Marina Waldén (Editor)

Proceedings of the 29th Nordic Workshop on Programming Theory

Turku Centre *for* Computer Science

TUCS Lecture Notes
No 27, November 2017

Foreword

This volume contains the extended abstracts of the talks to be presented at the *29th Nordic Workshop on Programming Theory, NWPT'17*, that will take place in Turku, Finland, 1-3 November, 2017.

The objective of Nordic Workshop on Programming Theory is to bring together researchers from the Nordic and Baltic countries interested in programming theory, in order to improve mutual contacts and co-operation. However, the workshop also attracts researchers outside this geographical area. In particular, it is targeted at early-stage researchers as a friendly meeting where one can present work in progress. Typical topics of the workshop include:

- semantics of programming languages,
- programming language design and programming methodology,
- programming logics,
- formal specification of programs,
- program verification,
- program construction,
- tools for program verification and construction,
- program transformation and refinement,
- real-time and hybrid systems,
- models of concurrency and distributed computing,
- language-based security.

This volume contains 21 extended abstracts of the presentations at the workshop including the abstracts of the three distinguished invited speakers:

Prof. Marjan Sirjani	Mälardalen University, Sweden and Reykjavik University, Iceland
Prof. Marieke Huisman	University of Twente, The Netherlands
Prof. John Hughes	Chalmers University of Technology, Sweden

After the workshop selected papers will be invited, based on the quality and topic of their presentation at the workshop, for submission to a special issue of *The Journal of Logic and Algebraic Methods in Programming***.**

Acknowledgements

The 29th Nordic Workshop on Programming theory is supported by *Åbo Akademi University Foundation* and the *City of Turku*. Technical and administrative support is provided by the Department of Information Technologies at *Åbo Akademi University* and by *Turku Centre for Computer Science* (TUCS).

Programme Committee

Lars Birkedal	Aarhus University, Denmark
John Gallagher	Roskilde University, Denmark
Michael R. Hansen	Technical University of Denmark, Denmark
Magne Haveraaen	University of Bergen, Norway
Keijo Heljanko	Aalto University, Finland
Fritz Henglein	University of Copenhagen, Denmark
Thomas T. Hildebrandt	IT University of Copenhagen, Denmark
Anna Ingolfsdottir	Reykjavík University, Iceland
Einar Broch Johnsen	University of Oslo, Norway
Jaakko Järvi	University of Bergen, Norway
Yngve Lamo	Bergen University College, Norway
Kim G. Larsen	Aalborg University, Denmark
Alberto Lluch Lafuente	Technical University of Denmark, Denmark
Fabrizio Montesi	University of Southern Denmark, Denmark
Wojciech Mostowski	Halmstad University, Sweden
Olaf Owe	University of Oslo, Norway
Philipp Rümmer	Uppsala University, Sweden
Gerardo Schneider	University of Gothenburg, Sweden
Cristina Seceleanu	Mälardalen University, Sweden
Jiri Srba	Aalborg University, Denmark
Tarmo Uustalu	Tallinn University of Technology, Estonia
Jüri Vain	Tallinn University of Technology, Estonia
Antti Valmari	Tampere University of Technology, Finland
Marina Waldén	Åbo Akademi University, Finland (chair)

Organizing Committee

Marina Waldén (chair)
Mojgan Kamali
Jonatan Wiik

Åbo Akademi University
Faculty of Sciences and Engineering
Department of Information Technologies
Vattenborgsvägen 3
FIN-20500 Turku, Finland

ISSN 1797-8831
ISBN 978-952-12-3608-2

Table of Contents

Event-based Analysis of Distributed Timed Actors

Marjan Sirjani

Mälardalen University, Sweden and Reykjavik University, Iceland

Abstract

Actor models have been used for modeling and analyzing distributed and asynchronous systems. Moreover, actors are being increasingly used in industry, and new actor-based languages are designed and used by Google and Microsoft, for example Go, P and Orleans. In the new era of cyber-physical systems, we need methods and techniques for safety and performance assurance of timed models. Floating Time Transition System (FTTS) is introduced as an event-based semantics for the actor-based language Timed Rebeca, and it is used for efficient model checking and performance evaluation of timed actors. I will explain FTTS and the action-based weak bisimulation relation between FTTS and the standard semantics in Timed Transition System, and how this relation guarantees preserving of the event-based properties. I will also show how Timed Rebeca is used in safety assurance and performance evaluation of different systems, like Network on Chip architectures, sensor network applications, Traffic Control systems, and quadcopter.

Verification of Concurrent Software with VerCors

Marieke Huisman

University of Twente, The Netherlands

Abstract

Concurrent software is inherently error-prone, due to the possible interactions and subtle interplays between the parallel computations. As a result, error prediction and tracing the sources of errors often is difficult. In particular, rerunning an execution with exactly the same input might not lead to the same error. To improve this situation, we need techniques that can provide static guarantees about the behaviour of a concurrent program. In this presentation, I present an approach based on program annotations, which is supported by the VerCors tool set. I will present the general set up of the approach, and discuss what kind of programs can be verified using this approach. Then I will dive into one concrete example, namely where we use the VerCors verification techniques to prove that compiler directives for program parallellisations (as done in OpenMP, for example) cannot change the behaviour of the program.

Testing the Hard Stuff and Staying Sane

John Hughes

Chalmers University of Technology, Sweden

Abstract

Even the best test suites can't entirely prevent nasty surprises: race conditions, unexpected interactions, faults in distributed protocols and so on, still slip past them into production. Yet writing even more tests of the same kind quickly runs into diminishing returns. I'll talk about new automated techniques that can dramatically improve your testing, letting you focus on what your code should do, rather than which cases should be tested–with plenty of war stories from the likes of Ericsson, Volvo Cars, and Basho Technologies, to show how these new techniques really enable us to nail the hard stuff.

Towards an Autonomous Robot Architecture Combining Complex Event Processing and Multilevel Modelling

Juan Boubeta-Puig[1], Fernando Macías[2], and Adrian Rutle[2]

[1] University of Cádiz, Spain, juan.boubeta@uca.es
[2] Western Norway University of Applied Sciences, Norway, {fmac,aru}@hvl.no

Complex Event Processing (CEP) [5] is a cutting-edge technology that allows the real-time analysis and correlation of large volumes of data, with the aim of detecting complex and meaningful events and of inferring valuable knowledge for end users and systems. In order to do this, so-called event patterns are used. These patterns specify which conditions must be met in order to detect such situations of interest.

Multilevel Modelling (MLM) enables the definition of Domain-Specific Modelling Languages (DSML) in a hierarchical manner [4]. In such a way a modelling language can be refined an arbitrary number of times while maintaining its typing relations with the more abstract languages, allowing for reusability and flexibility [7].

In this approach, we define an architecture for the control of autonomous systems using CEP, in which the behaviour of such systems is specified using MLM techniques. The configuration of the CEP engine is then generated through automatic code generation, removing the necessity of the developer having previous knowledge of CEP technology, or even of the whole paradigm.

The situations that the autonomous system must detect and react to are event occurrences or event sequences. A simple event is indivisible and happens at a point in time; a complex event contains more semantic meaning which summarises a set of other events. Events can be derived from other events by applying or matching event patterns; these are defined by using specific languages developed for this purpose, known as Event Processing Languages (EPLs) [1]. A CEP engine is the software used to match these patterns over continuous and event streams, and to raise alerts about complex events created when detecting such event patterns.

The main advantage of CEP is that complex events can be identified and reported in real time, thus reducing latency in decision making, unlike other traditional methods. Other relevant advantages are [3]: information overload prevention, human workload reduction, faster and automatic reply and decision quality improvement.

In order to describe the correct behaviour of a system, such as a robot, we make use of Model-Driven Software Engineering (MDSE), a software paradigm that uses abstractions for modelling different aspects – behaviour and structure – of software systems, considering models as first-class entities in all phases of software development [2]. Macías et al. [7] have proposed an approach for the definition of behaviour models focusing on multilevel modelling hierarchies.

Taking the advantages of both paradigms, in this paper we present an autonomous robot architecture that combines CEP and multilevel modelling to detect relevant situations in autonomous robots, as well as to automatically execute the appropriate actions. Our autonomous robot architecture is composed of three tiers: Hardware, Message and Logic. Figure 1 illustrates this architecture along with all its components.

The **Hardware tier** includes the hardware components (sensors and actuators) together with their controller modules. The *sensor module* receives readings of different sensors, such as infrarred and colour, and sends this sensor data to a *sensor message queue*. At the same time, the *robot state module* sends the state data to a *state message queue*. This module is also in charge of receiving complex events and transforming them into data executable by actuators.

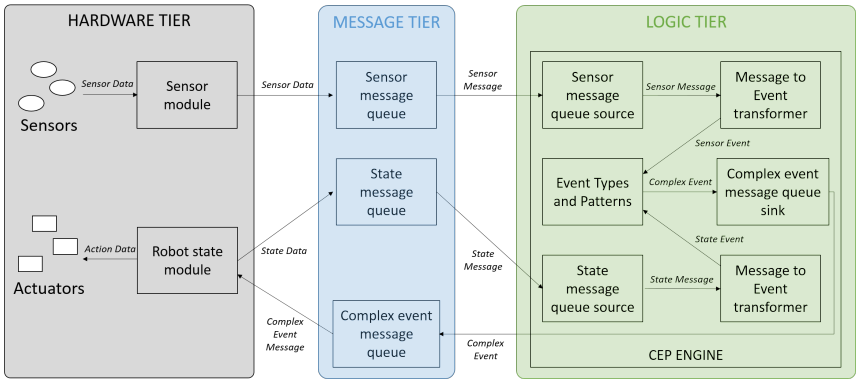


Figure 1: Our autonomous robot architecture combining CEP and multilevel modelling.

The **Message tier** provides the message queues which make possible the integration between the Hardware and Logic tiers. The *sensor message queue* receives the sensor data coming from the *sensor module*, while the *state message queue* receives the state data from the *robot state module*. Both queues forward this information into the CEP engine to be processed and analysed. This tier has also the *complex event message queue*, which receives the complex events generated by the CEP engine and sends them to the *robot state module*.

Finally, the **Logic tier** contains the CEP engine that provides the logic of the architecture. CEP is mainly performed in 3 stages: (1) event capture: it receives events to be analysed by the CEP engine; (2) analysis: from the event patterns previously defined, it will process and correlate the information (events) to detect relevant situations in real time; and (3) response: after detecting a particular situation, notify the system, software or device in question.

This engine supports the flow-based programming, i.e. a component-oriented programming paradigm that allows us to define programs as networks of "black box" processes. These processes can exchange data using predefined connections through message passing. Then, these processes can be reconnected to create other programs without modifying them internally.

We have created a data flow inside the CEP engine composed of the following components:

- A *sensor message queue source*: this component subscribes to the sensor messages from the *sensor message queue*. When a new message is received, then it is sent to the *sensor message to sensor event transformer*.

- A *sensor message to sensor event transformer*: this is in charge of transforming the sensor message into a specific event format: *SensorEvent(timestamp Long, infrarred Float, colour String)*, in which the *timestamp* event property indicates when (in epochs) the event has been created, while the *infrarred* and *colour* properties specify the values coming from the corresponding sensors. An example of a simple event of this type can be: *SensorEvent(1505401100, 5000, "F3421A")*.

- *event types and patterns*: this component represents all the simple event types and event patterns which have been previously registered in the CEP engine in order to detect situations of interest. When the conditions of a pattern are satisfied, then a complex event is created alerting which pattern has been detected. Afterwards, this complex event is sent to the *complex event message queue sink*.

- *complex event message queue sink*: this allows the communication from the CEP engine to the *Message tier* by sending forward every complex event received.

- *state message queue source*: this subscribes to the state messages from the *state message queue*. Forwards every new message to the *state message to state event transformer*.
- A *state message to state event transformer*: this is responsible for transforming the state message into a specific event format: *StateEvent(timestamp Long, currentTask String)*, in which the *timestamp* event property indicates when the event has been created, and the *currentTask* specifies which task is doing the robot at that moment. An example of a simple event of this type can be: *StateEvent(1505401200, "GoFwd2")*.

In Figure 2, we display a simple specification of the behaviour of a robot, extracted from [6]. This example can be used to briefly illustrate the automatic code generation part of our approach. The generated code concerns the definition of event types and patterns, through EPL files, based on the elements displayed in Figure 2 and their types (in blue). This particular robot has two kind of sensors, and hence two simple event types that can be generated from them. That is, one simple event type is generated per input (red boxes), except *Timeout*, which can be supported natively with EPL primitives. To process these simple event types, two event patterns are automatically generated to detect when the property values of such simple events pass a threshold (obstacle too close or border detected). Additionally, another pattern is generated for the initial task, which gets created without preconditions. The remaining code generation consists of encoding each particular transition (arrow) as a new event pattern that gets fired as a response to the state (simple event type) of the actuators and the complex event types related to the sensor values reaching a threshold.

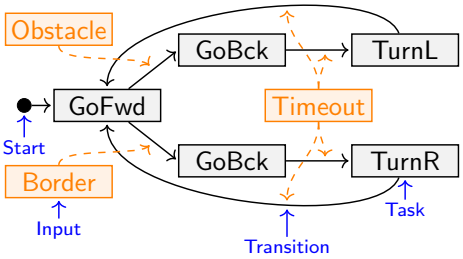


Figure 2: Specification example.

We are working on implementing this process and the presented architecture using the Lego EV3 platform and the Esper CEP engine, and generating Python, Java and Esper EPL code.

References

[1] J. Boubeta-Puig, G. Ortiz, and I. Medina-Bulo. ModeL4CEP: Graphical domain-specific modeling languages for CEP domains and event patterns. *Expert Systems with Applications*, 42(21):8095–8110, Nov. 2015.

[2] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[3] K. M. Chandy and W. R. Schulte. *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill, USA, 2010.

[4] J. de Lara, E. Guerra, and J. Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, 14(1):429–459, 2015.

[5] D. Luckham. *Event Processing for Business: Organizing the Real-Time Enterprise*. Wiley, New Jersey, USA, 2012.

[6] F. Macías, T. Scheffel, M. Schmitz, and R. Wang. Integration of runtime verification into metamodeling for simulation and code generation (position paper). In Y. Falcone and C. Sánchez, editors, *16th Intl. Conf. Runtime Verification, RV 2016*, volume 10012 of *LNCS*, pages 454–461. Springer, 2016.

[7] F. Macías, U. Wolter, A. Rutle, F. Durán, and R. Rodriguez-Echeverria. Multilevel coupled model transformations for precise and reusable definition of model behaviour. *Submitted to Journal of Logic and Algebraic Methods and Programming*, 2017.

Coordination and Amalgamation of Multilevel Coupled Model Transformations

Fernando Macías, Adrian Rutle, and Volker Stolz

Western Norway University of Applied Sciences, {fmac,aru,vsto}@hvl.no

The growing complexity of software systems is forcing industry to implement solutions which enable a more abstract manipulation of software artefacts. Model-Driven Software Engineering (MDSE) is one of the most suitable responses from the scientific communities to this challenge, since it allows for the structural and behavioural specification of software systems in manners that reconcile the mindsets and needs of software architects, developers, domain experts, clients and all stakeholders in general. We believe that Domain-Specific (Meta)Modelling (DSMM) [3, 6] is an approach that could unite software modelling and abstraction, software design and architecture, and organisational studies. This would help in filling the gap between these fields which "could solve all kinds of problems and make modelling even more widely applicable than it currently is" [13].

While structural modelling has advanced both in industry and academia due to mature tools and frameworks, behavioural modelling has still a long way to go especially because of the challenges related to the definition of dynamic semantics. One of the approaches for defining dynamic semantics is based on **model transformations**, as we see examples in [12, 9, 11]. Two characteristics of behavioural modelling are of special importance. First, since most behaviour models have some commonality both in concepts and their semantics, **reusing** these model transformations across behaviour models would be a huge gain. And secondly, behavioural modelling is inherently **multilevel** since we define a metamodel for the modelling language at one particular level while the semantics is described two levels below the metamodel [2, 10]. This is because the behaviour is reflected in the running instances of the models which conform to the metamodel.

To achieve reusable multilevel model transformations which are suitable for definition of behaviour, we have in earlier work [8] proposed the use of Multilevel Coupled Model Transformations (MCMTs): *multilevel* to support the inherent multilevelness of domains and achieve reusability by genericness, and *coupled* to support precision in rule definition and avoid repetition of very similar rules. Hence, by utilising Multilevel Modelling (MLM) techniques for DSMM we could exploit commonalities among Domain-Specific Modelling Languages (DSMLs) through abstraction, genericness and definition of behaviour by reusable model transformations. The reason for our choice is that existing approaches which employ reusable model transformations for the definition of behaviour models focus on traditional two-level modelling hierarchies and their affiliated two-level model transformations (see [7] for a survey). Moreover, multilevel aware model transformations [1] are relatively new and are not yet proven suitable for reuse and definition of behavioural models.

Defining behaviour by rules would require a proper coordination for the application of these rules. Running these MCMTs in different settings and applying them to different models would require the users of the framework to coordinate them properly so that the right rules are applied at the right time. An example is when two or more rules have overlap in their matches in a way that applying one of them would make the others inapplicable. This is a first dimension of conflict which needs to be resolved, where coordination and prioritisation might be helpful [5].

In a multilevel setting, rules defined at a lower abstraction level (more concrete) might overlap in their matches with rules defined at a higher one (more abstract). This second

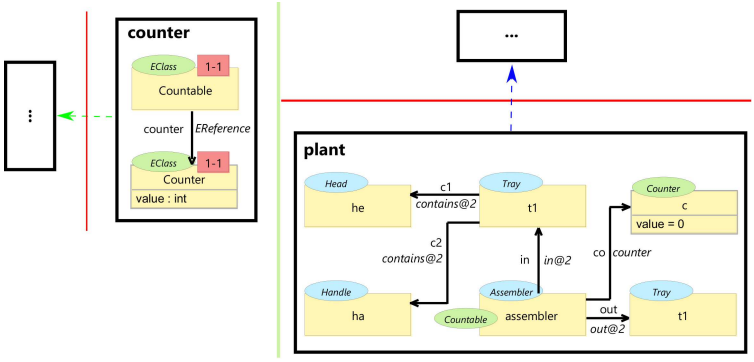


Figure 1: Example of double typing with orthogonal hierarchies.

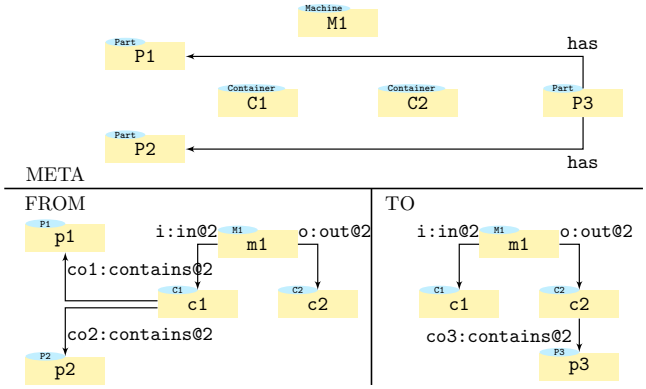


Figure 2: Rule to be applied on instances of *Assembler*.

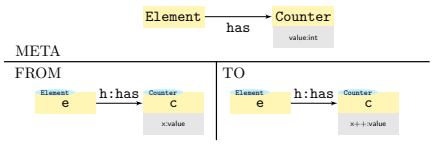


Figure 3: Rule to be applied on instances of *Assembler*.

conflict dimension can also be solved by layering, such that more abstract rules would get a lower priority than the more concrete rules with an isomorphic left-hand side (LHS). However, if the LHSs are not isomorphic, the rules need to be analysed and ultimately amalgamated.

The third dimension of conflict (which is the focus of this paper) arises when the system is specified by combining two or more orthogonal hierarchies, like the ones displayed in Figure 1. In this scenario, an existing hierarchy representing the domain of Product Line Systems (PLS) [9], is augmented with a supplementary hierarchy that allows to include counters, that should be increased under certain circumstances (see also [4]). In our particular example, the desired combined behaviour is that the counter is increased every time a new part is assembled.

The original PLS hierarchy (right of Figure 1) can be manipulated by an MCMT that defines the behaviour of machines of type *Assembler*, which take two separate parts and assemble them together, generating a new part (see Fig. 2). An overview of the semantics of this graphical representation for MCMTs can be found in [8]. For the supplementary hierarchy (left of Fig. 1) that includes the counter, the MCMT displayed in Fig. 3 just states how to increment the value of the counter.

The conflict between both MCMTs arises from the fact that both may be applicable at the same time on the same instance model – actually, the *increase counter* MCMT is always applicable in our example hierarchy. That is, while matches might be overlapping, these rules might not be conflicting in the sense that they won't disable each other. However, we will need

to amalgamate the rules in order to get the sum of the effects of applying both of them, and getting our desired behaviour of *increase the counter on every assembly*.

Manually solving conflicts among rules related to different, orthogonal hierarchies in an ad-hoc manner is not desirable, since it would eliminate both the structural decoupling that those hierarchies had in the first place, and make the resulting set of amalgamated rules unsuitable for further reuse. Moreover, modifying the rules by hand is an error-prone task that might modify their originally intended behaviour. Hence, since using standard techniques like NACs and priorities is not suitable in many scenarios, we are currently working on the possible adaptation of amalgamation techniques already proposed for non-multilevel model transformations (see [4]) into our MLM setting, so that the typing relations among hierarchies can provide all (or at least most of) the information required to automatically generate the amalgamated multilevel rules. This process could then be combined with the proliferation process presented in [8] to automatically generate a big set of two-level model transformation rules from a much smaller initial set of MCMTs defined by the domain experts, hence leveraging the full power of DSML creation through MLM techniques.

References

[1] C. Atkinson, R. Gerbig, and C. V. Tunjic. Enhancing classic transformation languages to support multi-level modeling. *Software & Systems Modeling*, 14(2):645–666, 2015.

[2] J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Model Driven Engineering Languages and Systems: 13th Intl. Conf., MODELS 2010, Proceedings, Part I*, pages 16–30. Springer, 2010.

[3] J. de Lara, E. Guerra, and J. Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, 14(1):429–459, 2015.

[4] F. Durán, A. Moreno-Delgado, F. Orejas, and S. Zschaler. Amalgamation of domain specific languages with behaviour. *J.LAMP*, 86(1):208–235, 2015.

[5] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[6] S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.

[7] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *Software & Systems Modeling*, 14(2):537–572, 2015.

[8] F. Macías, U. Wolter, A. Rutle, F. Durán, and R. Rodriguez-Echeverria. Multilevel coupled model transformations for precise and reusable definition of model behaviour. *Submitted to JLAMP*, 2017.

[9] J. E. Rivera, F. Durán, and A. Vallecillo. A graphical approach for modeling time-dependent behavior of dsls. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009*, pages 51–55. IEEE Computer Society, 2009.

[10] A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *Proceedings of BM-FA '12*, pages 5:1–5:10. ACM, 2012.

[11] A. Schürr and A. Rensink. Software and systems modeling with graph transformations. *Software & Systems Modeling*, 13(1):171–172, 2014.

[12] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, 2nd Intl. Workshop*, volume 3062 of *LNCS*, pages 446–453. Springer, 2003.

[13] J. Whittle, J. E. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

Code Diversification Mechanisms for Securing the Internet of Things *

Shukun Tokas, Olaf Owe, and Christian Johansen

University of Oslo, Norway

Internet of Things (IoT) is the networking of physical objects (or things) having embedded various forms of electronics, software, and sensors, and equipped with connectivity to enable the exchange of information. IoT is gaining popularity due to the great benefits it can offer in domestic and industrial settings as well as public infrastructures. However, securing IoT has proven a complex task, which is largely disregarded by industry for which the business driving force asks for functionality instead of safety or security. Securing IoT is also made difficult by of the resource constraints on the majority of these devices, which also need to be cheap.

IoT devices are often deployed in large numbers. The fact that such a large amount of devices are programmed in the same way allows an attacker to exploit one vulnerability in millions of devices at once, thus with much more gains at the same cost. To address this challenge we propose to consider inclusion of diversification and randomisation mechanisms, at program design, implementation, and execution levels of IoT systems, to diversify observable program behaviour and thus increase resilience. By resilience we mean the ability to resist against attacks and the ability to recover quickly and with limited damages in case of infringements. Although diversity cannot protect against all kinds of attacks, it has proven a strong defence mechanism.

Software diversity is a research topic with several recent comprehensive surveys [1, 2]. Diversity techniques can be simply summarized as introducing uncertainty in the targeted program. Detailed knowledge of the target software (i.e., the exact binary rather than the high level code) is essential for a wide range of attacks, like memory corruption attacks, including control injection [3, 4, 5]. Diversity techniques strive to include in software implementations high entropy so the attacker has a hard time figuring out the exact internal functioning of the system. The range of techniques for diversification through program transformation is large, and include approaches that vary with respect to threat models, security, performance, and practicality [1].

Software diversification has been applied at all levels of software, reaching the microprocessors level, the compiler or the network. Automated techniques from programming languages like information flow static analysis [6] have been extended to the dynamic setting to protect against code injection. Dynamic taint analysis [7] automatically detects injection attacks without need for source code or special compilation for the monitored program, and hence works on commodity software. TaintCheck [7] was an example tool that performs binary rewriting at run time. Such techniques are still very popular and have been e.g., adopted for mobile operating systems [8] to protect the privacy of mobile apps [9]. It is interesting to see how such modern dynamic analysis techniques can be coupled with diversification techniques. Automated software diversification can also be used to counter bugs in software at runtime, thus making the system more robust, and applications to embedded systems have been proposed [10].

However, the diversification techniques are usually developed for standard operating systems or processor architectures running on powerful computing devices like PCs or phones. There is very little research on which diversification mechanisms can be applied to IoT and how. Moreover, we are interested in automated diversification techniques, in particular, techniques that can be employed at design and compile time, because these could be deployed e.g., on

*This work was partially supported by the projects IoTSec and DiversIoT.

version servers that distribute updates or patches to upgrade IoT devices in a seamless manner. When trying to apply a diversification to IoT we are faced with two challenges: (I) IoT devices are resource constrained (with limited computational and memory capabilities), and (II) we need to generate a significant number of software variants (due to large number of IoT devices).

Following are some of the relevant techniques:

N-variant Technique One example of a manual diversification technique that one could think of automating is the software design methodology *N-variant* [11]. The need for N teams of developers developing N variants of the same software independently, from a common specification, should be replaced with automated techniques based on algorithms with mathematical guarantees (e.g., probabilistic or logical guarantees) that would produce the N variants from the same software specification, or implementation given by only one team of developers (e.g., [12]).

Overhead: Does not have any execution overhead, thus being good for the resource constraint nature of IoT devices. However, the overhead is in terms of programming resources (budget, skills, time) required during development of the variants. It moreover incurs an overhead proportional to N for maintenance and updates.

Applicability: This mechanism seems to be useful when it employs automated techniques based on algorithms with mathematical guarantees (e.g., probabilistic or logical guarantees) that would produce the N variants from the same software specification, or implementation given by only one team of developers. This mechanism is useful for developing a fault tolerant system, as diverse sources of faults leads to transient effects.

Program Obfuscation Code transformation techniques change the source program P into a (functionally) equivalent program P' [13]. The objective is to make low-level semantics of programs harder and more complex for attacker to comprehend, without affecting the program's observable behavior. However, to have effective security and diversity the obfuscated code should be difficult enough to reverse engineer. Collberg et al [13], identified four main classes of transformation for code and data obfuscation: lexical transformation, control flow transformation, data flow transformation, and preventive transformation. These may involve renaming variable, altering control flow of program by using opaque predicates or graph flattening, changing the data encoding, etc. After applying a series of transformations, the obfuscated code is distributed to clients. This technique is an effective defence against attacks based on reverse engineering and code tampering.

Overhead: However, it does incur an added cost due to memory usage and execution cycles required to execute obfuscated code.

Applicability: Benefit of this technique is that it can be automated to generate large number of code variants, in a platform independent manner (considering transformation at source code level). It diversifies the code in terms of code space and execution timings, and also it is effective against automated program analysis.

Insertion of Non-Functional Code Non-functional code can be inserted to generate delay in execution or to indicate some space reservation in program memory. Adding any number of NOP instruction does not change program semantics, but it generates diverse binaries and makes the program execution more unpredictable to the attackers as the variants will have different execution statistics. It can also be used to detect control flow change due to instruction misalignment.

Overhead: Consumes only one clock cycle, overhead is proportional to number of NOP instructions included.

Applicabiity: It doesn't degrade systems performance significantly and can be combined with other diversification mechanisms to have an effective diversification strategy.

We plan to adapt, implement, and test the above techniques for IoT systems, and to analyse how they can be combined. At a higher abstraction level, we want to propose and implement a new technique where we want to make use of modern concurrent programming languages like Creol [14] for developing the IoT system. We then take advantage of the inherent *non-determinism* of concurrent programs to produce numerous sequentialized versions based on varied thread scheduling policies (involving randomness). These sequential programs will be deployed on the actual IoT device, preferably also going through more transformations as above. This technique would prevent attacks based on knowledge of the precise timing of events. We plan to develop and demonstrate this idea in detail using a case study.

References

[1] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*, pp. 276–291, IEEE, May 2014.

[2] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.

[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *6th ASIACCS Symposium*, pp. 30–40, ACM, 2011.

[4] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*, pp. 601–615, IEEE, May 2012.

[5] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, pp. 2:1–2:34, Mar. 2012.

[6] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 5–19, Jan 2003.

[7] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, The Internet Society, 2005.

[8] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, pp. 5:1–5:29, June 2014.

[9] M. L. Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *IEEE Communications Surveys Tutorials*, vol. 15, no. 1, pp. 446–471, 2013.

[10] A. Höller, T. Rauter, J. Iber, and C. Kreiner, "Towards dynamic software diversity for resilient redundant embedded systems," in *Software Eng. for Resilient Systems*, pp. 16–30, Springer, 2015.

[11] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.

[12] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity.," in *USENIX Security Symposium*, pp. 105–120, 2006.

[13] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[14] E. B. Johnsen, O. Owe, and I. C. Yu, "Creol: A type-safe object-oriented model for distributed concurrent systems," *Theoretical Computer Science*, vol. 365, no. 1-2, pp. 23–66, 2006.

Time analysis of actor programs

Cosimo Laneve[1], Michael Lienhardt[2], Ka I Pun[3], and Guillermo Román-Díez[4]

[1] University of Bologna/INRIA, Italy
[2] University of Turin, Italy
[3] University of Oslo, Norway
[4] Universidad Politécnica de Madrid, Spain

1 Motivation

Time computation for programs running on multicore or distributed systems is intricate and demanding as the execution of a process may be indirectly delayed by other processes running on different machines due to synchronizations. In this paper, we analyze the time of a basic actor language by defining a compositional translation function that returns *cost equations*, which are fed to an automatic off-the-shelf solver for obtaining the time bounds. Our approach is based on a new notion of *Synchronization sets*, which captures possible difficult synchronization patterns between actors and helps make the analysis efficient and precise. The actor language is intended to be an effective model for staging the time cost of actor-based programming languages by defining ad-hoc compilers.

2 The Time Analysis

In cloud architectures, services are bound by so-called *service-level agreements* (SLAs), which regulate the costs in time and assign penalties for their infringements [3]. In particular, the service providers need guarantees that the services meet the SLA, for example in terms of the end-user response time, by deciding on a resource management policy, and by determining the appropriate number of virtual machine instances (or containers) and their parameter settings (e.g., their CPU speeds). In this contribution, we develop a technique allowing service providers to select resource management policies in *a correct way*, before actually deploying the service.

The technique we follow is similar to the one in [7], where a statically typed intermediate language has been defined in order to verify safety properties and certify code optimisations. However, different from [7], our language, called `alt`, short for *actor language with time*, is concurrent – includes task invocation and synchronization –, features dynamic actor creation, and contains an operation defining the number of processing cycles required to be computed, called $wait(n)$, which is similar to the `sleep`(n) operation in `Java`.

The present work builds upon a previous article by the authors [5], where the computational cost was estimated for functions with a very severe constraint: invocations were admitted only either on the same actor or on newly created ones, i.e., no invocation on parameters. For instance, according to this constraint, an invocation to a function `inner`(y), where the first parameter is the actor executing the function, cannot occur in the body of a function `outer`(x, y). The challenge is that, in this case, computing the cost of `outer`(x, y) requires to know whether there is a synchronization between the actors x and y. In case there is, one has to consider that `inner`(y) might be delayed by other functions running on y (which might be independent from `outer`(x, y)). To overcome this issue, we compute *synchronization sets*, which are the set of actors that potentially might interfere with the executions of each other. Then we compose the cost of an invocation with the cost of the caller in two ways: (1) it is *added*

```
1  main(x, a, b) =
2      νy; νz; νw;
3      wait(1);
4      νh: bar(w, b);
5      νf: foo(y, z);
6      wait(a);
7      νg: gee(z);
8      f✓;
9      wait(k3);
10     h✓; g✓;
11
12 bar(w, b) =
13     wait(b);
14 foo(y, z) =
15     wait(k2);
16     νh: huu(z);
17     h✓;
18     wait(k4);
19
20 gee(z) =
21     wait(k5);
22
23 huu(z) =
24     wait(k6);
```

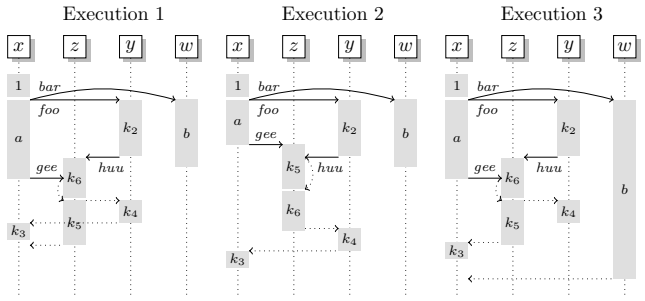


Figure 1: An `alt` program and three possible time computations

– corresponding to *sequential compositions* – if the arguments of the invocation and those of the caller are in the same synchronization set; (2) it is the *maximum value* – corresponding to *parallel composition* – otherwise. The analysis is carried out by a translation function that takes an `alt` program and returns a set of cost equations. In order to compute synchronization sets and to analyze cost compositions, this function has to manage *aliases*, which may be created when an `alt` function is invoked with several copies of the same name. The translation of `alt` programs into the solver input code [1, 4] is currently being prototyped. This tool, together with the compiler we have defined in the authors' earlier work [5], will allow us to automatically compute the cost of programs in `ABS`, a modeling language for programming the cloud [6].

2.1 The Language `alt`

To illustrate the language `alt`, we discuss a simple example. The function *main* in Fig. 1 has three arguments: x is the carrier actor, the other twos – a and b – are integer parameters. *main* creates three new actors, y, z and w at line 2, and spawns several tasks on them at lines 4, 5 and 7. As the tasks are spawned on actors different from x, they will execute in parallel with *main*. Their terminations are synchronized at lines 8 and 10 by means of $h^{\checkmark}$, $f^{\checkmark}$, and $g^{\checkmark}$. Note that *main* takes one of its integer arguments a, which is used in $wait(\cdot)$ operation at line 6. The statement $wait(e)$, where e is an integer expression, represents the advance of e time units. This is the only term in our model that consumes time (a.k.a. that has a cost). The expression e is a cost annotation specifying how many processing cycles are needed by the subsequent statement in the code. Thus, the computation time of *main* depends on a's concrete value. Function *foo* invokes function *huu* on actor z. The other $wait(\cdot)$-operations are executed with some constants.

Fig. 1 also highlights the graphical representation of three possible executions of the code therein. These three executions are obtained by choosing different values of a and b. Execution 1 describes the execution where $a > k_2$, leading to the execution of *huu* on z *begins before gee*. This case highlights how $wait(k_2)$, which is not executed on z, affects the subsequent execution orders, and therefore must be included in the cost of invocations on z. Execution 2 describes the execution where $a < k_2$. The execution of *huu* is postponed until *gee* is finished on z, which ultimately delays the execution of *foo* and its synchronization ($h^{\checkmark}$ at line 17) accordingly. Finally, Execution 3 describes an execution where the execution of *bar* takes longer than all the other methods due to value of b.

2.2 Translation

The translation of a `alt` program associates to each of its methods $\mathtt{m}(x, \overline{y}, \overline{n}) = s$ a cost equation of the form $\mathtt{m}(x, \overline{y}, \overline{n}) = e$ where e gives the cost of executing this method. Note that if s calls other methods, e may depend of the cost of these other methods. As previously discussed, the translation is based on the notion of *Synchronization Sets* which is an equivalence relation between actors that may have unknown and possibly complex synchronization patterns. Our analysis uses this relation to abstract every actors in the same synchronization set into one single actor: as the synchronization pattern between these actors is unknown, the only sound over-approximation is to consider that all of their tasks are synchronized, i.e., are executed in sequence in one single actor. Using this abstraction, our analysis traverses the code of each method, computing an abstract task queue for every synchronization set and accumulating costs for every wait instructions executed in the method and the different awaited calls.

Consider for instance the method *main* in Fig. 1. This method contains four actors x, y, z and w that result in three synchronization sets: $\{x\}$, $\{y, z\}$ and $\{w\}$. The actors y and z are in the same set because of the call $foo(y, z)$ at line 5, which makes the synchronization pattern between these two actors unknown from the *main* method. The translation of the *main* method starts considering the abstract tasks queue of $\{x\}$, $\{y, z\}$ and $\{w\}$ to be empty and starting at time 0 and accumulates the costs for all the queue in correspondances to the different method calls. Synchronizations like $h^{\checkmark}$ at line 10 empties the distant queue of $\{w\}$ and counts the possible waiting time of the synchronization with a cost equal to the max of the distant queue and the local one. For instance, the cost of $h^{\checkmark}$ is $\max(e, b)$ where e is the cost of the lines 5–9.

3 Summary

We have defined a low-level actor language and we study a technique for over-approximating the computational time of the corresponding programs when they run on multicore or distributed systems. Our results may be relevant in cloud computing because `alt` terms might be considered as abstract descriptions of methods suited for SLA compliance. In that context, our analysis could be used in combination with worst-case execution time (WCET) analysis [2] to display correct upper-bounds of the values of cost-expressions written in *wait*() terms.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[2] S. Blazy, A. Maroneze, and D. Pichardie. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Procs. of VSTTE'13*, volume 8164 of *LNCS*, pages 281–303. Springer, 2013.

[3] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Sys.*, 25(6):599–616, 2009.

[4] A. Flores Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of APLAS 2014*, volume 8858 of *LNCS*, pages 275–295. Springer, 2014.

[5] E. Giachino, E. B. Johnsen, C. Laneve, and K. I Pun. Time complexity of concurrent programs. In *Proceesdings of FACS 2015*, pages 199–216. Springer, 2016.

[6] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of FMCO 2010*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

Effects in deterministic parallel programs

Junia Gonçalves

Roskilde University, Roskilde, Denmark
ju@cyberglot.me

Abstract

This work introduces an effect discipline to the λ LVar calculus for deterministic parallel programs. We propose a Haskell library, `lfx`, that combines effect handlers and parallel programming with LVars.

1 Introduction

Parallel programming can be a challenge considering it requires a deterministic execution, i.e. it must provide an outcome observably equivalent to its sequential counterpart. LVars are shared monotonic data structures for guaranteed-deterministic parallel programming [Kup15]. The underlying idea of LVars is that order constraints in a shared data structure assist in the task of assuring determinism: information is partially ordered and can only grow within the structure, but never shrink. LVars are also a generalization of I-Structures for parallel computations – also known as IVars within the Haskell community [MNJ11]. Original works on LVars also introduce different calculi, targeting a broader algorithmic expressiveness. In this work, we only consider the most basic calculus named λ LVar that defines `put` and `get` operations to an LVar.

The LVish library introduced by Kuper [Kup15] was the first step towards a type discipline for the λ LVish calculus[1], which is a Haskell library leveraging parallel programming in the same spirit of the `monad-par` library [MNJ11]. The LVish library provides effect tracking of LVar operations, as well as enforcing the inviability of effectful operations under certain circumstances – e.g., trying to increment an LVar (defined as `bump`) that can only be `put` to (`bump` and `put` do not commute, incurring in non-determinism). However, effect composition is done via monad transformers and the mix does not create a cohesive effect system for the library due to effect tracking and effect composition existing in different domains. To illustrate the issue, we provide a code snippet (Listing 1) using the LVish library (available in [Kup15], page 108):

```
p :: (HasGet e, HasPut e) => ParVecT s1 String Par e s [String]
p = do { set `foo`; ptr <- reify;
         forkSTSplit 3 (write 0 `bar`) (write 0 `baz`);
         frozen <- liftST (freeze ptr); toList frozen   }
```

Listing 1: Disjoint parallel mutation in LVish

In the computation `p` above, the `Par` monad is being stacked with the `ST` monad[2] to enable disjoint parallel updates, where disjoint chunks of a vector can be mutated in parallel without loss of determinism. At type-level, no information suggests that disjoint mutation can happen while `put` and `get` effects are encoded.

In this work, our key contribution is a Haskell library for deterministic parallel programming with LVars using algebraic effects and handlers [Pre15]. `lfx` is briefly described here and available at https://github.com/cyberglot/lfx. The major improvement over the LVish library

[1] A more expressive version of the λ LVar calculus featuring data structure freezing and event handlers.

[2] The `ST` monad is Haskell's standard monad for performing mutations on state. Its usage is witnessed by `forkSTSplit` and `liftST` functions.

is an effect system that can track effects of operations defined in terms of `lfx`, explicitly encoding effects at type-level. On the other hand, `lfx` is less expressive than the LVish library (more on it in §4) and does not interface well with existing code written with monad transformers.

2 A tour of the `lfx` library in Haskell

As a trivial example (Listing 2), we declare a `Nat` data type and provide an implementation for the `BoundedJoinSemiLattice` typeclass[3], and we also define `four` as a computation with parallel effects: we `put` 3 to the LVar, and `get` a value from it that should match the threshold set of $\{1\}$[4] and the computation deterministically returns 4. `runPar` is the canonical handler of parallel computations over an LVar structure. The `new` function returns an LVar initialised with a `bottom` value. The type of `four` is a computation (`Comp`) containing the `Par` effect, which manipulates values of type `Nat`.

```
data Nat = Zero | Succ Nat
instance BoundedJoinSemiLattice Nat where
 a \/ b = if a <= b then a else b
 bottom = Zero

four :: Comp '[Par] Nat
four = do
  l <- new Zero
  put (Succ (Succ (Succ Zero))) l
  x <- get (>=(Succ Zero)) l
  return (Succ x)

*Main> runPar four
Succ (Succ (Succ (Succ Zero)))
```

Listing 2: Handling a parallel computation

```
new :: BoundedJoinSemiLattice a
    => a -> Comp (Par ': r) (LVar a)

get :: BoundedJoinSemiLattice a
    => (a -> Bool)
    -> LVar a
    -> Comp (Par ': r) a

-- NFData constraint required to
-- force the evaluation of lazy thunks
put :: (NFData a, BoundedJoinSemiLattice a)
    => a
    -> LVar a
    -> Comp (Par ': r) ()
```

Listing 3: Type signatures of LVar operations

In a second example (Listing 4), we have a new effect `Logger` and a handler `runLogger` that work similarly to the `Writer` monad. By handling the computation `four'` with `run . runPar' . runLogger`, we get a pair containing the result of the computation and the log of an LVar `get` operation. `runPar'` is a new handler that only performs LVar operations and returns another computation, not a value. As a result, we combined the effects of parallel execution with LVars and logging. Notice that `Par` and `Logger String` are explicitly stated in the type of `four'`.

```
four' :: Comp '[Par, Logger String] Nat
four' = do
  l <- new Zero
  put l (Succ (Succ (Succ Zero)))
  x <- get (>=(Succ Zero)) l
  log ("Get: " ++ show x)
  return (Succ x)



*Main> (run . runPar' . runLogger) four'
( Succ (Succ (Succ (Succ Zero)))
, "Get: Succ (Succ (Succ Zero))")
```

Listing 4: Handling a parallel computation with logging

```
-- runPar handlers for parallel computations
runPar  :: Comp '[Par]      a -> a
runPar' :: Comp (Par ': es) a -> Comp es a

-- canonical top-level handler for
-- computations without effects
run :: Comp '[] a -> a

-- runLogger handler logs to an
-- internal state
runLogger :: Monoid b
          => Comp (Logger b ': es) a
          -> Comp es (a, b)
```

Listing 5: Canonical handlers and `runLogger`

[3] We also suppose an `Ord Nat` instance that defines an implementation of `<=` for the `Nat` data type.

[4] Threshold sets have been oversimplified to be predicates and are required in order to keep the `get` operation deterministic. More on threshold sets in [Kup15].

3 The implementation of `lfx`

The `lfx` library implements the λ LVar calculus and features a work-stealing scheduler that exploits GHC's concurrent runtime in a similar fashion as the `monad-par` library. To inhabit an LVar as showed in §2, a data type must be an instance of the `BoundedJoinSemiLattice` typeclass and provide an implementation to `bottom` and `\/` (*least upper bound*) operations – correctness of such implementations must be verified by the programmer. The `lfx`'s effect system and handlers are based on Kiselyov et al.'s extensible effects [KSS13] and the `freer` package[5]. Effects are explicitly stated in a type-level list (formally, an open union) where they can be extracted from and performed by a handler. The effectful functions simply signal operations to be performed while the handler is responsible for the actual execution. A computation `Comp` is a tree of pure values (`Val y`) or effectul computations (`Eff e k` of effects and continuations). The `runPar` handler traverses the computation tree while it calls the scheduler's functions.

```
data Par a where -- algebraic data type defining LVar operations as the Par effect
  New :: BoundedJoinSemiLattice a => a -> Par (LVar a)
  Put :: BoundedJoinSemiLattice a => a -> LVar a -> Par ()
  Get :: BoundedJoinSemiLattice a => (a -> Bool) -> LVar a -> Par a

new :: BoundedJoinSemiLattice a => a -> Comp (Par ': r) (LVar a)
new a = send (New a) -- it signals a New operation to be performed

runPar x = Scheduler.runPar (go x) where
  go (Val y)   = return y
  go (Eff e k) = case extract e of -- operations being extracted from the list of effects
                   New a   -> do { l <- Scheduler.new a  ; go (kApp k l)  }
                   Put a v -> do {      Scheduler.put v a; go (kApp k ()) }
                   Get p v -> do { s <- Scheduler.get p v; go (kApp k s)  }
```

Listing 6: Fragments of `lfx`'s implementation

4 Future work

The current implementation of `lfx` library features the λ LVar calculus; however, our work is intended to contemplate the λ LVish calculus in order to be competitive. In many configurations the λ LVish calculus is not optimal: memoisation, task cancellation, disjoint parallel mutation, which have been tackled by the LVish library, and their replication in the context of algebraic effects and handlers is planned. A calculus formalising the semantics and the type system implemented in the `lfx` library will finally be introduced in future works.

References

[KSS13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013.

[Kup15] Lindsey Kuper. *Lattice-based Data Structures for Deterministic Parallel and Distributed Programming*. PhD thesis, Indiana University, 2015.

[MNJ11] Simon Marlow, Ryan Newton, and Simon L. Peyton Jones. A monad for deterministic parallelism. In Koen Claessen, editor, *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, pages 71–82. ACM, 2011.

[Pre15] Matija Pretnar. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electr. Notes Theor. Comput. Sci.*, 319:19–35, 2015.

[5] Available at https://hackage.haskell.org/package/freer.

A Language-Based Approach to Prevent DDoS Attacks in Distributed Object Systems *

Toktam Ramezanifarkhani, Elahe Fazeldehkordi, and Olaf Owe
Department of Informatics, University of Oslo, Norway

Abstract

Denial of Service (DoS) attacks and Distributed DoS (DDoS) attacks with higher severity are historically considered as one of the major security threats and among the hardest security challenges. Although there are lots of defense mechanisms to overcome such attacks, they are making the headlines frequently and have become the hugest cyberattacks, recently in 2016 and 2017. In this paper, our aim is to show how distributed program analysis can help to combat these attacks as an additional layer of defense. We consider a high-level imperative and object-oriented framework based on the actor model with support of asynchronous and synchronous method interaction, and shared futures, which are sophisticated features applied in many systems today. Since the preceding step in these attacks is flooding, we show how such communication can cause flooding and thus DoS or DDoS. Then, we provide a hybrid approach including the static and dynamic phases in distributed systems to prevent these attacks statically and to detect them at runtime based on the inline monitoring.

Introduction

Denial of Service (DoS) attacks are becoming crucial. Moreover, *Distributed DoS* (DDoS) attacks have even higher severity and the worst DDoS attacks happened (multiple times) in 2016 and 2017 [3]. More than 90 reports in the first month of 2017 were about DoS attacks. Recent DDoS attacks have imposed high financial overhead as well. Since 70 percent of the exploited devices are unmanaged and have weaknesses, and since there are tens of millions of such devices out there, we face a huge problem, and thus it is inevitable that applications in such devices can be used as bot-nets again. Although there are lots of proposed defense mechanisms to overcome these attacks [1, 2]such as packet filtering or intrusion detection systems, based on the recent experiences, they are not enough and it is required to strengthen them. Moreover, existing bots are likely to live and they are not going away for a while.

In our setting and underlying language, due to some sophisticated features such as asynchronous and non-blocking method calls, it is even easier for the attacker to launch a DoS, because then undesirable waiting by the attacker is avoided in the distributed setting. Therefore, we adapt a static technique to prevent flooding and thus DOS attacks. Moreover, instrument the code for dynamically checking of probable attacks to prevent them at runtime. By including the static analysis in the compilation phase, one obtains static and automatic built-in DoS prevention, and dynamic DoS detection at runtime. In this paper we consider a high-level imperative and object-oriented language based on the actor model with support of asynchronous and synchronous method interaction. We explain our hybrid approach including the static and dynamic phases in this model of distributed systems, and show some examples.

Static and Dynamic Attack Detection and Prevention: To launch a DoS attack, the attacker tries to submerge the target server under many requests to saturate its computing resources. To do so, flooding attack by method calls are effective especially when the server allocates a lot of resources in response to a single request. So, we detect

- call-flooding: flooding from one object to another, which is similar to GET-based flooding, and

*Work supported by the SCOTT and IoTSec (Norwegian Research Council) projects. SCOTT (www.scott-project.eu) has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, Norway.

- parametric-call-flooding: flooding from one object to another when the target object allocates resources or consume resources for each call.

For any set of methods that call the same target method, a call cycle could be harmful. The methods might belong to the same or different objects with the same or different interface. With the possibility of non-blocking calls, it is even more cost-beneficial for the attacker to launch a DoS, because then undesirable waiting by the attacker is avoided in the distributed setting. By means of futures and asynchronous calls, a caller process can make non-blocking method calls that we have considered in an example. This case can be detected statically, involving several factors:

- There should not be lots of methods that can call the same method, simultaneously. With respect to static detection, it is in general hard to see if the callee is the same for different calls. However, a category of self calls can be detected.
- Although we can not trace calls statically, for each target method we can automatically instrument a security code to check the number of calls it receives in a time frame, and block the callers as an anomaly detection and reaction. Moreover, to minimize the runtime overhead, we statically detect critical methods, such as those that are called as non-blocking or by the suspension method that are beneficial for the attackers and do the instrumentation for runtime detection.
- To prevent and detect parameterized DoS or DDoS attacks, the same static and dynamic approach is used while calls with parameters and resource allocations are considered as more serious situations.
- Since the possibility of infinite object creation as referred to as *instantiation flooding* could cause resource consumption and DoS which could be detected statically, especially if those objects and their communication can cause flooding requests in the bots such as clients in our example. Moreover, it is even worse if there is instantiation flooding at the target side of the distributed code. However, this can be detected by static analysis of the target.

Moreover, our anomaly detection is not based on source machine IP addresses that can be forged through a proxy or IP address spoofing. Therefore, for runtime anomaly detection it is possible to check the situations in which thousands of requests are coming to one object every single second specially when they have the same size or parameter settings which is common in automatic flooding attacks.

An Example of Instantiation Flooding: Fig. 1 (a) exploits unbounded creation of client objects where each client object is unaware of the attack. Interfaces are similar to those above and are not given. Each client is innocent in the sense that it does not cause any attack by itself. However, the attacker object makes an attack by using an unbounded number of clients to flood the same server s. The attacker does not wait for the connect calls to complete, therefore it is able to create more and more work load for s in almost no time. The execution of f=c!connect(s) causes an asynchronous call and assigns a future to the call. Thus no waiting is involved. It is immediately followed by a recursive asynchronous call, causing the current run execution to terminate before a new one is started. The attacker creates flooding by rapidly creating clients that each perform a resource-demanding operation on the same server.

Static Analysis of DoS Attacks: We apply the static analysis of flooding presented in [4] for detection of flooding of requests, formalized for the Creol setting. We adapt this notion of flooding to deal with detection of DDoS attacks, which have a similar nature. The static analysis will look for flooding cycles in the code. According to [4] flooding is defined as follows:

> An execution is *flooding with respect to a method* m if there is an execution cycle, call it C, containing a call statement $o!m(\overline{e})$ at a given program location, such that this statement may produce an unbounded number of uncompleted calls to method m, in which case we say that the call $o!m(\overline{e})$ is *flooding with respect to* C.

Flooding is detected by building the control flow graph of the program and locating control flow cycles as shown in Fig. 1 (b). Then, the sets of weakly reachable calls, denoted *calls*, and the set of strongly reachable call completions, denoted *comps*, in each cycle have to be analyzed. Flooding is reported for each cycle with a nonempty difference between *calls* and *comps*, as explained in Fig. 1 (c). Note that the abbreviated notations for synchronous calls and suspending calls are expanded to the more basic call primitives explained above.

Weakly reachable nodes are those that are reachable from the cycle by following a flow edge or a call edge. A node is strongly reachable if it is on the cycle or is reachable without passing a wait node (outside the cycle) unless the return node of the corresponding call is strongly reachable. Also nodes that lead to a strongly reachable

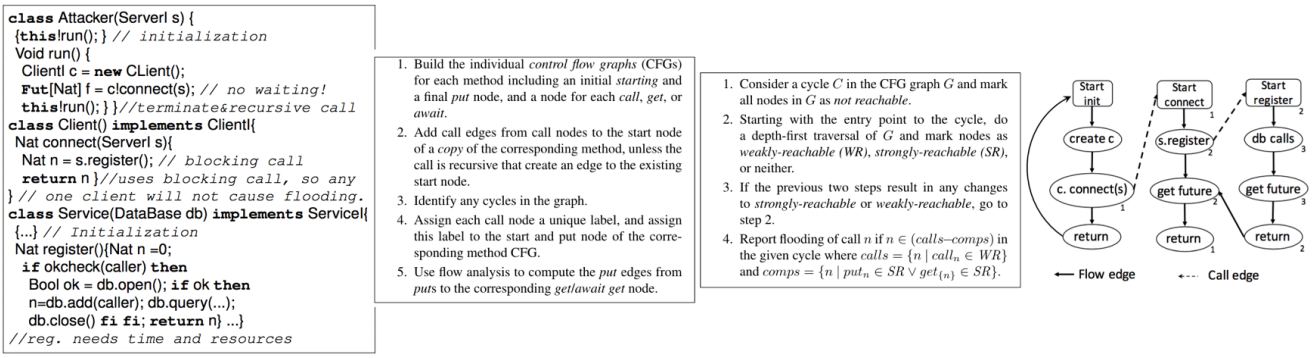


Figure 1. (a), Flooding by unbounded creation of innocent clients targeting the same server, (b) Static detection of flooding, (c) Control Flow Graph, and (d) Top Level algorithm for detecting flooding relative to a given cycle.

node without leaving an if-branch are also strongly reachable. A more precise detection is found in [4], which distinguishes between weak flooding and strong flooding. Strong flooding is flooding under the assumption of so-called *favorable* process scheduling. Strong flooding reflects the more serious flooding situations that persist regardless of the underlying scheduling policy. In the detection of strong flooding, an enabled node is considered strongly reachable if its predecessor flow node(s) are strongly reachable. With respect to DDoS, weak flooding of a server is in general harmless unless the flooding is caused by a large enough number of objects. And strong flooding is dangerous even from one single attacker. A server is often running on powerful hardware, even compared to that of an attacker, in which case it suffices to look for strong flooding. However, weak flooding may be used to discover botnet attacks, since in this case the combined speed of the attacker can be significantly higher than that of the attacked object.

Static Analysis of the Instantiation Example: For the creation of an attacker object, new Attacker(s), the following cycle is detected: - the initialization of the attacker calls *run*, *run* creates a client object *c*, *run* calls c!connect(s), *run* terminates and calls itself recursively. The *run* call has a call edge to the flow graph of *connect*, and *connect* has a call edge to the flow graph of *register*. The call to *register* waits for completion of *register* since it is a blocking call, and the database calls made by *register* wait for the completion of these database calls. The code for the database is not given, and therefore the analysis will be a worst case by considering such calls possibly non-terminating. The control flow graph is given in Fig. 1 (d) The weakly reachable call nodes of the cycle, i.e., *calls*, are $\{1, 2, 3\}$, and the strongly reachable calls, i.e., *comps*, are $\{1\}$. This gives that $calls - comps$ is $\{2, 3\}$. Thus call 2 gives a potential flooding, but in this case it does not reflect a real flooding since each instance of call 2 is on a separate object. This could be understood by (an extension of) the static checking analysis since it is on a new object generated in the same method. Furthermore, call 3 is detected as potentially flooding, and this reflects a real flooding situation. Thus the analysis detects the dangerous flooding of the server s, which is a DDoS attack.

References

[1] C. Douligeris and A. Mitrokotsa. DDoS attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks*, 44(5):643–666, 2004.

[2] N. Gruschka and N. Luttenberger. Protecting web services from dos attacks by soap message validation. In *IFIP International Information Security Conference*, pages 171–182. Springer, 2006.

[3] New Mirai variant hits target with 54-hour DDoS, 2016. *https://www.infosecurity-magazine.com/news/new-mirai-variant-hits-target-with/*.

[4] O. Owe and C. McDowell. On detecting over-eager concurrency in asynchronously communicating concurrent object systems. *Journal of Logical and Algebraic Methods in Programming*, 90:158 – 175, 2017.

Model-based Testing of the Gorums Framework for Fault-tolerant Distributed Systems

Rui Wang[1], Lars Michael Kristensen[1],
Hein Meling[2], Volker Stolz[1]

[1] Department of Computing, Mathematics, and Physics
Western Norway University of Applied Sciences
Email: {rwa@hvl.no,lmkr@hvl.no,vsto@hvl.no}
[2] Department of Electrical Engineering and Computer Science
University of Stavanger, Email: {hein.meling@uis.no}

Abstract

Building cloud computing services that are highly available and resilient to failures involves complex distributed system protocols. Ensuring the correctness of such protocols is a challenging undertaking, particularly when dealing with concurrency and communication [4]. Distributed systems typically leverage a quorum system [8] to achieve fault-tolerance, yet it remains challenging to implement fault-tolerance correctly. *Model-based testing (MBT)* [7] is a promising approach that can help to improve the correctness of such systems. It uses a model of the system under test (SUT) to generate test cases and test oracles, so that a test adapter can use the generated test cases to execute the SUT and compare the test results against the test oracles. This paper explores the use of Coloured Petri Nets (CPNs) [3] for model-based testing applied to quorum-based distributed systems. Recently, the Gorums framework [6] has been developed to ease the implementation of quorum-based distributed systems. We have used MBT to validate a single-writer, multi-reader distributed storage implemented using the Go language and the Gorums framework [9]. In this paper, we extend our existing testing framework to also validate expected behaviour in the case of failures. We report on the improved coverage and the framework's ability to discover defects in the implementation.

Quorum-based Distributed Systems and Gorums

Gorums is a library whose goal is to simplify the development effort for building advanced distributed algorithms for replication, such as Paxos [5] and distributed storage [8]. These algorithms are used to implement replicated services, and they rely on a quorum system [8] to achieve fault tolerance. That is, to access the replicated state, a process only needs to contact a quorum, i.e. a majority of the processes. In this way, a system can provide service despite the failure of some processes. However, communicating with and handling replies from sets of processes often complicates the protocol implementations. To reduce this complexity, Gorums provides two core abstractions (shown in Fig. 1): (a) a quorum call abstraction, used to invoke a set of RPCs (the gRPC [2] remote procedure calls developed from Google)

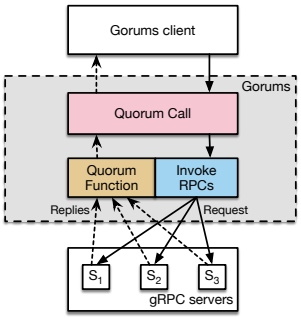


Figure 1: Overview of Gorums abstractions.

on a group of processes and to collect their responses, and (b) a quorum function abstraction which is used to process responses and to determine if a quorum has been obtained. These abstractions can help to simplify the main control flow of protocol implementations.

System Under Test: Gorums and Distributed Storage

Fig. 2 gives an overview of the testing approach. We have implemented a distributed storage system, with a single writer and multiple readers. The storage system has replicated servers for fault-tolerance, and is implemented using the Gorums framework. It also consists of read and write quorum functions and quorum calls. Additionally, we have implemented a client application together with the storage system and Gorums as the SUT. To test our system, we have designed a corresponding CPN testing model that we use to generate test cases and oracles. We also implemented a test adapter used to control the execution of the SUT using the generated test cases and compare the test results against test oracles.

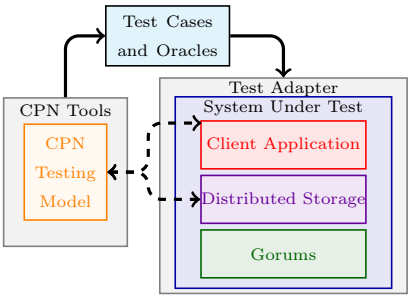


Figure 2: Overview of testing approach.

CPN Testing Model and Test Case Generation

Our constructed CPN testing model can generate both unit tests and system tests. The generation of test cases for the SUT is based on the analysis of executions of the CPN model. The test cases generated for the quorum functions are unit tests, whereas the test cases generated for quorum calls are system tests consisting of a fixed number of concurrent and interleaved invocations of read- and write quorum calls. Fig. 3 shows the Read module of the CPN model. Test cases for the Read quorum function can be obtained by considering the occurrences of the ApplyReadQF transition, and when this transition occurs, the variable readreplies' is bound to the list of all replies that have been received from the servers. The test oracle can be obtained by considering the value of the token on the place WaitingReply. The occurrences of the ApplyReadQF transition can be detected using either state spaces or simulations:

State-space based detection. We explore the full state space of the CPN model. Whenever an occurrence of the ApplyReadQF transition is encountered, we emit a test case based on the current coloured token, together with the test oracle. In this case, we obtain test cases for all the possible ways in which the quorum function can be invoked in the CPN model.

Simulation-based detection. We run a simulation of the CPN model and use the monitoring facilities of the CPN Tools [1] simulator to detect occurrences of the ApplyReadQF transition and emit the corresponding test cases. The advantage of this approach over the state-space based approach is scalability, while the disadvantage is potentially reduced test coverage.

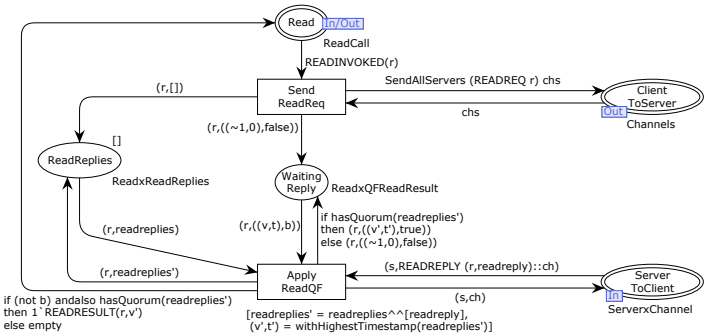


Figure 3: The Read module of CPN model.

Experimental Results

To perform an evaluation of our model-based test case generation, we consider the code coverage obtained using different test drivers for concurrent and sequential execution of generated test cases. In order to improve our previous work [9], we extended our CPN model and the test adapter so that our experiments can also consider scenarios involving server failures of the SUT: We implement a "process killer" in the test adapter that terminates one of the servers when executing such test cases. Due to the fault-tolerance through replication of our distributed storage, the system can still work successfully with one server failure. To evaluate the effectiveness of capturing programming mistakes, we manually injected errors in both the read and write quorum functions. After executing the generated test cases, the tests capture the injected errors for both unit tests and system tests. We obtain statement coverage for read and write quorum functions of 100 % for both system and unit tests, as long as both read and write calls are involved. The statement coverage for read and write quorum calls is 96.7 %. For the Gorums library as a whole, the statement coverage reaches 52.3 %. The reason for the lower coverage of the Gorums library is that it contains code generated by Gorums's code generator, and among them, various auxiliary functions that are never used by our current implementation.

References

[1] CPN Tools. CPN Tools homepage. http://www.cpntools.org.

[2] Google Inc. gRPC Remote Procedure Calls. http://www.grpc.io.

[3] K. Jensen and L. Kristensen. Coloured Petri Nets: A Graphical Language for Modelling and Validation of Concurrent Systems. *Communications of the ACM*, 58(6):61–70, 2015.

[4] Jepsen. Distributed Systems Safety Analysis. http://jepsen.io.

[5] L. Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[6] T. E. Lea, L. Jehl, and H. Meling. Towards New Abstractions for Implementing Quorum-based Systems. In *37th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2017.

[7] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22:297–312, 2012.

[8] M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus.* Morgan and Claypool, 2012.

[9] R. Wang, L. M. Kristensen, H. Meling, and V. Stolz. Application of model-based testing on a quorum-based distributed storage. In *PNSE17*, pages 177–196, 2017.

A High-Level Language for Active Objects with Future-Free Support of Futures *

Toktam Ramezanifarkhani[1], Farzane Karami[1], and Olaf Owe[1]

Department of Informatics, University of Oslo, Norway

Keywords: Active Objects; Asynchronous Methods; Distributed Systems; Futures;

Introduction

The Actor model [1] has been adopted by a number of languages as a natural way of describing distributed systems. The advantages are that it offers high-level system description and that the operational semantics may be defined in a modular manner. The Actor model is based on concurrent units communicating by means of message passing. A criticism of message passing has been that its one-way communication paradigm may lead to complex programming when there are dependencies among the incoming messages.

The Actor-Based Concurrent Language (ABCL) is a family of programming languages based on the Actor model [3]. It makes use of *futures* [2] in order to make the communication more efficient and convenient. A future is a read-only placeholder for a result that is desirable to share by several actors. Future identities can be passed around as first class objects. This model is suitable for modeling of service-oriented systems, and gives rise to efficient interaction, avoiding active waiting and low-level synchronization primitives such as explicit signaling lock operations. The notion of *promises* gives even more flexibility than futures by allowing the programmer to talk about the result of call even before the call has been made.

One may combine the Actor model and object-orientation using the paradigm of concurrent, active objects, and using methods rather than messages as the basic communication mechanism [7]. This opens up for two-way communication. This is for instance done by the Creol language [5] using so-called *call labels* to talk about calls, implementing method calls and replies by asynchronous method passing. Creol introduced *cooperative scheduling*, allowing mechanisms for suspension and process control. A process may suspend while waiting for a condition or a return value. For instance `await f?` makes a process suspend until the reply associated with label `f` appears, resulting in passive waiting. One may also make use of the future mechanism to generalize this setting so that several objects may share the same method result, given as a future. For instance the ABS language [6] is based on the Creol concurrency model, allowing the call labels of Creol to be first class, thereby supporting futures.

In this setting the two-way communication mechanism is replaced by a more complex pattern, namely that a method call generates a future object where the result value can be read by a number of objects, as long as they know the future identifier. Thus for a simple two-way call, the caller will need to ask or wait for the future. This means that each call has a future identity, and that the programmer needs to keep track of which future corresponds to which call. Our experience is that futures are only needed once in a while, and that basic two-way communication suffices in most cases. Thus the flexibility of futures (and promises) comes at a cost. Moreover, implementation-wise, garbage collection of futures is non-trivial, and static analysis of various aspects, such as deadlock, in presence of futures is more difficult. With futures, even normal calls are more complex due to the overhead of the future mechanism.

*Work supported by the *IoTSec* and *DiversIoT* projects (Norw. Research Council) and SCOTT (EU, JU).

In this paper we consider the setting of active objects and compare a future-less programming paradigm to the programming paradigm of future-based interaction. For the future-less programming paradigm we choose a core language derived from Creol, but without call labels nor futures. Comparison of paradigms can be done with respect to several dimensions and criteria. We will use the fairly obvious criteria given by *expressiveness*, *efficiency*, *syntactic complexity*, and *semantic complexity*. Other criteria could also be relevant, such as information security aspects and tool friendliness.

Future mechanisms

Languages may have explicit or implicit support of futures [4, 2]. Implicit futures support the "wait by need" principle. However, when considering cooperative scheduling it is essential that the suspension points are explicit, and we therefore focus on explicit support of futures in the comparison below. Languages based on explicit futures have (a subset of) the following mechanisms (providing ABS style syntax):

- creation of a future (f:=o!m(e))
- first class future operations (assignment, parameter passing)
- polling a future, i.e., using an if-statement to check if a future is resolved (if f? then .. else ..)
- waiting for a future while blocking, i.e., active waiting (x:= get f)
- waiting for a future while suspending, i.e., passive waiting (await f?)

Here f is a future variable, m a method, o an object, e a list of actual parameters, and x a program variable. A non-blocking version of get, can be done by await f?; x:= get f, and is abbreviated await x:= get f. In general, polling may lead to complicated branching structures, and is often avoided in languages with support of explicit futures.

A high-level, future-less language for active objects

We build on the Creol model for active objects, but avoid call labels (and futures). Object interaction is done by so-called asynchronous method calls, implemented by asynchronous message passing. This means that communication is two-way, passing actual parameters from the caller to the callee object when a method is called, and passing method return values from the callee to the caller when the method execution terminates. We include the Creol primitives for process control and conditional suspension, using the syntax await condition, where condition is a Boolean condition. The syntax for method calls is as follows:

- x:=o.m(e)[s] for a blocking call where s is done while waiting for the future to be resolved, and if needed, active waiting happens after s (as in f:=o!m(e); s; x:= get f, using Creol)
- await x:=o.m(e)[s] for a non-blocking call, where the suspension point is after s (as in f:=o!m(e); s; await x:= get f, using Creol/ABS)
- o!m(e), for calls where no return value is needed.

Here [s] may be empty as in x:=o.m(e)/await x:=o.m(e), or may include additional calls as in for instance await x:=o1.m1(e1)[<calculate e2>; await y:=o2.m2(e2)[s]], where the suspension point is after s, passively waiting for *both* calls to complete. In this manner, programs with nested call-get structures can be expressed without futures.

For the comparison we note that the future mechanism involves non-trivial garbage collection. Even if a future is short-lived, it may be complex to detect when it is no longer needed.

Comparison

By defining "future" classes supporting the future primitives above, as illustrated below, we show that our high-level core language is expressive enough to define futures, by means objects of (one of the) future classes. This means that efficient two-way interaction is directly supported, without garbage collection and future objects, while futures can be obtained, when needed, by using future objects. In the former case, efficiency is better than in an implementation using futures, in the second case it is similar (modulo optimizations). For programs with a majority of two-way interaction, efficiency is improved by our paradigm. We also note that programming with two-way interaction is conceptually simpler, since the declaration and usage of future variables are avoided. This is also beneficial for static analysis, since in static analysis of future retrieval (get) one typically needs to associate a call statement with each get statement. This can in general be difficult, and it is less modular when these associations cross class boundaries. Program reasoning is also more complex in the presence of first class futures [8].

Our language is able to encode futures in a straight forward manner. For instance the ABS code f:=o!m(e) is imitated by f:= new Fut_m(o,e) in our language, where class Fut_m is a predefined class, outlined below with initial code, a local method start , and exported methods:

```
class Fut_m(o,par) {
  Bool res:= false; // is the future resolved?
  T value; // the value of the future when resolved
 {start()} // initial code
  Void start(){await value:=o.m(par); res:=true} // see comment below
  Bool resolved(){return res} // polling
  Bool await_resolved(){await res; return true} // waiting until resolved
  T get(){await res; return value} // waiting for the resolved value
}
```

In start we use await when polling is allowed, then the future object will be able to perform incoming call requests, and for instance return the appropriate result of polling requests. (The class parameters should here have the types given by the method m.)

A more detailed comparison will be made in the full paper.

References

[1] C. Hewitt, P. Bishop, R. Steiger: A Universal Modular Actor Formalism for Artificial Intelligence. IJCAI. 1973.

[2] H. Baker, C. Hewitt: The Incremental Garbage Collection of Processes. Proc. Symposium on Artificial Intelligence Programming Languages, ACM Sigplan Notices 12, 8. pp. 55-59. 1977.

[3] ABCL: An Object-Oriented Concurrent System. A. Yonezawa ed, MIT Press 1990.

[4] R. H. Halstead: MultiLisp: A Language for Concurrent Symbolic Computation. TOPLAS, 1985.

[5] E. B. Johnsen, O. Owe: An Asynchronous Communication Model for Distributed Concurrent Objects, Journal of Software and Systems Modeling 6(1): 39-58, Springer 2007.

[6] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen: ABS: A Core Language for Abstract Behavioral Specification. Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, LNCS vol. 6957, pp. 142-164. 2010.

[7] F. de Boer et al: A Survey of Active Object Languages. ACM Computing Surveys 50(5):1-39, 2017.

[8] C. C. Din, O. Owe: Compositional reasoning about active objects with shared futures. Formal Aspects of Computing, vol. 27, Issue 3, pp 551-572. May 2015.

Stubborn Versus Structural Reductions for Petri Nets*

Frederik Bønneland, Jakob Dyhr, Mads Johannsen, and Jiří Srba

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.

State space analysis of distributed systems is often a time and resource consuming task. Explicit analysis methods perform an exhaustive search in the state space, generated by considering every interleaving action in the analysed system, in our case a Petri net [7]. The number of states can be of exponential size and is commonly referred to as the state-space explosion problem. Methods like partial order reductions such as stubborn sets [10,9,8], and structural reductions [3], alleviate the problem by directly or indirectly removing executions that do not need to be searched. Both reductions preserve correctness relative to a given reachability query, meaning that a query is satisfied in the original net iff it is satisfied in the reduced net.

In Petri nets, stubborn reductions remove redundant interleavings of transitions during the state space exploration, reducing on-the-fly the state space that has to be searched through. On the other hand, structural reductions are based on a number of syntax modifying rules that, when applicable, reduce the size (number of places and transitions) of the Petri net itself, hence decreasing the size of the resulting state space [3]. Based on models from the 2015 Model Checking Contest (MCC) [5], structural reduction reduced on average the size of nets by 35%. The winner of the contest was LoLA [11] using primarily stubborn reductions among other reduction techniques. Since structural reductions can be done as a preprocessing while stubborn reductions can be applied during the state space exploration, both techniques can be combined. Up to our knowledge, the effect of combining these two techniques have not yet been investigated in detail. The combination of these two techniques is the main contribution of this abstract, together with a precise definition of stubborn set reduction described in a general framework of labelled transition systems. Furthermore, it is specialized to the Petri net model extended with inhibitor arcs and handling all cardinality and fireability propositions that appear in the model checking contest.

We shall first present our approach to the partial order reduction technique via stubborn sets for Petri nets and prove that the reduction preserves the correctness of reachability queries. For applying this to Petri nets with inhibitor arcs, we define an interesting set of transitions for a given reachability query at a given marking, and a closure algorithm that transforms the interesting set into a stubborn set. Lastly, we implement stubborn reductions in the tool TAPAAL [2]. The tool already supports structural reductions [3], hence we can compare the combination of both stubborn and structural reductions, evaluated on a large benchmark of models from MCC'16 [4].

Reductions on Labelled Transition Systems (LTS). An LTS is a triple $(\mathcal{S}, A, \rightarrow)$ where $\mathcal{S}$ is a set of states, A is a set of actions (or labels), and $\rightarrow \subseteq \mathcal{S} \times A \times \mathcal{S}$ is a transition relation. A reduction identifies the sets of actions (called stubborn actions) in each state that are required to be executed in order to reach a state satisfying some property. For an LTS $T = (\mathcal{S}, A, \rightarrow)$, a reduction of T is a function $St : \mathcal{S} \rightarrow 2^A$. We let $\overline{St(s)} = A \setminus St(s)$. For a given reduction St, we define a reduced transition relation $\underset{St}{\longrightarrow} \subseteq \rightarrow$ such that $s \underset{St}{\xrightarrow{a}} s'$ iff $s \xrightarrow{a} s'$ and $a \in St(s)$.

We require that a reduction St satisfies two axioms. Axiom **W** allows us, in a series of executed actions, to move the stubborn actions to the beginning.

W For all $s \in \mathcal{S}$, all $a \in St(s)$, and all $w \in \overline{St(s)}^*$, if $s \xrightarrow{wa} s'$ then $s \xrightarrow{aw} s'$.

Let $G \subseteq \mathcal{S}$ be a given set of goal states. Axiom **R** states that when starting in a non-goal state, the execution of only non-stubborn actions cannot reach any goal state from G.

R For all $s \in \mathcal{S}$ if $s \notin G$ and $s \xrightarrow{w} s'$ where $w \in \overline{St(s)}^*$ then $s' \notin G$.

Axioms **W** and **R** ensure that for any path from an initial state to a goal state, there exists a path in the reduced transition relation leading also to a goal state, moreover preserving the lengths of minimal paths.

* Based on the master thesis [1].

Theorem 1 (Reachability preservation). *Let $(\mathcal{S}, A, \rightarrow)$ be an LTS, $G \subseteq \mathcal{S}$ a set of goal states, and $s_0 \in \mathcal{S}$. Let St be a reduction satisfying* ***W*** *and* ***R****. If $s_0 \rightarrow^n s$ where $s \in G$ then $s_0 \xrightarrow[St]{}^m s'$ where $s' \in G$ and $m \leq n$. If $s_0 \xrightarrow[St]{}^m s$ where $s \in G$ then $s_0 \rightarrow^m s$.*

Petri Nets. Let $N = (P, T, W, I)$ be a Petri net, where P is a finite set of places, T is a finite set of transitions, $W\colon (P \times T) \cup (T \times P) \rightarrow \mathbb{N}^0$ (where $\mathbb{N}^0 = \mathbb{N} \cup \{0\}$) is a weight function for regular arcs, and $I : (P \times T) \rightarrow \mathbb{N}^\infty$ (where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$) is a weight function for inhibitor arcs. Let $\mathcal{M}(N)$ be the set of all markings for N, and let $T(N) = (\mathcal{M}(N), T, \rightarrow)$ be the LTS for N such that $M \xrightarrow{t} M'$ iff for all $p \in P$ we have $W((p,t)) \leq M(p) < I((p,t))$ and $M'(p) = M(p) - W((p,t)) + W((t,p))$. For a formula φ, from the MCC Property Language [4], and a Petri net N, there is a goal set of markings defined as $G_\varphi = \{M \mid M \in \mathcal{M}(N) \text{ and } M \models \varphi\}$. The interesting set of transitions $A_M(\varphi)$ of a marking M are the transitions that, when fired, can alter the truth value of φ from *false* to *true*. Interesting sets are an extension of attractor sets from [8], and are generated recursively on the syntax of φ. Stubborn sets are generated by applying the closure described in [6] to the interesting set of transitions extended to inhibitor arcs, although it is known to be non-optimal [10].

Experiments. Experiments are executed using the database of models from MCC'16 [4]. For each model there are three categories of queries: reachability cardinality (RC), reachability fireability (RF), and deadlock (DL). For RC and RF there is a total of 16 queries for each category, where we select and perform experiments on the initial 5 queries. We focus on four reachability algorithms: **Base** TAPAAL using exhaustive search, **Stub** TAPAAL adding stubborn reduction, **Struct** TAPAAL adding structural reduction, and **StubStruct** TAPAAL using both stubborn and structural reductions. We do pairwise comparison of the algorithms. For each query, an algorithm gets a point relative to another algorithm, as follows. **Exclusive:** answers the query while the opponent algorithm does not. **Time:** answers the query at least 10% faster, disregarding queries that are solved in less than 10 seconds by both algorithms. **States:** answers the query by exploring fewer states, disregarding queries that are exclusively answered. **Memory:** answers the query by using at least 10% less peak memory, disregarding queries that are exclusively answered. Table 1 shows the number of queries solved by each algorithm (in 15 minute timeout for RC and RF, and 1 hour timeout for DL). Pairwise comparison between the four algorithms is shown in Table 2.

		number of queries solved			
cat.	**queries**	**Base**	**Stub**	**Struct**	**StubStruct**
RC	1565	817	961	879	996
RF	1565	1082	1184	1125	1212
DL	313	211	238	221	239
total	3443	2110	2383	2225	2447

Table 1: Number of queries solved by each algorithm

Conclusion. Adding stubborn and structural reduction improves the verification of base TAPAAL, as seen in Table 2a and Table 2b. In both cases, **Base** TAPAAL still has some exclusive answers, which we expect is due to models not being very reducible, leaving only the overhead of using the reduction techniques. Stubborn reduction performs overall better than structural reduction in Table 2d, however, there is a considerable number of queries where structural reduction answers exclusively or simply more efficient in terms of time, states and memory. When comparing Table 2a, 2b and 2c, we notice the combined efficiency of stubborn and structural reduction. There is a significant increase in the number of exclusive answers when combining the two reductions. We notice that, when adding stubborn reduction to structural reduction, there appear some new exclusive answers to **Base** as well as a considerable number of faster answers. This suggests that the reduction techniques conflict on a number of instances and the stubborn reduction loses efficiency after structural reduction is applied and we are left with the computational overhead. Table 2f displays the contribution of adding stubborn reduction to **Struct** (current TAPAAL implementation). The addition of stubborn reduction increases performance significantly, however, there are some queries that can be solved using only the structural reduction as seen in Table 2f, that the stubborn reduction does not follow to the same degree as seen in Table 2e.

Acknowledgements. We thank Peter Gjøl Jensen for his technical assistance with the implementation.

Base vs Stub								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	13	157	80	208	22	401	9	144
RF	9	111	71	176	22	584	10	184
DL	2	29	13	40	13	174	4	41
total	24	163	173	424	57	1159	23	369

(a)

Base vs Struct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	2	64	31	97	17	257	4	75
RF	2	45	40	89	16	306	4	95
DL	0	10	5	19	17	79	0	23
total	4	119	76	205	50	642	8	193

(b)

Base vs StubStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	11	190	90	245	22	477	9	155
RF	9	139	87	213	21	667	8	196
DL	2	30	10	47	23	154	4	45
total	22	359	187	505	66	1298	21	396

(c)

Stub vs Struct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	115	33	183	104	337	186	124	32
RF	87	28	169	94	512	186	154	37
DL	20	3	37	9	131	47	32	7
total	222	64	389	207	980	419	310	76

(d)

Stub vs StubStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	2	37	37	84	14	320	5	74
RF	6	34	50	69	23	326	6	61
DL	0	1	5	10	15	88	0	12
total	8	72	92	163	52	734	11	147

(e)

Struct vs StubStruct								
cat.	**exclusive**		**time**		**states**		**memory**	
RC	22	139	88	192	20	429	9	141
RF	12	99	73	156	22	589	8	158
DL	2	20	9	35	14	132	4	33
total	36	258	170	383	56	1150	21	332

(f)

Table 2: Algorithm comparisons—in RC and RF there are in total 1565 queries, in DL 313 queries

References

1. Frederik M. Bønneland, Jakob Dyhr, and Mads Johannsen. A Simplified and Stubborn Approach to CTL Model Checking of Petri Nets. Master's thesis, Department of Computer Science, Aalborg University, 2017.
2. Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H Møller, and Jiří Srba. TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In *TACAS'12*, volume 7214 of LNCS, pages 492–497. Springer Berlin Heidelberg, 2012.
3. Jonas F. Jensen, Thomas Nielsen, Lars K. Oestergaard, and Jirı Srba. TAPAAL and Reachability Analysis of P/T Nets. In *Transactions on Petri Nets and Other Models of Concurrency XI*, volume 9930 of LNCS, pages 307–318. Springer Berlin Heidelberg, 2016.
4. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trịnh, and K. Wolf. Complete Results for the 2016 Edition of the Model Checking Contest. http://mcc.lip6.fr/2016/results.php, June 2016.
5. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, A. Linard, M. Beccuti, A. Hamez, E. Lopez-Bobeda, L. Jezequel, J. Meijer, E. Paviot-Adet, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, and K. Wolf. Complete Results for the 2015 Edition of the Model Checking Contest. http://mcc.lip6.fr/2015/results.php, 2015.
6. Lars Michael Kristensen, Karsten Schmidt, and Antti Valmari. Question-guided stubborn set methods for state properties. In *Formal Methods in System Design*, volume 29, pages 215–251. Kluwer Academic Publishers-Plenum Publishers, 2006.
7. Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Mathematical Institute of the University of Bonn, 1962.
8. Karsten Schmidt. Stubborn sets for standard properties. In *International Conference on Application and Theory of Petri nets*, volume 1639 of LNCS, pages 46–65. Springer Berlin Heidelberg, 1999.
9. Antti Valmari. Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, volume 483 of LNCS, pages 491–515. Springer Berlin Heidelberg, 1989.
10. Antti Valmari and Henri Hansen. Stubborn set intuition explained. In *Proceedings of the International Workshop on Petri Nets and Software Engineering, PNSE*, volume 10470 of LNCS, pages 213–232. Springer Berlin Heidelberg, 2016.
11. Karsten Wolf. Running LoLA 2.0 in a Model Checking Competition. In *Transactions on Petri Nets and Other Models of Concurrency XI*, volume 9930 of LNCS, pages 274–285. Springer Berlin Heidelberg, 2016.

Resource Management of Cloud-Aware Programs using Coloured Petri Nets

Anastasia Gkolfi[1], Einar Broch Johnsen[1],
Lars Michael Kristensen[2], and Ingrid Chieh Yu[1]

[1] Department of Informatics, University of Oslo, Norway
{natasa, einarj, ingridcy}@ifi.uio.no
[2] Western Norway University of Applied Sciences, Norway
lmkr@hvl.no

1 Introduction

There is today an increasing interest in using cloud computing infrastructure, which provides on demand and elastic resource provisioning. Applications which are able to autonomously make use of this elasticity to adjust their resource usage to their resource needs, are said to be cloud aware. Cloud-aware applications need to explicitly express resource management policies, hence they should be written in appropriate programming languages. Real-Time ABS [7] (hereafter RT-ABS) proposes a programming model with many features for resource awareness, using a conceptual separation between the cost of a computation and the resource capacity of the infrastructure. These features allow deployment decisions to be made explicitly, which make RT-ABS suitable for modelling cloud applications. RT-ABS has been used to evaluate the performance of different deployment scenarios related to load balancing or different deployment architectures by means of simulation (e.g., [7]).

In this talk, we address similar questions from a model-analysis perspective. We have chosen Petri nets for their suitability to model resources, concurrency and synchronization. Specifically, we construct a hierarchical coloured Petri net [5] (hereafter CPN), whose net captures the semantics of RT-ABS and whose markings are abstractions of RT-ABS program configurations. The advantage is that since resource awareness is represented by the tokens, the model checker [1] of coloured Petri nets (CPN Tools) can provide answers related to resource management by means of state space exploration. Our hierarchical CPN model has been constructed with two levels (called the *imperative layer* and the *deployment layer*, see Fig. 1, each modelling the fragment of the language with the corresponding features). We first explain the kind of analysis that can be achieved by the (self-contained) imperative layer and then how the correlation between the two layers can lead to answers about resource management.

2 The Imperative Layer

The imperative layer of the model is self-contained. It models the communication mechanism of RT-ABS. RT-ABS is an actor-based language, where active objects combine the asynchronous communication of actors with object-oriented programming by means of asynchronous method calls and synchronisation on futures [6]. This can possibly lead to communication deadlocks. The implementation of the communication mechanism of the language as a coloured Petri net can detect static deadlocks (see [4]). In particular, CPN markings are abstractions of the RT-ABS program configurations such that active objects are represented as tokens in the model and deadlock situations can be detected by particular place markings. For example, in Fig. 3 we can

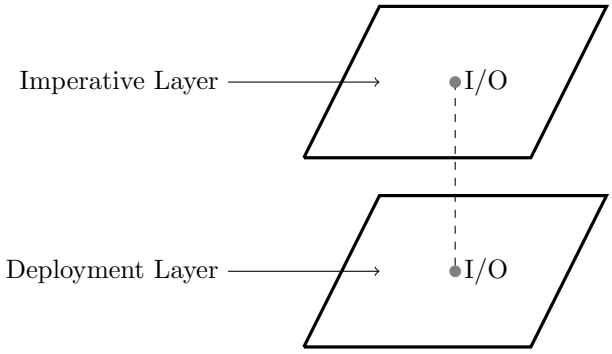


Figure 1: The two layered model

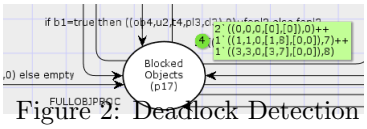


Figure 2: Deadlock Detection

see a module of the implementation related to the communication mechanism. The colourset of the place "Blocked Objects" caries information about the identity of the object which makes a method call (caller), as well as the process that the callee object creates in order to execute this method call. If there is a marking where both the caller object and the callee are blocked in the same place waiting each other to execute, then there is a deadlock (of course this can be generalised for more that two objects, see [4] for details). In Fig. 2 we can see such a deadlock situation detected from the model for a RT-ABS implementation of the publisher-subscriber example [4, 8].

The implementation is supported by a full soundness proof where we prove that program configurations are in abstract simulation relation with CPN markings. An important issue is that the model has been constructed according to the semantics of the language by using the Abstract Interpretation Framework (e.g., [2, 3]) and it supports dynamic creation of abstract objects. This led to the construction of *a single and fixed-size model* for all RT-ABS programs, which was important for gaining scalability in the size of the programs.

3 The Deployment Layer

The deployment layer, as it is shown in Fig. 1, is linked to the imperative layer and, in particular, it adds resource awareness to it. It is designed in a similar way, i.e. it is based on the semantic rules of the deployment fragment of RT-ABS [7] and it supports features like dynamic deployment component creation, resource reallocation, moving of objects to different deployment components, and discrete ABS time. As it was mentioned in the introduction section, the computation cost and the resource capacity are two features that, in RT-ABS, were kept separated for a more precise performance analysis. Since the cost is directly related to the communication between active objects, this layer is complementary to the imperative one. The markings are abstract configurations and the model checker (CPN Tools) can be used to answer questions related to resource analysis. In particular, it can detect whether there is starvation, check if a better resource distribution can avoid starvation and optimise resource usage.

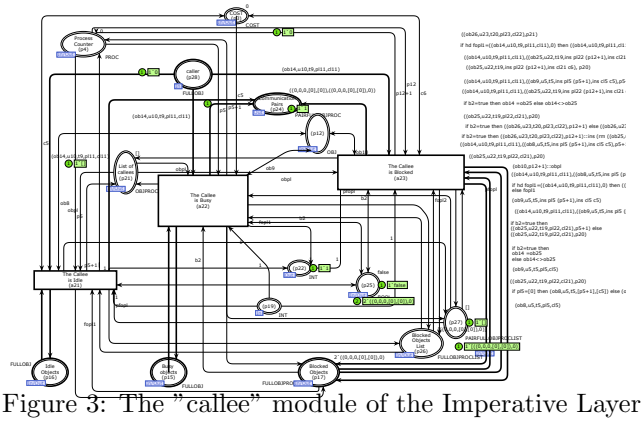


Figure 3: The "callee" module of the Imperative Layer

4 Conclusion and Future Work

In this abstract, we have presented a CPN model of the semantics of a programming language (RT-ABS) that is of fixed size for any program in terms of the net in the model, which thereby supports scalability in the size of programs. It contains two layers, the imperative layer which abstractly simulates the communication mechanism of the language and can be used to detect static deadlocks, and the deployment one, which adds resource awareness to the model. The CPN model can be used to verify starvation freedom, to find better deployment scenarios (if any) for starvation avoidance, and to optimise resource usage (ongoing work). A soundness proof for the deployment layer is also in progress. As a future work, we are planning to go towards the direction of CPN-based test case generation of resource aware programs and apply model based game theory and/or combinatorics to find optimal resource strategies.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[2] P. Cousot. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., 1981.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth Annual Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[4] A. Gkolfi, C. C. Din, E. B. Johnsen, I.C. Yu. Translating Active Objects into Colored Petri Nets for Communication Analysis. In *Proc. FSEN 2016*, LNCS 10522. Springer, 2017. To appear.

[5] K. Jensen and L. M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.

[6] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[7] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67—91, 2015.

[8] O. Owe and I. C. Yu. Deadlock detection of active objects with synchronous and asynchronous method calls. *In Proceedings of NIK 2014*, 2014.

On-the-fly solving of railway games (Work in progress)

Michael R. Hansen, Technical University of Denmark

Abstract

The goal of this work is to synthesize correct-by-construction control programs for railway networks by use of game theory. It builds upon [3], where a model of railway networks is introduced comprising specifications of the following elements:

- A set of named *linear segments*,
- a set of named *point segments*,
- a connector component describing how segments are put together,
- placement of *signals* on linear points,
- the initial positions of n trains, together with a specification of the direction of movement of each train, and
- the final destinations for the n trains.

A concrete example of railway network, from [3] is given in Fig. 1. This figure shows a layout

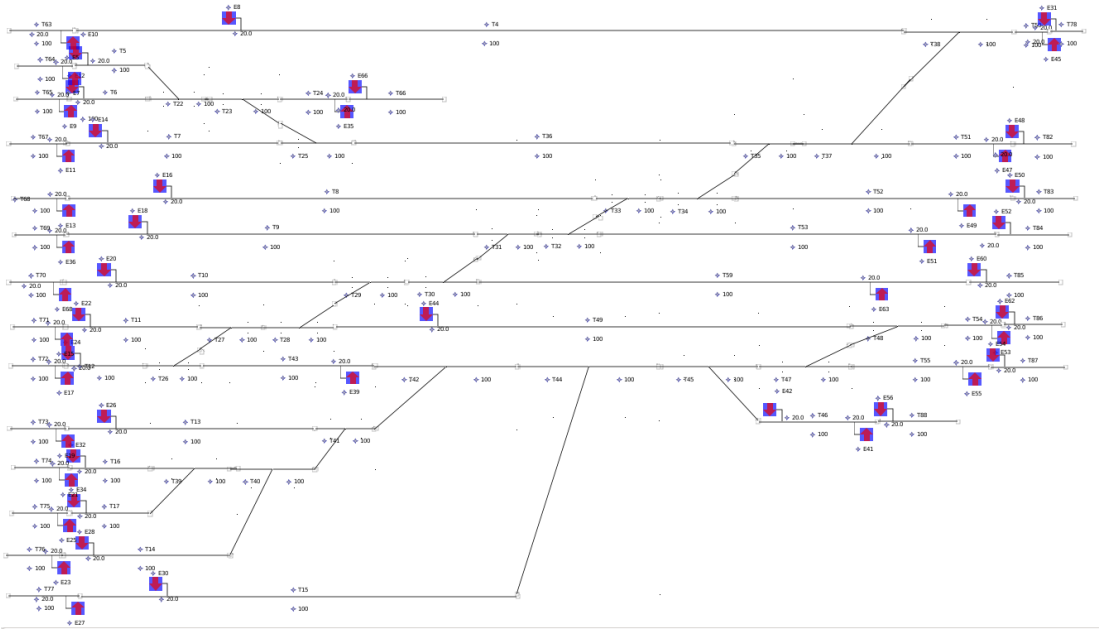


Figure 1: Florence Station, Italy. This drawing was kindly provided by Alessandro Fantechi.

of roughly a quarter of Florence Station, Italy. I consists of 69 linear segments, 23 points and 46 signals. The coloured boxes are signals that are placed on linear sections.

A *signalling plan* is a mapping from train positions to configurations of points of signals. A signalling plan can be considered a control program in the following sense: In a given state it describes how signals and points are configured, so that a train not hindered by a red signal can move to the next linear segment.

When a train moves, the system may enter a *crash state*

- if it bumps into another train, or

- if it drives into a point that does not allow movement in the direction of the train, or
- if there is no next segment in the direction of the trains movement.

A *correct signalling plan* is a plan that brings the trains from their initial positions to their destinations without ever entering a state from which a crash is possible.

The problem generating a correct signalling plan is, in [3], reduced to finding a winning strategy of a two-player turn-based reachability game, where one player is the control system and the other is a "vicious super train driver", where

- the control system in charge of points and signals, that is, when the control system has the turn, it sets points and signals (without moving the trains), and
- the super train driver controls movements of all trains (while obeying the signals), that is, he selects one train not hindered by a red signal to move (if possible).

The super train driver wins a play if the *destinations are not reached*, while the control system wins if all trains safely move from the initial positions to their destinations. A strategy is winning for the control system if it safely brings the trains from their initial positions to their destination, no matter what the opponent does.

The size of the game graph grows exponentially in the number of trains, the number of points and the number of signals, unless it is taken into account that only signals and points placed immediately in from of a train are relevant. In that case the size of the game graph grows exponentially "only" in the number of trains.

When solving the game using a global, backwards reachability algorithm [1], the biggest synthesis problem that was solved in [3] was for Florence Station having 4 trains moving from left to right (see Fig. 1). That took 8.1s on a machine having a 2.7-GHz Intel-Core-i5 processor and 8 GB of memory, and the generated game graph had 220773 vertices. The program ran out of memory in connection with problems for Florence Station having 5 trains.

In terms of the size of the railway networks that could be handled, the results in [3] were encouraging as Florence Station has a size resembling the biggest stations handled by model-checking approaches in the railway domain, e.g. [5, 6]. Note, however, that the models in [5] are more detailed and they focus on interlocking tables (and not on signalling plans).

The backward reachability algorithm used by [3] generates the complete game graphs and finds the complete winning regions for the two players, that is, it performs far too much work as only winning states on a path from initial states, where trains are in their initial positions, to final states, where trains are at their destinations, are relevant.

Therefore, an initial attempt on using local, on-the-fly techniques has been conducted. In particular, a backend solver based on Liu and Smolka's Local2 algorithm [4] for evaluating minimal fixed points of dependency graphs is implemented in F#. The core part of this implementation is shown in Fig 2.

This programs is basically performing a (clever) depth-first search that generates the relevant part of the game graph while solving the game. Initial experiments using this algorithm show significant improvements compared to using global solving techniques. For example,

- Florence Station with 4 trains is solved in 27 ms generating 382 states
- Florence Station with 8 trains (where 4 are moving from left to right and 4 are moving in the opposite direction) is solved in 19s generating 99720 states

These experiments were conducted on a machine having a 2.7-GHz Intel-Core-i7 processor and 16 GB of memory.

These preliminary results are encouraging, and a few examples of next steps could include:

- Improved data structure, including techniques for a more lazy edge generation. Initial experiments (not yet conducted on railway network problems) has shown 10-20 % reductions in the number of generated states

- Investigation of winning strategies. It is not clear what constitutes a good winning strategy.
- Sound transformations of winning strategies in order to avoid stop-and-go movement of the trains.

References

[1] D. Berwanger. Graph games with perfect information. Course notes, Master Parisien de Recherche en Informatique, 2013.

[2] M.R. Hansen and H. Rischel. Functional programming using F#. Cambridge Univ. Press, 2013

[3] P. Kasting, M.R. Hansen, and Steen Vester. Synthesis of Railway-Signaling Plans using Reachability Game. *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages (IFL 2016)*, pages 9:1–9:13, ACM, New York, NY, USA 2017

[4] X. Liu and S.A. Smolka S.A. Simple linear-time algorithms for minimal fixed points. In *Automata, Languages and Programming. ICALP 1998.* LNCS 1443. Springer, 1998, pp 53-66

[5] L. H. Vu. *Formal Development and Verification of Railway Control Systems - In the context of ERTMS/ETCS Level 2.* PhD thesis, Technical University of Denmark, 2015.

[6] L. H. Vu, A.E. Haxthausen and J. Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Science of Computer Programming*: 133, 2017


```
type Game<'N when 'N: comparison>() =
   abstract member edges: 'N -> 'N list list
   abstract member goal: 'N list
   member this.solve (start: 'N list) =
      let win = new C5.ArrayList<bool option>()
      let toNode  = new C5.HashDictionary<int,'N>()
      let toVertex = new C5.HashDictionary<'N,int>()
      let deps = new C5.ArrayList<(int * int list) list>()
      let vertexOf n = ...
      let load v = ...
      let startVertices = List.map vertexOf start
      ...
      for v in startVertices do
         win.[v] <- Some false
      for m in this.goal do
         win.[vertexOf m] <- Some true

      let rec loop = function
        | []                                  -> toList win
        | (k,vs)::hes when win.[k] = Some true  -> loop hes
        | (k,vs)::hes                         ->
            match List.skipWhile (fun v -> win.[v] = Some true) vs with
            | []  -> win.[k] <- Some true; loop(hes@(deps.[k]))
            | v::vs' when win.[v] = Some false -> deps.[v] <- (k,vs')::deps.[v]; loop hes
            | v::vs' -> win.[v] <- Some false; deps.[v] <- [(k,vs')]; loop (load v @ hes)
      loop (List.collect load startVertices)
```

Figure 2: The core of Liu and Smolka's Local2 algorithm [4] formulated in F#

Symbolic Synthesis for Non-Negative Multi-Weighted Games*

Lasse S. Jensen, Isabella Kaufmann, Kim G. Larsen, Søren M. Nielsen & Jiří Srba
Department of Computer Science, Aalborg University, Denmark

Complex systems are an integral part of everyday life and the correctness of these systems is an area of great interest. For several safety-critical application areas the cost of undefined behaviour in a system can be very high, thus creating the demand for a more thorough verification. Traditionally model checking has been applied to verification of quantitative models like weighted timed automata [1] and 1-weighted extensions CTL [3]. In this area we provide a decidability result for a multi-weighted CTL (WCTL). While model checking verifies an existing system, the next step is synthesis which allows for the automatic generation of a controller derived from a formal specification. Synthesis is a very active topic and recent algorithmic contributions include optimised algorithms for LTL and CTL [5, 7] as well as energy games with multiple weights [6] and on-the-fly algorithms for reachability and safety games [2]. We present a generalised approach to synthesis which extends the current state-of-the-art by handling multiple weights with both lower- and upper-bounds.

Multi-weighted formalism We define a multi-weighted extension of a Kripke structure (n-WKS) as a tuple $K = (S, s_0, \mathcal{AP}, L, T)$ where S is a set of states, $s_0 \in S$ is the initial state, $\mathcal{AP}$ is a set of atomic propositions, $L : S \to \mathcal{P}(\mathcal{AP})$ is a labelling function and $T \subseteq S \times \mathbb{N}_0^n \times S$ is the transition relation. Let $\overline{w} \in \mathbb{N}_0^n$ then we denote the ith component of $\overline{w}$ by $\overline{w}[i]$, where $1 \leq i \leq n$. With this in mind we are ready to formulate a WCTL over an n-WKS as follows:

$$\begin{aligned}\varphi &:= \text{TRUE} \mid \text{FALSE} \mid a \mid \psi_1 \bowtie \psi_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 \mid \\ &\quad \mathit{reset}\ \#i\ \text{in}\ \varphi \mid AX\ \varphi \mid EX\ \varphi \mid AG\ \varphi \mid EG\ \varphi \mid AF\ \varphi \mid EF\ \varphi \mid E\varphi_1 U \varphi_2 \mid A\varphi_1 U \varphi_2 \\ \psi &:= \#i \mid c \mid \psi_1 \oplus \psi_2\end{aligned}$$

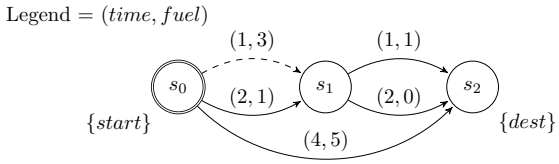
where $a \in \mathcal{AP}$, $\bowtie \in \{\leq, \geq\}$, $\oplus \in \{+, \cdot, -\}$, $c \in \mathbb{N}_0$, and $1 \leq i \leq n$ is a component index in a vector. A run ρ in the n-WKS K is an infinite or finite sequence of states and transitions and we define the cost of a run as the sum of all vectors along the traversed edges. It is based on this cost that we evaluate the sub-formula $(\psi_1 \bowtie \psi_2)$ defined with a vector component.

Theorem 1 *The model checking problem for WCTL is undecidable on a finite* 3*-WKS, but decidable on a finite n-WKS when we restrict comparison to $(\psi \bowtie c)$ and remove subtraction s.t. $\oplus \in \{+, \cdot\}$.*

Game Theoretic Framework One way to work with software synthesis is from a game theoretic view. We look at two-player games where one player acts as the controller and the other player acts as the environment. The game is defined as a tuple $(\mathcal{G}, \varphi)$ where $\mathcal{G}$ is the game graph and φ is the winning condition. We define the game graph as an n-WKS where the edges are annotated with n-dimensional non-negative vectors and are divided between the players into two disjoint sets $T_e \cup T_c$, where T_e belongs to the environment and T_c belongs to the controller. The winning conditions are expressed in WCTL. To gain familiarity with this formalism we refer to example 1 which depict a game modelling a self-driving car.

Example 1: Self-driving car
In Figure 1 we show a simple model of a self-driving car, where the dotted lines belong to the environment and the solid lines to the controller. The winning condition is illustrated as a WCTL specification φ, bounding the consumption of time and fuel.

*Based on the master thesis [4]



$$\varphi = AF \begin{pmatrix} \#1 \leq 3 \wedge \\ \#2 \leq 3 \wedge \\ dest \end{pmatrix}$$

Figure 1: The game graph $\mathcal{G}$ modeling a self-driving car and an anti congestion system. The first component on each weight, is the discrete units of time spent, and the second is the units of fuel consumed. The winning condition is expressed as a WCTL specification φ, where $\#i$ is the accumulated weight of the ith component in the cost

To define the state of a game we say that a configuration $(s, \overline{w}) \in S \times \mathbb{N}_0^n$ in the game consist of the current state s and the accumulated cost $\overline{w}$. Notice that the accumulated cost is always non-negative as the vectors on the edges are non-negative. We define the set of all configurations as $\mathcal{C}$. The game is played by moving from configuration to configuration, where $(s_0, 0^n)$ is the initial position. Given a configuration $(s, \overline{w})$ we proceed in the following manner:

- If all outgoing transitions belong to T_c, then the controller must choose.
- If all outgoing transitions belong to T_e, then the environment must choose.
- If there are both outgoing transitions in T_c and T_e, then the environment may choose from T_e or force the controller to choose from T_c.

Once either the controller or the environment has chosen an edge $(s, \bar{c}, s') \in T_e \cup T_c$ the next configuration becomes $(s', \overline{w} + \bar{c})$. The controller's choices are based on a strategy σ that, given history of the game, outputs the controller's next move. Based on the strategy's choices and all choices available to the environment, we unfold the game into an n-WKS. When the unfolded game restricted by the strategy, satisfy the winning condition φ we define it as a winning strategy. Hence the controller wins the game if they have a winning strategy. Below we show the winning strategy, as well as the unfolding, for the game presented in example 1.

Example 2: Winning strategy for the self-driving car
The example below is a strategy σ for the controller based on Figure 1.

$$\sigma(s_0) = s_0 \xrightarrow{(2,1)} s_1 \qquad \sigma(s_0 \xrightarrow{(1,3)} s_1) = s_1 \xrightarrow{(2,0)} s_2 \qquad \sigma(s_0 \xrightarrow{(2,1)} s_1) = s_1 \xrightarrow{(1,1)} s_2$$

Notice that the unfolded game satisfies the specification φ, thus σ is a winning strategy,

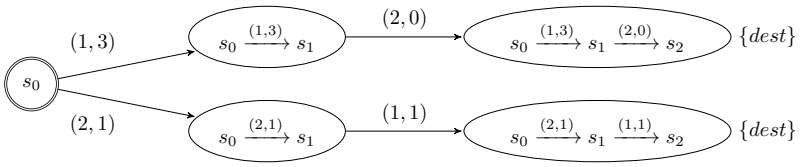


Figure 2: The unfolded game $\mathcal{G}\restriction\sigma$ where $\mathcal{G}\restriction\sigma \vDash_{0^n} \varphi$

Synthesis As WCTL is undecidable we define the reachability sub-class RWCTL as follows:

$$\varphi := AF\psi$$
$$\psi := a \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \#i \bowtie c.$$

Synthesis for WReach holds two main challenges; ensuring that we account for all possible behaviours of the environment and representing a possibly infinite set of vectors satisfying some lower-bound. Our goal is to create an algorithm that can detect and extract a winning strategy if one exist. Intuitively we do this by calculating a set of winning configurations $(s, \overline{w})$. A configuration $(s, \overline{w})$ is winning with respect to a specification $AF\psi$ if there is a winning strategy σ s.t. for the unfolded game, denoted $\mathcal{G}\restriction\sigma$, we have that $\mathcal{G}\restriction\sigma, s \vDash_{\overline{w}} AF\psi$. We begin by identifying all configurations which trivially satisfy the objective. The set is

defined as $F_0 = \{(s, \overline{w}) \in \mathcal{C} \mid s \models_{\overline{w}} \psi\}$. From the initial set we identify additional winning configuration by backwards traversal of the game graph. This is captured in the set F_i that is inductively defined for all $i \in \mathbb{N}$ s.t. $F_i = F_{i-1} \cup Add(F_{i-1})$ where Add calculates the set of configurations guaranteed to be able to reach a winning configuration in F_{i-1} in a single transition. The function accounts for the environment's choices, by only including a new configuration whenever all environmental edges and at least one controllable edge (if one exist), can lead to a winning configuration. Hence we create an increasing sequence of winning configurations $F_0 \subseteq F_1 \subseteq \ldots F_i = F_{i+1}$. We say that when $F_i = F_{i+1}$ the set is *final*.

Lemma 1 *A configuration $(s, \overline{w}) \in F_i$ for some $i \in \mathbb{N}$ iff $(s, \overline{w})$ is winning.*

To illustrate this approach we calculate F_i for the game presented in example 1 and show that the final set contain the configuration $(s_0, 0^n)$, implying that there is a winning strategy from the initial position.

Example 3: Calculation of winning configurations
Recall Figure 1, for the game $(\mathcal{G}, \varphi)$ then we have that $\varphi = AF(\#1 \leq 3 \wedge \#2 \leq 3 \wedge dest)$ and we compute $F_0 = \{(s_2, (k, j)) \mid 0 \leq k \leq 3 \text{ and } 0 \leq j \leq 3\}$ and F_i for any $i \in \mathbb{N}$ until $F_i = F_{i+1}$ s.t.,

$$F_1 = F_0 \cup \{(s_1, (k, j)) \mid 0 \leq k \leq 2 \text{ and } 0 \leq j \leq 2\} \cup \{(s_1, (k, j)) \mid 0 \leq k \leq 1 \text{ and } 0 \leq j \leq 3\}$$
$$F_2 = F_1 \cup \{(s_0, (k, 0)) \mid 0 \leq k \leq 1\}$$
$$F_3 = F_2 \cup \emptyset$$

Note that we cannot add s_0 before F_2 because the only transition which leads directly to s_2 is $(s_0, (4, 5), s_2) \in T_c$ which breaks the upper-bounds.

If we only considered upper-bounds this algorithm would be sufficient as it would ensure a finite amount of configurations. However, the challenge of handling the possible infinite sets satisfying a lower-bound still remains.

Symbolic representation We find that an infinite set of non-negative vectors can be represented symbolically- As all configurations have a non-negative cost we can define these with this symbolic representation. Furthermore we can easily adapt the definition of F_0 and F_i to work with this symbolic representation, thus giving us an algorithm for WReach synthesis.

Theorem 2 *The synthesis problem for a finite game $(\mathcal{G}, \varphi)$, where φ is a WReach formula, is EXPTIME-complete.*

Future Work We propose to further investigate synthesis for WCTL, as well as extend the formalism to include partial observations and stochastic variables, to more realistically model the behaviour of the environment. This work will be part of the PhD studies of Isabella Kaufmann.

References

[1] Patricia Bouyer, Kim G. Larsen, and Nicolas Markey. "Model-Checking One-Clock Priced Timed Automata". In: *FOSSACS 2007*. Vol. 4423. LNCS. Springer, 2007, pp. 108–122.

[2] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. "Efficient on-the-fly algorithms for the analysis of timed games". In: *CONCUR 05*. Vol. 3653. LNCS. Springer, 2005, pp. 66–80.

[3] Jonas Finnemann Jensen, Kim Guldstrand Larsen, Jiří Srba, and Lars Kaerlund Oestergaard. "Efficient model-checking of weighted CTL with upper-bound constraints". In: *STTT* 18.4 (2016), pp. 409–426.

[4] Lasse S. Jensen, Isabella Kaufmann, and Søren M Nielsen. *Symbolic Synthesis of Non-Negative Multi-Weighted Games with Temporal Objectives*. 2017.

[5] Barbara Jobstmann and Roderick Bloem. "Optimizations for LTL Synthesis". In: *FMCAD '06*. FMCAD '06. IEEE Computer Society, 2006, pp. 117–124.

[6] Marcin Jurdziński, Ranko Lazić, and Sylvain Schmitz. "Fixed-Dimensional Energy Games are in Pseudo-Polynomial Time". In: *ICALP 2015*. Vol. 9135. LNCS. Springer, 2015, pp. 260–272.

[7] Tobias Klenze, Sam Bayless, and Alan J. Hu. "Fast, Flexible, and Minimal CTL Synthesis via SMT". In: *CAV 2016*. Vol. 9779. LNCS. Springer, 2016, pp. 136–156.

An Energy-aware Mutation Testing Framework for EAST-ADL Architectural Models

Raluca Marinescu*, Predrag Filipovikj*, Eduard Enoiu*, Jonatan Larsson†, and Cristina Seceleanu*

*{first.last}@mdh.se, †jln13010@student.mdh.se
Mälardalen University, Västerås, Sweden.

Background and Motivation. Early design artifacts of embedded systems, such as architectural models, represent convenient abstractions for reasoning about a system's structure and functionality. One such example is the Electronic Architecture and Software Tools-Architecture Description Language (EAST-ADL) [3], a domain-specific architectural language that targets the automotive industry. EAST-ADL is used to represent both hardware and software elements, as well as related extra-functional information (e.g., timing properties, triggering information, resource consumption). Testing architectural models [1] is an important activity in engineering large-scale industrial systems, which sparks a growing research interest. Modern embedded systems, such as autonomous vehicles and robots, have low-energy computing demands, making testing for energy usage increasingly important. Nevertheless, testing resource-aware properties of architectural models has received less attention than the functional testing of such models. In our previous work [11], we have outlined a method for testing energy consumption in embedded systems using manually created faults based on statistical model checking of a priced formal system model. In this paper, we extend our previous work by showing how mutation testing [6] can be used to generate and select test cases based on the concept of energy-aware mutants– small syntactic modifications in the architectural model, intended to mimic real energy faults. Test cases that can distinguish a certain behavior from its mutations are sensitive to changes in the model, and hence considered to be good at detecting faults. The main contributions of this paper are: (i) an approach for creating energy-related mutants for EAST-ADL architectural models, (ii) a method for overcoming the equivalent mutant problem [9] (i.e., the problem of finding a test case which can distinguish the observable behavior of a mutant from the original one), (iii) a test generation approach based on UPPAAL Statistical Model Checker (SMC) [4], and (iv) a test selection criteria based on mutation analysis using our MATS tool[1] [8].

Proposed Framework. In this section, we describe our mutation testing framework that uses energy consumption goals to automatically select test suites based on random system simulations. The framework is enabled by transforming the EAST-ADL model into a network of priced timed automata (PTA) [10]. It is composed of several steps, mirrored in Figure 1:

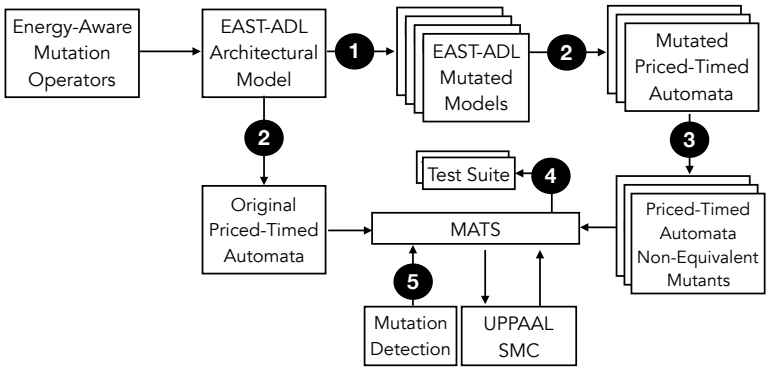


Figure 1: Overview of the energy-aware mutation testing framework.

[1] MATS is an open source software and is available at https://github.com/JLN93/MATS-Tool

(1) Energy-Aware Mutant Generation. The consumption of a resource r for an East-adl component represents the accumulated resource usage up to some point in time. Based on this assumption, resources can be classified as continuous or discrete [13] . In this paper, we focus on energy consumption, which is a continuous resource assumed to evolve linearly in time ($r(t) = n \times t$, where $n \in \mathbb{N}$ and t is the elapsed time). Since in East-adl the resource usage annotation is provided at component level, we define the total energy consumption of the system as $r_{total}(t) = \sum_{i=1}^{m} r_i(t)$, where m is the number of functional components. In mutation testing, faults are injected based on a predefined set of mutation operators. Ideally, such mutants should represent commonly-occurring faults, but to the best of our knowledge, there is no previous work on identification of resource-related faults at the architectural level. Given this, we propose a set of mutation operators applied on: (i) the East-adl resource annotation (Energy Consumption Replacement Operator (ero)), (ii) the timing behavior of an East-adl component (Period Replacement Operator (pro), Execution Time Replacement Operator (ero)), and (iii) the structure of the functional architecture (Component Removal Operator (cro), Component Insertion Operator (cio), and Triggering pattern Replacement Operator (tro)). These mutation operators are systematically applied to the entire East-adl model, resulting in a set of energy-aware mutants, each simulating one syntactic model change.

(2) East-adl to PTA. In order to use Uppaal SMC for test case generation, we transform the East-adl model (with energy consumption annotations) into a PTA model. Each East-adl component is automatically transformed into a network of two PTA: an *interface* automaton, which encodes the interface of the component, and a *behavior* automaton, used to model the component's internal behavior. The triggering of each component, timing information, as well as the resource annotations, are included in the interface PTA. The energy consumption starts at the moment data is read from the input ports until the component writes the data to the output ports. This means that the energy consumed by each component increases with the execution time, modeled as a cost "c" in PTA ($c(t) = n_c \times t$, where $n_c \in \mathbb{N}$ is the rate of consumption over time t), but not when the component is idle ($c'(t) = 0$). A monitor automaton is added to compute the energy used by the system based on the energy consumed by each component. For more details, we refer the reader to our previous work [10].

(3) Detection of Equivalent Mutants. Let O_n be the original PTA model and M_m be a mutant of the former obtained by applying a predefined mutant operator. We say that models O_n and M_m are *equivalent* if there is no input parameter for which the difference in energy consumption of the models exceeds some predefined threshold within some bounded time limit. Otherwise, there is a valuation of the input parameters for which the mutant can be detected. From the above, it is obvious that the *mutant equivalence check* can be reduced to a *satisfiability problem* [5]. Let $\Phi = \{\varphi_1, \varphi_2, ..., \varphi_k\}$ and $\Psi = \{\psi_1, \psi_2, ..., \psi_l\}$ denote the set of constraints for the energy consumption of c_n and c_m in O_n and M_m, respectively. The mutant M_m is not equivalent to O_n if the following conjunction evaluates to true: $\exists l_k. \bigwedge_{i=1}^{k} \varphi_i(l_k) \wedge \bigwedge_{j=1}^{l} \psi_j(l_k) \wedge (|c_n - c_m| \geq threshold)$, where l_k is an arbitrary input parameter. Reducing the mutant equivalence checking to a satisfiability problem has been considered in other frameworks. Brillout et. al [2] exploits a similar technique for functional testing of Simulink models. In comparison, our framework is specifically tailored for resource-aware mutation testing of architectural models. The HiLiTe tool [12] is another example of using an SMT solver for improving test case generation for large-scale complex and constrained models. Given the fact that the energy consumption is of continuous nature, we have to resort to a specialized type of SMT-solving suitable for hybrid systems [7].

(4) Test Suite Generation. We create executable test cases using the Mats tool [8], by extracting the input parameters and the energy values at predefined time points from the simulation traces produced by Uppaal SMC. Each test input is a vector of signals where the time-dependent behavior of the model is executed using an ordered sequence of signals. Mats uses Uppaal SMC for obtaining simulation traces over a predefined number of runs of the system model. A simulation can be formulated as the property: *simulate* $n[bound]\{E_1, .., E_k\}$ in Uppaal SMC, where n is the number of simulations to be performed, *bound* is the time bound on the simulations, and $E_1, .., E_k$ are the monitored expressions. Each test case is executed on both the original model and its mutated counterpart. In order to minimize the final set of test cases we remove the test cases not contributing to the mutation score [8].

(5) Mutant Detection Criteria. We show how to detect energy mutants using the Mats tool. A

mutant is detected by a test suite if the energy signal diverges drastically at certain time points from the expected values (e.g, substantial energy deviations). To measure the mutant-revealing ability of a test suite, we use a quantitative measure of a mutant detection oracle. Let a test case T be generated for a mutated model M, and let $E_M = E_{M1}, ..., E_{MN}$ be the set of energy signals obtained by running M for the test inputs in T and sampled at N time points. Let $E_O = E_{O1}, ..., E_{ON}$ be the corresponding expected energy signals. We use a threshold to check if the distance between each value of E_O and E_M at each time point is larger than this threshold. If there is at least one energy value in E_M for which the distance is larger than the expected threshold then we consider the mutant M detected.

Conclusions and Future Work. In this paper we have outlined a framework for energy-aware mutation testing of EAST-ADL architectural models. Given the large number of energy mutations we aim to reduce the number of equivalent mutants by employing an SMT-solver. In addition, this framework selects test suites contributing to the overall mutation score using UPPAAL SMC and MATS. Future work aims to apply this framework on an industrial case to expose its strengths as well as limitations both in terms of test efficiency and effectiveness.

Acknowledgements. The authors of this work are supported by the following projects: Swedish Governmental Agency for Innovation Systems (VINNOVA) and ECSEL (EU's Horizon 2020) under grant agreement No 737494, VINNOVA VeriSpec project 2013-01299, Swedish Research Council (VR) project "Adequacy-based testing of extra functional properties of embedded systems" and the Swedish Knowledge Foundation (KKS) project DPAC – "Dependable Platforms for Autonomous systems and Control".

References

[1] Antonia Bertolino, Paola Inverardi, and Henry Muccini. Software architecture-based analysis and testing: a look into achievements and future challenges. *Computing*, 95(8):633–648, 2013.

[2] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for Simulink models. In *Formal Methods for Components and Objects*, pages 208–227. Springer, 2010.

[3] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Papadopoulos, et al. The EAST-ADL architecture description language for automotive embedded software. In *Model-based engineering of embedded real-time systems*, pages 297–307. Springer, 2010.

[4] Alexandre David, Kim Larsen, Axel Legay, Marius Mikučionis, Danny Poulsen, Jonas Van Vliet, and Zheng Wang. Statistical model checking for networks of priced timed automata. *Formal Modeling and Analysis of Timed Systems*, pages 80–96, 2011.

[5] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[6] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[7] Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. dreach: δ-reachability analysis for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 200–205. Springer, 2015.

[8] Jonatan Larsson. Automatic Test Generation and Mutation Analysis using UPPAAL SMC. In *Bachelor of Science Thesis Report*. MDH Diva, 2017.

[9] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1):23–42, 2014.

[10] Raluca Marinescu, Eduard Enoiu, and Cristina Seceleanu. Statistical Analysis of Resource Usage of Embedded Systems Modeled in EAST-ADL. In *VLSI Symposium*, pages 380–385. IEEE, 2015.

[11] Raluca Marinescu, Eduard Enoiu, Cristina Seceleanu, and Daniel Sundmark. Automatic Test Generation for Energy Consumption of Embedded Systems Modeled in EAST-ADL. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 69–76. IEEE, 2017.

[12] Hao Ren, Devesh Bhatt, and Jan Hvozdovic. Improving an Industrial Test Generation Tool Using SMT Solver. In *NASA Formal Methods Symposium*, pages 100–106. Springer, 2016.

[13] Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. REMES: A Resource Model for Embedded Systems. In *International Conference on Engineering of Complex Computer Systems*. IEEE, 2009.

Consequence Testing for Automotive Software through Mocking (Extended Abstract)*

Wojciech Mostowski

Center for Research on Embedded Systems, Halmstad University, Sweden
wojciech.mostowski@hh.se

1 Introduction

In an earlier paper [9], using a small but complete case study from an automotive domain, we have described the method and challenges in model-based testing (MBT) [11] of low level C code designed to run in a multi-vendor library scenario [4]. In this short paper we discuss the idea of consequence testing of possible defects in the libraries stemming from the differences in particular vendor implementations and inevitable specification drifts [2]. The principle idea and approach is through injecting faults into *executable* models, rather than the actual (typically very complicated) code, and plugging this faulty model into the software to be tested using a *mocking mechanism* [10] – a model-based "emulation" of a software component. A faulty model is much easier to construct than injecting the faults in the actual implementation. Moreover, having full control over the model and its execution-time behaviour, it is much easier to lead the faulty component behaviour during testing to witness a failure in the complete system. Such faulty models can be constructed by hand to introduce particular kinds of faults, or can be generated with automata learning techniques [8] applied on known faulty or specification non-compliant implementations.

This work is part of an on-going project AUTO-CAAS [3], where techniques to aid and automate model-based testing of automotive software adhering to the Autosar [4] industry standard are investigated.

2 Model-Based Testing of Autosar using QuickCheck

The simplest concrete example of an Autosar component and a client code that utilises it is that of a message box (`MBox`) implemented over a circular buffer (`CirqBuff`). The particular method and tool that we use in our project is QuickCheck [1, 6]. In QuickCheck, a model is an Erlang [5] functional program that follows a predefined structure and API to form a behavioural description of a given software component. The properties that can be checked with generated tests are postconditions (test oracles) of single operations and model invariants. The model also defines the *symbolic execution* state of the component under test and operation preconditions, both of which define the valid execution traces for the tests to be generated. For the software under test written in C, which is the case for the Autosar components, QuickCheck provides a flexible interface that allows to abstract the C function calls in the model and seamlessly run the underlying C implementation during the actual testing. The general philosophy of QuickCheck is to run several short tests quickly to detect problems early, rather than creating large and complex testing scenarios for which failures are extremely difficult to analyse. Should a larger

*This work is supported by the Swedish Knowledge Foundation grant for the AUTO-CAAS project.

test case need to be generated to exhibit a bug, QuickCheck offers a mechanism to *shrink* the test data to a smaller, possibly minimal, set that leads to the same failure.

The method for specifying the message queue and testing it with QuickCheck is described in detail in [9], which includes full specification for the circular buffer library and the message box implementation[1]. Though simplistic, this method is representative, in terms of the process and associated challenges, of specifying and testing large scale Autosar projects with QuickCheck [2].

3 Consequence Testing Through Mocking

The two key enabling mechanisms in QuickCheck for consequence testing are the *call-out* specifications and the mocking API. Call-out specifications are given in a process algebra-like language [10] and describe the calls to other components that the operation is allowed or required to make. The mocking API allows to trace these calls by providing a C wrapper around function calls in the system under test to intercept and analyse the lower-level calls. More importantly, the mocking API can *replace* the calls in the system under test with its own implementations. This allows us to execute a program over a specified library that is essentially emulated by QuickCheck with a modelled behaviour. This way, we can introduce faults on the level of libraries without modifying the library original C code (which in the worst case might not even be available). Instead, we plug in different models instead of the library. These plugged in models can either have a fully correct behaviour or a faulty one. The faulty behaviour in turn, can be either purposely injected to test the resistance of the top-level software against buggy libraries, or represent known drifts or non-compliances from the specification to check if (and how) these drifts affect the top-level behaviour of the system. In other words, to test whether the software accounts for these known possible drifts. In the general methodology, we keep plugging in such faulty components into the system to test for the (absence of) consequences of low-level faults in the top-level behaviour.

For example, Figure 1 shows a concrete snapshot from our message box specification where the call to `CirqBuffPush` of the circular buffer library is specified to have a hidden fault in that it silently limits the buffer size to 128 bytes, regardless of what the user specified during initialisation. Such a fault could be, e.g., caused by memory limitation on a small embedded platform. The top of the figure gives a specification that assumes a correct behaviour of the circular buffer provided the user fulfils the precondition, i.e., one is not trying to overflow the buffer, while the bottom part specifies the faulty behaviour – `CirqBuffPush` returns 1 indicating an error, when elements past the 128 boundary are pushed into the queue.

4 On-Going Research

The testing of the message box with QuickCheck based on the specification in Fig. 1 does not immediately reveal the problem with the limited circular buffer. In fact, the tests generated by default simply pass. This is because the MBT strategy of QuickCheck needs to be driven towards hitting the 128 byte mark, which is not the default behaviour (following the minimal testing approach mentioned earlier).

In our small example, the bug is triggered easily by a couple of additions to the specification to weight the post message calls more and to extend the average length of the test cases. However, in a more general case, this is not sufficient. Our technique so far assumes the introduction of single faults and manually driving the testing process towards hitting these faults. Ideally,

[1] `https://github.com/parai/OpenSAR`.

```
% Only allow posting messages to a non-full box:
post_message_pre(S) -> S#mbox_state.ptr /= undefined
  andalso length(S#mbox_state.elements) < S#mbox_state.size.
...
% Underlying call to CirqBuffPush always succeeds (returns 0):
post_message_callouts(S, [_, Value]) ->
  ?CALLOUT(mbox, 'CirqBuffPush', [?WILDCARD, Value], 0).
```

```
% Faulty implementation fails with return value 1 when over 128 elements
% are placed in the buffer:
post_message_callouts(S, [_, Value]) ->
  ?CALLOUT(mbox, 'CirqBuffPush', [?WILDCARD, Value],
    if length(S#mbox_state.elements) < 128 -> 0; true -> 1 end).
```

Figure 1: Model-based fault injection.

one would prefer to indicate which parts of the specifications should be considered faults, and let the tool drive the test generation process automatically to reach these specification areas. It can be just one state in the specification or a whole group of states that represent a family of faults. Thus, the first challenge is to provide specification means to mark faulty behaviours, the second is to develop a test generation routine geared towards reaching these faulty behaviours.

One possible approach is to use automata learning techniques to build such fault models [8] by learning the fault state space of a component from a collection of successful and failed test case runs. A fault model of this kind, which generalises single faults into fault scenarios, can be then used to drive test generation by only accepting model transitions that possibly lead to the fault states. Driving the test generation this way is not difficult and can be solved by stating specific pre-conditions that only enable model transitions eventually leading to faults. With automata learning, the main challenge are the scalability issues when applied to non-trivial systems. Initial ideas for this are presented in [8].

5 Conclusions

We briefly discussed a technique and some associated challenges for discovering high-level faults in software that works with multi-vendor automotive libraries, which may or may not adhere to the official specification. Our technique builds on model-based testing and utilises features available in our particular MBT tool QuickCheck – call-out specifications and library calls mocking. In the long run, the developed techniques should serve in improving testing of (future) autonomous functions in automotive software, which due to the inherent complexity and safety requirements are the next difficult target in testing [7].

References

[1] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with QuviQ QuickCheck. In *Proceedings of ERLANG'06*, pages 2–10. ACM, 2006.

[2] T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *Eighth IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 1–4, 2015.

[3] T. Arts and M.R. Mousavi. Automatic consequence analysis of automotive standards (AUTOCAAS). In *First International Workshop on Automotive Software Architectures (WASA 2015)*, pages 35–38. ACM Press, 2015.

[4] *AUTOSAR BSW and RE Conformance Test Specification, Release 4.0, Revision 2*, 2011.

[5] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly, 2009.

[6] J. Hughes. QuickCheck testing for fun and profit. In *Proceedings of PADL'07*, pages 1–32. Springer, 2007.

[7] P. Koopman and M. Wagner. Challenges in autonomous vehicle testing and validation. In *2016 SAE World Congress*, 2016.

[8] S. Kunze, W. Mostowski, M.R. Mousavi, and M. Varshosaz. Generation of failure models through automata learning. In *Second International Workshop on Automotive Software Architectures (WASA 2016)*, pages 22–25. IEEE Society, 2016.

[9] Wojciech Mostowski, Thomas Arts, and John Hughes. Modelling of Autosar libraries for large scale testing. In Holger Hermanns and Peter Höfner, editors, *2nd Workshop on Models for Formal Analysis of Real Systems (MARS 2017)*, volume 244 of *Electronic Proceedings in Theoretical Computer Science*, pages 184–199. Open Publishing Association, April 2017.

[10] J. Svenningsson, H. Svensson, N. Smallbone, T. Arts, U. Norell, and J. Hughes. An expressive semantics of mocking. In *Fundamental Approaches to Software Engineering*, volume 8411 of *LNCS*, pages 385–399. Springer, 2014.

[11] J. Tretmans. Model-based testing and some steps towards test-based modelling. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 297–326. Springer, 2011.

Analyzing Changes on Configurable Systems with #ifdefs

Larissa Braz[1], Rohit Gheyi[1], Volker Stolz[2], Márcio Ribeiro[3]

[1] Federal University of Campina Grande, Brazil
[2] Høgskulen på Vestlandet, Norway
[3] Federal University of Alagoas, Brazil

Introduction

Most software systems are frequently changed by developers during the development process. These changes may have consequences in several parts of the system. Change impact analysis is the task of finding the consequences of a change to the system code [1]. However, manual approaches to detect the impacts of a change may not be applicable on most systems, due to the large size of current systems and the number of dependencies between involved entities [2].

Mongiovi et al. [3] present SAFIRA that analyzes two versions of a Java/AspectJ program and identify the methods impacted by the change. It decomposes the transformation into a set of small-grained transformations and formalizes a set of laws to calculate the impact of each small-grained transformations separately. Impact analysis approaches may be useful when analyzing changes performed on Java code. For example, SAFEREFACTORIMPACT [3] uses SAFIRA to analyze code changes and generate tests only for the impacted methods.

In the context of the C language, Frama-C [4] is a plug-in that allows the automatic computation of the set of statements impacted by a selected statement of a system. CppDepend[1] is a static analysis tool for C/C++ code. It allows analyzing and visualizing code dependencies by doing impact analysis. However, most of the C available tools, such as GCC,[2] consider only one version of the system at a time. In addition, they consider only one configuration of the system per analysis. Generating, compiling, and testing all configurations may be costly and not feasible for most configurable systems due to the high number of potential variants [5, 6]. Therefore, in practice developers usually only check few configurations of the code or the default one.

Variability-aware parsers, such as TypeChef [7], analyze the code by considering the complete configuration space. They generate abstract syntax trees (AST) enhanced with all variability information. However, the time-consuming setup and compilation process of these tools hinder the analysis of some systems. TypeChef still misses interactions between preprocessor features and lacks flexibility. Both aspects are addressed in SuperC, another variability-aware parser [8]. It is faster than TypeChef, but it does not perform type-checking analysis. Moreover, none of the previous approaches consider code changes to reduce the effort of evaluating configurable systems.

In this work, we propose an analyzer of changes on configurable systems with `#ifdefs`. We use change impact analysis to identify impacted macros. We consider that a macro is directly impacted when the change modifies it. A macro is indirectly impacted when it enables the compilation of semantically impacted code. For example, if a change affects a variable that is later used under the `M1` macro, then `M1` is indirectly impacted by the change. We show that this approach removes false negatives that we have not been able to handle in our previous impact analysis [9].

Improving the Impact Analysis Approach

In our previous work [9], we propose CHECKCONFIGMX, a change-centric tool to compile configurable systems with `#ifdefs`. As input, it receives two versions of a file from a configurable system

[1] https://www.cppdepend.com/

[2] https://gcc.gnu.org/

```
1  int x = 0;
2  #ifdef M1 && !M2
3  int y = x + 1;
4  #endif
```

(a) Code snippet of the original file.

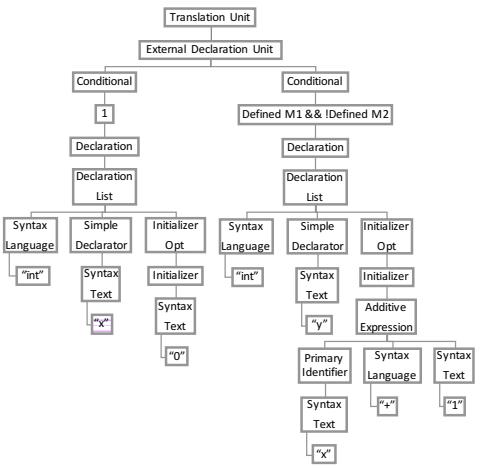
```
1
2  #ifdef M1 && !M2
3  int y = x + 1;
4  #endif
```

(b) Code snippet of the modified file.

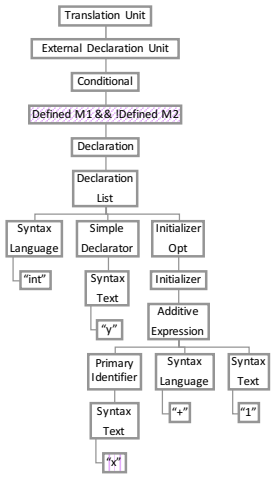
Listing 1: Code changes introducing a compilation error *(use of undeclared variable)*.

implemented with `#ifdefs`. Next, it performs an impact analysis using text diff to identify the modified lines and the impacted macros. It yields the set of impacted configurations as a set of enabled `#defines` and compiles only them. As a result, it returns the introduced compilation errors and their related configurations. However, CHECKCONFIGMX impact analysis may have false negatives since it does not consider the structure of the system to identify impacted macros that have no code changes within `#ifdefs`. For example, consider the code snippet presented by Listing 1a. The developer performs a change where she removes the declaration of the variable `x`. However, this variable is still used within the `#ifdef M1 && !M2` block. Listing 1b presents the modified code. CHECKCONFIGMX does not detect the compilation error (*use of undeclared variable*) introduced by Listing 1.

In this work, we propose an improvement for CHECKCONFIGMX impact analysis. After receiving two versions of a configurable system, we use xtc [8] to generate an AST of each one. Figures 1a and 1b represent xtc's output for Listings 1a and 1b, respectively. Mandatory codes are represented under a *Conditional* and an *1* (always true) nodes, while codes under `#ifdefs` blocks are represented under a *Conditional* and a *<macro(s) condition(s)>* nodes. Next, we compare both ASTs to identify the changes. In this example, we note that the left branch of the *External Declaration Unit* node was removed from the first AST to the second one. Then, we search this branch leaf nodes to identify variables or function names. As a result, we find the node `x`, which we marked with horizontal lines in Figure 1a. Following, we search all occurrences of `x` in the modified file AST, which we marked with vertical lines in Figure 1b. Finally, for each occurrence, we walk its branch up to identify if its inside an `#ifdef` block, if positive we



(a) AST of original code (Listing 1a).



(b) AST of modified code (Listing 1b).

Figure 1: ASTs generated by xtc for both file versions of Listing 1.

```
1  static const char hush_version_str[] ALIGN1 = "HUSH_VERSION="BB_VER;
```

Listing 2: Code from `hush.c` from BusyBox before preprocessing.

consider the macro(s) as impacted. In our example, we identify that the `M1` and `M2` macros are impacted, which we marked with diagonal lines in Figure 1b.

However, parsers, such as TypeChef and SuperC, may break when parsing C code that has not been preprocessed. For example, these parsers may not parse the code presented by Listing 2 due to missing declarations of the `ALIGN1` and `BB_ver` macros. In cases like that, we use an alternative approach to identify impacted macros after code changes. For example, receiving Listing 1 as input, if the parser fails, we identify that the first line of Listing 1a (`int x = 0;`) changed, since it is removed from the code in Listing 1b. Next, we remove the keywords (`int`), the operators (`=` and `+`), and the numbers (`1`). We yield a set of words: `{x}`. Then, we search all occurrences of the words in the modified code. As a result, we identify the set of impacted lines of the modified code: `{2}`. Then, we identify `{M1, M2}` as the set of impacted macros. Next, we follow the same steps as in the AST-approach: we generate and compile all impacted configurations based on the impacted macros. As a result, we identify the use of an undeclared variable in configuration `M1`.

Conclusions

By not considering the impact of the transformations applied to a configurable system, exhaustive approaches may have to analyze all possible configurations. Our previous evaluation results [9] show evidence that CHECKCONFIGMX can be useful in analyzing transformations applied to files with a large number of macros. However, due to its simple impact analysis, CHECKCONFIGMX may miss some compilation errors. To reduce the number of false negatives, in this work, we extend our previous work [9] by improving its change impact analysis. We identify impacted macros using AST or through a word-search approach. As future work, we intend to evaluate our new change impact analysis on open source projects, and compare it with CHECKCONFIGMX previous change impact analysis.

Acknowledgement *Supported by the bilateral SIU/CAPES/10032 project "Modern Refactoring".*

References

[1] X. Ren, B. Ryder, M. Stoerzer, and F. Tip. Chianti: A change impact analysis tool for Java programs. In *ICSE*, pages 664–665, 2005.

[2] S. Lehnert. A taxonomy for software change impact analysis. In *IWPSE-EVOL*, pages 41–50, 2011.

[3] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba. Making refactoring safer through impact analysis. *SCP*, 93:39–64, 2014.

[4] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *FAC*, 27(3):573–609, 2015.

[5] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *TOSEM*, 21(3):1–39, 2012.

[6] A. Iosif-Lazar, J. Melo, A. Dimovski, C. Brabrand, and A. Wasowski. Effective analysis of c programs by rewriting variability. *Programming Journal*, 1(1):1, 2017.

[7] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA*, pages 805–824, 2011.

[8] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *PLDI*, pages 323–334, 2012.

[9] L. Braz, R. Gheyi, M. Mongiovi, M. Ribeiro, F. Medeiros, and L. Teixeira. A change-centric approach to compile configurable systems with #ifdefs. In *GPCE*, pages 109–119, 2016.

Operational Semantics of a Weak Memory Model
inspired by Go

Daniel S. Fava,[1] Martin Steffen,[1] Volker Stolz[1,2] and Stian Valle[1]

[1] Dept. of Informatics, University of Oslo
[2] Western Norway University of Applied Sciences

A *memory model* dictates which values may be returned when reading from memory. In a parallel computing setting, the memory model affects how processes communicate through shared memory. The design of a proper memory model is a balancing act. On one hand, memory models must be lax enough to allow common hardware and compiler optimizations. On the other, the more lax the model, the harder it is for developers to reason about their programs. In order to alleviate the burden on programmers, a weak memory model should provide what is called the *data-race freedom guarantee*, which allows reasoning in terms of sequential consistency provided a program is data-race free.

In this paper we present a theory for weak memory with channel communication as the sole synchronization primitive. There are few studies on channel communication as a synchronization primitive for a weak memory model. We formalize the memory model in a small-step operational semantics and implement it in an executable semantics framework [4] from which we obtain an interpreter. Similar to Boudol and Petri, we favor an operational semantics because it allows us to prove the DRF guarantee "at the programming language level." This yields a more concrete interpretation of the DRF guarantee as compared to formalisms in which the notion of a program is abstracted away, often in the form of a graph [1].

The calculus we propose is inspired by the Go programming language developed at Google, which recently gained traction in networking applications, web servers, distributed software and the like. It features goroutines (i.e., asynchronous execution of function calls resembling lightweight threads) and buffered channel communication in the tradition of CSP or Occam.

The Go memory model

The *happens-before relation* is used in the Go memory model to describe which reads can observe which writes to the same variable. It says, for example, that *within a single goroutine, the happens-before relation boils down to program order* and, between goroutines, events can appear to happen out of program order. If the effects of a goroutine are to be observed by another, a synchronization primitive must be used in order to establish a relative ordering between events belonging to the different goroutines. The Go memory model advocates channel communication as the main method of synchronization [3]. In particular, it states that *a send on a channel happens before the corresponding receive from that channel completes.* Our semantics incorporates channels for message passing, goroutines for asynchronous code execution, and it allows for out-of-order execution where writes to memory can be arbitrarily delayed.

Abstract syntax

The abstract syntax of the calculus is given in Table 1. *Values* are written generally as v and include booleans, integers, and etc (these more obvious values are not explicitly listed on the table). Note that local variables (or registers) are also counted as values and are denoted r. Names (or references) are also considered values and are denoted n. Names are used, for example, when referring to different channels – when presenting the semantics, we will use c for indicating a reference to a channel.

A new channel is created by `make` $(\texttt{chan}\ T, v)$, where T represents the type of values carried by the channel, and the non-negative integer v the channel's capacity. Sending a value over a channel and

v	::=	$r \mid \underline{n}$	values
e	::=	$t \mid v \mid \texttt{load}\, z \mid z := v$	expressions
	$\mid$	$\texttt{make}\,(\texttt{chan}\, T, v) \mid \leftarrow v \mid v \leftarrow v \mid \texttt{close}\, v \mid \underline{\texttt{pend}}\, v$	
	$\mid$	$\texttt{if}\, v\, \texttt{then}\, t\, \texttt{else}\, t \mid \texttt{go}\, t$	
g	::=	$v \leftarrow v \mid \leftarrow v \mid \texttt{default}$	guards
t	::=	$\texttt{let}\, r = e\, \texttt{in}\, t \mid \sum_i \texttt{let}\, r_i = g_i\, \texttt{in}\, t_i$	threads

Table 1: Abstract syntax

receiving a value as input from a channel are written respectively as $v_1 \leftarrow v_2$ and $\leftarrow v$. After the operation `close`, no further values can be sent on the specified channel. Attempting to send values on a closed channel leads to a panic. The expression `pend` v represents the state immediately after sending a value over a channel. Note that `pend` is part of the *run-time* syntax as opposed to the user-level syntax, i.e., it is used to formulate the operational semantics of the language but is not part of the syntax available to the programmer.

Starting a new asynchronous activity (i.e., goroutine) is done using the `go`-keyword. Select-statements, written using the $\sum$-symbol, consist of a finite set of guarded branches. The `let`-construct $\texttt{let}\, r = e\, \texttt{in}\, t$ combines sequential composition and the use of scopes for local variables r. It becomes *sequential composition* when r does not occur free in t. We use semicolon as syntactic sugar in such situations.

Operational semantics with delayed writes

Programs consist of the parallel composition of goroutines $\langle \sigma, t \rangle$, write events $n(\!|z{:=}v|\!)$, and channels $c[q_f, q_b]$. *Write events* are 3-tuples from $N \times X \times Val$; they record the shared variable being written to and the written value, together with a unique identifier n. In the current semantics, read accesses to the main memory cannot be delayed; consequently, there are no read events.

In addition to the code t to be executed, goroutines $\langle \sigma, t \rangle$ contain local information about earlier memory interaction. *Local states* σ are are tuples of type $2^{(N \times X)} \times 2^N$ abbreviated as Σ. We use the notation (E_{hb}, E_s) to refer to the tuples. The first component of the local state, E_{hb}, contains the identities of all write events that have happened before the current stage of the computation of the goroutine. The second component of the local state, E_s, represents the set of identities of write events that, at the current point, are shadowed (i.e., no longer visible to the goroutine).

The reduction rules for reads and writes are given on Table 2. From a goroutine's point of view, its reads and writes appear in program order. This is guaranteed by the absence of delayed reads and by disallowing reads from obtaining values of writes that have been shadowed. Writes from other goroutines, however, may appear out of order: writes are placed on a global pool and subsequent reads can read any write from the pool as long as the event has not been shadowed from the reader's point of view.

Synchronization between goroutines is achieved by communicating via channels, as shown in Table 2. A channel is of the form $c[q_f, q_b]$, where c is a name and (q_f, q_b) a pair of queues referred to as *forward* and *backward* queue. For convenience, we use c_f and c_b when referring to channel's c forward and backward queues. When creating a channel (cf. rule R-MAKE), the forward channel is initially empty but the backward is not: it is initialized by a queue of length $|c|$, which corresponds to the capacity of the channel (the channel is synchronous when capacity is 0). In order to account for the synchronization power of channels, in addition to communicating a value, the queues are managed so that "happened before" and "shadowed" knowledge are also exchanged between communicating partners.

See our technical report for a detailed description of the semantics [2].

$$\frac{q = [\sigma_\bot, \ldots, \sigma_\bot] \qquad |q| = v \qquad \mathit{fresh}(c)}{p\langle \sigma, \texttt{let}\ r = \texttt{make}\ (\texttt{chan}\ T, v)\ \texttt{in}\ t\rangle \to \nu c\ (p\langle \sigma, \texttt{let}\ r = c\ \texttt{in}\ t\rangle \parallel c_f[] \parallel c_b[q])}\ \text{R-MAKE}$$

$$\frac{\neg \mathit{closed}(c_f[q])}{p\langle \sigma, c \leftarrow v; t\rangle \parallel c_f[q] \to p\langle \sigma, \texttt{pend}\ c; t\rangle \parallel c_f[(v, \sigma) :: q]}\ \text{R-SEND}$$

$$\frac{\sigma' = \sigma + \sigma''}{c_b[q_2 :: \sigma''] \parallel p\langle \sigma, \texttt{pend}\ c; t\rangle \parallel c_f[q_1] \to c_b[q_2] \parallel p\langle \sigma', t\rangle \parallel c_f[q_1]}\ \text{R-PEND}$$

$$\frac{\sigma' = \sigma + \sigma'' \qquad v \neq \bot}{c_f[q_1 :: (v, \sigma'')] \parallel p\langle \sigma, \texttt{let}\ r = \leftarrow c\ \texttt{in}\ t\rangle \parallel c_b[q_2] \to c_f[q_1] \parallel p\langle \sigma', \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel c_b[\sigma :: q_2]}\ \text{R-RECEIVE}$$

$$\frac{\sigma' = \sigma + \sigma''}{c_f[(\bot, \sigma'')] \parallel p\langle \sigma, \texttt{let}\ x = \leftarrow c\ \texttt{in}\ t\rangle \to c_f[(\bot, \sigma'')] \parallel p\langle \sigma', \texttt{let}\ x = \bot\ \texttt{in}\ t\rangle}\ \text{R-RECEIVE}_\bot$$

$$\frac{\neg \mathit{closed}(c_f[q])}{c_f[q] \parallel p\langle \sigma, \texttt{close}\ (c); t\rangle \to c_f[(\bot, \sigma) :: q] \parallel p\langle \sigma, t\rangle}\ \text{R-CLOSE}$$

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (n, z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(n)}{p\langle \sigma, z := v; t\rangle \to \nu n\ (p\langle \sigma', t\rangle \parallel n(\!|z{:=}v|\!))}\ \text{R-WRITE}$$

$$\frac{\sigma = (\_, E_s) \qquad n \notin E_s}{p\langle \sigma, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel n(\!|z{:=}v|\!) \to p\langle \sigma, \texttt{let}\ r = v\ \texttt{in}\ t\rangle \parallel n(\!|z{:=}v|\!)}\ \text{R-READ}$$

Table 2: Operational semantics: message passing and memory reads/writes

Contributions

- We define a novel semantics for weak memory with channel communication as synchronization primitive;
- We prove that our proposed weak memory upholds the *data-race freedom* guarantee;
- We present our implementation in an executable semantics framework, which allows us to derive an interpreter for programs in the target language.

References

[1] G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *Proceedings of POPL '09*, pages 392–403. ACM, January 2009.

[2] Daniel Fava, Martin Steffen, Volker Stolz, and Stian Valle. An operational semantics for a weak memory model with buffered writes, message passing, and goroutines. Technical Report 466, University of Oslo, Dept. of Informatics, April 2017.

[3] Go memory model. The Go memory model. https://golang.org/ref/mem, 2016.

[4] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Methods in Programming*, 79(6):397–434, 2010.

A diagrammatic approach for bracing heterogeneous models

Fazle Rabbi and Yngve Lamo

Western Norway University of Applied Science, Bergen, Norway
Fazle.Rabbi@hvl.no, Yngve.Lamo@hib.no

1. Introduction: Today's software systems are complex and involves interaction among several devices and applications using heterogeneous platforms. For modeling software systems we need to consider various aspects of systems. In order to deal with the complexity of software systems, software engineers usually separate the aspects of software systems which leads to different models. Decomposing a system based on aspects (such as structure and behavior) facilitate abstraction and provides flexibility in updating the decomposed sub-modules. However, to reason about the system as an integrated whole and to support concurrent coevolution of subsystems we need to coordinate and synchronize models that are representing different aspects of a system and represent distributed systems. Requirements for integrating heterogeneous distributed systems are increasing with the rapid technological advancements. Therefore, model composition is becoming a key issue in requirements analysis and design of complex systems. This requires formalization to understand and develop satisfactory solutions. The study of integrating heterogeneous systems is a complex process consisting of information and expert knowledge management, modeling, simulation, and decision making support [2].

The modeling concerns for the representation of complex systems require techniques for handling composition of heterogeneous artefacts. Efforts have been made in [3, 4] to specify the integration of heterogeneous systems by the integration of heterogeneous modeling languages. They categorized the need for language integration into three groups: language aggregation, language embedding, and language inheritance. An algebraic approach for integrating languages have been studied in [1, 6] where the authors used a Common Algebraic Specification Language (CASL) for the specification and development of modular software systems. However, these approaches are based on textual languages. We propose to use a diagrammatic approach for modeling complex software systems and propose a formal modeling approach to compose heterogeneous models in a coherent way. We demonstrate its application for the optimization of distributed resources. The formal modeling approach is presented by means of composite specifications which provide reusable patterns for the structural composition of software models. It can be used for modeling in the small in a sense that it can deal with modeling artefacts such as models, metamodel elements and relations between them; it can also be used for modeling in the large in a sense that we can coordinate heterogeneous modeling systems with composite specifications.

2. Composite specification: The formalization of composite specification is discussed as an approach for constructing complex software models. In this formalization, we integrate several modeling artefacts to define software structures. Composite specification can be used for constructing the structure of individual software models as well as correlating complex structures of distributed software models. As a motivation consider the following model integration problem. Consider two distributed systems for an orthopedic department and a radiology department in a hospital. To model these distributed systems we need to define

different domain concepts and constraints. These two systems share some resources and we need to model the interdependency of the systems with an integrated system model. There are some overlapping of concepts and constraints in these two systems and we need to specify inter-model constraints representing the global constraints governing the overall system. To model the resource allocation of the distributed systems and their optimization we wish to use a game theory model. Therefore we need to model the game theoretic perspective of the distributed resource allocations and link them with the distributed software models. To cope with this situation, we require a modeling framework that allows heterogeneous model integration and supports both modeling small and large distributed systems. We propose to use a modeling formalism with so-called composite specification that supports such model composition requirements. Inter-model constraints of heterogeneous systems can be specified using composite specification and it provides a building block for integrating software models in a coherent way. Composite specification may have numerous applications in software engineering as various applications can be modeled by combining different elements/aspects of models and/or modeling artefacts.

Formalization: We adapt the formalization of the Diagram Predicate Framework (DPF)[8, 5] by generalizing the components of diagrammatic specifications. DPF uses the concept of diagram predicates to specify constraints on a diagrammatic specification. A DPF predicate signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$ consists of a collection of predicate symbols and a mapping of the predicate symbols to their arity graph. In DPF, a diagrammatic specification $\Im = (G, C^S : \Sigma)$ is given by a graph G and a set of atomic constraints $C^S : \Sigma$. The shape graph G is given by a collection of nodes, edges, source, and target maps. The atomic constraints are added to the graph G by means of graph homomorphism from the shape of a predicate to G, $\delta : \alpha^\Sigma(p) \to G$. In this paper, we generalize the concepts of diagrammatic specification and propose to use composite specification for modeling heterogeneous systems. Composite specification is formalized as a set of specifications, relationship among the specifications, and a set of structural constraints. The specifications and relationships constitute the shape of a composite specification. Similar to DPF, structural constraints are added to the shape of a composite specification by a structure preserving map from the arity of a predicate.

Definition 1 (Composite specification)**.** *Given a predicate signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$, a composite specification $\mathfrak{C}$ is defined recursively as follows:*

- *a DPF specification $\Im = (G, C^S : \Sigma)$ is a composite specification;*
- *let N be a set of composite specifications, then a composite specification is given by*
  - *a set of composite specifications $S \subseteq N$,*
  - *a set of edges between elements of S,*
  - *a set of atomic constraints on the shape of $\mathfrak{C}$ with $p \in P^\Sigma$*

A *Composite specification* describes the structure of a model which allow us to compose modeling elements through substitution. Complex structures may be formulated with composite specifications by specifying structural constraints. In order to be a valid composition, the structural constraints must be satisfied.

We obtain a modeling language $\mathcal{M}_\mathfrak{C}$ for model composition by means of a composite specifications. The semantic of a composite specification is described in a fibred manner i.e., the semantics of a composite specification is given by the set of its instances. Models

are usually underspecified in the early phases and therefore we need to define models with abstract information. Composite specifications are often defined with underspecification such that there is usually not a single system that realizes a model, but a larger set of realizations is allowed. Further refinement of a composite specification is achieved by replacing abstract information with concrete modeling elements. During the early stage software developer often deals with an incomplete or inconsistent model. Model transformation techniques can play an important role in completing an incomplete model and/or repairing an inconsistent model.

In our presentation at the workshop we will present a variety of composition patterns and show how composite specifications can be used for both modeling in the small and modeling in the large. To show the applicability of the proposed method, we will demonstrate an application of model composition for optimizing distributed resource allocations in a healthcare context. Figure 1 illustrates an example of a composite specification where $S0 - S5$ are representing composite specifications. Specification $S0$ is representing a game theory model consisting of three choices of a player $P1$ and two choices of a player $P2$. The choices are linked to software model specifications ($S1 - S5$) representing the resource allocation of the system. For simplicity the detailed specification of the models are not presented in this abstract. We wish to present a formal approach for composing heterogeneous models based on the MDE techniques presented in [7].

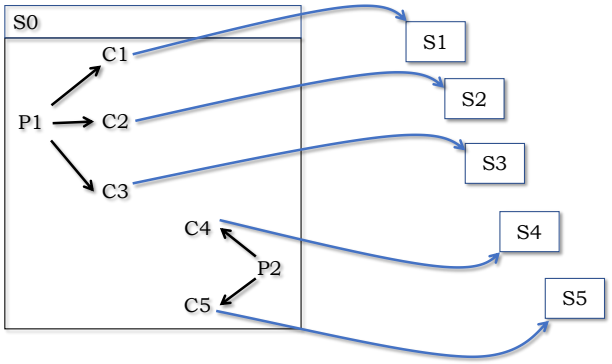


Figure 1: Example of a composite specification

References

[1] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brckner, P. D. Mosses, D. Sannella, and A. Tarlecki. Casl: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153 – 196, 2002.

[2] A. Bagdasaryan. Systems theoretic techniques for modeling, control, and decision support in complex dynamic systems. *CoRR*, abs/1008.0775, 2010.

[3] S. Erdweg, P. G. Giarrusso, and T. Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8, New York, NY, USA, 2012. ACM.

[4] A. Haber, M. Look, A. Navarro Perez, P. Mir Seyed Nazari, B. Rumpe, S. Völkel, and A. Wortmann. Integration of heterogeneous modeling languages via extensible and composable language components. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2015, pages 19–31, Portugal, 2015. SCITEPRESS - Science and Technology Publications, Lda.

[5] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle. *DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment*, pages 37–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[6] T. Mossakowski. Relating casl with other specification languages: The institution level. *Theor. Comput. Sci.*, 286(2):367–475, Sept. 2002.

[7] F. Rabbi. *MDE Techniques for Modeling and Analysis of Complex Software Systems*. PhD thesis, Department of Informatics, University of Oslo, Norway, 2017.

[8] A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

Expressive Power and Encoding of Transition System Models for Software Product Lines

Mahsa Varshosaz[1], Lars Luthmann[3], Malte Lochau[3,4], and Mohammad Reza Mousavi[1,2]

[1] Halmstad University, Halmstad, Sweden
mahsa.varshosaz@hh.se
[2] University of Leicester, Leicester, UK
mm789@leicester.ac.uk
[3] Technische Universität Darmstadt, Darmstadt, Germany
{lars.luthmann,malte.lochau}@es.tu-darmstadt.de
[4] University of Passau, Passau, Germany

Abstract

Several formalisms have been proposed for modeling Software Product Lines (SPLs), such as Modal Transition Systems (MTSs) and their extensions, Feature Transition Systems (FTSs), and Product Line Transition Systems (PL-LTSs). In this talk, we review our past work on comparing the expressive power of these formalisms and providing encodings between them. Namely, we show that FTSs are strictly more expressive than MTSs and are exponentially more succinct than PL-LTS.

As MTSs are well-studied and there are tools for analysis of such models, we move on to find a connection between FTSs and MTSs. To this end, we seek an encoding from FTSs into sets of MTSs, as FTSs are more expressive than individual MTSs. We present initial ideas on an encoding that preserves the behavior of FTS and results in a set of MTSs.

1 Introduction

Software Product Line (SPL) engineering is a commonly used development technique aiming for mass production and customisation of families of software systems. In SPL engineering, a set of software systems, with well-defined commonalities and variabilities, are developed based on a common core and platform. Furthermore, systematic reuse is enabled in different phases of software development by using SPL engineering.

There are different analysis techniques that have been used for quality assurance of SPLs. Some of these analysis techniques such as model checking and model-based testing require a model of the system. Using conventional techniques for modeling individual products in an SPL can be time consuming and costly, as the number of the products can be potentially large. Several formalisms have been extended and introduced for efficient modeling of software product lines. Feature Transition Systems (FTSs) [3], Modal Transition Systems [7] and their extensions [4, 6, 1], and Product Line Calculus of Communicating Systems (PL-CCSs) [5] are among these formalisms that have been used for modeling SPLs. These models can be compared in terms of properties such as expressive power and compactness.

In [2], we have compared the expressive power of three fundamental models used for modeling SPLs, namely, FTSs, MTSs, and PL-CCSs. The results show that FTSs are strictly more expressive than MTSs as persistent choices and exclusive behavior cannot be represented by MTSs.

Modal transition systems have been extensively studied and there are tools supporting analysis of this type of models. Hence, in this work in progress, we are developing a connection between MTSs and FTSs and possibly a translation between them that preserves the behavior.

Modal transition systems are extensions of Labeled transition Systems (LTSs) in which the set of transitions are divided into two sets, namely, *may* and *must* transitions. An MTS is defined as a tuple $(S, A, \rightarrow_{\Box}, \rightarrow_{\Diamond}, s_{init})$, where S is a set of states, s_{init} is the initial state, A is a set of actions, and $\rightarrow_{\Box} \subseteq \rightarrow_{\Diamond} \subseteq S \times A \times S$, where $\rightarrow_{\Box}$ is the set of must transitions, which can be used for representing the mandatory behavior and $\rightarrow_{\Diamond}$ is the set of may transitions, which can be used for representing the optional behavior. Based on the refinement relation given for MTSs in [7], an LTS that refines an MTS, should implement all the must-transitions and also can include some (or none) of the may transitions.

Feature transition systems are also extensions of LTSs used for modding SPLs. An FTS can be defined as a tuple $(S, A, F, \rightarrow, \Lambda)$, where S is a set states, A is a set of actions, F is a set of features, $\rightarrow \subseteq S \times A \times \mathbb{B}(F) \times S$, where $\mathbb{B}(F)$ represents the set of all propositional formulae over a set of variables that represent features in F, is the set of transitions, and Λ represents the set of valid products in the product line. Each transition in an FTS has a presence condition (represented as a propositional formula) that determines in which product models the transition is present. A projection operator is given in [3], for generating valid LTSs that implement an FTS.

The main goal in this work is to present an encoding from FTSs to MTSs such that the behaviour is preserved. As mentioned before, MTSs cannot represent exclusive behavior and also persistent choices. Hence, the result of translation of an FTS is a set of MTSs. By splitting the behavior between different MTSs, the exclusive behavior can be included in separate MTSs.

Assume that $\mathbb{FTS}$, $\mathbb{MTS}$, and $\mathbb{LTS}$, respectively, denote the class of FTSs, MTSs, and LTSs; Assuming that $E : \mathbb{FTS} \rightarrow 2^{\mathbb{MTS}}$, denotes our encoding from FTSs into MTSs. Furthermore, for an arbitrary $mts \in \mathbb{MTS}$, we denote the set of all LTSs implementing mts by $[\![mts]\!]$, and the set of LTSs implementing an arbitrary $fts \in \mathbb{FTS}$ is denoted by $[\![fts]\!]$. One of the properties that is proved for the encoding is as follows:

$$\forall fts \in \mathbb{FTS} \;\; \forall lts \in [\![fts]\!] \cdot \; \exists lts' \in \bigcup_{mts \in E(fts)} [\![mts]\!] \cdot \; lts \underline{\leftrightarrow} lts' \wedge$$

$$\forall fts \in \mathbb{FTS} \;\; \forall \; lts \in \bigcup_{mts \in E(fts)} [\![mts]\!] \cdot \; \exists \; lts' \in [\![fts]\!] \; \cdot \; lts \underline{\leftrightarrow} lts'$$

where $\underline{\leftrightarrow}$ represents bisimilarity relation. Considering an arbitrary FTS *fts*, the above statements mean that for each *lts* that is a valid implementation of *fts*, there is an LTS implementing one of the MTSs resulted from the encoding of *fts* that is bisimilar with *lts* and vice versa.

As a part of our work we are aiming for implementing the encoding and application of this approach on different case studies, in which the behavior of the SPL is represented as an FTS and then generating MTSs that preserve the behavior.

References

[1] Nikola Beneš, Jan Křetínský, Kim G. Larsen, Mikael H. Møller, and Jiří Srba. Parametric modal transition systems. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, pages 275–289, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] H. Beohar, M. Varshosaz, and M. R. Mousavi. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. *Science of Computer Programming*, 2015. In Press, available online.

[3] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.

[4] H. Fecher and H. Schmidt. Comparing disjunctive modal transition systems with an one-selecting variant. *The Journal of Logic and Algebraic Programming*, 77(1-2):20–39, 2008. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006).

[5] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.

[6] K. G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90)*, pages 108–117, Jun 1990.

[7] K.G. Larsen and B. Thomsen. A modal process logic. In *Proc. of the 3rd Annual Symposium on Logic in Computer Science*, pages 203–210. IEEE, 1988.

Abstract refinement algebra: a survey

Kim Solin

Åbo Akademi, Åbo, Finland

Refinement algebras are axiomatic algebras intended for total-correctness reasoning about programs. The most basic refinement algebra, von Wright's demonic refinement algebra [13], is a modification of Kozen's Kleene algebra with tests [5] to accommodate total correctness and, consequently, the refinement calculus [2]. Technically, a refinement algebra is an idempotent semiring with an iteratation operator $(S, \sqcap, ;, ^\omega, \top, 1)$ where $\sqcap$ caputures demonic choice, ; captures sequential composition, $^\omega$ caputures a possibly non-terminating iteration, 1 is skip, and $\top$ is the miraculous program.

The axiomatic framework allows for multiple models, including the motivating model of predicate transformers, and for significant reuse of established results in related algebras. Moreover, the axiomatic framework also easily lends itself to reasoning about angelic choice [9], enabledness and termination [12], probabilistic programs [7, 6], data refinement [11], and concurrency [3]. Algebras of this kind are highly suitable for automated reasoning [1, 4].

In this talk, I survey the background to abstract, axiomatic reasoning about program refinement based on semiring structures, and the main achievements of this line of research. I also discuss a number of research avenues and open problems, including combining semirings for reasoning about epistemic agents [8, 10] with refinement algebra to achieve a framework for reasoning about security.

References

[1] Alasdair Armstrong, Victor B. F. Gomes, and Georg Struth. Kleene algebra with tests and demonic refinement algebras. *Archive of Formal Proofs*, 2014, 2014.

[2] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus - A Systematic Introduction.* Graduate Texts in Computer Science. Springer, 1998.

[3] Ian J. Hayes, Robert J. Colvin, Larissa A. Meinicke, Kirsten Winter, and Andrius Velykis. An algebra of synchronous atomic steps. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 352–369, 2016.

[4] Peter Höfner and Georg Struth. Automated reasoning in Kleene algebra. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2007.

[5] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[6] Larissa Meinicke and Ian Hayes. Probabilistic choice in refinement algebra. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*, volume 5133 of *Lecture Notes in Computer Science*, pages 243–267. Springer, 2008.

[7] Larissa Meinicke and Kim Solin. Refinement algebra for probabilistic programs. *Formal Asp. Comput.*, 22(1):3–31, 2010.

[8] Kim Solin. A sketch of a dynamic epistemic semiring. *Inf. Comput.*, 208(5):594–604, 2010.

[9] Kim Solin. Dual choice and iteration in an abstract algebra of action. *Studia Logica*, 100(3):607–630, 2012.

[10] Kim Solin. Modal semirings with operators for knowledge representation. In Joaquim Filipe and Ana L. N. Fred, editors, *ICAART 2013 - Proceedings of the 5th International Conference on Agents and Artificial Intelligence, Volume 2, Barcelona, Spain, 15-18 February, 2013*, pages 197–202. SciTePress, 2013.

[11] Kim Solin. Encoding and decoding in refinement algebra. In Wolfram Kahl, Michael Winter, and José Nuno Oliveira, editors, *Relational and Algebraic Methods in Computer Science - 15th International Conference, RAMiCS 2015, Braga, Portugal, September 28 - October 1, 2015, Proceedings*, volume 9348 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2015.

[12] Kim Solin and Joakim von Wright. Enabledness and termination in refinement algebra. *Sci. Comput. Program.*, 74(8):654–668, 2009.

[13] Joakim von Wright. Towards a refinement algebra. *Sci. Comput. Program.*, 51(1-2):23–45, 2004.

Towards Domain-Specific CPN Modelling Languages

Alejandro Rodríguez Tena[1], Fernando Macías[1],
Lars Michael Kristensen[1], and Adrian Rutle[1]

Department of Computing, Mathematics, and Physics
Western Norway University of Applied Sciences, Bergen
{arte,fmac,lmkr,aru}@hvl.no

Software systems engineering is a comprehensive discipline involving a multitude of activities such as requirements engineering, design and specification, implementation, testing, and deployment. Model-driven software engineering (MDSE) [4] is one of the emergent responses from the scientific and industrial communities to tackle the increasing complexity of software systems. MDSE utilises abstractions for modelling different aspects – behaviour and structure – of software systems, and treats models as first-class entities in all phases of software development. One way to increase the adoption of MDSE is to develop modelling approaches which reflect the way software architects, developers and designers, as well as organisations, domain experts and stakeholders handle abstraction and problem-solving. In this context, domain-specific (meta)modelling (DSMM) [5] has been proposed as an approach to unite software modelling and abstraction, software design and architecture, and organisational studies. The main aim is to fill the gap between these fields and make modelling more widely applicable than it currently is [15].

The development of distributed software systems is particularly challenging. A major reason is that these systems possess concurrency and non-determinism which means that the execution of such systems may proceed in many different ways. To cope with the complexity of modern concurrent systems, it is therefore crucial to provide methods that enable debugging and testing of central parts of the system design prior to implementation and deployment [7]. Since concurrency, communication and synchronisation are increasingly present in our lives, it is priority to put efforts in improving the current techniques to deal with them. One way to approach the challenge of developing concurrent systems is to build an executable model of the system. Constructing a model and simulating it usually leads to significant insights into the design and operation of the system considered and often results in a simpler and more streamlined design.

Modelling of distributed systems. Coloured Petri Nets (CPNs) form a graphical language designed to construct models of distributed systems i.e. communication protocols [6], data networks [3], distributed algorithms [11] and embedded systems [2]. CPNs combine classical Petri nets [10] with the functional programming language Standard ML [14]. The modelling language is suited for discrete-event processes that include choice, iteration, and concurrent execution. A CPN model of a system is an executable model representing the states of the system and the events (transitions) that can cause the system to change state. The CPN modelling language also makes it possible to organise a model into a hierarchically related set of modules, and it has a time concept to represent the time taken to execute events.

One advantage of CPNs is that they contain few but powerful modelling constructs. This means that the modeller has few constructs that need to be mastered in order to apply the language. However, several recent applications of CPNs [13] have shown that it would be beneficial to be able to develop domain-specific variants that would make it possible to support:

Modelling patterns representing commonly used approaches to capture concepts from the problem domain.

Modelling restrictions forcing the modeller to use only certain constructs in the language when modelling concepts from the problem domain.

Subtyping of elements allowing specific interpretation of certain model elements such as places, transitions, and arcs.

Figure 1 (left) shows an example from the embedded software domain [8] in which substitution transitions and certain places have been subtyped as representing interfaces and events for code generation purpose. Figure 1 (right) shows an example from the control system domain in which modelling patterns are used to consume events (Fig 1(right,top)) and update a process variable (Fig 1(right,bottom)) based on input received from the environment. The patterns in turn put restrictions on the arcs and arc expressions.

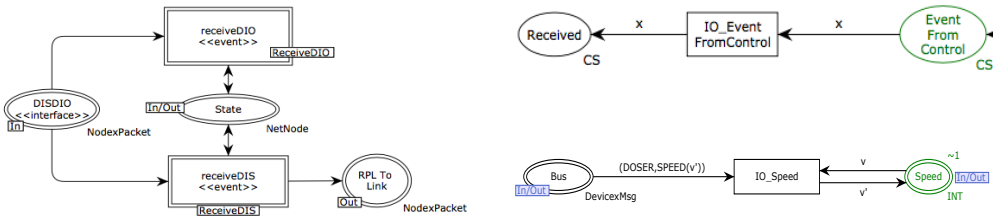


Figure 1: Examples of domain-specific concepts in CPN models.

A Metamodel for Coloured Petri Nets. The lack of extensibility of the CPN language and lack of adaptability provided by CPN Tools have motivated us to develop a model-driven infrastructure for CPN. The first step towards this has been to develop a metamodel for CPN. The definition of this metamodel will support application of model transformations (for definition of model semantics), domain-specific metamodelling (for creation of domain-specific versions of CPN), and abstraction (for definition of modelling patterns and restrictions).

There exists work on metamodels for Petri Nets [1]. These metamodels have been developed for the purpose of tool interoperability for general purpose Petri nets, and not the domain-specific aspects that we aim to address. Figure 2 shows a first attempt to develop a metamodel with the Eclipse Modeling Framework (EMF). The next step is to put this metamodel in a multilevel context using MultEcore [9] to facilitate refinement of concepts from CPN to reflect domain concepts. This metamodel captures all models that can be built using CPN [7]. In addition to the concepts represented by the class model in the figure, we have the following constrains in the metamodel:

- A module cannot be a submodule of itself, i.e., if we follow the associations through substitution transitions and modules then we cannot encounter the same module twice.
- If a port is associated with a socket place, then the socket place must be connected to a substitution transition that has the module to which the port belongs as its submodule.
- Associated port and (socket) places must have identical colour sets, and if the port place has an initial marking it must be equal to the associated socket place.

These constrains can be expressed using the Object Constraints Language (OCL). However, in order to obtain a more uniform metamodel, we are currently investigating how they can be expressed directly using formal diagrammatic notations in DPF [12].

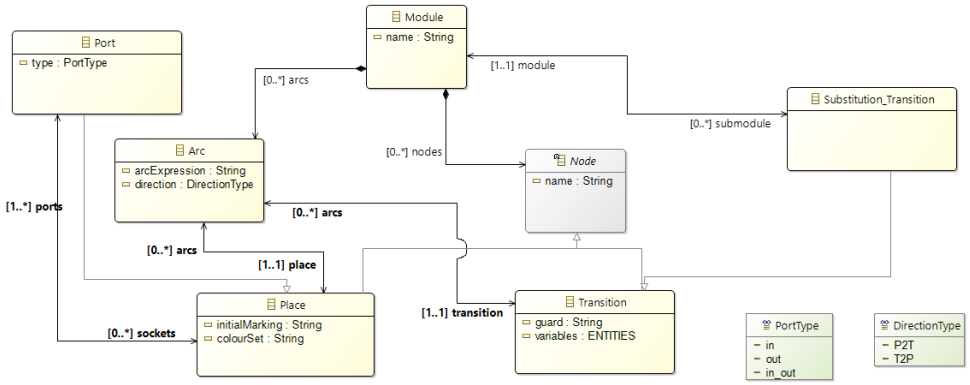


Figure 2: Metamodel for Colored Petri Nets

References

[1] Petri nets markup language. http://www.pnml.org/papers.php.

[2] M. A. Adamski, A. Karatkevich, and M. Wegrzyn. *Design of embedded control systems*, volume 267. Springer, 2005.

[3] J. Billington and M. Diaz. *Application of Petri nets to communication networks: Advances in Petri nets*. Number 1605. Springer Science & Business Media, 1999.

[4] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.

[5] J. de Lara, E. Guerra, and J. Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, 14(1):429–459, 2015.

[6] J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3018 of *Lecture Notes in Computer Science*. Springer, 2004.

[7] K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, Jun 2007.

[8] L. Kristensen and V. Veiset. Transforming cpn models into code for tinyos: A case study of the rpl protocol. In *Proc. of ICATPN'16*, volume 9698 of *LNCS*, pages 135–154. Springer, 2016.

[9] F. Macías, A. Rutle, and V. Stolz. MultEcore: Combining the best of fixed-level and multilevel metamodelling. In *MULTI*, volume 1722 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[10] W. Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012.

[11] W. Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer Science & Business Media, 2013.

[12] A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

[13] K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Pragmatics annotated coloured petri nets for protocol software generation and verification. *TopNoC*, 11:1–27, 2016.

[14] J. D. Ullman. *Elements of ML programming*. Prentice-Hall, Inc., 1994.

[15] J. Whittle, J. E. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Software*, 31(3):79–85, 2014.

RFun Revisited

Robin Kaarsgaard and Michael Kirkedal Thomsen

DIKU, Department of Computer Science, University of Copenhagen
{robin, m.kirkedal}@di.ku.dk

We describe here the steps taken in the further development of the reversible functional programming language RFun. Originally, RFun was designed as a first-order untyped language that could only manipulate constructor terms; later it was also extended with restricted support for function pointers [6, 5]. We outline some of the significant updates to the language, including a static type system based on relevant typing, with special support for ancilla (read-only) variables added through an unrestricted fragment. This has further resulted in a complete makeover of the syntax, moving towards a more modern, Haskell-like language.

Background In the study of reversible computation, one investigates computational models in which individual computation steps can be uniquely and unambiguously inverted. For programming languages, this means languages in which programs can be run *backward* and get a unique result (the exact input). Though the field is often motivated by a desire for energy and entropy preservation though the work of Landauer [3], we are more interested in the possibility to use reversibility as a property that can aid in the execution of a system; an approach which can be credited to Huffman [1]. In this paper we specifically consider RFun. Another notable example of a reversible functional language is Theseus [2], which has also served as a source of inspiration for some of the developments described here.

Ancillae Ancillae (considered ancillary variables in this context) is a term adopted from physics to describe a state in which entropy is unchanged. Here we specifically use it for variables for which we can guarantee that their values are unchanged over a function call. We cannot put too little emphasis on the guarantee, because we have taken a conservative approach and will only use it when we statically can ensure that it is upheld.

1 RFun version 2

In this section, we will describe the most interesting new additions to RFun and how they differ from the original work. Rather than showing the full formalisation, we will instead argue for their benefits to a reversible (functional) language.

Figure 1 shows an implementation of the Fibonacci function in RFun, which we will use as a running example. Since the Fibonacci function is not injective (the first and second Fibonacci numbers are both 1), we instead compute *Fibonacci pairs*, which *are* unique. Hence, the first Fibonacci pair is $(0, 1)$, the second to $(1, 1)$, third $(2, 1)$, and so forth.

The implementation in RFun can be found in Figure 1 and consists of a type definition `Nat` and two functions `plus` and `fib`. Here, `Nat` defines the natural numbers as Peano numbers, `plus` implements addition over the defined natural numbers, while `fib` is the implementation of the Fibonacci pairfunction. Further, Figure 2 shows an implementation of the map function.

1.1 Type system

With Milner's motto that "well-typed programs cannot go wrong," type systems have proven immensely successful in guaranteeing fundamental well-behavedness properties of programs. In

```
data Nat = Z | S Nat
plus :: Nat → Nat ↔ Nat
plus Z     x = x
plus (S y) x =
  let x' = plus y x
   in (S x')
```

Figure 1: RFun program computing Fibonacci pairs.

```
fib :: Nat ↔ (Nat, Nat)
fib Z     = ((S Z), Z)
fib (S m) =
  let (x, y) = fib m
      y'     = plus x y
  in (y', x)
```

```
map :: (a ↔ b) → [a] ↔ [b]
map fun [ ] = [ ]
map fun (l:ls) =
  let l'  = fun l
      ls' = map fun ls
  in (l': ls')
```

Figure 2: Map function in RFun.

reversible functional programming, linear type systems (see, *e.g.*, [2]) have played an important role in ensuring reversibility.

Fundamentally, a reversible computation can be considered as an injective transformation of a state into an updated state. In this view, it seems obvious to let the type system guarantee linearity, *i.e.*, that each available resource (in this case, variable) is used exactly once. Though linearity is not enough to guarantee reversibility, it enables the type system to statically reject certain irreversible operations (*e.g.*, projections). However, linearity is also more restrictive than needed: if we accept that functions may be partial (a necessity for r-Turing completeness), first-order data *can* be duplicated reversibly. For this reason, we may relax the linearity constraint to *relevance*, *i.e.*, that all available variables must be used *at least* once. This guarantees that values are never lost, while also enabling implicit duplication of values.

A useful concept in reversible programming is access to ancillae, *i.e.*, values that remain unchanged across function calls. Such values are often used as a means to guarantee reversibility in a straightforward manner. For example, in Figure 1, the first input variable of the `plus` function is ancillary; it's value is tacitly returned automatically as part of the output. To support such ancillary variables at the type level, a type system inspired by Polakow's combined reasoning system of ordered, linear, and unrestricted intuitionistic logic [4] is used. The type system splits the typing contexts into two parts: a static one (containing ancillary variables and other static parts of the environment), and a dynamic one (containing variables not considered ancillary). This gives a typing judgment of $\Sigma; \Gamma \vdash e : \tau$, where Σ is the static context, and Γ the dynamic one.

Whereas we must ensure that variables in the dynamic context Γ are used in a relevant manner to guarantee reversibility, there are no restrictions on the use of variables in the static context – these can used as many or as few times (including not at all) as desired. To distinguish between ancillary and dynamic variables at the type level, two different arrow types are used: $t_1 \to t_2$ denotes that the input variable is ancillary, whereas $t_1 \leftrightarrow t_2$ denotes that it is dynamic. As such, the type of `plus` in Figure 1 signifies that the first input variable is ancillary, and the second is dynamic.

A neat use of ancillae is to provide limited support for behaviour similar to higher-order functions. For example, the usual `map` function is not reversible, as not every function of type `[a] ↔ [b]` arises as a functorial application of some function of type `a ↔ b`. However, if we consider the input function `f :: a ↔ b` to be ancillary, one can straightforwardly define a reversible `map` function (see Figure 2) as one of type `(a ↔ b) → ([a] ↔ [b])`. In this way, ancillae can be considered as a slight generalization of the *parametrized maps* found in Theseus [2].

1.2 Duplication/Equality

In the first version of RFun, duplication and equality was included as a special operator, which could perform deep copying or uncopying reversibly, depending on the usage. However, we have

found that understanding the semantics this operator often poses a problem for programmers.

To remedy this, we propose to use type classes, and implement equality instead using a type class similar to the `EQ` type class found in Haskell. As in Haskell, the functions needed to be member of this class can often be automatically derived.

1.3 First-match policy

The first-match policy (FMP) is essential to ensuring injectivity of individual functions. It states that a returned value of a function *must not* match any previous leaf of the function; this can be compared to checking the validity of an assertion on exit.

In the first version of RFun, the check to ensure that the first-match policy was upheld was always performed at run-time, and, thus, posed a limitation to the performance. However, with the type system, it will now often be possible to perform this check statically, as the types of the leaves or even the ancillae inputs can be orthogonal. *E.g.* in the `plus` function (in Figure 1), the use of ancillae input ensures that the FMP is always upheld, while `map` (in Figure 2) is reversible by orthogonality of the leaves. Unfortunately, this cannot always be guaranteed statically, and the `fib` function (in Figure 1) is an example where a runtime check is still required.

1.4 Conclusion

In this paper we have outlined the future development of the reversible function language RFun. A central element of this is the development of the type system. However, it is important to note that type correctness is a sufficient condition for reversibility *only* because we assume partiality. Thus, it might be that case that the domain (and range) of the function is very small or even empty. It is valid to consider how useful a reversible function with an empty domain actually is.

The work shows both an interesting new application of relevant type systems, and gives RFun a more modern design that will make it easier for programmers to understand.

Acknowledgements This work was partly supported by the European COST Action IC 1405: Reversible Computation - Extending Horizons of Computing.

References

[1] D. A. Huffman. Canonical forms for information-lossless finite-state logical machines. *IRE Transactions on Information Theory*, 5(5):41–59, 1959.

[2] R. P. James and A. Sabry. Theseus: A high level language for reversible computing. Work in progress paper at RC 2014. Available at `www.cs.indiana.edu/~sabry/papers/theseus.pdf`, 2014.

[3] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[4] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001.

[5] M. K. Thomsen and H. B. Axelsen. Interpretation and programming of the reversible functional language. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, pages 8:1–8:13. ACM, 2016.

[6] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In A. De Vos and R. Wille, editors, *Reversible Computation, RC '11*, volume 7165 of *LNCS*, pages 14–29. Springer-Verlag, 2012.

Turku
Centre *for*
Computer
Science

http://www. tucs.fi

tucs@abo.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Faculty of Science and Engineering
- Computer Engineering
- Computer Science

Faculty of Social Sciences, Business and Economics
- Information Systems

ISBN 978-952-12-3608-2
ISSN 1797-8831

Marina Waldén (Editor)

Proceedings of the 29th Nordic Workshop on Programming Theory