Report from Dagstuhl Seminar 17462

A Shared Challenge in Behavioural Specification

Edited by

Klaus Havelund[1], Martin Leucker[2], Giles Reger[3], and Volker Stolz[4]

1 **Jet Propulsion Laboratory, US,** klaus.havelund@jpl.nasa.gov
2 **Universität Lübeck, DE,** leucker@isp.uni-luebeck.de
3 **University of Manchester, GB,** giles.reger@manchester.ac.uk
4 **West. Norway Univ. of Applied Sciences – Bergen, NO,** vsto@hvl.no

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 17462 "A Shared Challenge in Behavioural Specification". The seminar considered the issue of behavioral specification with a focus on its usage in Runtime Verification. The seminar was motivated by the observations that, whilst the field of Runtime Verification is becoming more mature, there is a lack of common specification language, in the main part due to the rich setting allowing for highly expressive languages. The aim of the Seminar was to shed light on the similarities and differences between the different existing languages, and specifically, suggest directions for future collaboration and research. The seminar consisted of two talk sessions, two working group sessions, and a feedback and reflection session. Working group topics were suggested and agreed in response to points raised in talks. One significant outcome was the proposal of a shared challenge project in which different Runtime Verification approaches can be compared, as outlined in one of the working group reports.

Seminar November 12–15, 2017 – http://www.dagstuhl.de/17462
1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.3 Formal Languages
Keywords and phrases behavioural specification, dynamic properties, runtime verification, temporal logic
Digital Object Identifier 10.4230/DagRep.7.11.59

1 Executive Summary

Giles Reger
Klaus Havelund
Martin Leucker
Volker Stolz

This seminar dealt with the issue of behavioural specification from the viewpoint of runtime verification. Runtime verification (RV) as a field is broadly defined as focusing on processing execution traces (output of an observed system) for verification and validation purposes. Of particular interest is the problem of verifying that a sequence of events, a trace, satisfies a temporal property, formulated in a suitable formalism. Examples of such formalisms include state machines, regular expressions, temporal logics, context-free grammars, variations of the mu-calculus, rule systems, stream processing systems, and process algebras. Of special

A Shared Challenge in Behavioural Specification, *Dagstuhl Reports*, Vol. 7, Issue 11, pp. 59–85
Editors: Klaus Havelund, Martin Leucker, Giles Reger, and Volker Stolz

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

interest is how to specify data-rich systems, where events themselves carry data. Applications cover such domains as security monitoring and safety monitoring.

Such techniques are characterised by highly expressive languages for specifying behaviour, enabled by the concreteness of dealing directly with single runtime traces, which makes the verification problem tractable. However, this permitted expressiveness has also led to a divergence in such languages. The aim of this Dagstuhl Seminar was to shed light on the similarities and differences between these different formalisms, and specifically, suggest directions for future collaboration and research. This effort can potentially lead to an attempt to standardize an RV formalism.

The seminar included a mixture of tool developers, theoreticians, and industry experts and the above aim was addressed by two main activities.

The first activity was that each tool developer was asked to produce a brief summary of their specification language in the form of a set of short examples. These were then presented as talks during the Seminar, alongside other general contributed talks on issues surrounding behavioural specification. The examples were uploaded to a shared repository (which will be available via runtime-verification.org) and eleven participants added their tool descriptions and examples to this repository, producing a lasting resource from the seminar.

The second activity was carried out through eight working groups formed during the Seminar to discuss topics raised by the talks. The results of this working groups are detailed in this report. We take this opportunity to detail the topics (in the form of questions) proposed during the seminar that were not chosen for discussion in working groups:

- *Where should we get specifications from?* This question addressed both the issue of designing specification languages that can be usable by engineers but also the trending topic of inferring specifications from various artifacts and how specification languages can support this.
- *How can we measure specification quality?* What is a good specification, or when is one specification better than another? This might be related to coverage of the system being specified, or might be about interpretability or some other measure of usability.
- *How do we ensure our specification language is not broken?* This question was inspired by the experience of one speaker with developing the industrial-strength PSL language and the issues surrounding getting it right.
- *How can we balance different levels of abstraction (e.g. local and global behaviour) in a specification?* It was noted that specification languages are often closely associated with specifications at a certain level of abstraction. Is this an inherent restriction or a positive feature? Should we build specification languages with a certain level of abstraction in mind?
- *How do we unify the different uses of a specification?* This was inspired by the observation that a specification may be used to explain behaviour, check behaviour, or synthesize behaviour, and different presentations may be preferred in these different contexts.

This seminar was the first time the runtime verification community has reflected on the broad issue of specification and has fed into further developments including new perspectives for the international runtime verification competition, a proposed shared challenge involving the NASA core flight system, and the first informal survey and categorisation of actively developed runtime verification tools.

2 Table of Contents

Working groups

3 Overview of Talks

3.1 Trace Focussed and Data Focussed Specification: Complementary, Competing, Combined?

Wolfgang Ahrendt (Chalmers University of Technology – Göteborg, SE)

5 Years ago, I participated in Dagstuhl seminar *Divide and Conquer: the Quest for Compositional Design and Analysis*. In effect, the seminar could have been named *Model Checking meets Deductive Verification*. It was a very interesting seminar, but we had some difficulties to identify the types of *properties*, and with them specification formalisms, which both communities are interested in, or can cope with using their respective technologies.

Property languages are often technology driven, and so are properties themselves. To analyse one system with *different methods*, we end up using *different fomalisms*, specifying *disconnected views*.

In Runtime Verification, as well as in Model Checking, there is a strong focus on traces, often traces of events of some kind. In Deductive Versification, as well as in Runtime Assertion Checking, the focus on properties of the data, at specific points in the execution. Is the difference really motivated by what either communities consider important system properties, or rather by what the respective technologies are good at checking? To which extent should specification formalisms make a pre-choice?

In my talk, I suggest the following community effort:

- Integrated/coordinated specification of trace *and* data focused aspects,
- Front-ends mapping divers aspects of the specification to tool/method-oriented formats,
- Delegation of sub-tasks to appropriate tools
- Delegation of sub-tasks to static or dynamic analysis
- Integration of the results from diverse analyses

I mentioned ppDATE [1] as one specific attempt to combine multiple aspects on the specification level. With that, I hope to trigger a more general discussion about the role and integration of trace focused and data focused specification.

References

1 Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design*, Apr 2017.

3.2 Domain-Specific Languages with Scala, and model-based testing as an example

Cyrille Artho (KTH Royal Institute of Technology – Stockholm, SE)

Joint work of Cyrille Artho, Klaus Havelund, Rahul Kumar, Yoriyuki Yamagata
Main reference Cyrille Artho, Klaus Havelund, Rahul Kumar, Yoriyuki Yamagata: "Domain-Specific Languages with Scala", in Proc. of the Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings, Lecture Notes in Computer Science, Vol. 9407, pp. 1–16, Springer, 2015.
URL https://doi.org/10.1007/978-3-319-25423-4_1

Domain-Specific Languages (DSLs) are often classified into external and internal DSLs. An external DSL is a stand-alone language with its own parser. An internal DSL is an extension of an existing programming language, the host language, offering the user of the DSL domain-specific constructs as well as the constructs of the host language.

This presentation gives a brief overview of the concepts and also looks at an internal DSL used for model-based testing with the tool "Modbat".

3.3 The Tale of Dr Jekyll and Mr Hyde in Pattern-based Specification Languages

Domenico Bianculli (University of Luxembourg, LU)

Joint work of Marcello Maria Bersani, Lionel Briand, Wei Dou, Carlo Ghezzi, Srdan Krstic, Pierluigi San Pietro
Main reference Domenico Bianculli, Carlo Ghezzi, Pierluigi San Pietro: "The Tale of SOLOIST: A Specification Language for Service Compositions Interactions", in Proc. of the Formal Aspects of Component Software, 9th International Symposium, FACS 2012, Mountain View, CA, USA, September 12-14, 2012. Revised Selected Papers, Lecture Notes in Computer Science, Vol. 7684, pp. 55–72, Springer, 2012.
URL http://dx.doi.org/10.1007/978-3-642-35861-6_4
Main reference Wei Dou, Domenico Bianculli, Lionel C. Briand: "A Model-Driven Approach to Trace Checking of Pattern-Based Temporal Properties", in Proc. of the 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017, pp. 323–333, IEEE Computer Society, 2017.
URL http://dx.doi.org/10.1109/MODELS.2017.9

This talk presents two specification languages, SOLOIST [6] and TemPsy [8, 7]. Both are based on property specification patterns [5, 9] and have been defined in the context of an industrial collaboration.

SOLOIST (*SpecificatiOn Language fOr servIce compoSitions inTeractions*) is a metric temporal logic with new, additional temporal modalities that support aggregate operations on events occurring in a given time window; it can be used to specify both functional and quality-of-service requirements of the interactions of a composite service with its partner services. The trace checking algorithms proposed for SOLOIST rely on the use of SMT solvers [4, 1] and of distributed and parallel computing frameworks [2, 3].

TemPsy (Temporal Properties made easy) is a pattern-based, domain-specific specification language for temporal properties. Its goal is to make as easy as possible the specification of the temporal requirements of business processes, by supporting only the constructs needed to express temporal requirements commonly found in business process applications. TemPsy comes with an efficient trace checking algorithm [8] which relies on a mapping of temporal requirements written in TemPsy into Object Constraint Language (OCL) constraints on a conceptual model of execution traces.

References

1 Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srdan Krstić, and Pierluigi San Pietro. SMT-based checking of SOLOIST over sparse traces. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014), Grenoble, France*, volume 8411 of *Lecture Notes in Computer Science*, pages 276–290. Springer, April 2014.

2 Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srdan Krstić, and Pierluigi San Pietro. Efficient large-scale trace checking using MapReduce. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), Austin, TX, USA*, pages 888–898. ACM, May 2016.

3 Domenico Bianculli, Carlo Ghezzi, and Srdan Krstić. Trace checking of metric temporal logic with aggregating modalities using MapReduce. In *Proceedings of the 12th International Conference on Software Engineering and Formal Methods (SEFM 2014), Grenoble, France*, volume 8702 of *Lecture Notes in Computer Science*, pages 144–158. Springer, September 2014.

4 Domenico Bianculli, Carlo Ghezzi, Srdan Krstić, and Pierluigi San Pietro. Offline trace checking of quantitative properties of service-based applications. In *Proceedings of the 7th International Conference on Service Oriented Computing and Application (SOCA 2014), Matsue, Japan*, pages 9–16. IEEE, November 2014.

5 Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zürich, Switzerland*, pages 968–976. IEEE Computer Society Press, June 2012.

6 Domenico Bianculli, Carlo Ghezzi, and Pierluigi San Pietro. The tale of SOLOIST: a specification language for service compositions interactions. In *Proceedings of the 9th International Symposium on Formal Aspects of Component Software (FACS'12), Mountain View, CA, USA*, volume 7684 of *Lecture Notes in Computer Science*, pages 55–72. Springer, September 2012.

7 Wei Dou, Domenico Bianculli, and Lionel Briand. OCLR: a more expressive, pattern-based temporal extension of OCL. In *Proceedings of the 2014 European Conference on Modelling Foundations and Applications (ECMFA 2014), York, United Kingdom*, volume 8569 of *Lecture Notes in Computer Science*, pages 51–66. Springer, July 2014.

8 Wei Dou, Domenico Bianculli, and Lionel Briand. A model-driven approach to trace checking of pattern-based temporal properties. In *Proceedings of the 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, USA*. IEEE, September 2017.

9 Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Proc. ICSE 1999*, pages 411–420, New York, NY, USA, 1999. ACM.

3.4 Asynchronous HyperLTL

Borzoo Bonakdarpour (McMaster University – Hamilton, CA)

HyperLTL is a temporal logic for expressing a subclass of hyperproperties. It allows explicit quantification over traces and inter-trace Boolean relations among traces. The current semantics of HyperLTL evaluate formula by progressing a set of traces in a lock-step

synchronous manner. In this talk, we will present our recent work on relaxing the semantics of HyperLTL to allow traces to advance with different speeds. While this relaxation makes the verification problem undecidable, the decidable fragment is expressive enough to express most commonly used security policies. Our new semantics has also application in model-based runtime monitoring.

3.5 PSL: The good, the bad and the ugly

Cindy Eisner (IBM – Haifa, IL)

Joint work of Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, Johan Mårtensson, David Van Campenhout

For a specification language to be suitable for formal verification by model checking, it must have sufficient expressive power, its semantics must be formally defined in a rigorous manner, and the complexity of model checking it must be well understood and reasonable. In order to allow widespread adoption in the industry, there is an additional requirement: behavioral specification must be made easy, allowing common properties to be expressed intuitively and succinctly. But while adding syntax is simple, defining semantics without breaking important properties of the existing semantics is surprisingly difficult. In this talk I will discuss various extensions to temporal logic incorporated by PSL, their motivation, and the subtle semantic issues encountered in their definition. I will emphasize where we succeeded, where we were less successful, and point out some features that are still missing.

References

1 C. Eisner and D. Fisman. Augmenting a regular expression-based temporal logic with local variables. In A. Cimatti and R. B. Jones, editors, Formal Methods in Computer Aided Design (FMCAD), pages 1–8. IEEE, 2008.
2 C. Eisner and D. Fisman. Structural contradictions. In H. Chockler and A. J. Hu, editors, Intl. Haifa Verification Conference (HVC), volume 5394 of Lecture Notes in Computer Science, pages 164–178. Springer, 2008.
3 C. Eisner, D. Fisman, and J. Havlicek. A topological characterization of weakness. In M. K. Aguilera and J. Aspnes, editors, Symp. on Principles of Distributed Computing (PODC), pages 1–8. ACM, 2005.
4 C. Eisner, D. Fisman, and J. Havlicek. Safety and liveness, weakness and strength, and the underlying topological relations. ACM Trans. Comput. Log., 15(2), 2014.
5 C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In W. A. H. Jr. and F. Somenzi, editors, Intl. Conf. on Computer-Aided Verification (CAV), volume 2725 of Lecture Notes in Computer Science, pages 27–39. Springer, 2003.
6 C. Eisner, D. Fisman, J. Havlicek, and J. Mårtensson. The top,bot approach for truncated semantics.Technical Report 2006.01, Accellera, May 2006.
7 C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout. The definition of a temporal clock operator. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J.Woeginger, editors, Intl. Coll. on Automata, Languages and Programming (ICALP), volume 2719 of Lecture Notes in Computer Science, pages 857–870. Springer, 2003.

3.6 Two to Tango: A pair of specification languages for runtime monitoring

Adrian Francalanza (University of Malta – Msida, MT)

The choice of a specification language is ultimately determined by its intended use. In this talk we motivate the need to employ two specification languages to be able to study the problem of monitorability. In particular, we will outline why we chose a variant of the modal-mu calculus on the one hand, and a process calculus on the other to understand formally what can and cannot be monitored for at runtime. We will also overview how the choice of two formalisms can be used to asses the correctness of a monitor that is entrusted with checking the execution of a system against a specification.

3.7 A "Do-It-Yourself" Specification Language With BeepBeep 3

Sylvain Hallé (University of Quebec at Chicoutimi, CA)

Main reference Sylvain Hallé: "When RV Meets CEP", in Proc. of the Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings, Lecture Notes in Computer Science, Vol. 10012, pp. 68–91, Springer, 2016.
URL https://doi.org/10.1007/978-3-319-46982-9_6

BeepBeep is an event stream processing engine [1]. In BeepBeep, streams of events (of any kind) are piped into computing units called *processors*. The output of processors can be used as the input of other processors, forming potentially complex processor chains.

BeepBeep is organized along a modular architecture. The main part of BeepBeep is called the *engine*, which provides the basic classes for creating processors and functions, and contains a handful of general-purpose processors for manipulating traces. The rest of BeepBeep's functionalities is dispersed across a number of optional *palettes*.

BeepBeep provides multiple ways to create processor pipes and to fetch their results. A first obvious way is programmatically, using BeepBeep as a library and Java as the glue code for creating the processors and connecting them. In addition to directly instantiating and connecting processors, BeepBeep also offers the possibility to create domain-specific languages with subsets of all the available processors. To this end, BeepBeep provides a parser, which can be given a BNF grammar at runtime, and create a parse tree from any string. With the help of a grammar file and a custom-made "expression parser", one can hence create, in a few lines of code, a domain-specific language with an arbitary syntax, and a parser that converts an expression of this language into a BeepBeep processor chain.

References

1 S. Hallé. "When RV Meets CEP". Proc. RV 2016, Springer Lecture Notes in Computer Science 10012, 68-91, 2016.

3.8 Linking Heterogeneous Models for Intelligent-Cyber-Physical Systems

Zhiming Liu (Southwest University – Chongqing, CN)

Compared to the challenges in traditional ICT applications that engineers used to face, CPS systems and their software development are to, based on the infrastructures of existing systems, develop new components or subsystems, new applications and front end services and to integrate them onto the existing systems. Such development and integration have to deal with the complexity of ever evolving architectures digital components, physical components, together with sensors and smart devices controlled and coordinated by software. The architectural components are designed with different technologies, run on different platforms and interact through different communication technologies. Software components run in these systems for data processing and analytics, computation, and intelligent control. The requirements and environment of a CPS components keep changing with significant uncertainty. Thus, a CPS must contain dynamic monitors and adapters. In this talk we intend to discuss the need of a foundation for the combination of traditional software engineering and AI (or knowledge-based engineering) and the emerging Big Data technologies. We propose research problems including monitoring AI (including learning systems) components, end-to-end specification of composition learning and non-learning components, and a unifying modeling theory to link the different modeling paradigms of non-learning software components and learning software components. We believe that the unified modeling framework need to combine models data functionality, interaction protocols, and timing in both declarative and operational languages, but yet it has to support separation of different design and verification concerns.

3.9 Specification Languages for CPS

Dejan Nickovic (AIT Austrian Institute of Technology – Wien, AT)

Continuous and hybrid behaviors naturally arise from cyber-physical systems (CPS). In this talk, we will present a brief overview of the specification languages that were designed to tackle CPS-specific properties. We will mainly focus on Signal Temporal Logic (STL) and Timed Regular Expressions (TRE), but will also present their syntactic and semantic extensions. We will discuss what are the strength and weaknesses of these languages and in which situations they should or should not be used.

3.10 Automaton-Based Formalisms for Runtime Verification

Gordon Pace (University of Malta – Msida, MT)

An open question is the appropriateness of logic-based vs. visual, graph-based formalisms to be used as specification languages. Clearly there are different ways in which one may measure appropriateness, ranging from ease of writing, ease of comprehension, maintainability of specifications, efficiency of verification and conciseness to mention but a few. Over the past years, we have used graph-based formalisms in various projects and domains, with a particular focus on their use in runtime verification. The formalisms used range from DATEs (the formalism used by the Larva runtime verification tool), ppDATEs (an extension of DATEs used by StaRVOOrs static+dynamic analysis tool), contract automata (used to formalise contracts, including obligations, prohibitions and permissions) and policy automata (used to formalise social network privacy policies). The primary drive towards using these formalisms was the ease of adoption from industrial partners, and correspondence to models typically documented (or not) of the lifecycles of entities in such systems. We present these formalisms and issues arising from their use, and go on to discuss formalisms lying above and below this automaton-based level of abstraction – outlining our experience with controlled natural languages (above) and guarded-commands (below).

3.11 What is parametric trace slicing good for?

Giles Reger (University of Manchester, GB)

Parametric trace slicing [2, 6] is an approach for parametric runtime monitoring that was introduced by tracematches and JavaMOP [1] and later extended by QEA [3, 4, 5]. In this talk I will briefly discuss what it is good for and, perhaps more interesting, what it is not good for. I argue that this approach is very efficient where we have a few quantified variables and care about the cross-product of their domains (such situations arise reasonably often when reasoning about API usage). I argue that language based on this approach tend to be less intuitive and 'non-local' (i.e. you always need to consider the whole specification when considering each part). Additionally, specifications tend not to be composable.

References

1 Meredith, P.O., Jin, D., Griffith, D. et al. Int J Softw Tools Technol Transfer (2012) 14: 249.
2 Chen F., Roşu G. (2009) Parametric Trace Slicing and Monitoring. In: Kowalewski S., Philippou A. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2009. Lecture Notes in Computer Science, vol 5505. Springer, Berlin, Heidelberg
3 Barringer H., Falcone Y., Havelund K., Reger G., Rydeheard D. (2012) Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou D., Méry D. (eds) FM 2012: Formal Methods. FM 2012. Lecture Notes in Computer Science, vol 7436. Springer, Berlin, Heidelberg

4 Havelund K., Reger G. (2015) Specification of Parametric Monitors. In: Drechsler R., Kühne U. (eds) Formal Modeling and Verification of Cyber-Physical Systems. Springer Vieweg, Wiesbaden
5 Reger G., Cruz H.C., Rydeheard D. (2015) MarQ: Monitoring at Runtime with QEA. In: Baier C., Tinelli C. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2015. Lecture Notes in Computer Science, vol 9035. Springer, Berlin, Heidelberg
6 K. Havelund and G. Reger and E. Zalinescu and D. Thoma Monitoring Events that Carry Data, In E. Bartocci and Y. Falcone, editors, Lectures on Runtime Verification – Introductory and Advanced Topics, volume 10457 of *LNCS*, pages 60–97, Springer, 2018

3.12 Specification: The Biggest Bottleneck in Formal Methods and Autonomy

Kristin Yvonne Rozier (Iowa State University, US)

Advancement of increasingly AI-enhanced control in autonomous systems stands on the shoulders of formal methods, which make possible the rigorous safety analysis autonomous systems require. Formal methods are highly dependent on the specifications over which they reason; not only are specifications required for analysis, but there is no escaping the "garbage in, garbage out" reality. Correct, covering, and formal specifications are thus an essential element for enabling autonomy. However, specification is difficult, unglamorous, and arguably the biggest bottleneck facing verification and validation of aerospace, and other, autonomous systems. This talk will examine the outlook for the practice of formal specification, and highlight the on-going challenges of specification, from design-time to runtime system health management. We will pose challenge questions for specification that will shape both the future of formal methods, and our ability to more automatically verify and validate autonomous systems of greater variety and scale.

3.13 TeSSLa: A Real-Time Specification Language for Runtime Verification of Non-synchronized Streams

Torben Scheffel (Universität Lübeck, DE)

Joint work of Torben Scheffel, Sebastian Hungerecker, Martin Leucker, Malte Schmitz, Daniel Thoma

The Temporal Stream-based Specification Language (TeSSLa) operates on non-synchronized real-time streams. It was first created for specifying properties about programs running on multi core systems and it is currently used and developed in the COEMS project.

TeSSLa can express a lot of different properties like real time properties, reasoning about sequential executions orders, calculating and comparing statistics and more. From the beginning on, TeSSLa was built in a way that the generated monitors can be synthesized and executed in hardware (more concrete: FPGAs) such that we are able to still process even huge amounts of data online by exploiting the high parallelism of the hardware.

Furthermore, the goal was that industry software engineers are able to understand and write TeSSLa specifications easily. Hence, TeSSLa is equipped with a strong macro system such that we are able to define a huge standard library on top of the basic functions so software engineers can use TeSSLa. It is also easy to define new functions based on existing ones if needed. Besides using TeSSLa for hardware supported monitoring, it is also feasible to use TeSSLa for creating software monitors, which might reason about objects.

This talk shows the basic motivation for TeSSLa, the basic operators of TeSSLa, application areas and examples.

3.14 E-ACSL, an Executable Behavioural Interface Specification Language for C Programs

Julien Signoles (CEA LIST – Gif-sur-Yvette, FR)

This talk introduces E-ACSL, a behavioral specification language for C programs. It is based on a typed first order logic whose terms are pure C expressions extended with a few specific keywords. Every construct may be executed at runtime. Among others, it provides assertions, contracts, invariants, data dependencies and ghost code. It is powerful enough to express most functional properties of C programs and encode other properties such as LTL properties and information flow policies.

References

1 E-ACSL manual available at http://www.frama-c.com/download/e-acsl/e-acsl.pdf
2 Mickaël Delahaye and Nikolaï Kosmatov and Julien Signoles. Common Specification Language for Static and Dynamic Analysis of C Programs. In Symposium on Applied Computing SAC'13, pages 1230–1235, 2013, ACM.

3.15 LOLA

Hazem Torfah (Universität des Saarlandes, DE)

LOLA is a specification language and stream processing engine for monitoring temporal properties and computing complex statistical measurements. Lola combines the ease-of-use of rule-based specification languages with the expressive power of heavy-weight scripting languages or temporal logics previously needed for the description of complex stateful dependencies. The language comes with two key features: template stream expressions, which allow parameterization with data, and dynamic stream generation, where new properties can be monitored on their own time scale. We give an overview on the development and the current state of our tool in addition to a series of applications.

3.16 Metric First-Order Dynamic Logic and Beyond

Dmitriy Traytel (ETH Zürich, CH)

I present Metric First-Order Dynamic Logic (MFODL), the "supremum" of the specification languages Metric First-Order Temporal Logic (MFOTL) [4] and Metric Dynamic Logic (MDL) [2] used in the MonPoly [1] and Aerial [3] monitoring tools. Moreover, I discuss a few missing features of MFODL, which can be useful in applications: context-free or context-sensitive temporal dependencies, aggregations, and absolute time references.

References

1 D. Basin, F. Klaedtke and E. Zalinescu. The MonPoly monitoring tool. In International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, 2017.
2 D. Basin, S. Krstić, and D. Traytel. Almost event-rate indepedent monitoring of metric dynamic logic. In S. Lahiri and G. Reger, editors, *RV 2017*, volume 10548 of *LNCS*, pages 85–102. Springer, 2017.
3 D. Basin, S. Krstić, and D. Traytel. Aerial: Almost event-rate indepedent algorithms for monitoring metric regular properties. In International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, 2017.
4 J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

3.17 Behavioural Type-Based Static Verification Framework for Go

Nobuko Yoshida (Imperial College London, GB)

Main reference Julien Lange , Nicholas Ng , Bernardo Toninho , Nobuko Yoshida: "A Static Verification Framework for Message Passing in Go using Behavioural Types", in Proc. of the 40th Int'l Conf. on Software Engineering (ICSE 2018), to appear, ACM 2018.

I first give an introduction of our group working on session types.

Go is a production-level statically typed programming language whose design features explicit message-passing primitives and lightweight threads, enabling (and encouraging) programmers to develop concurrent systems where components interact through communication more so than by lock-based shared memory concurrency. Go can detect global deadlocks at runtime, but does not provide any compile-time protection against all too common communication mismatches and partial deadlocks.

In this work we present a static verification framework for liveness and safety in Go programs, able to detect communication errors and deadlocks by model checking. Our toolchain infers from a Go program a faithful representation of its communication patterns as behavioural types, where the types are model checked for liveness and safety.

4 Working groups

4.1 How do we integrate trace behaviour with state properties

Wolfgang Ahrendt (Chalmers University of Technology – Göteborg, SE), Stijn de Gouw (Open University – Heerlen, NL), Adrian Francalanza (University of Malta – Msida, MT), Zhiming Liu (Southwest University – Chongqing, CN), and Julien Signoles (CEA LIST – Gif-sur-Yvette, FR)

As a follow-up of the talk *Trace Focused and Data Focused Specification: Complementary, Competing, Combined?*, this group discussed how the community could move towards integrated/coordinated specification and analysis (static or dynamic) of trace *and* data focused aspects. We first made a distinction of specifications and models. The former describe properties a system is supposed to have, while the latter are a vehicle for system design, and finally, development. Following the theme of the seminar, we focused on specifications. Further, we soon converged on the view that the tasks of combining, relating, or even integrating divers specification styles and aspects requires a common semantic foundation.

An archetypal semantic base for both trace and data focused properties could, among others, be sequences $\langle (e_0, v_0), (e_1, v_1), (e_2, v_2), \ldots \rangle$ of pairs (e_i, v_i), where e_i are relevant events, and v_i are valuations, assigning variables (or locations of a heap, or alike) to values. It is obvious how to interpret trace oriented properties on such traces. But also data oriented properties can be interpreted on that basis. For instance, a Hoare triple $\{\phi\}\texttt{m()}\{\psi\}$ could be defined by using method entry and exit events, here $\texttt{m}^{\downarrow}$ and $\texttt{m}^{\uparrow}$, stating that $\texttt{m}^{\downarrow} = e_i$, $\texttt{m}^{\uparrow} = e_j$, and $v_i \models \phi$ implies $v_j \models \psi$.

We arrived at the following (sketch of) a method for relating and combining different formalisms, trace and data oriented.

1. Unify the events relevant for the formalisms.
2. Unify the data (valuations) relevant for the formalisms.
3. Design a semantic domain representing the above, suited to naturally give meaning to the divers properties of interest.

4.2 Specifications that are like implementations

Cyrille Artho (KTH Royal Institute of Technology – Stockholm, SE), Cindy Eisner (IBM – Haifa, IL), Keiko Nakata (SAP Innovation Center – Potsdam, DE), Dejan Nickovic (AIT Austrian Institute of Technology – Wien, AT), and Volker Stolz (West. Norway Univ. of Applied Sciences – Bergen, NO)

To be flexible and expressive with a specification, it is helpful to combine declarative and imperative modes. If the model is rich, this avoids the need for a "bridge specification" that refines an existing model, for example, if parameters are very complex and beyond the expressiveness of the initial modelling language.

4.2.1 Why is this the case?

Many formalisms are too restricted. Temporal logics often feel unnatural to use. Furthermore, logics and finite-state machines cannot count; to cope with that, either formulas become unnecessarily complex, or the formalisms have to be extended with counters. (This is the case, for example, with extended finite-state machines, which are much closer to a possible implementation of a system than what a basic finite-state machine can express.) To be flexible and expressive with a specification, it is helpful to combine declarative and imperative modes. If the model is rich, this avoids the need for a "bridge specification" that refines an existing model, for example, if parameters are very complex and beyond the expressiveness of the initial modelling language

4.2.2 Trade-offs

The table below shows some of the trade-offs with the choice of the specification language:

	Simple, declarative specification	**Complex (imperative?) specification**
Specific burden (human)	High if specification becomes complex due to limitations	May be lower; feels "natural" to developers
Level of abstraction	High	Low
Semantics	Well-defined but perhaps difficult for a human	Loosely-defined but perhaps clearer to developers
Analysis burden (tool)	Low	High
Link to implementation	Remote	Close
Refining specification to implementation	Difficult	More direct
Integration with existing tools	Difficult	More straightforward

A specification that is very low-level may make it difficult to express complex data types or properties. Conversely, though, the high level of abstraction required by typical specification languages is also a desirable trait. The specification should be significantly simpler than the implementation and not just mirror it. Otherwise, the level of abstraction leads to the same type of thinking, and hence the same bugs in both the specification and the implementation; the distinction of "what" (specification) vs. "how" (implementation) becomes blurred. In many cases, a "rich" specification language may be enticing because it offers the same features like a programming language. Conversely, certain features are not available in a programming language. For example, it is not possible (without much extra code) to quantify over all instances on the heap in a programming language, but some specification languages may allow that.

On the practical side, using the implementation language (or a subset of it) for specifications eliminates the need to learn about a new language or platform. This choice makes tool chain integration straightforward and allows the same tools to be used for static and dynamic analysis. It also allows properties to refer directly to the implementation. Refinements are easier to reason about if the language is the same. The semantics of the language may be clearer to developers, making this a more "practical" choice.

4.2.3 Solutions

Domain-specific languages seem to be a great way to hide low-level aspects of "classical" specification languages while restricting the complexity of the specification. For example, PSL has been very successful because it is on a higher level of abstraction than LTL, allows for different styles of specification, and has several ways to make specifications more succinct. However, building better specification languages is difficult; it takes experience to extract common patterns and "package" them in a nice way. Visual specification languages are popular for software engineering, but their semantics are often not well-defined and not executable. A mathematical or textual specification language should still be reasonably close in appearance to a programming language. This explains why very mathematical notations like LTL and Z are not so popular with engineers.

The choice of the language depends on the problem to be solved: If problems to be specified are close to the implementation platform, such as in run-time verification, programming languages can be suitable. For something that should be platform-agnostic and abstract, other choices may be better.

If a programming language is used for verification, the language features used should be limited, and side effects in statements must be avoided. It is good to forgo completeness (being able to express any property): some abstraction is good and should be enforced at the level of the specification language. In the extreme case, a reference implementation is an executable specification, but properties may not be explicit. In this case, the implementation part should be separate from the monitor that checks results.

A good specification toolkit includes validation tools (vacuity, consistency, realizability checking; simulation; visualization). High-level languages may be easier to validate in principle, but programming languages often have good tools for this purpose, too.

Most importantly, we need to support a good V and V process, not just to provide tools. The given specification language should be used in the spirit of proving a formal model, not an implementation.

4.3 Property Specification Patterns for Runtime Verification

Domenico Bianculli (University of Luxembourg, LU), Borzoo Bonakdarpour (McMaster University – Hamilton, CA), Bernd Finkbeiner (Universität des Saarlandes, DE), Gordon Pace (University of Malta – Msida, MT), Giles Reger (University of Manchester, GB), Kristin Yvonne Rozier (Iowa State University, US), Gerardo Schneider (Chalmers University of Technology – Göteborg, SE), Dmitriy Traytel (ETH Zürich, CH), and Nobuko Yoshida (Imperial College London, GB)

Property specification patterns have been proposed as a means to express recurring properties in a generalized form, to aid engineers writing system requirements precisely and map them to formalisms like temporal logics. Starting from the seminal work by Dwyer et al. [4], several systems (i.e., catalogues) of property specification patterns have been proposed in the literature [1, 3, 2, 6, 5, 7]. This working group discussed the use of property specification patterns for writing specifications in the context of Runtime Verification (RV).

4.3.1 Why Specification Patterns in RV?

A first discussion point focused on the identification of the main reasons for using property specification patterns in RV:

abstraction: to raise the level of abstraction when writing specifications;
conciseness: to express complex specifications in a concise way;
reusability: to reuse blocks of specifications in different systems and programming languages;
compositionality: to be able to compose and intertwine different aspects of a specification;
extensibility: to provide a system that can be extended in a coherent way for different application domains;
automation: to enable the automated transformation of a high-level, abstract specification into a low-level, concrete specification, by supporting different formalisms, technologies, and instrumentation strategies.

4.3.2 Towards a System of Property Specification Patterns for RV.

The second part of the discussion pinpointed the main attributes that would characterize a system of property specification patterns tailored for RV applications. Such a system shall:

- support the three main "planes" of RV (i.e., time, data, and algorithms/behaviors) as first-class entities;
- enable the definition of multi-level patterns;
- provide pattern combinators;
- be built based on existing pattern systems.

Furthermore, several complementary extensions could be envisioned for this system:

- anti-patterns, to give guidance about which specifications are better avoided;
- design patterns for building observers, verifiers, and enforcers;
- an "interface language" to describe the capabilities of observers, verifiers, and enforcers with respect to certain patterns;
- transformational patterns, for transforming specifications into various target formalisms.

Finally, the working group touched upon the main design choices that the designers of such a system would face:

- How deep should the system be embedded in a target formalism?
- Should the system account only for functional specifications or also non-functional ones?

References

1 M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, July 2015.
2 P. Bellini, P. Nesi, and D. Rogai. Expressing and organizing real-time specification patterns via temporal logics. *J. Syst. Softw.*, 82(2):183–196, February 2009.
3 Domenico Bianculli, Carlo Ghezzi, Cesare Pautasso, and Patrick Senti. Specification patterns from research to industry: a case study in service-based applications. In *ICSE 2012: Proceedings of the 34th international conference on Software engineering*, pages 968–976. IEEE Computer Society, 2012.
4 Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420. IEEE Computer Society Press, 1999.
5 Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electron. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.

6 Lars Grunske. Specification patterns for probabilistic quality properties. In *ICSE 2008: Proceedings of the 30th international conference on Software engineering*, pages 31–40, New York, NY, USA, 2008. ACM.
7 Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 372–381, New York, NY, USA, 2005. ACM.

4.4 Exploring the tradeoffs between Declarative and Operational Specification

Adrian Francalanza (University of Malta – Msida, MT), Wolfgang Ahrendt (Chalmers University of Technology – Göteborg, SE), Cindy Eisner (IBM – Haifa, IL), Zhiming Liu (Southwest University – Chongqing, CN), and Gordon Pace (University of Malta – Msida, MT)

This working group explored the tradeoffs between declarative and operational/algorithm approaches to specification. We begin with defining these two terms:

- By "Declarative" (D) what we understand is "What" is expected to hold. A good example here would be regular expressions, defined (denotationally) as the sets of strings that they represent.
- By "Algorithmic" (A) what we understand is "How" we intend to check what is expected to hold. A good example here would be DFSAs: they may be used to denote the sets of strings that they accept, via the operational procedure of processing a string from the start state to the final state.

Another analogy could be the one between state-based vs action-based specification, though it is not clear that it fits exactly. In practice, this clear delineation is often blurred.

Next we consider what criteria one may want to consider when assessing/evaluating/deciding whether to prefer one style over the other.

Readability and Understandable. In general (D) tend to be more concise, hence potentially more readable. This, of course, need not be the case and is also very subjective. Certain (A) approaches have a graphical representation which improves substantially the understandability (e.g. automata). Various anecdotal evidence was brought forward. Whereas it was certainly the case for small specifications, it was unclear whether these advantages would scale for larger/real-world properties. Syntactic sugaring is also very useful in these cases, and (D) may be more amenable to this due to the compositional operators. These points were also linked to the need to have Domain-Specific language adaptations of these formalisms where, perhaps, only a subset of the expressive power may be needed.

Maintainability/adaptability. In general (D) are more algebraic by nature and come equipped with operations to compose specifications. This tends to affect maintainability/adaptability/compositionality/decompositionality.

Specification procurement. To give a bit of context, there seems to be a general aversion of engineers towards using specifications altogether. One criteria could thus be to lean towards the form of specification that facilitate best specification procurement. To this end, techniques such as visualisations and connections to controlled natural languages are

definitely a plus. Neither form, (D) or (A), offered a particular advantage over the other. For instance the box modality $[\alpha]\psi$ can be expressed as *whenever* α *then* ψ *must hold*. A case can be made that logical operators are not as intuitive to the"general public" e.g. by *or* they often understand *exclusive or*, believe that *implication* is associated with causality etc. Another aspect that was briefly touched upon was that of integrating these things into the workflow of a company, having good error reporting etc.

4.5 Event Stream Processing

Sylvain Hallé (University of Quebec at Chicoutimi, CA), Martin Leucker (Universität Lübeck, DE), Nicolas Rapin (CEA – Gif sur Yvette, FR), César Sánchez (IMDEA Software – Madrid, ES), Torben Scheffel (Universität Lübeck, DE), and Hazem Torfah (Universität des Saarlandes, DE)

This working group concentrated on the topic of behavioural specification through Event Stream Processing (ESP). All the participants in this working group were involved in the design and development of one of several ESP tools: ARTiMon [5], BeepBeep [4], LOLA [1], and TeSSLa [3]. The discussion revolved around a few broad questions, which are summarized in the following.

4.5.1 Motivation

A first discussion topic focused on the reasons the various ESP tools have been developed in the first place. This point is relevant, given that most of us come from the Runtime Verification (RV) community, where specification languages are generally based on logic or automata. For most of us, the shift to ESP came "out of necessity" –that is, some use cases we were faced with were difficult or flatly impossible to model using traditional logic or automata.

Case in point, one participant recalled an experience with members of the industry, who were shown specifications expressed in temporal logic in a first meeting, and specifications expressed as stream equations during a subsequent meeting. The overall reception to the stream equations was much more positive than for the temporal logic formulæ ("why didn't you show this in the first place?" was the reaction of one of the attendees). For some classes of problems, and for some audiences, the use of streams to model a specification may prove a better fit than existing logics and automata-based notations.

It shall be noted that, although the use of event streams in a verification or testing context is relatively recent, it can be seen as a generalization of much older concepts. For example, *temporal testers* have been introduced for the verification of Linear Temporal Logic specifications more than twenty years ago [2]. By using the same core principles (composition, incremental update upon each event), contemporary ESP tools extend temporal testers beyond Boolean streams and temporal logic, and generalize them to allow the computation of aggregations over numerical values and many other functions.

4.5.2 Declarative or Imperative?

A second point of discussion is the classification of event stream languages as *declarative* or *imperative*. For example, the main mode of operation of BeepBeep is to programmatically instantiate and connect "boxes" that each perform a simple computation, which is close to an imperative language. However, one can also see an event processing specification as a set of equations defining relationships between input and output streams; this is especially apparent in the input languages of ARTiMon, LOLA and TeSSLa. It can be argued that these equations define constraints on the inputs and the outputs that must be satisfied, but do not explicitly describe *how* to compute a satisfying output for a given input (or if such an output even exists). These traits would classify event stream specifications as declarative.

4.5.3 Relative Expressiveness

Although ESP tools share many similarities (the possibility to compute aggregation functions over time windows, for example), their expressiveness is not identical. One particular point where they differ is in their interpretation of time. Some systems assume synchronous streams where computation occurs in discrete steps, while others accept input streams that produce events at their own rate. Again, the particular implementation of time in each tool has been motivated by concrete use cases that the system had to handle.

4.5.4 Should there be fewer tools?

Another question that was discussed is whether it would be desirable to have fewer stream languages. The general consensus among panel members was that, since their expressiveness is not identical (see above), each tool fulfills a different need. Moreover, having multiple tools maintains a healthy level of competition and drives innovation.

It was also observed that, in the past fifteen years, we have witnessed the introduction of a large number of new programming languages. In many cases, new languages have been invented, because "starting from scratch" was easier than adapting an existing (and only partially appropriate) solution; a similar argument can be made about stream languages. To sum it up, if the existence of multiple programming languages is generally accepted and is not seen as abnormal, why would that be different for specification languages?

4.5.5 The future

As a next step for the short- and medium-term, it was suggested that authors of ESP tools should give themselves a standardized vocabulary to define the features of each specification language and each system. This vocabulary, in turn, would make it possible to compare the various solutions.

References

1 B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, Z. Manna. (2005). LOLA: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning, TIME 2005*, IEEE Computer Society, 166–174.

2 Y. Kesten, A. Pnueli, and L. Raviv. (1998). Algorithmic verification of linear temporal logic specifications. In *Proc. 25th Int. Colloq. Aut. Lang. Prog.*. Springer: Lecture Notes in Computer Science, vol. 1443, 1–16.

3 M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, A. Schramm. (2018). TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams. To appear on *ACM Symposium on Applied Computing, SAC 2018*.
4 S. Hallé. (2016). When RV Meets CEP. In Y. Falcone, C. Sánchez (eds), *Runtime Verification, RV 2016*. Springer: Lecture Notes in Computer Science, vol. 10012, 68–91.
5 N. Rapin. (2016) Reactive Property Monitoring of Hybrid Systems with Aggregation. In Y. Falcone, C. Sánchez (eds), *Runtime Verification, RV 2016*. Springer: Lecture Notes in Computer Science, vol. 10012, 447–453.

4.6 A shared challenge – NASA's Core Flight System

Volker Stolz (West. Norway Univ. of Applied Sciences – Bergen, NO), Borzoo Bonakdarpour (McMaster University – Hamilton, CA), Martin Leucker (Universität Lübeck, DE), Nicolas Rapin (CEA – Gif sur Yvette, FR), Kristin Yvonne Rozier (Iowa State University, US), Julien Signoles (CEA LIST – Gif-sur-Yvette, FR), Hazem Torfah (Universität des Saarlandes, DE), and Nobuko Yoshida (Imperial College London, GB)

4.6.1 Introduction

This working group focused on identifying a single, unified challenge which could be used by the RV community to show-case their tools and methods. In other communities, there have been case studies that are still around and referred to frequently, such as the Steam Boiler Control [3], the Mondex case study grand challenge [4], and the Common Component Modelling Example CoCoME [5]. We intend to kick-start a similar "future-proof" arena for joint activity through further discussion among the seminar participants. The following summarises the discussion within the working group.

In particular, we were interested in a topic that not only serves those goals (and has possibilities for most community members to contribute), but also is current and and engaging. Unmanned (aerial) vehicles (UAVs/drones) seem to fit this purpose very well: it is a topic that easily engages students, has many "moveable parts" that make it easy and interesting to instrument (sensor readings and engine data), and as a platform UAVs are easily extensible with additional sensors to tailor them to particular projects.

Although consumer-grade products usually come with their own control software, operating systems are easily interchangeable, and are frequently Android-based. To fulfil the requirement of a single unified system, NASA's Core Flight System (CFS, https://cfs.gsfc.nasa.gov/), can serve as a middleware tying together the various functionalities. Furthermore, prominent members of the community are, or have been, affiliated with NASA which developed and maintains the stack, and can thus serve as contact point for others seeking help or guidance with the system. In the following, we discuss some of the possibilities and challenges that those devices together with the CFS present as a platform for a shared RV challenge.

4.6.2 NASA's Core Flight System

The system covers the full range of a cyber-physical system: on top of a Linux kernel, a modular architecture provides a middleware for communication using publish/subscribe between the different components. Specific applications sit on top of this middleware. The

entire system is open-source and available from NASA's open-source repositories. It also includes a simulator. The Linux-based open-source architecture should make it easy to interface with additional tools, such as the instrumentations required for applied runtime verification.

A wide range of hardware is already supported, and the software has already been used on off-the-shelf UAVs. For new hardware, some configuration may be required, and contacts with NASA confirm that they are willing to provide some remote support, and have already made steps to incorporate the latest updates into the publicly available repositories.

Even within constrained budgets, consumer-grade drones or a computationally more powerful octocopter can be purchased for below a 1000 USD, allowing even smaller groups to participate after a small up-front investment. Within this budget, a subgroup of the community could also create a reference architecture with a well-defined set of sensors and motors. For the lower-end hardware, CFS does not directly run on the hardware itself, but on a separate computer that uses a communications link (e.g. wifi) to remote-control the vehicle.

4.6.3 Requirements from the RV Perspective

For the RV community, it is important that as many of the diverse approaches can be served by this platform. Most importantly, this includes the division between online- and offline processing, runtime reflection, and solutions requiring specific software. In general, we remark that the open-source nature of the entire platform gives ample opportunity to customize it.

For online runtime verification, the publish/subscribe communications-style of the middleware makes it easy to receive events from subsystems. Runtime reflection, where the behaviour of the system is actively influenced, can be equally easily achieved if the component to be influenced reacts to messages on the middleware bus, but otherwise may of course require some modification, or the design of new components.

Offline runtime verification, where event data is analysed through RV techniques (evaluating for example temporal logic properties), can be easily achieved by recording event data from the system, e.g. by logging messages from the middleware. Even if no hardware is available, trace data can be obtained from the simulator. Even if a community member lacks the technical expertise to run the system themselves, traces can be generated and published (e.g. in the upcoming trace data repository of the EU COST Action IC1402 "ARVI – Runtime Verification Beyond Monitoring", see [1]) by other members.

Several seminar participants work with particular implementations of runtime verification technology, e.g. those that are implemented on top of the Java Virtual Machine. Naturally, those approaches do not easily carry over into the world of embedded systems, where often specific operating systems requirements do not allow running full user-land applications, or memory or CPU are too constrained to enable such approaches. The suggested platform in principle offers a full operating system, and is only constrained by the underlying hardware. This allows groups with higher computational requirements to move to more powerful hardware, or use the simulator.

4.6.4 Relevant Properties

This working group also took a precursory look at relevant runtime verification-related properties that may be investigated on such a system. In particular, the general nature of the middleware should offer plenty of opportunity to look for relevant events. Although of course one could potentially devise application specific properties, and existing set of specifications

for the platform would be an interesting possibility of directly applying RV techniques in this domain.

We easily found existing specifications in particular for the LADEE mission such as "while in a flight mode, with active control disabled, a particular parameter shall also be disabled" (Karen Gundy-Burlet, SPIN 2014 keynote). The same mission has also already been utilised by Havelund as an application scenario for TraceContract [2].

Furthermore, we also found MatLab/Simulink models in the repositories which are of direct interest to runtime verification.

4.6.5 Conclusion

In conclusion, we are confident that such a system gives ample opportunity to either find, or come up with our own, properties that our current tools can then monitor, and that the majority of the community members hopefully have the resources and expertise to participate.

References

1 COEMS – Open traces from industry. S. Jaksic, M. Leucker, D. Li, and V. Stolz. In: RV-CuBES workshop, Kalpa, 2017.
2 Checking Flight Rules with TraceContract: Application of a Scala DSL for Trace Analysis. H. Barringer, K. Havelund, E. Kurklu, and R. Morris. In: Scala Days 2011
3 Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. J.-R. Abrial, E. Börger, and H. Langmaack. In: LNCS 1165, Springer, 1996.
4 First steps in the verified software grand challenge. J. Woodcock. In: Computer, 39(10):57–64, October 2006.
5 The Common Component Modeling Example: Comparing Software Component Models. A. Rausch, R. H. Reussner, R. Mirandola, F. Plasil (eds.). In: LNCS 5153, Springer, 2008

4.7 Data Quantification in Temporal Specification Languages

Dmitriy Traytel (ETH Zürich, CH), Domenico Bianculli (University of Luxembourg, LU), and Giles Reger (University of Manchester, GB)

In this working group, we tried to collect and characterize the different kinds of quantification or parametrization that can be encountered in temporal specification languages used in the runtime verification community. Our selection, albeit far from being comprehensive, shows that the main semantic difference is the domain of quantification. We have identified five different groups.

4.7.1 Standard First-Order Quantification

An early approach taken by Emerson's first-order linear temporal logic (FOLTL) [13] is to add standard first-order logic quantifiers to LTL. Thereby, the quantifiers range over a fixed domain, which is independent of the trace (sequence of structures). Chomicki's real-time extension of FOLTL, called metric first-order temporal logic (MFOTL) [9] follows this approach. The MonPoly monitoring tool [4] demonstrates how such quantifiers can be handled algorithmically. The new Dejavu tool [16] (and logic) quantifies over fixed (infinite) domains.

4.7.2 Quantification over the Active Domain

A different approach, inspired by the database community, is to quantify over the active domain, i.e., values that occur in the trace, rather than a fixed, separately given domain. For certain classes of properties, e.g., where all quantifiers are bounded by atomic propositions as in $\forall x.p(x) \rightarrow \phi$ or $\exists x.p(x) \wedge \phi$, active domain quantification coincides with the standard first-order quantification.

Several specification languages favor the active domain quantification, as it appears to be algorithmically more tractable. They differ in their definition of what the *active domain* in a trace is. The traditional database definition as *all values contained in the trace*, used in LTL-FO [12], is hard to realize in the online setting, where the trace is an infinite stream of events. An adaptation of the active domain to *all previously seen values* would fit this setting better. However, the most widespread interpretation is to restrict the quantification to *values seen at the current time-point*. For example, the languages LTL^{FO} [7], LTL-FO^{+} [15], and Parametrized LTL [20] use this semantics. Additionally, DejaVu [16] in its current form can quantify over "seen" values in the past. There are in fact two sets of quantifiers: `forall` and `exists` for quantifying over seen values, and `Forall` and `Exists` for quantifying over all values in the fixed (possibly infinite) domain.

4.7.3 Freeze Quantification

Freeze quantification is a further refinement of the quantification over the current time-point approach. The usage of registers restricts the quantification to be a singleton: the only value that populates the register at a given time-point. Timed propositional temporal logic (TPTL) [1] uses such quantifiers to extend LTL with real-time constraints. Here, we are interested in quantification over the data dimension rather than the time dimension, as used in Freeze LTL [11] and its extensions [10]. A recent extension of MTL with freeze quantification over data $\text{MTL}^{\downarrow}$ [5] was used as the specification language when online monitoring our-of-order traces.

4.7.4 Templates and Parametric Trace Slicing

Some approaches avoid explicit quantification in their formalisms. Yet, they allow parametric specifications, which are handled by decomposing traces containing data into propositional ones. This approach is known as parametric trace slicing [8, 19, 21], which is at the core of the JavaMOP system [18] and in quantified event automata QEA [2].

More recently, the stream-based specification language LOLA [14] introduced parametrization in terms of template specifications. Semantically, templates behave similarly to parametric trace slicing, but the precise connections are yet to be explored.

4.7.5 Quantitative Quantifiers

Finally, some data quantifiers in addition to binding a variable also perform an arithmetic operation (be it filtering, grouping, or aggregation) on the quantified values (be them data or the number of satisfied instances). Example languages in this space are LTL^{FO} extended with counting quantifiers [6], $\text{LTL}_4\text{-C}$ [17] with its probabilistic quantifiers, and the extension of MFOTL with aggregations [3].

References

1 R. Alur and T. A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–204, 1994.

2 H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, editors, *FM 2012*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.

3 D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.

4 D. A. Basin, F. Klaedtke, S. Müller, and E. Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.

5 D. A. Basin, F. Klaedtke, and E. Zalinescu. Runtime verification of temporal properties over out-of-order data streams. In R. Majumdar and V. Kuncak, editors, *CAV 2017*, volume 10426 of *LNCS*, pages 356–376. Springer, 2017.

6 A. Bauer, R. Goré, and A. Tiu. A first-order policy language for history-based transaction monitoring. In M. Leucker and C. Morgan, editors, *ICTAC 2009*, volume 5684 of *LNCS*, pages 96–111. Springer, 2009.

7 A. Bauer, J. Küster, and G. Vegliach. The ins and outs of first-order runtime verification. *Formal Methods in System Design*, 46(3):286–316, 2015.

8 F. Chen and G. Rosu. Parametric trace slicing and monitoring. In S. Kowalewski and A. Philippou, editors, *TACAS 2009*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.

9 J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

10 N. Decker and D. Thoma. On freeze LTL with ordered attributes. In B. Jacobs and C. Löding, editors, *FoSSaCS 2016*, volume 9634 of *LNCS*, pages 269–284. Springer, 2016.

11 S. Demri, R. Lazic, and D. Nowak. On the freeze quantifier in constraint LTL: decidability and complexity. *Inf. Comput.*, 205(1):2–24, 2007.

12 A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In S. Vansummeren, editor, *PODS 2006*, pages 90–99. ACM, 2006.

13 E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.

14 P. Faymonville, B. Finkbeiner, S. Schirmer, and H. Torfah. A stream-based specification language for network monitoring. In Y. Falcone and C. Sánchez, editors, *RV 2016*, volume 10012 of *LNCS*, pages 152–168. Springer, 2016.

15 S. Hallé and R. Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC 2008*, pages 63–72. IEEE Computer Society, 2008.

16 First Order Temporal Logic Monitoring with BDDs. K. Havelund, D. Peled, and D. Ulus 17th Conference on Formal Methods in Computer-Aided Design (FMCAD 2017), 2-6 October, 2017, Vienna, Austria. IEEE.

17 R. Medhat, B. Bonakdarpour, S. Fischmeister, and Y. Joshi. Accelerated runtime verification of LTL specifications with counting semantics. In Y. Falcone and C. Sánchez, editors, *RV 2016*, volume 10012 of *LNCS*, pages 251–267. Springer, 2016.

18 P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

19 G. Reger and D. E. Rydeheard. From first-order temporal logic to parametric trace slicing. In E. Bartocci and R. Majumdar, editors, *RV 2015*, volume 9333 of *LNCS*, pages 216–232. Springer, 2015.

20 V. Stolz. Temporal assertions with parametrized propositions. *J. Log. Comput.*, 20(3):743–757, 2010.

21 K. Havelund and G. Reger and E. Zalinescu and D. Thoma Monitoring Events that Carry Data, In E. Bartocci and Y. Falcone, editors, Lectures on Runtime Verification – Introductory and Advanced Topics, volume 10457 of *LNCS*, pages 60–97, Springer, 2018

Participants

- Wolfgang Ahrendt
Chalmers University of Technology – Göteborg, SE
- Cyrille Artho
KTH Royal Institute of Technology – Stockholm, SE
- Domenico Bianculli
University of Luxembourg, LU
- Borzoo Bonakdarpour
McMaster University – Hamilton, CA
- Stijn de Gouw
Open University – Heerlen, NL
- Cindy Eisner
IBM – Haifa, IL
- Bernd Finkbeiner
Universität des Saarlandes, DE
- Adrian Francalanza
University of Malta – Msida, MT
- Sylvain Hallé
University of Quebec at Chicoutimi, CA
- Martin Leucker
Universität Lübeck, DE
- Zhiming Liu
Southwest University – Chongqing, CN
- Keiko Nakata
SAP Innovation Center – Potsdam, DE
- Dejan Nickovic
AIT Austrian Institute of Technology – Wien, AT
- Gordon Pace
University of Malta – Msida, MT
- Nicolas Rapin
CEA – Gif sur Yvette, FR
- Giles Reger
University of Manchester, GB
- Kristin Yvonne Rozier
Iowa State University, US
- César Sánchez
IMDEA Software – Madrid, ES
- Torben Scheffel
Universität Lübeck, DE
- Gerardo Schneider
Chalmers University of Technology – Göteborg, SE
- Julien Signoles
CEA LIST – Gif-sur-Yvette, FR
- Volker Stolz
West. Norway Univ. of Applied Sciences – Bergen, NO
- Hazem Torfah
Universität des Saarlandes, DE
- Dmitriy Traytel
ETH Zürich, CH
- Nobuko Yoshida
Imperial College London, GB