniques are most suitable for those cases.

We then need to change the attitude to system design. When setting up a system, the software architecture that is chosen must not only be suitable for the purpose of the software, but also need to fit the verification needs.

1.3 A Unified Analysis Framework

The diverse, and relatively separate visions and solutions for analysing the behaviour of systems within various specialised problem domains have led to a situation in which results that are obtained in one domain are not easily transfered to other domains. There is a serious risk that this will inhibit the progress we can achieve in the near future.

What is needed is to identify a formalism that unifies existing behavioural analysis techniques. Such a formalism should be independent of the specification languages used to describe the systems, but, more importantly, the formalism should be sufficiently powerful to unify all the techniques developed in the separate specialised verification disciplines.

A serious candidate is the formalism of *parameterised Boolean equation systems* [7]. This formalism is firmly rooted in mathematical fixpoint theory and logic, and admits an elegant game theoretical interpretation. Model checking in mCRL2 is done via a translation to PBESs. Moreover, the techniques developed for PBESs can be studied in their own right, leading to new insights into existing theories. A beautiful example thereof is the theory of abstraction for PBESs [3], which, inspired by abstraction theories for transition systems, substantially improves upon it.

The PBES theory has, by now, successfully demonstrated its status as a unified framework for analysing data-dependent systems and real-time systems. However, more is needed. Apart from continuing the prime research line of unifying existing specialised techniques (*e.g.* partial order and symmetry reduction) for data-dependent and real-time systems into the the theory of PBESs we need to:

- define specialised theories for PBESs that involve (fragments of) real-valued and complex numbers; such theories are needed to deal with continuous variables and problems stemming from *e.g.* hybrid systems and hybrid approaches to fluid dynamics. Of particular interest is the algorithmic study of PBESs that include such numbers;
- extend the PBES theory to deal with quantitative analysis problems to allow for analysing probabilistic and stochastic systems, optimisation problems, *etcetera*, and demonstrating that these extensions are sufficiently powerful to solve the analytical problems in the respective application areas.
- address the fundamental open questions in the theory of PBESs; *e.g.* establishing the long standing open problem of the exact computational complexity of solving PBESs or parity games [2]; identifying a complete (as in *finite*) abstraction theory for PBESs; identifying the limits of PBESs as a unifying framework.

References

[1] Guy H. Broadfoot. 2005. ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software. In *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings (Lecture Notes in Computer Science)*, John S. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki (Eds.), Vol. 3582. Springer, 548–551. https://doi.org/10.1007/11526841_39

[2] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. 2017. Deciding parity games in quasipolynomial time. In *STOC*. ACM, 252–263.

[3] Sjoerd Cranen, Maciej Gazda, Wieger Wesselink, and Tim A.C. Willemse. 2015. Abstraction in Fixpoint Logic. *ACM Trans. Comput. Log.* 16, 4 (2015), 29:1–29:39.

[4] Jan Friso Groote, Jeroen Keiren, Anson van Rooij, Vikram Saralaya, and Anton Wijs. 2013. *Verificatie van de correctheid van de PLC-software van de Algerabrug.* Technical Report. Technical report for Rijkswaterstaat (confidential).

[5] Jan Friso Groote, Tim W. D. M. Kouters, and Ammar Osaiweran. 2015. Specification guidelines to avoid the state space explosion problem. *Softw. Test., Verif. Reliab.* 25, 1 (2015), 4–33.

[6] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and Analysis of Communicating Systems.* MIT Press.

[7] Jan Friso Groote and Tim A.C. Willemse. 2005. Parameterised boolean equation systems. *Theor. Comput. Sci.* 343, 3 (2005), 332–369.

[8] Yi-Ling Hwong, Jeroen J. A. Keiren, Vincent J. J. Kusters, Sander J. J. Leemans, and Tim A. C. Willemse. 2013. Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. *Sci. Comput. Program.* 78, 12 (2013), 2435–2452.

[9] J.W.C. Koorn J.F. Groote and S.F.M. van Vlijmen. 1995. The safety guaranteeing system at station Hoorn-Kersenboogerd (Extended abstract). In *Proceedings 10th Annual Conference on Computer Assurance (COMPASS'95)*. IEEE, Gaithersburg, Maryland, 57–68.

[10] Ruben Jonk. 2016. *The semantics of ALIAS defined in mCRL2.* Technical Report. Eindhoven University of Technology, Department of Computer Science, The Netherlands. master thesis.

[11] Ammar Osaiweran, Mathijs Schuts, Jozef Hooman, Jan Friso Groote, and Bart J. van Rijnsoever. 2016. Evaluating the effect of a lightweight formal technique in industry. *STTT* 18, 1 (2016), 93–108. https://doi.org/10.1007/s10009-015-0374-1

Practical Formal Methods

Jeroen J.A. Keiren
Department of Computer Science
Open University of the Netherlands
Department of Digital Security, ICIS
Radboud University
Nijmegen, The Netherlands
Delft University of Technology
Delft, The Netherlands
Jeroen.Keiren@ou.nl

1 Introduction

Today, we heavily depend on software. We do not only use the computers on our desktops and the mobile phones in our pockets. Financial infrastructures and automatic stock trading are controlled by computers, and computer systems are embedded in home appliances such as televisions, safety critical systems such as cars and airplanes, as well as systems controlling (access to) infrastructure such as bridges and tunnels.

During the development of software, inevitably, mistakes are made. In fact, on average, every 1000 lines of code contain up to 10-16 bugs [? ?]. Some of these bugs will only show up once the system is running. At that point, the consequences can range from being harmless – e.g. needing to restart your phone because it freezes –, to very severe – such as a car crashing [?], hundreds of millions of dollars being lost in in the stock markets [?]. Furthermore, a growing reliance on battery-powered devices and the effects of climate change have resulted in an increased interest in *green computing*. Bugs that lead to quick draining of batteries have gained a lot of publicity [?].

To effectively detect or avoid such bugs early (preferably before software is used) calls for a range of different tools and techniques. This ranges from exact and exhaustive verification methods such as (automated) theorem proving and model checking, to formal testing techniques such as model based testing, automated testing at the level of graphical user interfaces, as well as more traditional testing techniques such as unit- and integration testing. The formal methods and software engineering communities in the Netherlands have a long track record in developing such techniques.

When considering, especially, the more formal approaches such as model checking, theorem proving and model-based testing, there is a lot of anecdotal evidence that they are effective in finding and avoiding bugs, for instance [? ? ?]. However, the techniques typically require experts that are well-versed in both the application domain as well as in the formal methods applied. Furthermore, a clear business case for the application of formal methods is missing.

Besides the mainly theoretical research into the foundations of formal methods based verification and testing techniques, there are three research directions that deserve our undivided attention.

1. Develop formal methods that can be used by software engineers that are not formal methods experts.
2. Establish a business case for the industrial application of formal methods through empirical research.
3. Determine how formal methods can best be integrated in software engineering and computer science curricula.

I will detail each of these three points in the rest of this abstract.

2 Bringing formal methods to the masses

The application of formal methods such as model checking or model based testing to industrial cases is still very much an expert activity. Tools often rely on models of the software, instead of the software itself, and the models are specified in domain-specific languages whose syntax is far from the the programming languages software engineers are used to. Throughout the years many applications of such techniques have been reported as a success, e.g., [? ?], but large-scale industrial application has yet to gain traction.

In software model-checking, some successes have been obtained, e.g., at Microsoft using SLAM[1] [?] and TERMINATOR, continued as T2 [?], for the verification of C code, where verifying a limited set of properties on an actual (C-code) implementation seems to be one of the success criteria.

To bring formal methods to the masses, as a community we need to invest not just in new techniques, but also in the engineering of tooling and languages that can be used in the industrial software engineering process, and recognise that this step is not "just trivial engineering". We should therefore not stop at the development of prototype tools, but also look at industry needs and invest in making the tooling mature enough for use in production systems. At CERN, for example, we went from high-level academic tools, down to an integration of the verification in the IDE used by

[1] https://www.microsoft.com/en-us/research/project/slam/, accessed 9 August 2018

the developers, effectively providing developers with push-button access to model checking [**?**].

3 Empirical evidence for the merits of formal methods

Industrial application of formal methods are typically reported as succes stories. However, from an industrial perspective, it is important to know the effects on applying such techniques. How does the application of formal methods affect, for example, time-to-market, number of bugs found after deployment, etc.

Unfortunately, such information is currently lacking from the formal methods literature. Some vendors, such as Verum with its Dezyne toolkit – which is based on mCRL2 [**?**] –, do report benefits such as "50% reduction in development costs, 25% reduction in the cost of field defect and 20% decrease in time-to-market"[2], but the sources and reliability of such data are unclear. In order to build a successful business case for the application of formal methods in software engineering practice, a collaborative effort should be made to collect data about the application of formal methods. Based on the collected data, a business case for the application of formal methods should be developed. One possibility could be the collection of data in a large number of student projects, such as [**?**].

4 Teaching formal methods

A final advance in the acceptance of formal methods lies in the way we teach formal methods. It appears commonplace in software engineering and computer science curricula to present formal methods more as a research topic than as a software engineering topic. A typical question one gets from students is "so, where and how are these techniques actually applied in industry". In recent years, I have seen more critical attitudes of students towards formal methods courses. Should we do a better job at integrating formal methods into our curricula, to really show the students what the can *do with formal methods* instead of *how the formal methods work*?

Note that there is more to the teaching of formal methods. How can we effectively teach such methods using modern tools and techniques such as massive open online courses (MOOCs)? In particular, with (sometimes dramatically) increasing numbers of students, and in the context of distance learning, we need to find ways of providing feedback in formal methods education that does not rely on expert feedback. Is it possible to build interactive online tools that present feedback to students about their solutions? Are there more effective ways of teaching formal methods than we currently use in our courses? Can we, in this respect, learn from programming education research such as [**? ?**]?

[2]https://www.verum.com, accessed 9 August 2018.

More Code, More Quantities

Joost-Pieter Katoen
RWTH Aachen University, Software Modeling and Verification
Aachen, Germany

University of Twente, Formal Methods and Tools
Enschede, The Netherlands
katoen@cs.rwth-aachen.de,katoen@utwente.nl

Abstract

Formal methods are pretty strong in the Netherlands. It is however primarily focused on correctness in the Boolean sense: an artefact (being it a program or a system) is correct or not. Without doubt correctness is of pivotal importance, but in my view we should be ready to make a paradigm shift to a more quantitative setting. Big data, machine learning, robots (to mention a few) rapidly are becoming dominant for programs and (safety-critical) ICT systems. They come with a high degree of uncertainty. Reasoning about this requires that formal methodists leave the Boolean territory — *it is quantitative reasoning that is sought for!* In addition, model-based techniques prevail in Dutch formal methods. This contrasts the international trend where more and more emphasis is put on (continuously) verifying software at the code level. It is time that Dutch formal methodists take software more seriously — *it is software in cars, washing machines, and aerospace systems that becomes the bottleneck!*

1 Formal Verification

Key considerations for computer programs are whether they will terminate, and if so, whether the result of their computation is correct. The fundamental questions "does a computer program terminate?" and "does a computer program work as expected?" are core — if not the most important — research topics in program analysis ever since Turing's seminal paper on providing a program proof in 1949. As hard guarantees cannot be achieved using common techniques such as peer review (aka: manual inspection), extensive simulation and testing, more rigorous means are needed. This is where *formal verification techniques* enter the scene. They strive for establishing that a program is correct with respect to a given specification, a precise description of the input-output behaviour of the program. This is a challenge as proving program correctness is undecidable in general. Main techniques are deductive techniques à la Floyd-Hoare and model checking.

Model checking is a successful technique for finite-state systems making it very popular for checking hardware; e.g., the temporal logic PSL is an international standard for formal hardware specifications. The state-of-the art in *software verification* is to use a carefully balanced mixture of deductive techniques (to support computing weakest preconditions), model checking, program analysis, and satisfiability-modulo theory (SMT) techniques [5]. Powerful abstraction-refinement techniques enable the automated verification of a large class of programs.

2 The Rise of Software Verification

Formal verification has been adopted by the main software companies. On the world-wide level, Microsoft uses program verification to find bugs (or better: show their absence) in device-driver software, Facebook provides their programmers with tools employing lightweight formal verification to improve their software quality by tracing memory leaks and null pointer dereferencing, and Amazon web services attacks security leaks with formal verification [2]. Microsoft, Google, Amazon, and Facebook all provide funding possibilities for applying and further developing formal verification. Facebook's continuous verification program to boost the scalability of formal verification is an interesting example of this [7].

Also in Germany, formal verification is (finally) on the agenda of some major companies. This includes car manufacturers such as BMW and Mercedes, but also companies such as Bosch and Siemens. In close collaboration with — and completely funded by — Siemens we have recently finished building an IC3-based software model checker for C programs in the last five years [6]. Although it is applicable to general C programs, it is tailored to programmable logic controllers (PLCs) with specific support for arrays, floating points (for treating trigonometric functions occurring in motor control), and bit-vector analysis. The underlying technology is based on aggressive abstraction (IC3 and generalization), SMT, and deductive techniques for computing weakest preconditions. Together with — and fully funded by — Ford motor company (in Germany and the USA), we are currently applying software verification to C code that is automatically generated from Simulink [1].

3 More Programs, Less Models

I observe two trends. The first trend: software companies have adopted formal verification to quite some extent. They have entire research groups, bought out famous academics in the field, and invest substantial amounts in further developing the field. And is not safety-critical software per se that is focused on. *It seems that Dutch software industry does not follow this international trend at all.* Why is this? Is it their ignorance? Is software development not major enough?

Or does it have perhaps to do with the second trend: there is a clear shift from model-based to code-based analysis. Whereas until about a a decade ago, model-based verification (UML, AADL, process algebra, etc.) was prevailing, software verification — directly proving properties on the code — is taken over. *It seems that Dutch formal methodists do not follow this trend.* There are a few Dutch groups doing (excellent) research on (semi-)automated program verification, but it seems (to me) that model-based techniques are still in their majority. Why? Is this because of the strong Dutch tradition on model-based formalisms such as process algebras? Or are we more system oriented rather than software oriented? Or is it because in the Netherlands the primary use of software is in trading and finance, and not in industry?

4 The Rise of Uncertainty

Uncertainty is nowadays more and more pervasive in computer science. It is important both in big data and at the level of events and control. Applications have to treat lots of data, often from unreliable sources such as noisy sensors and untrusted web pages. Data may also be subject to continuous changes, may come in different formats, and is often incomplete. Systems have to deal with unpredictable and sometimes hostile environments. A different, also inevitable, kind of uncertainty arises from abstractions in system models focusing on the control of events.

Probabilistic modelling and randomization are key techniques for dealing with uncertainty. Many trends witness this. Real-world modelling in planning is advancing by probabilistic programs describing complex Bayesian networks. In security, hostile environments are often captured by probabilistic adversaries. Probabilistic databases deal with uncertain data by associating probabilities to the possible worlds. In systems verification, probabilistic model checking has emerged as a key technique allowing for correctness checking and performance analysis. Similar developments take place in logic and game theory. The pervasiveness of uncertainty urges to make substantial enhancements in probabilistic modelling and reasoning so as to understand, reason about, and master uncertainty. *This requires a paradigm shift from reasoning about Boolean correctness — a system is correct or not — to a more quantitative notion.* This is in line with Henzinger's convincing arguments of a few years ago [3]. With the advent of machine learning, big data, and robotics, I feel that such shift is more needed than ever.

5 More Quantities, Less Booleans

With the rapidly growing application of machine learning in e.g., self–driving cars, the need for verified guarantees against potentially fatal accidents is self–evident. To substantiate this, let me quote the U.S. government report *"Preparing for the Future of Artificial Intelligence"*:

> If practitioners cannot achieve justified confidence that a system is safe and controllable, so that deploying the system does not create an unacceptable risk of serious negative consequences, then the system cannot and should not be deployed.

Machine learning naturally cannot guarantee safe behavior. However, growing application areas such as autonomous driving, require the exclusion or likely avoidance of unsafe behaviors. Formal verification has the potential to indicate confidence in system behaviors obtained from machine learning. Vice versa, leveraging the capabilities of machine learning to quickly assess large data sets may help to enable verification for more realistic systems. Machine learning typically obtains results but provides no evidence about this.[1] Formal verification techniques such as counterexample generation have the potential to obtain some explainability. A recent list of challenges on the edge of machine learning and formal verification [4] includes a.o. safety verification of deep neural networks, formal program synthesis and analysis using machine learning, explainable AI, machine learning in motion planning, and guarantees on reinforcement learning in verification. *Evidently, quantities is what is sought for; it is no longer sufficient to treat correctness in a Boolean, absolute sense.*

References

[1] Philipp Berger, Joost-Pieter Katoen, Erika Ábrahám, Md Tawhid Bin Waez, and Thomas Rambow. 2018. Verifying Auto-generated C Code from Simulink - An Experience Report in the Automotive Domain. In *FM (LNCS)*, Vol. 10951. Springer, 312–328.

[2] Byron Cook. 2018. Formal Reasoning About the Security of Amazon Web Services. In *CAV (1) (LNCS)*, Vol. 10981. Springer, 38–47.

[3] Thomas A. Henzinger. 2013. Quantitative reactive modeling and verification. *Computer Science - R&D* 28, 4 (2013), 331–344.

[4] Nils Jansen, Joost-Pieter Katoen, Pushmeet Kohli, and Jan Kretinsky. 2018. Machine Learning and Model Checking Join Forces (Dagstuhl Seminar 18121). *Dagstuhl Reports* 8, 3 (2018), 74–93.

[5] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Comput. Surv.* 41, 4 (2009), 21:1–21:54.

[6] Tim Lange, Frederick Prinz, Martin R. Neuhäußer, Thomas Noll, and Joost-Pieter Katoen. 2018. Improving Generalization in Software IC3. In *SPIN (LNCS)*, Vol. 10869. Springer, 85–102.

[7] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *LICS*. ACM, 13–25.

[1]The NIPS 2017 test-of-time award presentation claims that *machine learning has become alchemy.*

Correct Message-Passing Software Systems

Current and Future Research Challenges

Jorge A. Pérez
Bernoulli Institute for Mathematics, Computer Science, and Artificial Intellgence
University of Groningen
The Netherlands
j.a.perez@rug.nl

Abstract

Many critical software systems rely on message-passing between distributed artefacts, such as (micro)services. To a large extent, certifying system correctness amounts to ensuring that these artefacts respect some intended protocol.

Rooted in programming models from concurrency theory, *behavioral type systems* are a rigorous verification technique for message-passing programs. While these type systems are well understood by now, a main challenge remains: as their precision varies ostensibly, they induce seemingly unrelated forms of correctness. Since behavioral type systems still lack unifying foundations, it is hard to establish a most necessary dialogue with other researchers and software practitioners.

This note briefly introduces behavioral type systems and describes current research that targets the above challenge, framed within my VIDI career grant, recently awarded. It also suggests links between type-based program verification and other approaches in the intersection of formal methods, programming languages, and software engineering.

Keywords message-passing concurrency, verification, types

1 Communication Correctness

Ensuring software systems correct is widely recognized as a societal challenge (see, e.g., [8]). Establishing correctness for concurrent programs is much harder than for sequential programs. For message-passing programs, we may define *communication correctness* as the interplay of four properties:

- *Fidelity*: programs respect interaction protocols;
- *Safety*: programs do not get into errors, e.g., communication mismatches;
- *Deadlock-freedom*: programs do not get stuck;
- *Termination*: programs do not have infinite internal runs.

Certifying communication correctness is notoriously hard. Developers need *effective* and *practical* verification tools. Effective tools detect insidious bugs (e.g., communication errors, protocol mismatches, deadlocked services) before programs are run. Practical tools reconcile the *programming view* that developers understand well (where programs are implemented) and the *engineering view* needed to build large systems (where protocols are conceived).

2 Behavioral Types

Framed within concurrency theory and programming languages, *behavioral type systems* (or just *behavioral types* [9]) are a rigorous technique for certifying that message-passing programs satisfy communication correctness.

From Data Types to Behavioral Types Type systems prevent the occurrence of errors during the execution of a program [4]. The most basic property of a type system, *soundness*, ensures that "well-typed programs can't go wrong" [12].

While usual data types classify values, behavioral types organize concurrent interactions into *communication structures*, which are explicit for developers and architects [19]. Implementation agnostic, behavioral types are *effective*: the interaction sequences they codify are useful to exclude bugs that jeopardize communication correctness. Behavioral types are *practical*: communication structures can be compositionally expressed at different abstraction levels. Soundness of a behavioral type system typically ensures fidelity and safety; deadlock-freedom and termination are harder to enforce.

Foundations and (Too Many) Variants Behavioral types are usually defined on top of *process calculi*, formal languages that treat concurrent processes much like the λ-calculus treats computable functions. The π-calculus [13], the paradigmatic calculus of concurrency, is the specification language for many behavioral type systems. The intent is to define robust type systems for the π-calculus, and then transfer such foundations to specific languages. This is how verification tools based on behavioral types have been successfully developed for Haskell [16], Go [11, 15], and Scala [18].

Many behavioral type systems exist; see [9] for a survey. Primarily intended as *static* verification techniques, which enforce correctness prior to a program's execution, behavioral types may help also in defining *dynamic* verification techniques that couple message-passing programs with so-called *monitors* that ensure fidelity and safety at run-time.

3 Two Pressing Challenges

While specific theories of behavioral types (most notably, *session types* [6, 7]) are gradually reaching maturity, the body of knowledge on behavioral type systems as a whole is still immature, as two basic issues still lack proper justifications.

First, different behavioral type systems enforce *different notions of communication correctness*: hence, a program accepted as well-typed (i.e., communication-correct) by one type system could be ruled out as incorrect by another.

Second, and related to the above point, since the precise relations between different behavioral types are still poorly understood, their derived verification tools are *hardly interoperable*. As such, we still do not know how to combine distinct verification tools in the formal analysis of communication correctness in complex software systems.

Our ongoing research, recently supported by a VIDI career grant, aims at jointly tackling both challenges.[1]

In short, my VIDI career grant aims to unify distinct notions of communication correctness induced by different theories of behavioral types. To this end, various verification techniques for message-passing programs will be rigorously related in terms of their *relative expressiveness* [17]. To articulate a unified theory of correctness, we will crucially rely on the Curry-Howard correspondence for concurrency [3], the most principled link between concurrency and logic. Our preliminary results [2, 5, 10] are rather promising. We plan to validate these foundational results through case studies and tool prototypes.

4 A Research Agenda for the Future?

Tackling the two above challenges appears to be a pre-requisite step for addressing other relevant research questions in the intersection of formal methods, programming languages, and software engineering. Such questions include:

- ***Integration of techniques***. Can we combine techniques based on behavioral types with complementary techniques developed by (Dutch) researchers on formal methods, such as model checking and (concurrent) program logics?
- ***Cyber-security and trustworthiness***. Verification based on behavioral types uses protocols as key abstraction for enforcing communication correctness. There is potential for using this abstraction in other settings. For instance, can we reconcile this view of protocols with the one typically required to certify *security protocols* in critical software systems? (We have given a preliminary answer in [14].)
- ***Industrial transfer and impact***. To what extent verification based on behavioral types has a place in validation phases used by (Dutch) software industries?
- ***Usability***. Are verification frameworks based on behavioral types usable by software developers in practice? Can practice inform further theoretical research on type-based program verification? (A preliminary answer is in [20].)

While this list is by no means exhaustive, in our opinion these questions already suggest potential grounds for collaboration with other (Dutch) researchers, as well as also medium- and long-term challenges to be tackled in cooperation with practitioners and software industries.

[1]See http://www.jperez.nl/vidi for details.

References

[1] E. Albert and I. Lanese, editors. *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016*, volume 9688 of *Lecture Notes in Computer Science*. Springer, 2016.

[2] L. Caires and J. A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In Albert and Lanese [1], pages 74–95.

[3] L. Caires, F. Pfenning, and B. Toninho. Towards concurrent type theory. In B. C. Pierce, editor, *Proceedings of TLDI 2012*, pages 1–12. ACM, 2012.

[4] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[5] O. Dardha and J. A. Pérez. Comparing deadlock-free session typed processes. In S. Crafa and D. Gebler, editors, *Proceedings of EXPRESS/SOS 2015*, volume 190 of *EPTCS*, pages 1–15, 2015.

[6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *Programming Languages and Systems - ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[7] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In G. C. Necula and P. Wadler, editors, *Proceedings of POPL 2008*, pages 273–284. ACM, 2008.

[8] M. Huisman, H. Bos, S. Brinkkemper, A. van Deursen, J. F. Groote, P. Lago, J. van de Pol, and E. Visser. Software that meets its intent. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016), Proceedings, Part II*, volume 9953 of *Lecture Notes in Computer Science*, pages 609–625, 2016.

[9] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira, and G. Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3, 2016.

[10] D. Kouzapas, J. A. Pérez, and N. Yoshida. On the relative expressiveness of higher-order session processes. In P. Thiemann, editor, *25th European Symposium on Programming, ESOP 2016*, volume 9632 of *Lecture Notes in Computer Science*, pages 446–475. Springer, 2016.

[11] J. Lange, N. Ng, B. Toninho, and N. Yoshida. Fencing off go: liveness and safety for channel-based programming. In G. Castagna and A. D. Gordon, editors, *Proceedings of POPL 2017*, pages 748–761. ACM, 2017.

[12] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[14] D. Nantes and J. A. Pérez. Relating process languages for security and communication correctness (extended abstract). In C. Baier and L. Caires, editors, *Proc. of FORTE 2018*, volume 10854 of *Lecture Notes in Computer Science*, pages 79–100. Springer, 2018.

[15] N. Ng and N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In A. Zaks and M. V. Hermenegildo, editors, *Proceedings of CC 2016*, pages 174–184. ACM, 2016.

[16] D. A. Orchard and N. Yoshida. Effects as sessions, sessions as effects. In R. Bodik and R. Majumdar, editors, *Proceedings of POPL 2016*, pages 568–581. ACM, 2016.

[17] J. A. Pérez. The challenge of typed expressiveness in concurrency. In Albert and Lanese [1], pages 239–247.

[18] A. Scalas, O. Dardha, R. Hu, and N. Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In P. Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl, 2017.

[19] V. T. Vasconcelos. Sessions, from types to programming languages. *Bulletin of the EATCS*, 103:53–73, 2011.

[20] A. L. Voinea and S. J. Gay. Benefits of session types for software development. In C. Anslow, T. D. LaToza, and J. Sunshine, editors, *Proc. of the 7th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU@SPLASH'16*, pages 26–29. ACM, 2016.

The Usability of Static Type Systems

Jurriaan Hage
Universiteit Utrecht
The Netherlands
j.hage@uu.nl

1 Introduction

Types are used in statically typed languages to guarantee that "well-typed programs do not go wrong" (for the right definition of wrong). Typically, this means that the compiler for the language prevents certain programs from compiling, because it has discovered that while running the program a value may be passed to an operation that works on values of an incompatible type. Such a type system is called an intrinsic type system, as it is part of the language definition that defines what are valid programs (for the given language).

Rice's theorem [7] implies that no type system for a Turing complete language can precisely characterize the set of programs that always go right. In a statically typed language this means that the type system allows only a subset of such programs. For example, a compiler for such a language will enforce type correctness for all parts of a program, even for dead code.

Another example of such over-approximation is the following. According to the Hindley-Milner type discipline [6] (used as the basis for the type systems of many statically typed functional languages), an expression like $(\lambda x.xx)(\lambda y.y)$ is considered ill-typed. The reason is that in the first lambda-abstraction x is applied to itself, leading to an infinite type to be inferred for it, something which is not allowed within the discipline. For that reason, people have sought to extend the type system leading to the invention of so-called higher-ranked types.

Extending and refining the intrinsic type system of a programming language to ensure that it can soundly (with respect to the semantics of the language) accept more and more programs is a game the programming language community has been playing for some time. For a language such as Haskell alone, type system concepts such as Generalized Abstract Datatypes (GADTs) [10], and many of Haskell's other type system extensions have been developed to allow the programmer to express more precise correctness properties for their programs. At the extreme end of that spectrum we find the dependently typed languages, such as Agda and Idris, in which the worlds of type and values have merged so that almost any property can be expressed within the type system [9].

The development of new type system features to extend the set of known well-behaved programs has some of the characteristics of an armament race. Consider the following, highly idealized, scenario for the programming language X. At one point, users of X discover that a particular class of properties cannot be easily expressed in X, or that certain classes of programs that clearly do not go wrong are forbidden by its type system. This prompts programming language researchers to develop extensions to or refinements of the existing type system of X to deal with this issue; prototype implementations are made to experiment with the new "weapon"; interactions with existing features of X are (hopefully) considered and if everything proceeds as planned, the new feature can be placed into the hands of the programmer by an implementation into a industry-strength compiler for X, unaware that some of these new features can easily blow up in your face.

2 The diagnosis of type errors

Fortunately, most programs and compilers do not actually blow up in anyone's face. What a compiler might do, is refuse to compile a program and generate a type error message. And, the more complicated and advanced the type system and its implementation is, the harder it will be to convey such a message correctly and comprehensibly. This is not only due to the intricacy of the new concept, but also to the tendency to focus only on the implementation of the concept itself: dealing with all the complicated interactions with other features often takes second place. If the usability of a new feature receives any attention at all it gets very little, and often as an afterthought.

Sometimes, the design of a new feature is very pleasing from a programming language viewpoint, but unintuitive from a programmer's viewpoint. A good example is the notion of type classes in Haskell to model ad-hoc polymorphism (aka overloading, the ability to refer to, say, the implementation of equality for strings and for booleans with the same name). In the setting of Haskell, when a programmer accidentally compares two functions, the type error message may suggest to add a method for comparing functions, although no sensible implementation can in fact be given. In practice, the programmer simply forgot to provide arguments to the functions. In the presence of overloaded numerals such error messages can become even more confusing.

Other challenges include the diagnosis of type errors in Domain-Specific Languages (DSLs) embedded into a general purpose host language [4]. The aim in this case is to introduce a new language specialized for a domain as part of an existing (host) language. As part of the embedding, we may even want to use the type system of the host to encode type-like properties for the embedded language. Type error diagnosis in such a setting is quite different from ordinary type error diagnosis, in that the host language compiler has absolutely no knowledge of the domain we are dealing with. In particular, error messages will typically fail to talk to the programmer in terms of the domain. This information then has to be provided as part of the DSL definition. Some solutions to this problem can be found in the literature [8], and although some implementations exist in realistic languages, much work still has to be done.

At the farthest extreme in terms of the guarantees that can be provided by the programmer we find the dependently typed languages (see [9] for more details). In this setting computation and type have all become one. This means that very precise properties can be checked, although typically the language itself imposes certain demands on the programs to make type checking work. The core idea of this paradigm is that code comes with a correctness proof for the code, integrated seamlessly into a single program. In practice, these languages are very hard to use, introducing diagnosis problems at various levels: on top of the usual unification errors, e.g., a function is called with arguments in the wrong order, at another level we might want the compiler to suggest strengthening a lemma to make the proof of a theorem go through.

3 Transparent programmer assistance for optimising functional programs

It may not be at all clear that the problems observed in the previous section carry over to the optimisation of statically typed functional languages. But here is how.

The advent of intrinsic type systems has led to the development of what are often called type and effect systems (some call these non-standard type system, or annotated type systems) [5]. The idea is to annotate the intrinsic types with properties of interest (e.g., strictness information for a lazy language [3], usage information [1], or control-flow information [2]). The advantages of this approach are twofold: types provide additional structure that we can exploit during the analysis, and we can reuse vocabulary and implementation techniques from the world of type systems. For example, let-polymorphism in the Hindley-Milner type system gives rise to so-called let-polyvariance in type and effect systems.

We can, however, go one step further. Optimisations are usually performed deep down inside a compiler on a core language to which the original program has been desugared. Whether an analysis actually leads to an optimisation, and how that is affected by how the program was written is not known to the programmer unless he/she is willing to spend time inspecting the generated object code. Typically, after some profiling, we simply try something, and see how that affects the running time and memory consumption.

Unsurprisingly, this ad-hoc approach can be very time consuming, a problem that becomes more pronounced as the abstraction level of a language rises. Although performance is not often a problem these days, when it is then developing applications in very high-level languages becomes a risk in itself and developers may prefer to resort to lower-level languages instead. To counter this development, what we need is that compilers for such high-level languages are transparent in that the information they collect for a given program are made available to programmers at a suitably high level of abstraction. One way to present this information is as type signatures decorated with the analysis information. Going one step further, these signatures can be supplied by the programmer so that the compiler can either verify that it can derive that information (showing that the programmer's expectations are correct). In other situations, the progrsammer may want the compiler to take this information at face value, and to use it to generate more efficient code (albeit potentially unsafe, this is not any different from allowing programmers in Haskell to use `seq` to enforce strictness.)

Here again, we have a situation in which the compiler needs to communicate with the programmer and the same problems that have bothered implementors of statically typed functional languages show up in another guise, but also bring their own set of challenges to the table.

References

[1] J. Hage, S. Holdermans, and A. Middelkoop. 2007. A generic usage analysis with subeffect qualifiers. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1–3, 2007.* ACM Press, 235–246.

[2] S. Holdermans and J. Hage. 2010. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *Proceedings of the 15th ACM SIGPLAN 2010 International Conference on Functional Programming (ICFP '10).* ACM Press, 63–74.

[3] S. Holdermans and J. Hage. 2011. Making "stricterness" more relevant. *Higher-Order and Symbolic Computation* 23 (2011), 315–335. Issue 3.

[4] Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Comput. Surv.* 28, Article 196 (December 1996). Issue 4es.

[5] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, New York, NY, USA, 47–57.

[6] R. Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17 (1978), 348–375.

[7] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366.

[8] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Proceedings.* 672–698.

[9] Wouter Swierstra. 2018. Programming with dependent types. (2018).

[10] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'06).* ACM Press, 50–61.

Towards Reliable Concurrent Software

Marieke Huisman
University of Twente
m.huisman@utwente.nl

Abstract

As the use of concurrent software is increasing, we urgently need techniques to establish the correctness of such applications. Over the last years, significant progress has been made in the area of software verification, making verification techniques usable for realistic applications. However, much of this work concentrates on sequential software, and a next step is necessary to apply these results also on realistic concurrent software. In this abstract, we argue that current techniques for verification of concurrent software need to be further developed in multiple directions: extending the class of properties that can be established, improving the level of automation that is available for this kind of verification, and enlarging the class of programs that can be verified.

1 Introduction

Software is everywhere! Every day we use and rely upon enormous amounts of software, It has become impossible to imagine what life would be like without software. This creates the risk that one day software failures will bring our everyday life to a grinding halt. In fact, all software contains errors that cause it to behave in unintended ways [4, 6], and substantial research is needed to to help software developers to make software that is reliable under all circumstances, without compromising its performance.

A commonly used approach to improve software performance is the use of *concurrency* and *distribution*. For many applications, a smart split into parallel computations can lead to a significant increase in performance. Unfortunately, parallel computations make it more difficult to guarantee *reliability* of the software. The consequence is unsettling: the use of concurrent and distributed software is widespread, because it provides efficiency and robustness, but the unpredictability of its behaviour makes that errors can occur at unexpected, seemingly random moments.

The quest for reliable software builds on a long history, and significant progress has already been made. Nevertheless, ensuring reliability of efficient software remains an open challenge. Ultimately, it is our dream that program verification techniques are built into software development environments. When a software developer writes a program, he explicitly writes down the crucial desired properties about the program, as well as the assumptions under which the different program components may be executed. Continuously, an automatic check is applied to decide whether the desired properties are indeed established, and whether the assumptions are respected. If this is not the case, this is shown to the developer – with useful feedback on why the program does not behave as intended.

2 Abstraction Techniques for Functional Verification

One of the main challenges for the verification of concurrent software that we see is to automatically verify *global functional* correctness properties of concurrent software. To reach this goal, we advocate an approach where a *mathematical model* of a concurrent application is constructed, which provides an *abstract view* of the program's behaviour, leaving out details that are irrelevant for the properties being checked [3, 7], see Figure 1. The main verification steps in this approach are

1. *algorithmic verification* over the mathematical model to reason about global program behaviour, and
2. *program logics* to verify the formal connection between the software and its mathematical model.

Typically, the basic building blocks of the abstract mathematical model are *actions*, for which we can prove a correspondence between abstract actions and concrete code fragments. A *software designer* specifies the desired *global properties* for a given application in terms of abstract actions. The *software developer* then specifies how these *abstract actions map to concrete program state*: in which states is the action allowed, and what will be its effect on the program state. Global properties may be safety properties, *e.g.*, an invariant relation between the values of variables in different components, or a complicated protocol specifying correct interface usage, but we believe that extensions of the approach to liveness and progress properties are also possible.

To further develop this approach and make it scale, we believe the following challenges should be addressed:

1. identify a good abstraction theory,
2. extend the abstraction theory to reason about progress and liveness properties of code, and
3. use the abstraction theory to guide the programmer to develop working code through refinement.

3 Automating the Verification Process

Another major challenge is how to automate the verification process. At the moment, program verification requires many user annotations, explicitly describing properties which are often obvious to developers. We believe that many of the required annotations can be generated automatically, using a combination of appropriate static analyses and smart heuristics.

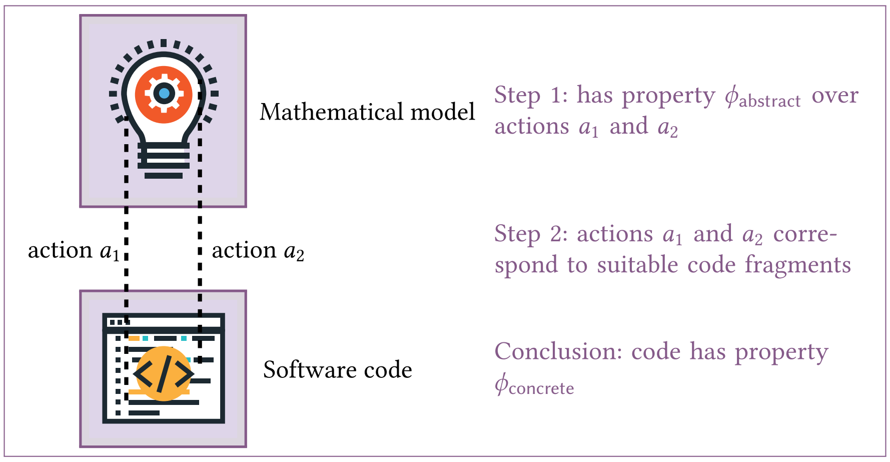

Mathematical model
action a_1
action a_2
Software code
Step 1: has property ϕ_{abstract} over actions a_1 and a_2
Step 2: actions a_1 and a_2 correspond to suitable code fragments
Conclusion: code has property ϕ_{concrete}

Figure 1. Using abstraction for the verification of concurrent and distributed software

We advocate a very pragmatic approach to annotation generation, where any technique that can be used to reduce the annotation burden is applied, combined with a smart algorithm to evaluate the usability of a generated annotation, removing any annotations that do not help automation. This will lead to a framework where for a large subset of non-trivial programs, we can automatically verify many common safety properties (absence of null-pointer dereferencing, absence of array out of bounds indexing, absence of data races etc.), and if we wish to verify more advanced functional properties, the developer might have to provide a few crucial annotations, but does not have to spell out in detail what happens at every point in the program (in contrast to current program verification practice).

4 Verification of Programs using Different Concurrency Paradigms

Finally, verification techniques need to support different programming languages, and different concurrency paradigms. In particular, we believe that it is important to investigate how to reason about programs written using the structured parallel programming model where all threads execute the same instructions. Recently, we have shown how our verification techniques can be adapted in a straightforward manner to GPUs (including atomic update instructions) [1, 2]. It turns out that the restricted setting of a GPU has a positive impact on verification: the same verification techniques can be used, and verification actually gets simpler. We believe that this direction should be explored further, as typical GPU programs are usually quite low-level, which makes them more error-prone.

An interesting extension of this work is to automatically transform a verified sequential program with annotations into an annotated GPU program, which will be directly verifiable [5]. We believe this idea can also be used for other compiler optimisations, such that they do transform not only the program, but also the correctness annotations, such that the result is a (hopefully) verifiable program again. Instead of proving correctness of the transformation, both the program and the annotations are transformed, such that after the transformation the resulting program with annotations can be reverified.

References

[1] S. Blom, S. Darabi, and M. Huisman. 2015. Verification of loop parallelisations. In *FASE (LNCS)*, Vol. 9033. Springer, 202–217.

[2] S. Blom, M. Huisman, and M. Mihelčić. 2014. Specification and Verification of GPGPU programs. *Science of Computer Programming* 95 (2014), 376–388. Issue 3.

[3] S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. 2015. History-based verification of functional behaviour of concurrent programs. In *SEFM (LNCS)*, Vol. 9276. Springer, 84 – 98.

[4] Archana Ganapathi and David A. Patterson. 2005. Crash Data Collection: A Windows Case Study.. In *Dependable Systems and Networks (DSN)* (2005-08-01). IEEE Computer Society, 280–285.

[5] M. Huisman, S. Blom, S. Darabi, and M. Safari. 2018. Program Correctness by Transformation. In *8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA) (LNCS)*. Springer. To appear.

[6] Rivalino Matias, Marcela Prince, Lúcio Borges, Claudio Sousa, and Luan Henrique. 2014. An Empirical Exploratory Study on Operating System Reliability. In *29th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 1523–1528. https://doi.org/10.1145/2554850.2555021

[7] W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. 2017. An Abstraction Technique for Describing Concurrent Program Behaviour. In *VSTTE (LNCS)*, Vol. 10712. Springer, 191 – 209.

Saving the World

A long and windy road towards sustainability and formal verification in practice

Marko van Eekelen
Full Professor, Head of Department
Computer Science Department
Faculty of Management, Science and Technology; Open University of The Netherlands
Heerlen
Marko.vanEekelen@ou.nl
Associate Professor
Digital Security
Institute for Computing and Information Sciences, Radboud University
Nijmegen
marko@cs.ru.nl

Abstract

My research is spread over two universities, mainly in the following two different research areas:

- *Resource consumption analysis*
- *Formal verification methods for verifying security and correctness in cyber physical systems.*

1 Introduction

1.1 Computer Science Research at the Open University

The OU CS department has an emerging research group within the Netherlands. Only since 2009 de OU formally has a disciplinary research task. Since 2014 the author is chairing the OU Computer Science Department with the intention that research at the Open University is just as important as education. This has lead to a growth of the department to a current size of over 30 members (4 full professors, 1 associate professor, 17 assistant professors, 5 lecturers, 4 postdocs, 13 external Ph.D. students) performing research in 3 focal points:

- Learning (in 3 topics: Tools for Supporting Learning, Computer Science Education, Computer Science Didactics),
- Resilience - Trustworthy Systems (in 2 topics: Verification, Security & Privacy)
- Innovation (Artificial Intelligence, Machine Learning).

The research is embedded in the Faculty of Management, Science and Technology promoting a culture of interdisciplinary research.

The members of the department recently acquired several grants (on Regional, National, European and American level) among which a Rubicon and a Veni grant.

2 Resource Consumption Analysis

Functional properties of programs are widely studied. It is however less common to study non-functional properties of code. Recently, the resources studied are diversifying [12]. In particular, the study of the consumption of other resources than time is an opening field. Studying resources such as memory and energy seems to the most promising [22].

From the practical point of view, the results discussed in [16] improve polynomial resource analysis of computer programs as presented in [19]. There the authors consider the size of output as a polynomial function on the sizes of inputs [18, 21]. In the NL NWO AHA project (2006-2011), the EU Charter Artemis project (2009-2012) and the NL GoGreen IOP GenCom project (2011-2015) the ResAna tool [10, 15, 20, 25] was developed that applies polynomial interpolation to generate an upper bound on Java loop iterations. The tool requires the user to input the degree of the solution. In [16] a partial result for that was provided. The results of recent work [17] make it possible to automatically obtain the degree of the polynomial in all cases for quadratic algebraic difference equation with constant coefficients.

Building upon this work, the focus moved from size, memory and loop bounds to sustainability of software [24] in general and of energy consumption analysis in particular.

2.1 A Moral Appeal

Computer Science is not the most sustainable discipline, to say the least. Every few years new equipment 'has' to be bought. The energy consumption due to digital equipment is seldom an issue. In software development energy consumption is rarely an issue. Instead of paying attention to the sustainability of software in such a way that an important design concern is that during the software life cycle as less as possible energy is consumed, the sole focus seems to be to keep legacy systems running in terms of functionality whatever the influence is on energy consumption.

As a discipline we need to do better with respect to sustainability. In fact, I would like to make a moral appeal for performing research in the area of energy analysis consumption paraphrasing famous words of John F. Kennedy:

"*And so my fellow Formal Method researchers: ask not what the world can do to reduce the energy consumption for you - ask how you can apply Formal Methods to reduce the energy consumption of the world: ask not what other researchers will do for you, but what together we can do for reducing the energy consumption of man.*"

The good news is that interest in energy consumption and in greenIT in the Netherlands is growing, e.g. at the Free University of Amsterdam [13], at the Software Improvement Group [7], at Utrecht University [5, 6] and at the Open University.

2.2 Energy Consumption Analysis at the Open University

Building upon practical resource analysis work [8] a research track on static analysis of energy consumption. This started with defining a suited Hoare logic that enabled a safely approximating static analysis [9]. This resulted in a webtool, ECAlogic [14], that made it possible to derive energy consumption bounds for small systems (hardware components controlled by a software application) in a hardware-parametric way. Due to this work the focus of the research changed to analysing IT controlled systems parametrised by hardware finite state machine models [3]. The corresponding approach was to focus on systems with multiple components, model the components and analyse the control software to estimate the energy consumption of the system. Using dependent types the analysis was made ready for a practical, precise and parametric energy analysis of IT controlled systems [4]. In working towards actual practice a first, small case study revealed that instead of doing a full analysis it can be worthwhile to focus solely on finding energy hot spots and energy bugs [2].

The OU memory and energy consumption analysis work was disseminated at the 2013 IPA Winterschool on Software Technology in Eindhoven, at the 2016 EU COST action TACLe Summerschool in Vienna and at the 2017 IPA Fall Days on System and Software Analysis.

2.3 Formal verification methods for verifying security and correctness in cyber physical systems at Radboud Unversity

My Radboud research in formal verification started with work on a dedicated proof assistant for the functional programming language Clean with special support for generic type classes and explicit strictness [1, 23, 26]. In the context of LaQuSO (Laboratory for Quality Software) we were able to verify the core decision algorithm of the Dutch Storm Surge Barrier 'Maeslantkering' protecting the Rotterdam area against flooding. The algorithm was formally specified in Z. We checked the code against the specification and we validated the specification. As a result firstly some minor changes were needed both in the specification and in the code and secondly a scenario popped up from model checking in which the barrier would not close according to the specification while it should close according to the experts [11]. Everything was fixed such that the Dutch are saved from 'getting their feet wet'.

Figure 1. Maeslantkering.

Currently, together with Herman Geuvers I am leading the STW Sovereign project (2016-2020) supported by RWS (the Dutch ministry of Transport, Public Works and Water Management) and NRG (the Dutch Nuclear Research Group). The goal of this project is to develop verification techniques for safety critical software based on the following challenging principles. Verification should be (1) scalable (costs should not grow exceedingly as the size of the system increases), (2) compositional (global properties are directly inferable from local properties of the subsystems), (3) incremental (the verification process can be performed iteratively while previous intermediate results are still usable), and (4) effective (the proposed methodology will be applied successfully in some real-world case studies). The fundamental idea of our proposal can be illustrated best with our motto: ÕScalability through modularityÕ. Modularity is commonly recognized as the key for managing complex software systems. With regards to programs, we will elaborate on the concept of design pattern (a description of a solution to a recurring problem) as a modularizing construct. We will investigate both general and security specific design patterns, and develop accompanying proof patterns that simplify the formal verification process. Moreover, as an follow-up of our work on the formalization of the C11 standard, we aim to make an important step in improving the scalability of the C verification process.

3 Moral Discussion

Answer the following questions:

- Is n't it about time that IT starts saving the world instead of consuming it?

- Is n't it about time that IT starts saving the world before it is too late?
- Is n't it about time that designs and implementations of safety critical cyber physical systems are subject to formal verification on a regular basis?

With every 'yes' we contribute to saving the world....

References

[1] Maarten de Mol and Marko C. J. D. van Eekelen. 1999. A Proof Tool Dedicated to Clean - The First Prototype. In *Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE'99, Kerkrade, The Netherlands, September 1-3, 1999, Proceedings.* 271–278. https://doi.org/10.1007/3-540-45104-8_22

[2] Pascal van Gastel, Bernard van Gastel, and Marko van Eekelen. 2018. Detecting Energy Bugs and Hotspots in Control Software Using Model Checking. In *Conference Companion of the 2Nd International Conference on Art, Science, and Engineering of Programming (Programming Along the System Stack &#39;18 Companion).* ACM, New York, NY, USA, 93–98. https://doi.org/10.1145/3191697.3213805

[3] Bernard van Gastel, Rody Kersten, and Marko van Eekelen. 2016. Using dependent types to define energy augmented semantics of programs. In *Proceedings of the Fourth International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'15) (LNCS)*, Vol. 9964. Springer, 1–20. https://doi.org/10.1007/978-3-319-46559-3_2

[4] Bernard van Gastel and Marko van Eekelen. 2017. Towards practical, precise and parametric energy analysis of IT controlled systems. In *Proceedings of the Fifth International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'17).*

[5] Erik Jagroep, Jordy Broekman, Jan Martijn E. M. van der Werf, Patricia Lago, Sjaak Brinkkemper, Leen Blom, and Rob van Vliet. 2017. Awakening Awareness on Energy Consumption in Software Engineering. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Society Track, ICSE-SEIS 2017, Bueons Aires, Argentina, May 20-28, 2017.* 76–85. https://doi.org/10.1109/ICSE-SEIS.2017.10

[6] Erik Jagroep, Jan Martijn E. M. van der Werf, Slinger Jansen, Miguel Alexandre Ferreira, and Joost Visser. 2015. Profiling energy profilers. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015.* 2198–2203. https://doi.org/10.1145/2695664.2695825

[7] Georgios Kalaitzoglou, Magiel Bruntink, and Joost Visser. 2014. A Practical Model for Evaluating the Energy Efficiency of Software Applications. In *ICT for Sustainability 2014 (ICT4S-14), Stockholm, Sweden, August 25, 2014.* https://doi.org/10.2991/ict4s-14.2014.9

[8] Rody Kersten, Olha Shkaravska, Bernard van Gastel, Manuel Montenegro, and Marko van Eekelen. 2012. Making resource analysis practical for real-time Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES) (JTRES'12)*, Martin Schoeberl and Andy J. Wellings (Eds.). ACM, New York, NY, USA, 135–144. https://doi.org/10.1145/2388936.2388959

[9] Rody W.J. Kersten, Paolo Parisen Toldin, Bernard E. van Gastel, and Marko C.J.D. van Eekelen. 2014. A Hoare Logic for Energy Consumption Analysis. In *Proceedings of the Third International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'13) (LNCS)*, Vol. 8552. Springer, 93–109. https://doi.org/10.1007/978-3-319-12466-7_6 Referenced in the thesis as [BvG-9], see appendix ??.

[10] Rody W. J. Kersten, Bernard van Gastel, Olha Shkaravska, Manuel Montenegro, and Marko C. J. D. van Eekelen. 2014. ResAna: a resource analysis toolset for (real-time) JAVA. *Concurrency and Computation: Practice and Experience* 26, 14 (2014), 2432–2455. https://doi.org/10.1002/cpe.3154

[11] Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. 2010. A Formal Verification Study on the Rotterdam Storm Surge Barrier. In *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings.* 287–302. https://doi.org/10.1007/978-3-642-16901-4_20

[12] Reinhard Wilhelm Florian Zuleger Marco Gaboardi, Jan Hoffmann (Ed.). 2018. Resource Bound Analysis: Report from Dagstuhl Seminar 17291. *Dagstuhl Reports* (2018).

[13] Fahimeh Alizadeh Moghaddam, Patricia Lago, and Iulia Cristina Ban. 2018. Self-adaptation approaches for energy efficiency: a systematic literature review. In *Proceedings of the 6th International Workshop on Green and Sustainable Software, GREENS@ICSE 2018, Gothenburg, Sweden, May 27, 2018.* 35–42. https://doi.org/10.1145/3194078.3194084

[14] Marc Schoolderman, Jascha Neutelings, Rody W.J. Kersten, and Marko C.J.D. van Eekelen. 2014. ECAlogic: Hardware-parametric Energy-consumption Analysis of Algorithms. In *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages (FOAL'14).* ACM, New York, NY, USA, 19–22. https://doi.org/10.1145/2588548.2588553

[15] Olha Shkaravska, Rody Kersten, and Marko C. J. D. van Eekelen. 2010. Test-based inference of polynomial loop-bound functions. In *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java, PPPJ 2010, Vienna, Austria, September 15-17, 2010*, Andreas Krall and Hanspeter Mössenböck (Eds.). ACM, 99–108. https://doi.org/10.1145/1852761.1852776

[16] O. Shkaravska and M. van Eekelen. 2014. Univariate polynomial solutions of algebraic difference equations. *Journal of Symbolic Computation* 60 (2014), 15 – 28. https://doi.org/10.1016/j.jsc.2013.10.010

[17] O. Shkaravska and M. van Eekelen. 2018. Polynomial solutions of algebraic difference equations and homogeneous symmetric polynomials. *Journal of Symbolic Computation* (2018). Under Submission.

[18] Olha Shkaravska, Marko C. J. D. van Eekelen, and Alejandro Tamalet. 2013. Collected Size Semantics for Strict Functional Programs over General Polymorphic Lists. In *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers (Lecture Notes in Computer Science)*, Ugo Dal Lago and Ricardo Peña (Eds.), Vol. 8552. Springer, 143–159. https://doi.org/10.1007/978-3-319-12466-7_9

[19] Olha Shkaravska, Marko C. J. D. van Eekelen, and Ron van Kesteren. 2009. Polynomial Size Analysis of First-Order Shapely Functions. *Logical Methods in Computer Science* 5, 2 (2009). http://arxiv.org/abs/0902.2073

[20] Olha Shkaravska, Ron van Kesteren, and Marko C. J. D. van Eekelen. 2007. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science)*, Simona Ronchi Della Rocca (Ed.), Vol. 4583. Springer, 351–365. https://doi.org/10.1007/978-3-540-73228-0_25

[21] Alejandro Tamalet, Olha Shkaravska, and Marko C. J. D. van Eekelen. 2008. Size Analysis of Algebraic Data Types. In *Proceedings of the Nineth Symposium on Trends in Functional Programming, TFP 2008, Nijmegen, The Netherlands, May 26-28, 2008. (Trends in Functional Programming)*, Peter Achten, Pieter W. M. Koopman, and Marco T. Morazán (Eds.), Vol. 9. Intellect, 33–48.

[22] Marko van Eekelen. 2018. ECA: Energy Consumption Analysis of software controlled systems, In Resource Bound Analysis: Report from Dagstuhl Seminar 17291, Reinhard Wilhelm Florian Zuleger Marco Gaboardi, Jan Hoffmann (Ed.). *Dagstuhl Reports*, 84.

[23] Marko C. J. D. van Eekelen and Maarten de Mol. 2005. Proof Tool Support for Explicit Strictness. In *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers.* 37–54. https://doi.org/10.1007/11964681_3

[24] Bernard van Gastel. 2016. *Assessing sustainability of software.* Ph.D. Dissertation. Open University of the Netherlands.

[25] Ron van Kesteren, Olha Shkaravska, and Marko C. J. D. van Eekelen. 2008. Inferring Static Non-monotone Size-aware Types Through Testing. *Electr. Notes Theor. Comput. Sci.* 216 (2008), 45–63. https://doi.org/10.1016/j.entcs.2008.06.033

[26] Ron van Kesteren, Marko C. J. D. van Eekelen, and Maarten de Mol. 2004. Proof support for generic type classes. In *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004, München, Germany, 25-26 November 2004.* 1–16.

Research Challenges in Supervisory Control Theory

Michel Reniers
Eindhoven University of Technology
M.A.Reniers@tue.nl

1 Introduction

In the era of Cyber-Physical Systems (of Systems) and the Internet of Things, much effort goes into the development of supervisory controllers that need to provide a safe (and efficient) coordination of subsystems.

One of the approaches that has been researched in the past decades is *supervisory control theory* [9], where based on a model of the uncontrolled system and a model of the (safety) requirements, a model of the supervisory controller is generated such that the controlled system (i.e., the uncontrolled system under the control of the supervisory controller) is (1) controllable, (2) safe w.r.t. the requirements, and (3) nonblocking. Traditionally, SCT works with so-called discrete-event systems, typically in the form of (extended) finite automata [3, 12].

In this short position paper, research challenges that are currently worked on and for which progress is expected / needed in the near future are described. This is not intended to be an overview of all the activities that take place in the field, but merely presents the authors current view on some interesting ones.

2 Core challenges in SCT

Challenges in the area of supervisory control synthesis, in no particular order, are

1. development of a *discipline of modelling* that facilitates use of the model-based engineering approach towards supervisory control synthesis;
2. scalability of supervisory controller synthesis;
3. expressivity of requirements;
4. development of synthesis techniques for networked supervisors, i.e., supervisors that are connected with the uncontrolled system by means of a network with its inherent communication characteristics (e.g., communication delays and losses);
5. integration of performance optimization techniques and supervisory controller synthesis.

2.1 A discipline of modelling for SCT

Inspired by experiences from application of supervisory control synthesis to relevant cases such as manufacturing systems [15], automotive systems [7], and waterway locks [11], recent (unpublished) research attempts to provide a set of sufficient conditions on models of uncontrolled system and requirements that provide a trivial controllable system without blocking.

Of course there are also relevant systems for which these conditions do not hold, and it is important to establish heuristics for modelling such systems that on the one hand allow application of supervisory controller synthesis, and on the other hand make the modelling effort itself manageable in terms of compositionallity and evolvability.

It is our ambition to formulate a discipline of modelling that allows practical and efficient application of supervisory control synthesis.

2.2 Scalability of supervisory controller synthesis

Scalability issues with monolithic synthesis algorithms have led to the study of techniques that decompose the synthesis problem into a number of smaller synthesis problems from the solution of which a supervisor can be obtained. Most of these techniques require a creative/manual effort to decide the decomposition of the system.

In recent research, Design Structure Matrices and clustering techniques are used to obtain a decomposition automatically [5]. Still much research is needed to provide guarantees for the supervisor obtained by composing the supervisors for the subproblems as in general nonblockingness and maximal permissiveness are sacrificed in such an approach.

Ideas for a form of compositional synthesis (involving intermediate abstractions and synthesis steps) are emerging in literature [4] and are actively researched by research groups involved with the tool sets Supremica [1] and CIF [2].

Adaptation of the well-known partial-order reduction techniques from model checking for use in the domain of supervisory control theory are studied in [14]. Such reductions need to preserve properties such as controllability and nonblocking, which are not standard in model checking in general.

2.3 Expressivity of requirements

Traditional synthesis is restricted to requirements specified by (extended) finite automata and state-based expressions. In recent years there have been some attempts to generalize the synthesis to requirements in fragments of temporal logics such as LTL and modal μ-calculus (see, e.g., [6]), but still much more expressivity is needed to capture meaningful behavioural requirements.

For (mechanical and control) engineers, capturing informal requirements in formal models is more than a challenge, and much better support is needed in formulating such properties. Candidates are formulation of properties in terms of scenarios-based formalisms such as life sequence charts. Validation of complex requirements is hardly supported by tool sets in the domain of supervisory control.

2.4 Synthesis of networked supervisors

A basic assumption in the supervisory control theory framework is that the supervisor and uncontrolled system interact synchronously. Practical applications require to relax this assumption since mostly there is some communication medium between supervisory controller and (parts of) the uncontrolled system. Such communication media introduce asynchronicity between plant and supervisor and may result in overtaking of messages and even message losses. Under such conditions obtaining a safe and nonblocking supervisory control becomes more challenging, obviously. Initial work is reported (see [10], and references therein), but both conceptually as well as in terms of applicability many improvements are still expected and required before such theories may become usable.

2.5 Synthesis of performance-optimal supervisors

It would be interesting and practically relevant if we could combine techniques for obtaining a supervisor that provides functional properties of the system and techniques for obtaining a supervisor that adheres to some performance properties, such as a guaranteed throughput. For fully controllable systems, promising first results have been developed in [13], and for partially controllable systems, initial work is given in [8].

References

[1] K. Åkesson, M. Fabian, H. Flordal, and R. Malik. 2006. Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems. In *2006 8th International Workshop on Discrete Event Systems*. 384–385. https://doi.org/10.1109/WODES.2006.382401

[2] D.A. van Beek, W. J. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J.M. van de Mortel-Fronczak, and M.A. Reniers. 2014. CIF 3: Model-based engineering of supervisory controllers. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2014), 575–580. https://doi.org/10.1007/978-3-642-54862-8_48

[3] Christos G. Cassandras and Stephane Lafortune. 2008. *Introduction to Discrete Event Systems*. Springer. 769 pages. https://doi.org/10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C arXiv:arXiv:1011.1669v3

[4] Hugo Flordal, Robi Malik, Martin Fabian, and Knut Åkesson. 2007. Compositional synthesis of maximally permissive supervisors using supervision equivalence. *Discrete Event Dynamic Systems: Theory and Applications* 17, 4 (2007), 475–504. https://doi.org/10.1007/s10626-007-0018-z

[5] Martijn Goorden, Joanna M. van de Mortel-Fronczak, Michel A. Reniers, and Jacobus E. Rooda. 2017. Structuring multilevel discrete-event systems with dependency structure matrices. In *56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, December 12-15, 2017*. 558–564. https://doi.org/10.1109/CDC.2017.8263721

[6] A.C. van Hulst, Michel A. Reniers, and Wan J. Fokkink. 2017. Maximally permissive controlled system synthesis for non-determinism and modal logic. *Discrete Event Dynamic Systems* 27, 1 (2017), 109–142. https://doi.org/10.1007/s10626-016-0231-8

[7] Tim Korssen, Victor S. Dolk, Joanna M. van de Mortel-Fronczak, Michel A. Reniers, and Maurice Heemels. 2018. Systematic Model-Based Design and Implementation of Supervisors for Advanced Driver Assistance Systems. *IEEE Trans. Intelligent Transportation Systems* 19, 2 (2018), 533–544. https://doi.org/10.1109/TITS.2017.2776354

[8] Berend Jan Christiaan van Putten. 2018. *Tackling Uncontrollability in the Specification and Performance of Manufacturing Systems*. Master's thesis. Eindhoven University of Technology. Available online: `michelreniers.files.wordpress.com/2018/06/masterthesisvanputten2018.pdf`.

[9] P.J. Ramadge and W.M. Wonham. 1987. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization* 1987, 1 (1987), 206–230. https://doi.org/10.1109/ROBOT.2001.932624

[10] Aida Rashidinejad, Michel Reniers, and Lei Feng. 2018. Supervisory Control of Timed Discrete Event Systems Subject to Communication Delays and Non-FIFO Observations. In *14th International Workshop on Discrete Event Systems WODES 2018*. IEEE.

[11] F. F. H. Reijnen, M. A. Goorden, J. M. van de Mortel-Fronczak, and J. E. Rooda. 2017. Supervisory control synthesis for a waterway lock. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*. 1562–1563. https://doi.org/10.1109/CCTA.2017.8062679

[12] Markus Sköldstam, Knut Åkesson, and Martin Fabian. 2007. Modeling of discrete event systems using finite automata with variables. In *46th IEEE Conference on Decision and Control, CDC 2007, New Orleans, LA, USA, December 12-14, 2007*. 3387–3392. https://doi.org/10.1109/CDC.2007.4434894

[13] Bram van der Sanden, João Bastos, Jeroen Voeten, Marc Geilen, Michel Reniers, Twan Basten, Johan Jacobs, and Ramon Schiffelers. 2016. Compositional Specification of Functionality and Timing of Manufacturing Systems. In *2016 Forum on Specification and Design Languages (FDL)*. Bremen, Germany.

[14] Bram van der Sanden, Marc Geilen, Michel Reniers, and Twan Basten. 2018. Partial-Order Reduction for Supervisory Controllers. (2018). In submission.

[15] Bram van der Sanden, Michel Reniers, Marc Geilen, Twan Basten, Johan Jacobs, Jeroen Voeten, and Ramon Schiffelers. 2015. Modular model-based supervisory controller design for wafer logistics in lithography machines. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MODELS 2015 - Proceedings*. 416–425. https://doi.org/10.1109/MODELS.2015.7338273

Sound and modular formal methods

Robbert Krebbers
Delft University of Technology
Delft, The Netherlands

Abstract

This short paper is a contribution to the Lorentz Center workshop on “A Research Agenda for Formal Methods in The Netherlands” on September 3 and 4, 2018 in Leiden.

1 Introduction

Depending on the way software is used, and the resources that are available to ensure its quality, it is desired to establish different correctness properties. In some cases, safety properties may be sufficient, whereas in other cases (in particular, mission critical software), it would be desired to establish full functional correctness.

As such, I believe it is important that the formal methods community in the Netherlands develops a large variety of formal methods techniques and tools—ranging from static analysis, type systems and model checkers, to methods for establishing full functional correctness. This way, we can make sure that we—as the Dutch formal methods community—cover the whole spectrum.

Regardless of the actual tools or techniques that are being used, it is crucial that these are *sound* and *modular*. Soundness means that whenever the tool says that software satisfies a certain property, that is undoubtedly the case. After all, we do not want to give false pretenses—software that is said to be formally verified should not go wrong. To achieve this, we should use sound mathematical principles as the basis to build tools and develop techniques. On top of that, we should make sure that the implementations of the tools we are using are mathematically verified.

Modularity means that each component of a piece of software can be verified in isolation, and that when all of the components are combined into a larger piece of software, we obtain the intended correctness property for the software as a whole. Modularity is crucial to achieve scalability, which is crucial to use formal methods in the large.

2 Current research

My research has focused on the following topics:

1. How to develop formal semantics of realistic programming languages (like C and Rust), and use said formal semantics to reason about such languages.
2. How to develop *separation logics* and *logical relations* that support modular reasoning about daring programming features such as fine-grained concurrency, higher-order functions, non-local control, *etc.*
3. How to modularly establish meta theoretical properties of *whole* programming languages, like type safety.
4. How to develop tools that enable convenient and proved sound reasoning about programs.

As the basis of my research I am using proof assistants, which can be used to specify programming languages, to reason about programs, and to validate mathematical results, in the most reliable and trustworthy way. I am an active user of the Coq proof assistant [2] and nearly all of my recent research has been entirely formalized using it.

Below I will list some noteworthy research projects that I have been involved in:

- As part of my PhD [9, 10, 13], I have developed a formal semantics of a large part of the C programming language, based on the official specification of C from the C11 standard. My C semantics, called **CH_2O**, comes in the form of a type system, an operational and executable semantics, and a separation logic. All of these components have been defined in Coq and are proved to match up with each other.
- I am one of the main developers of **Iris** [5, 6, 11]—a framework for higher-order concurrent separation logic, which has been implemented in the Coq proof assistant and deployed very effectively in a wide variety of verification projects world-wide. Among many other things, we have used Iris to establish the correctness of the Rust type system and some of its standard libraries [4], and to develop an expressive logic to reason about refinements of concurrent programs [3].
- I have worked on so-called *tactic* languages for carrying out proofs in a proof assistant. Notably, I have co-developed the tactic languages **Iris Proof Mode** [12] and **MoSeL** [8] for separation logic proofs in Coq, and have co-developed **Mtac2** [7], a dependently typed language for safe tactic programming in Coq.
- Most of the aforementioned results use separate proofs to establish properties of programs and programming languages. In other work [14], we have investigated how to use dependent types to define programming language specifications so that certain properties (like type safety) hold *by construction.*

3 Future research

There are many directions for future work in formal methods. In this section I will a (non-exhaustive) list of some directions that I plan to work on in the coming years:

- I have recently been awarded an NWO **Veni** grant to apply formal verification to programs written in a combination of different programming languages.

This is needed, because actual software is not written in a single programming language, but consist of many components written in different languages that interact with each other. As part of the Veni, I will develop formal semantics and reasoning tools for shared-memory interaction via foreign function interfaces, and message passing via sockets or signals, and apply this to the web, where programming language interaction is omnipresent.

- A lot of the formal verification research has focused on functional correctness, *i.e.,* that the program has the correct output given some input. However, I think we should go beyond that, and develop sound and modular techniques and tools for establishing non-functional properties such as resource usage, complexity, security properties like non-interference, *etc.* In a recent manuscript, we have developed a separation logic to establish correct disposal of resources in a programming language with concurrency [1].

References

[1] Aleš Bizjak, Daniel Gratzer, Krebbers, Robbert, and Lars Birkedal. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic, 2018. Manuscript under submission.

[2] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2015. Available at https://coq.inria.fr/doc/.

[3] Dan Frumin, Robbert Krebbers, and Lars Birkedal. Reloc: A mechanised relational logic for fine-grained concurrency. In *LICS*, pages 442–451, 2018.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.

[5] Ralf Jung, Krebbers, Robbert, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic, 2018. Accepted to Journal of Functional Programming (JFP).

[6] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016.

[7] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: Typed tactics for backward reasoning in Coq. *PACMPL*, 2(ICFP):78:1–78:31, 2018.

[8] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL*, 2(ICFP):77:1–16:30, 2018.

[9] Robbert Krebbers. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *POPL*, pages 101–112, 2014.

[10] Robbert Krebbers. *The C standard formalized in Coq.* PhD thesis, Radboud University Nijmegen, 2015.

[11] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*, volume 10201 of *LNCS*, pages 696–723, 2017.

[12] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017.

[13] Robbert Krebbers and Freek Wiedijk. Separation Logic for Non-local Control Flow and Block Scope Variables. In *FoSSaCS*, volume 7794 of *LNCS*, pages 257–272, 2013.

[14] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *PACMPL*, 2(POPL):16:1–16:34, 2018.

Model checking in biology and health care

Rom Langerak
Formal Methods and Tools
University of Twente
r.langerak@utwente.nl

Abstract

Model checking is a useful technique for analyzing models in biology and health care. Here we point out several research topics concerning the modeling of various networks in biology, and modeling of diagnostic and treatment protocols in health care.

1 Introduction

Many complex biological phenomena can be modeled as networks. Prominent examples within biological cells are metabolic neworks, signaling networks, and gene regulatory networks. Understanding the quantitative and qualitative behavior of such networks is an important prerequisite for curing various diseases. Modeling these networks within a framework that allows model checking is an attractive way of gaining such understanding [Brim et al. 2013; David et al. 2015].

In Twente the contribution of computer science to this research has concentrated on the use of UPPAAL [UPPAAL website 2018]. UPPAAL is attractive as it is a mature tool allowing a compositional approach, and with a graphical interface that facilitates communication with non-experts in formal methods.

Kinase signaling networks have been modeled in the context of osteoarthritis [Scholma et al. 2013, 2014]. A tool called ANIMO has been created [ANIMO 2018] that makes use of UPPAAL and is intended to be used by molecular biologists. ANIMO has been succesfully used to model realistically sized biological networks [Schivo et al. 2014; Schivo et al. 2016, 2013; Schivo et al. 2012; Scholma et al. 2014]. Below we list several research topics for the modeling and analysis of biological networks.

Another line of research is modeling protocols for diagnosis and treatment, and then using model checking for analysing these protocols for effectiveness, efficiency, and costs. In Twente such analysis has been performed using UPPAAL for prostrate cancer [Degeling et al. 2017; Schivo et al. 2015] and tooth wear [Choudry 2018; van Rooijen 2018] (in collaboration with the Academic Centre for Dentistry Amsterdam). Below we list several research topic for the modeling and analysis of diagnosis and treatment protocols.

2 Research topics for modeling and analysis of biological networks

Tools Modeling and analysis techniques should be offered to biologists and medical researchers via tools that hide as much as possible the technical complexities of the underlying computer science models. One way of achieving this is to try to stick as close as possible to user interfaces of the mostly informal network tools that have been developed already in biology.

Model generation Creating a model can be a time consuming task. Often the information needed for constructing a model can be found in literature or databases. Therefore we need to investigate techniques for building initial models from information that is extracted from heterogeneous sources. It is important to be able to deal with missing or incomplete information.

Parameter fitting and sensitivity Much of the effort in creating a model goes into providing the right parameters for a model. This is related to the issue of parameter sensitivity and robustness. Especially for parameters that vary from patient to patient is is important to establish that model properties are not critically dependent on such parameters.

Simulation and visualization Simulations are needed initially to gain confidence in the correctness of a model. Once confidence has been gained in the correctness, a model can be explored by simulating it with various stimuli. In this way hypotheses can be checked by in-silico experimentations. Challenges are the efficiency of such simulations, especcially if reactions may take place at different timescales. In addition, the way results are visually and graphically presented to researchers of medical practitioners is of crucial importance.

Relating models and experiments Experiments will always form an indispensible component of biological research, and modeling can geatly enhance the effectivity and efficiency of experiments. Models can be of great help0 in suggesting experiments and thereby pruning the large amount of possible experiments. Models are also important in interpretating the (often verly large) amount of experimental data - just indicating which data is in accordance with the model sofar, and which data is not, is already extremely useful. And more research should be performed on automatic suggestions for model improvement in the light of new experimentalo data.

Using model checking for drug synthesis For some goal in a network model, model checking can provide the stimuli that have to be offered to a network in order to reach that goal, by analyzing the trace leading to the goal. In this way model checking is a powerful technique supporting drug synthesis. Since network models may have an enormous state space, the challenge is to find abstraction and high performance computing technqiues that enable to check large scale network models.

3 Research topics for modeling and analysis of treatment protocols

Tools The ambition is to create a tool that enables health practitioners to create their own treatment protocols, and analyze them. This asks for a domain specific language that is both easy to use and sufficiently flexible and expressive to deal with many different scenarios.

Educated guesses for parameters Usually when modeling a treatment protocol many parameters are unknown or not known precisely, and it would be too costly or time consuming to establish such parameters by clinical trials. What is needed is a framework to deal with such educated guesses; by sensitivity analysis or parameter sweeps it could be established which parameters need to be established with more precision, and which parameters are not so crucial for the analysis outcomes.

Optimization It would be very useful to establish the optimal protocol under some constraint. What is the most effective protocol given a certain budget, or what is the most economic protocol that is able to obtain a given level of effectiveness? This asks for a theory of optimization of timed stochastic processes with costs.

References

ANIMO. 2018. http://fmt.cs.utwente.nl/tools/animo. (2018).

Luboš Brim, Milan Češka, and David Šafránek. 2013. Model Checking of Biological Systems. In *Formal Methods for Dynamical Systems: 13th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2013, Bertinoro, Italy, June 17-22, 2013. Advanced Lectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 63–112. https://doi.org/10.1007/978-3-642-38874-3_3

Umarah Choudry. 2018. *Timed Automata Modeling for the Tooth Wear Evaluation System*. Master's thesis. Academic Centre for Dentistry Amsterdam (ACTA), The Netherlands.

Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. 2015. Statistical model checking for biological systems. *International Journal on Software Tools for Technology Transfer* 17, 3 (2015), 351–367.

Koen Degeling, Stefano Schivo, Niven Mehra, Hendrik Koffijberg, Romanus Langerak, Johann de Bono, and Maarten Joost IJzerman. 2017. Comparison of Timed Automata with Discrete Event Simulation for Modeling of Biomarker-Based Treatment Decisions: An Illustration for Metastatic Castration-Resistant Prostate Cancer. *Value in health* 20, 10 (12 2017), 1411–1419. https://doi.org/10.1016/j.jval.2017.05.024

Stefano Schivo, K. Degeling, Koen Degeling, Hendrik Koffijberg, Maarten Joost IJzerman, and Romanus Langerak. 2015. PRM113 - Timed Automata Modeling of The Personalized Treatment Decisions In Metastatic Castration Resistant Prostate Cancer. In *ISPOR 18th Annual European Congress Research Abstracts (Value in Health)*. International Society for Pharmacoeconomics and Outcomes Research (ISPOR), A702–A703. https://doi.org/10.1016/j.jval.2015.09.2630 eemcs-eprint-26885.

S. Schivo, J. Scholma, H. B. J. Karperien, J. N. Post, J. C. van de Pol, and R. Langerak. 2014. Setting Parameters for Biological Models With ANIMO. http://eprints.eemcs.utwente.nl/24659/. In *Proceedings 1st International Workshop on Synthesis of Continuous Parameters, Grenoble, France (Electronic Proceedings in Theoretical Computer Science)*, É André and G. Frehse (Eds.), Vol. 145. Open Publishing Association, 35–47.

Stefano Schivo, Jetse Scholma, Paul E. van der Vet, Marcel Karperien, Janine N. Post, Jaco van de Pol, and Rom Langerak. 2016. Modelling with ANIMO: between fuzzy logic and differential equations. *BMC Systems Biology* 10, 1 (2016), 56. https://doi.org/10.1186/s12918-016-0286-z

Stefano Schivo, Jetse Scholma, Brend Wanders, Ricardo A. Urquidi Camacho, Paul E. van der Vet, Marcel Karperien, Rom Langerak, Jaco van de Pol, and Janine N. Post. 2013. Modelling biological pathway dynamics with Timed Automata. *IEEE Journal of Biomedical and Health Informatics* 18, 3 (2013), 832–839. https://doi.org/10.1109/JBHI.2013.2292880

S. Schivo, J. Scholma, B. Wanders, R. A. Urquidi Camacho, P. E. van der Vet, H. B. J. Karperien, R. Langerak, J. C. van de Pol, and J. N. Post. 2012. Modelling biological pathway dynamics with Timed Automata. http://eprints.eemcs.utwente.nl/22597/. In *12th IC on Bioinformatics and Bioengineering (BIBE 2012)*. IEEE Computer Society, 447–453.

J. Scholma, J. Kerkhofs, S. Schivo, R. Langerak, P. E. van der Vet, H. B. J. Karperien, J. C. van de Pol, L. Geris, and J. N. Post. 2013. Mathematical modeling of signaling pathways in osteoarthritis. http://eprints.eemcs.utwente.nl/23972/. In *2013 Osteoarthritis Research Society International (OARSI) World Congress, Philadelphia, USA*, S. Lohmander (Ed.), Vol. 21, Supplement. Elsevier, Amsterdam, S123–S123. https://doi.org/10.1016/j.joca.2013.02.259

J. Scholma, S. Schivo, J. Kerkhofs, R. Langerak, H. B. J. Karperien, J. C. van de Pol, L. Geris, and J. N. Post. 2014. ECHO: the executable chondrocyte. http://eprints.eemcs.utwente.nl/24845/. In *Tissue Engineering & Regenerative Medicine International Society, European Chapter Meeting, Genova, Italy*, Vol. 8. Wiley, Malden, 54–54.

Jetse Scholma, Stefano Schivo, Ricardo A. Urquidi Camacho, Jaco van de Pol, Marcel Karperien, and Janine N. Post. 2014. Biological networks 101: Computational modeling for molecular biologists. *Gene* 533, 1 (2014), 379–384. https://doi.org/10.1016/j.gene.2013.10.010

UPPAAL website. 2018. www.uppaal.org. (2018).

Jasper van Rooijen. 2018. *Sensitivity and Optimalisation of Uncertain Parameters in UPPAAL models*. Master's thesis. University of Twente, Enschede, The Netherlands.

Actionable Feedback during Software Development

Delft University of Technology
The Netherlands

Sebastian Erdweg

My team works on programming tools that supply developers with actionable feedback during software development. Feedback is actionable if it is relevant to the programmer's task, if the programmer can rely on its correctness, and if it arrives in a timely manner. By providing actionable feedback, we protect developers against introducing performance bottlenecks, unsafe code, security vulnerabilities, or specification violations. Our feedback also influences development tools such as compiler optimizations and refactorings. We tackle this challenging research program in two focus areas: incremental computing and practical correctness proofs.

1 Incremental computing

My team develops building blocks for incremental algorithms that achieve high performance when reacting to a change in their input. Rather than repeating the entire computation over the changed input, an incremental algorithm only updates those parts of the previous result that are affected by the input change. This way, incremental algorithms provide asymptotical speedups in theory and we have observed multiple orders of magnitude speedups in practice.

Incremental algorithms are crucial for providing actionable feedback because the feedback needs to be updated after every code change the developer makes. Yet, existing algorithms, such as the Java type checker in Eclipse JDT, are one-off solutions that required years of engineering that cannot be reproduced. We develop building blocks for incremental computing and collect them in frameworks that execute regular algorithms incrementally.

In the **IncA project** [13–15], we study algorithms for incremental program analysis. The basic idea is to store the syntax tree of programs in a relational database and to run incremental Datalog queries over these relations. However, incremental solvers of Datalog are inherently limited in expressiveness. In particular, they lack support for lattices, which are ubiquitous in program analysis. We were the first to discover techniques for incrementally solving lattice-based Datalog queries and we have applied our techniques to achieve order-of-magnitudes speedups when analyzing C, Java, and Rust code.

In the **PIE project** (formerly pluto) [2, 8, 9], we develop incremental build systems. Incremental build systems are essential for fast, reproducible software builds and enable short feedback cycles when they capture dependencies precisely and selectively execute build tasks efficiently. A much-overlooked feature of build systems is the expressiveness of the scripting language, which directly influences the maintainability of build scripts. We develop new incremental build algorithms that allow build engineers to use a full-fledged programming language and where task dependencies can be discovered during building.

In the **CoCo project** [1, 10], we explore novel ways for co-contextual reasoning about code and how that can be used to achieve incrementality. Specifically, we are developing co-contextual type checkers for functional and object-oriented programming languages. A co-contextual type checker produces context requirements rather than reading context information as it traverses a syntax tree bottom-up. We have mostly focused on applying this technique to incremental type checking so far, yet applications to parallelization and streaming seem promising.

2 Practical correctness proofs

Correctness proofs ensure algorithmic results are correct. Conversely, incorrect feedback gives developers a false sense of security that is not actually warranted by their code. We develop theory and tools that simplify correctness proofs. Our long-term goal is to enable analysis developers to prove correctness in little time, without requiring extensive training. Provably correct analysis results will boost the confidence of programmers when reacting to analysis feedback.

In the **Sturdy project** [6, 7], we explore techniques for compositional correctness proofs. The key idea of compositional proofs is to decompose complex verification tasks into much simpler ones. Developers then only need to prove the simple tasks, from which overall algorithmic correctness follows by construction. Key to our approach is to capture the similarities between the specification and the implementation in a single shared program, parameterized over an arrow-based interface. We have instantiated our technique for program analyses and proved simple analyses sound with modest effort. To better understand and support practical scalability, we currently apply our framework to analyses of Java and JavaScript, as well as to code generators.

In the **Veritas project** [3–5], we explore techniques for automated verification of complex tasks. Specifically, we explore how existing off-the-shelve SMT solvers and first-order theorem provers can be applied to prove domain-specific verification problems. The key idea is to translate such problems into first-order logic in a way that existing provers can support. Due to the unpredictability of off-the-shelve solvers, this research is largely driven by empirical experiments that indicate how such translation can be successful.

In the **Soundx project** [11, 12], we develop techniques to guarantee the type safety of code generators. Code generators are hard to get right because they operate at the meta-level, where programs are data. This makes it is easy to generate code that does not type check, which in turn is hard to debug for users since the type errors refer to generated code. We develop automated techniques for ensuring that code generators can only produce code that is well-typed.

References

[1] S. Erdweg, O. Bračevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 880–897. ACM, 2015.

[2] S. Erdweg, M. Lichter, and M. Weiel. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 89–106. ACM, 2015.

[3] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward)*, pages 137–150. ACM, 2015.

[4] S. Grewe, S. Erdweg, A. Pacak, and M. Mezini. An infrastructure for combining domain knowledge with automated theorem provers (system description). In *Proceedings of Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2018.

[5] S. Grewe, S. Erdweg, A. Pacak, M. Raulf, and M. Mezini. Exploration of language specifications by compilation to first-order logic (extended version). *Science of Computer Programming*, 155, 2018.

[6] S. Keidel and S. Erdweg. Toward abstract interpretation of program transformations. In *International Workshop on Meta-Programming Techniques and Reflection*, pages 1–5. ACM, 2017.

[7] S. Keidel, C. B. Poulsen, and S. Erdweg. Compositional soundness proofs of abstract interpreters. *Proceedings of the ACM on Programming Languages*, 2(ICFP), 2018.

[8] G. Konat, S. Erdweg, and E. Visser. Scalable incremental building with dynamic task dependencies. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. ACM, 2018.

[9] G. Konat, M. J. Steindorfer, S. Erdweg, and E. Visser. PIE: A domain-specific language for interactive software development pipelines. *Art, Science, and Engineering of Programming*, 2(3), 2018.

[10] E. Kuci, S. Erdweg, O. Bračevac, A. Bejleri, and M. Mezini. A co-contextual type checker for Featherweight Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2017.

[11] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 331–342. ACM, 2013.

[12] F. Lorenzen and S. Erdweg. Sound type-dependent syntactic language extension. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 204–216. ACM, 2016.

[13] T. Szabó, S. Erdweg, and M. Völter. IncA: A DSL for the definition of incremental program analyses. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. ACM, 2016.

[14] T. Szabó, M. Völter, and S. Erdweg. IncAL: A DSL for incremental program analysis with lattices. In *International Workshop on Incremental Computing (IC)*, 2017.

[15] T. Szabó, E. Kuci, M. Bijman, M. Mezini, and S. Erdweg. Incremental overload resolution in object-oriented programming languages. In *International Workshop on Formal Techniques for Java-like Programs*. ACM, 2018.

DSLs for Protocols: Open Problems

Sung-Shik Jongmans
Department of Computer Science
Open University of the Netherlands
ssj@ou.nl

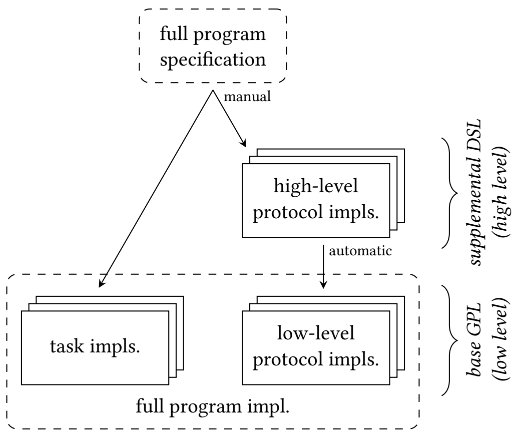

full program specification
manual
high-level protocol impls.
automatic
supplemental DSL (high level)
task impls.
low-level protocol impls.
base GPL (low level)
full program impl.

Figure 1. Approach

1 Introduction

With the advent of multicore processors, concurrent programming has become an important skill for many to acquire. However, concurrent programming remains difficult: despite contemporary general-purpose languages (GPL) offering higher-level abstractions on top of bare threads and locks, developers continue to struggle with classical concurrency errors, such as deadlocks and data races [2].

A major challenge that developers of concurrent programs face, pertains to the implementation of *protocols* (i.e., synchronization/communication patterns) among *tasks* (i.e., sequential computations, run concurrently): although GPLs offer concurrency primitives to implicitly enact global *interactions* of protocols (e.g., a communication of an integer from task T_1 to T_2) indirectly through local *actions* of tasks (e.g., T_1 writes integer `5` to variable `x`, then releases to semaphore `s`; concurrently, T_2 blocks until it acquires from `s`, then reads from `x`), GPLs lack linguistic support to explicitly enact interactions, directly. Aggravated by the exponentially many, seemingly nondeterministic, interleavings in which tasks can be scheduled, purely *action-centric protocol programming* techniques are hard to reason about and error-prone to use.

In recent years, *interaction-centric protocol programming* techniques have been developed to overcome these and other issues. The idea is that developers should continue to use an existing GPL (e.g., Java, C, etc.) to implement tasks. But complementary, developers should use a new domain-specific language (DSL) to implement protocols. A separate code generator can subsequently translate high-level protocol implementations in the *supplemental DSL* to low-level protocol implementations in the *base GPL*; the full program thus emerges in the base GPL and can be compiled/run using its standard tools. Figure 1 illustrates this approach.

A fundamental strength of this approach is that it enforces *modularity* of protocol code, through the separation of computations and synchronizations/communications. This allows developers to program with procotol-tailored abstractions, exposed through the DSL; it enables different developers to implement tasks and protocols separately; it improves ease of reuse and maintenance; and it better supports reasoning independently about tasks and protocols. Premier examples of DSLs for protocols are Reo [1] and Scribble [8].

2 Example

Full program. To set the stage for presenting a number of open problems in Sect. 3, imagine we need to implement a Java program with two tasks, Alice and Bob: Alice repeatedly rolls a die and communicates the outcome to Bob; Bob repeatedly checks if the outcome equals some value `n` (unknown to Alice); once Alice rolls `n`, the program terminates.

Tasks. To implement *purely* Alice and Bob, *without the protocol between them*, concurrency primitives are needed that allow Alice and Bob to indicate *just* that they are ready to engage in *some* interaction; in turn, the protocol implementation will decide which *particular* interaction ensues, beyond Alice's and Bob's control. With Java as our base GPL, we will use this API that offers such concurrency primitives:

```
interface Env {            interface Pr {
  Object in();               Env env(String name); }
  void out(Object o); }
```

- `Env` – Represents the *environment* of a task; tasks can use methods `in`/`out` to receive/send values from/to their environments, without knowing what exactly their environments consist of. Both methods are *blocking*: they complete only once the environment is ready.
  `in`/`out` have no arguments to indicate the intended sender/receiver: only `Env` objects (i.e., protocol implementations) control which values "flow" between which tasks.
- `Pr` – Represents a protocol P; it is a container for environments, one for every tasks participating in P.

We can now implement Alice and Bob as Java methods:

```
void alice(Env e) {
  Random r = new Random();
  boolean b = false;
  while (!b) {
    e.out(r.nextInt(6));
    b = (boolean)e.in(); } }
```

```
void bob(Env e, int n) {
  boolean b = false;
  while (!b) {
    b = (int)e.in() == n;
    out(b); } }
```

The `Env` objects passed to methods `alice` and `bob` are instances of `Env` classes that are generated from a high-level protocol implementation in the supplemental DSL. Essentially, the `Env`s ensure `alice` and `bob` follow the protocol. For instance, the `Env`s ensure that `alice`'s first `out` blocks until `bob`'s first `in` is called (and vice versa); then, they transport the integer; finally, they unblock `alice` and `bob`. None of this logic is in `alice` and `bob`: it is fully encapsulated in the `Env`s.

The actual integration of the implementations of tasks and protocols is done in a separate method `main`:

```
void main() {
  Pr p = new AliceBobPr(); // generated
  new Thread(() -> { alice(p.env("A")); }).start();
  new Thread(() -> { bob(p.env("B")); }).start(); }
```

Protocol. For simplicity, let us use a representative toy supplemental DSL, called Sequential Binary Communications (SBC); its syntax is inspired by Scribble, while its formal semantics is inspired by Reo. Let t range over *value types*, and let n range over *task names*. SBC's grammar looks as follows:

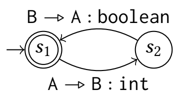
$$P \ ::= \ t \ \texttt{from} \ n_1 \ \texttt{to} \ n_2 \ \mid \ \texttt{repeat} \ \{ \ P \ \} \ \mid \ P_1 \, ; \ P_2$$

- t `from` n_1 `to` n_2 – synchronous communication of a value typed t from a task named n_1 to a task named n_2.
- `repeat` P – finite number ($\geq$0) of iterations of P.
- P_1; P_2 – sequential composition of P_1 and P_2.

This code implements the protocol between Alice and Bob:

```
repeat { int from A to B; boolean from B to A }
```

In words, repeatedly, first Alice communicates an integer to Bob, and then Bob communicates a boolean to Alice.



The semantics of SBC can be formalized using *automata* over alphabets of synchronous communications [5]. For instance, the automaton for the code above is shown here on the right. This automaton-based formal semantics is instrumental in three activities:

- *Generation of low-level code* – `Env` classes generated for an SBC term P essentially simulate P's automaton in an event-driven fashion: whenever `in`/`out` is called on an `Env` object (an event), it checks if this new `in`/`out` *enables* a transition out of the current state. If so, the transition is made, and a communication ensues; if not, the new `in`/`out` remains pending until it can be completed as part of the handling of a next event (cf. Reo [4]).
- *Unit testing/verification of safety properties* – A program is *(protocol-)safe* iff "wrong" `in`/`out` calls never complete. To establish safety, one needs to show that `Env` objects for a protocol never enact interactions that violate some specification. Assuming the `Env` objects faithfully simulate the protocol's automaton, it suffices to show that this automaton meets the protocol's specification. This can be done using model-checking (cf. Reo [7]).
- *Integration testing/verification of liveness properties* – A program is *(protocol-)live* iff "right" `in`/`out` calls always eventually complete. To establish liveness, one needs to show every task always eventually calls `in`/`out` in conformance with the protocols it participates in (e.g., Alice should first call `out` and then `in`; because, if she first calls `out` and then `in`, the program deadlocks). This can be done by extracting behavioral types from an automaton and type-checking tasks against those types (cf. Scribble [3]).

3 Open Problems

1. How to formally model protocols in a more scalable way? (New automaton models? Event structures? Step traces?) *This is a pivotal open problem: currently, the practical applicability of DSLs for protocols is limited by the fact the automata of many realistic protocols grow exponentially in the number of tasks. While techniques exist to mitigate state explosion, transition explosion is still problematic.*
2. How to optimize the performance of generated code with provably correct model transformations, beyond [6]?
3. How to prove, instead of assume, that `Env` objects faithfully simulate the automaton for a protocol?
4. How to efficiently *test* safety when protocol models are too large to exhaustively *verify*? (Model-based testing?)
5. How to reduce the invasiveness of establishing liveness? (Static code analysis without type annotations?)

References

[1] Farhad Arbab. 2004. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14, 3 (2004), 329–366.

[2] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. 2017. Concurrency bugs in open source software: a case study. *J. Internet Services and Applications* 8, 1 (2017), 4:1–4:15.

[3] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.

[4] Sung-Shik Jongmans and Farhad Arbab. 2016. PrDK: Protocol Programming with Automata. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9636. Springer, 547–552.

[5] Sung-Shik Jongmans, Tobias Kappé, and Farhad Arbab. 2017. Constraint automata with memory cells and their composition. *Sci. Comput. Program.* 146 (2017), 50–86.

[6] Sung-Shik Jongmans. 2016. *Autamata-Theoretic Protocol Programming*. Ph.D. Dissertation. Leiden University.

[7] Natallia Kokash, Christian Krause, and Erik de Vink. 2012. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *Formal Asp. Comput.* 24, 2 (2012), 187–216.

[8] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *TGC (Lecture Notes in Computer Science)*, Vol. 8358. Springer, 22–41.

In Search for Scalable Correctness Assurance in Virtual Realities

Wishnu Prasetya
Utrecht University
w.prasetya@uu.nl

One of the main research focuses at Utrecht University is in advanced mediums of human and computer interactions. One of such a medium is Virtual Reality (VR) which has become popular, with various applications: entertainment, education, training, marketing, and even in health care. Compared to traditional User Interface, VR offers realism, smooth interactions, and immersion, of which many applications can benefit. Like any software though, we do want our virtual realities to function correctly. In related short papers, authors have addressed Utrecht University's research in advanced type checking [9], static analysis [5], and program synthesis. These belong to the class of static approaches to correctness assurance, along with e.g. program verification [2]. In this short paper we will focus on a complementary approach. Inevitably, real world software will have plenty of aspects which cannot be verified statically using the afore mentioned approaches. To complement them, we will look into a dynamic way of assuring correctness, in particular *automated testing*. We define 'automated testing' as an approach to verify a given formal specification ψ (capturing some desired property) by generating actual executions and checking that these executions satisfy ψ. Producing just one execution is usually not considered as adequate. Test adequacy is usually expressed in terms of a coverage requirement, which can be thought as a set of target predicates representing different classes of similar executions. While simply producing an execution is trivial, producing an execution that solves a particular coverage target is often hard (in general undecidable).

The challenge

Testing a virtual world poses an interesting challenge. Its interaction space is huge. Furthermore, its virtue of smooth interaction means a single interaction can be very fine grained, e.g. moving just one pixel to the right, while on the other hand there is always a whole spectrum of possible actions to do, resulting in a combinatoric explosion in the interaction space. Obviously, programs with large enough state space will pose a challenge for manual testing, but in virtual worlds this challenge is further aggravated to a new level. Even creating a single test case is laborious. For example, suppose a new entity has been added into the virtual world and we want to verify that it interacts correctly. Very often entities cannot be tested independently because their behavior depend on the virtual world context in which they are placed. It is hard for testers to create this context without the help of the visualization and other senses provided by virtual world itself, which implies that the only practical way to test them is by testing them in the virtual world itself. This would require the tester to first navigate through the virtual world to the place where the said entity resides. Multiplied over multiple test cases, this overhead is really significant, hence scalability is a real threat if we ever want to properly test a virtual world, which matters e.g. if it is to be used for something mission critical.

A virtual world is also inhabited with many dynamic entities. These are entities that can autonomously move around or change their state (e.g. it may become at some point uninteractable). Some of them may even actively try to obstruct the user, e.g. if they simulate enemy units in a combat training program. These not only further complicate the situation for human testers, but they practically make the virtual world non-deterministic. Although theoretically we can control every random generator in a program, and even control its internal concurrency, in practice this is just very hard to engineer. The consequence of this is that the old trick of recording test plays to replay them when we need to re-test the virtual world does not work anymore, which implies that testers will just have to toil manually again every time the virtual world is updated or patched, and hence needs to be re-tested.

So we turn to computers, seeking our salvation. But can computers do all these things that usually require humans to accomplish?

Where should we look?

Let's first briefly consider where we currently stand in the field of automated testing. Bounded model checking (BMC) [1] has been shown to be feasible. A program can be thoroughly verified, up to a certain execution depth. However, this is not likely to work on fine grained interaction space of a VR, where the space immediately blows up after the initial state. Many states that need verification would be just too far way for BMC. Combinatoric testing [7] offers at least an alternative perspective. The idea is to split the interaction space into several dimensions the tester believe to be relevant in influencing the target property ψ to test. Then each dimension is partitioned into equivalence classes. To test ψ we exhaustively generate executions such that every combination of the equivalence classes over all dimensions is covered by at least one execution. Such abstract reduction would allow us to, in principle, and complementary to BMC, reach far away states. Unfortunately, finding even one execution that satisfies a given coverage constraint is, as remarked before, a hard problem. So, although combinatoric testing suggests a promising position, it alone does not enlighten us as to how to get to that position in the first place.

The last decade has also seen the advance of search based testing [6] that treats the problem for solving a given coverage predicate as a search problem, for which there are various algorithms we can try. The most popular one is probably the family of evolutionary algorithms, e.g. as implemented in the tool Evosuite [4] which has been very successful in constantly winning unit testing tool competitions [8]. Despite being evolution-inspired, an evolutionary algorithm still works by, to some degree randomly, trying out different executions, and then applying mutation and cross over to converge them towards solutions. The process is guided by a so-called fitness function which is a metric expressing how far we are from finding a solution. While such a process works well at the unit testing level, it is unlikely to scale up to deal with the huge interaction space of a VR. Fitness functions would become far too abstract/rough to properly capture the dynamic of such a space.

The direction that we have been investigating at Utrecht University is to combine those automated testing approaches with human like cognitive skills. Agents with such skills can be deployed into a VR to simulate human testers to autonomously test the VR. The underlying premise is that with just enough of such skills the agents would be able to get some grasp on the VR's semantical structure, hence becoming more effective in choosing the right interactions rather than just brute-forcely or randomly trying all sorts of interactions. We hypothesize thus that cognitive skills would enable test agents to prune the search space into fragments which are again tractable for traditional approaches.

A simple example is navigation skill, which we can add by simply incorporating a path finding algorithm [3] into the agents. This would enable the agents to go from one place to another in the VR, without having to brute force the path using BMC or an evolutionary algorithm.

The agents will need more than just navigation skill though. A path may be blocked, for which specific interactions are needed to clear it. In turn, this may requires the corresponding entities to be in a certain state, which in turn requires more interactions to trigger. Rather than just brute forcing all sorts of possible interactions, the test agents need to make educated guesses. Entities in a virtual world, and interactions that we can do on them, represent concepts meaningful to humans, e.g. building, door, vehicle. Interactions on a door would be for example 'open' and 'close', and so on. Agents would need to be able to *learn* and *infer* associations between concepts, to at least make a guess which ones would be relevant, and which ones can be ignored, towards solving some goal at hand. We are not there yet in our research, but this is the future direction that we want to go. At least, 'learning' and 'inference' are two themes which have been well researched. The more specific research question for us would be to find out which learning and inference paradigms suit best for VR testing.

Further down the road, more challenges await: what kind of cognitive skill is needed to deal with dynamic entities, including those which are not friendly? And how about collaboration? Many VRs allow multiple users, and hence allowing us to deploy multiple agents. Can we exploit this? For many VRs, user experience is very important. Bad reviews from users can be detrimental for a VR's reputation, potentially wasting the millions USD invested into developing it. Can we extend the agents so that they can appraise which general emotion the VR would invoke? Is it boring? Is it too distressing?

Closing words

Virtual reality poses an interesting scientific challenge for the formal method community. Here we have outlined how it challenges the current state of the art of automated testing, but this actually applies to other areas of formal method. In automated testing we focuses on generating executions, given specifications. But producing a formal specification of a virtual world is not trivial either. What is the right paradigm to formalize it? What can we do (or what should we trade off) to make it scalable?

The tech-world does not remain still either —of course we know that. The next kind of "reality" is already around the corner: augmented reality. So, how do we deal with that...?

References

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[2] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer, 2004.

[3] X. Cui and H. Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011.

[4] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*, pages 416–419, 2011.

[5] J. Hage. The usability of type systems. In *A Research Agenda for Formal Methods in The Netherland*. Utrecht University, 2018.

[6] P. McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[7] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[8] U. Rueda, T. Vos, and I. S. W. B. Prasetya. Unit testing tool competition – round three. In *IEEE/ACM 8th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2015.

[9] W. Swierstra. Programming with dependent types. In *A Research Agenda for Formal Methods in The Netherland*. Utrecht University, 2018.

Programming with dependent types

Wouter Swierstra
Universiteit Utrecht
The Netherlands
w.s.swierstra@uu.nl

Computer programs manipulate data. This data comes in many different shapes and sizes: the telephone numbers stored in our smartphone; the salary calculations done in Excel spreadsheets; or the customer addresses stored in a database. Many modern programming languages use a *type system* to classify data and rule out nonsensical calculations, such as trying to multiply a customer's telephone number and address. A (static) type system rules out such calculations before a program is run. Indeed, Simon Peyton Jones famously described static type systems as "*the world's most successful application of formal methods.*"

Yet not all developers are enamored of statically typed languages. Some of the most common points of critique include:

- Simply typed languages can only prevent simple errors, such as mixing up the order of arguments passed to a method. In reality, programs are full of rich properties that cannot be enforced effectively by a type system.
- Requiring a program to be type correct before it can be executed bogs down the development process. Programmers should not spend their time writing code, rather than fixing type errors. Static types may help ensure safety, but make programs inherently harder to write.
- Not all type information may be known when a program is first written. Code that needs to interface with a database or other external data source may not know the type of all data involved before its execution. Similarly, some methods—such as `printf` or those using C's `varargs`—are notoriously difficult to type statically.

This note aims to illustrate how some of these limitations may be tackled by embracing *programming languages with dependent types*. In particular, it aims to show how *current research* on such languages provides a novel perspective on the benefits of statically typed programming—and formal methods more generally.

Static types cannot...

There is a deep connection between types and mathematical logic known as the *Curry-Howard correspondence*.Simply stated, this correspondence states that every type system may be viewed as a logic. Every type in written in a type system corresponds to a unique logical proposition; every program corresponds to a unique proof. To illustrate this point, consider the rules for *modus ponens* in logic and the typing rule for function application:

$$\frac{p \to q \qquad p}{q} \qquad\qquad \frac{f : a \to b \qquad x : a}{f\ (x) : b}$$

These rules are strikingly similar! The Curry-Howard correspondence shows how this is no coincidence—this similarity extends to richer logics and language constructs.

A common complaint about statically typed languages is *the lack of expressivity.* While static type systems are certainly no silver bullet, the properties that may be enforced by most type systems of mainstream languages is very limited. The 'logic' underlying such type systems is an (embarrassingly unsound) propositional logic in which you cannot express any interesting properties. To say anything meaningful about the behaviour of programs, we need *quantifiers* in our logic.

Viewed through the Curry-Howard lens, adding quantifiers our logic amounts to shifting from a simply typed programming language to one with *dependent types.* Per Martin-Löf was one of the first to propose a single (dependently typed) language to express both proof and computation. To illustrate the importance of quantification, consider the following three type signatures:

$$\begin{aligned} &f_1 : \mathit{List\ Int} \to \mathit{List\ Int}\\ &f_2 : \forall\, a\,.\ \mathit{Order}\ a \to \mathit{List}\ a \to \mathit{List}\ a\\ &f_3 : \forall\, a\,.\ \mathit{Order}\ a \to \forall (xs : \mathit{List}\ a)\,.\ \exists\ ys : \mathit{List}\ a, \mathit{Sorted}\ xs\ ys \end{aligned}$$

Judging from its type, the first function could do anything from reversing the list to incrementing every element. The type associated with f_2 is much more restrictive: the parametric polymorphism—induced by the universal quantifier—ensures that any elements in the output list must also occur in the input.Crucially, the universal quantifier is restricted to abstract over *types* in most languages. In contrast, the last type signature uses *dependent types* to specify that for every input list *xs*, the function must compute a new list, *ys*, that must be the sorted permutation of *xs*.

Not all code has as clear a specification as sorting algorithms. Nonetheless, *all* code has some (partial) specification or invariant that programmers track in their head. Rather than abandoning static typing altogether, wouldn't it be better to explore languages where these properties can be described and enforced? Doing so offers programmers a spectrum of correctness: from basic code hygiene to full-blown mechanised proofs of functional correctness.

"Computer says know"

A common perspective on static types states that the type-correct programs are only a subset of all meaningful programs. A static type system polices the development process, slapping the wrist of any careless developer that dares adventure outside the subset of programs considered valid.

I would like to offer a slightly different perspective. Developers do not produce software by writing random strings and subsequently checking which ones happen to correspond to meaningful programs. Instead, software design is a creative intellectual activity: a program is constructed methodologically,breaking a large problem into smaller pieces. Once the pieces are small enough, we can proceed to figure out the inputs and outputs of each piece. Programmers should write example inputs and outputs—turning these into tests—before implementing the functions that solve the individual problem pieces.

See how types fit naturally in this philosophy? Types form a partial specification; type checking the signatures of the individual pieces show how each individual function—once it has been implemented—may be composed to solve the original problem.

But types have much more to offer than such simple correctness properties. By starting to write down the types, the code often follows naturally. Integrated Development Environments (IDEs) such as Visual Studio use static type information to help a programmer navigate a complex codebase. Programmers who have worked with dependently typed systems such as Agda and Idris, will know how most of the thought goes into the careful design of the *types* of a function; once the types are fixed, the *program* almost writes itself. The programmer only need provide hints about which argument to pattern match on or when to make a recursive call; the IDE is often happy to use the static type information of the values in scope to find any missing values automatically.

Going even further, research on *dataype generic programming* has shown how we can generate new functions from the structure of our data types, or even refactor functions automatically exploiting structured changes to the types involved.

The purpose of a static type system is not *only* to rule out bad programs; a type provides information about the values that inhabit it. *Knowing* the types provide a valuable clue to a program's construction. To cite Conor McBride it: "*is a type a lifebuoy or a lamp*"?

Just-in-time static typing

How can static types help if the *types* of the data my program manipulates are not all known before a program is run? One of the key rules in a dependently typed programming language is the *conversion rule*:

$$\frac{\Gamma \vdash t : \sigma \qquad \sigma \equiv_\beta \tau}{\Gamma \vdash t : \tau}$$

This rule states that types are equal if they *evaluate* to the same value. This simple rule has profound consequences: by mixing evaluation and type checking, it allows us to *compute* new types on the fly. For example, a function like *printf* is hard to assign a *single* static type, but the format string passed as its first argument may be used to *compute* the types of any remaining arguments that it expects.

There are many similar situations that simple statically typed languages cannot handle well. When interfacing with a database in a dependently typed language, for example, we might ask for a description of certain tables, parse the server's response, and compute the corresponding types. Even if these types are not know statically—we still have the guarantee that the pieces of our program will behave well when composed. *Type providers*, such as those implemented by F#, show how computing new types provide a welcome foothold when interfacing with foreign data. Dependently typed languages, that freely mix evaluation and type checking, take these ideas one step further.

Looking ahead

Despite their promise, there is still much research and development necessary before dependently typed languages can expect more widespread adoption. This research ranges from fundamental questions—such as finding a suitable definition of equality—to more mundane engineering challenges.

Many beginners still experience a high barrier to entry when learning to program with dependent types. Better training material, more informative type error messages, and more robust compiler implementations would all facilitate their more widespread adoption. With the richer design space that dependent types offer, beginners often find themselves making the wrong design choice early on in the development process. Categorising the design principles of experienced users and facilitating the automatic refactoring of typed programs would certainly help explore this richer design space—and prevent beginners from painting themselves into a corner.

Static types cannot possibly solve all the problems in the construction of modern software. But the field of programming with dependent types, while based on fundamental research almost a century old, addresses many of the limitations of traditional static type systems, offering new perspectives on the future of high-assurance program development.