Programming - What is Next?

Klaus Havelund[1⋆] and Bernhard Steffen[2]

[1] Jet Propulsion Laboratory, California Institute of Technology, USA
[2] TU Dortmund University, Germany.

Abstract. The paper provides an introduction to the track: "Programming - What is Next?", organized by the authors as part of ISoLA 2021: the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. A total of 14 papers were presented in the track, with responses to the question: *what are the trends in current more recent programming languages, and what can be expected of future languages?*. The track covers such topics as general-purpose programming languages, domain-specific languages, formal methods and modeling languages, textual versus graphical languages, and application programming versus embedded programming.

Keywords: Programming, general-purpose languages, domain-specific languages, formal methods, modeling, textual languages, graphical languages, application programming, embedded programming.

1 Introduction

High-level main-stream programming languages (high-level wrt. to assembler languages and machine code) have evolved dramatically since the emergence of the Fortran language well over half a century ago, with hundreds of languages being developed since then. In the last couple of decades we have seen several languages appearing, most of which are oriented towards application programming, and a few of which are oriented towards systems and embedded close-to-the-metal programming. More experimental programming languages focusing e.g. on program correctness, supporting proof systems have appeared as well.

In addition, we see developments in the area of Domain-Specific Languages (DSLs), including visual as well as textual languages, easy to learn for experts in dedicated fields. Combined with approaches like generative and meta-programming this may lead to very different styles of system development. Related to these developments, we can also observe developments in modeling languages meant to support abstraction, verification, and productivity.

This paper provides an introduction to the track: "Programming - What is Next?", organized by the authors as part of ISoLA 2021: the 9th International

⋆ The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Symposium On Leveraging Applications of Formal Methods, Verification and Validation. 14 papers were presented in the track, with responses to the question: *what are the trends in current more recent programming languages, and what can be expected of future languages?*.

The papers presented cover various topics. There is a core of papers focusing on different ways of programming applications, and in particular embedded systems, using general purpose programming languages. Here classical programming languages such as C and C++ have been dominating for decades. However, these are low-level and unsafe, and better abstractions are needed. This includes formal specification and proof support. Related to this topic is the question how modeling and programming interacts, and it is emphasized that modeling and programming ought to be tightly integrated. Several papers discuss programming language concepts and constructs. New concepts are proposed, such as time as a core concept, an alternative to object-orientation, advanced type systems, and a suggestion to focus on non-linear dynamic systems. The alternative to general-purpose programming languages is domain-specific languages. Several papers advocate for their use, both textual and graphical. General-purpose as well as domain-specific languages are typically used/developed in IDEs. A browser-based approach is advocated in one paper.

The track can be seen as a followup of the tracks named *"A Unified View of Modeling and Programming"*, organized at ISoLA 2016 [2] and 2018 [3]. These tracks focused on the similarities and differences between programming languages and modeling languages. Whereas those tracks considered programming and modeling of equal interest, the "Programming - What is Next?" track is more focused on the programming activity.

2 Contributions

The papers presented in the track are introduced below. They are divided into subsections according to the track sessions, covering approaches to program development, programming language concepts, and domain-specific languages.

2.1 Program Development

Lethbridge [8] (*Low-code is often high-code, so we must design low-code platforms to enable proper software engineering*), argues that software written on low code platforms often accumulates large volumes of complex code, which can be worse to maintain than in traditional languages, because the low-code platforms tend not to properly support good engineering practices such as version control, separation of concerns, automated testing and literate programming. Based on his experience with low code platforms he claims that such technical debt can only be avoided by providing low-code platforms with just as deep a capability to support modern software engineering practices as traditional languages. As a side result he see this as a sign that traditional programming will maintain its value also in the (long) future.

Lee and Lohstroh [7] (*Time for all programs, not just real-time programs*), argue that the utility of time as a semantic property of software is not limited to the domain of real-time systems. This paper outlines four concurrent design patterns: alignment, precedence, simultaneity, and consistency, all of which are relevant to general-purpose software applications. It is shown that a semantics of logical time provides a natural framework for reasoning about concurrency, makes some difficult problems easy, and offers a quantified interpretation of the CAP theorem, enabling quantified evaluation of the trade-off between consistency and availability.

Havelund and Bocchino [6] (*Integrated modeling and development of component-based embedded software in Scala*), explore the use of Scala for modeling and programming of embedded systems represented as connected components. Four internal Scala DSLs are presented, inspired by an actual C++ framework, for programming space missions. The DSLs support programming of software components, hierarchical state machines, temporal logic monitors, and rule-based test generators. The effort required to develop these DSLs has been small compared to the similar C++ effort. It is argued that although Scala today is not suitable for this domain, several current efforts aim to develop Scala-like embedded languages, including the works [11, 5] reported on in this volume.

Robby and Hatcliff [11] (*Slang: The Sireum programming language*), present the programming language Slang, syntactically is a subset of the Scala programming language, for programming high assurance safety/security-critical systems. The language supports specification and proof of properties, and omits features that make formal verification difficult. A subset, Slang Embedded, can be compiled to e.g. C. Slang can be used for prototyping on a JVM, and later re-deployed to an embedded platform for actual use. Slang is used as programming language in HAMR, see [5] in this volume, a High Assurance Model-based Rapid engineering framework for embedded systems. Developers here specify component-based system architectures using the AADL architecture description language.

Hatcliff, Belt, Robby, and Carpenter [5] (*HAMR: An AADL multi-platform code generation toolset*), present HAMR, a tool-kit for High-Assurance Model-based Rapid engineering for embedded cyber-physical systems. Architectures are modeled using AADL. HAMR is based on an abstract execution model that can be instantiated by back-end translations for different platforms. Elements of models can be programmed in the Slang programming language, translatable to C, also reported on in this volume [11]. The framework supports automated formal verification of models and code written in Slang. Since the infrastructure code and code generators are written in Slang, HAMR provides the convenience of a single verification framework to establish the correctness of code generation.

2.2 Program Language Concepts

Mosses [10] (*Fundamental constructs in programming languages*), presents a framework for defining the semantics of programming constructs at a high level of abstraction. A programming language construct is defined by translating it to fundamental constructs, referred to as funcons, in a compositional manner.

The use of funcons is meant as a precise and complete alternative to informal explanations of languages found in reference manuals. Furthermore, specifying languages by translation to funcons appears to be significantly less effort than with other frameworks. Funcons abstract from details related to implementation efficiency, and are defined using a modular variant of structural operational semantics. A library of funcons has been developed, available online, along with tools for generating funcon interpreters from them.

Harel and Marron [4] (*Introducing dynamical systems and chaos early in computer science and software engineering education can help advance theory and practice of software development and computing*), argue that the concept of nonlinear dynamic systems, their theory, and the mathematical and computerized tools for dealing with them, should be taught early in the education of computer scientists. These systems are found in diverse fields, such as fluid dynamics, biological population analysis, and economic and financial operations. Such systems are complex and embody the notion of chaotic behavior. Focus on dynamic systems can lead to enrichment of e.g. programming languages, tools and methodologies in computer science.

Wadler [15] (*GATE: Gradual effect types*), highlights the value of advanced type systems, including effect types, and discusses how they can become mainstream. Traditional type systems are concerned with the types of data. Effect types are concerned with the effects that a program may invoke, such as input, output, raising an exception, reading or assigning to state, receiving or sending a message, and executing concurrently. It is argued that in order to make such advanced type systems main stream, a gradual approach is needed (the "gate" to types), where types can be gradually added, and which allow untyped languages to interoperate with strongly typed languages. The paper provides a survey of some of the work on these different advanced type systems.

Selić and Pierantonio [12] (*Fixing classification: a viewpoint-based approach*), argue that the classification scheme realized in traditional object-oriented computer languages is insufficient for modern software development, which is becoming increasingly more integrated with the highly dynamic physical world. The limitations of the traditional binary classification approach makes it difficult to model dynamic reclassification of objects, classification of objects from different perspectives, and representing in-between cases, where an entity may be categorized as belonging in more than one class. The paper outlines a new approach to classification based on viewpoints, overcoming these limitations. The proposed approach replaces the static multiple-inheritance hierarchy approach with multiple dynamic class hierarchies, including overlapping class membership.

2.3 Domain-Specific Languages

Stevens [13] (*The future of programming and modelling: a Vision*), argues that, despite impressive achievements, software development now suffers from a capacity crisis which cannot be alleviated by programming as currently conceived. Rather, it is necessary to democratise the development of software: stakeholders who are not software specialists must, somehow, be empowered to take more

of the decisions about how the software they use shall behave. She proposes to describe this behaviour via a collection of models, each expressed in a (domain-specific) language appropriate to its intended users. Bi-directional transformations are then meant to serve for the corresponding global semantics. The paper also discussed required advances to guarantee the required progress.

Balasubramanian, Coglio, Dubey, and Karsai [1] (*Towards model-based intent-driven adaptive software*), argue that a model-based workflow for adaptive software may reduce the burden caused by system evolution like requirement changes and platform updates. In their vision, a modeling paradigm centered around the concepts of objectives, intents, and constraints, may uniformly comprise required functionalities as well as all managerial aspects. These concepts define, respectively, (1) what the system must do in terms of domain-specific abstractions, (2) the concretization choices made to refine a model into implementation, and (3) the system requirements not already expressed in terms of domain-specific abstractions.

Margaria, Chaudhary, Guevara, Ryan, and Schieweck [9] (*The interoperability challenge: building a model-driven digital thread platform for CPS*), argue that the traditional approach to achieve interoperability is inadequate and requires a model-driven platform approach supporting low-code application development on the basis of dedicated domain-specific languages. The paper illustrates the impact of such a platform by examples about robotics, Internet of Things, data analytics, and Web applications. In particular, it is shown how REST services can generically be extended, external data bases can be integrated, and new data analytics capabilities can be provided.

Voelter [14] (*Programming vs. that thing subject matter experts do*), argues that allowing subject matter experts to directly contribute their domain knowledge and expertise to software through DSLs and automation does not necessarily require them to become programmers. In his opinion, the requirement to provide precise information to unambiguously instruct a computer can be achieved more easily, of course requiring the basics of computational thinking. Völter believes that it is possible and economically important to provide accordingly 'CAD programs for knowledge workers'.

Zweihoff, Tegeler, Schürmann, Bainczyk, and Steffen [16] (Aligned, Purpose-driven cooperation: the future way of system development), argue that the future of software and systems development is collaborative, and will be supported globally in a cloud-based fashion. This way individual contributors do not need to worry about the infrastructural aspects which are all taken care of globally in the Web. This eases also the use of so-called purpose-specific languages that aim at directly involving application experts in the development process. The presentation of the vision is supported by details about the realization which, in particular, explain a simplicity-oriented way of language integration which can happen in a deep and shallow fashion.

3 Conclusion

The contributions of this track clearly indicate the expanse of what can be considered programming. It is therefore not surprising that the visions of where the evolution of programming will lead to, or should aim at, are very diverse. This diversity, however, does not imply that the visions are contradictory. Hopefully on the contrary. The embedded systems perspective, e.g., is envisaged to even deal with phenomena like chaos, application programming to successively comprise more computational paradigms and constructs to enable experts to elegantly solve dedicated tasks, and the future of user-level programming seems dominated by increasing ease and collaboration. Tools play a major role in all scenarios, which seem to have in common that programming and modelling will increasingly converge.

References

1. Balasubramanian, D., Coglio, A., Dubey, A., Karsai, G.: Towards model-based intent-driven adaptive software. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
2. Broy, M., Havelund, K., Kumar, R., Steffen, B.: Towards a unified view of modeling and programming (track summary). In: Margaria, T., Steffen, B. (eds.) 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2016, Part 2, Corfu, Greece, October 10-14. LNCS, vol. 9953, pp. 3–10. Springer (2016)
3. Broy, M., Havelund, K., Kumar, R., Steffen, B.: Towards a unified view of modeling and programming (track summary). In: Margaria, T., Steffen, B. (eds.) 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2018, Limasol, Cyprus, November 5-9. LNCS, vol. 11244, pp. 3–21. Springer (2018)
4. Harel, D., Marron, A.: Introducing dynamical systems and chaos early in computer science and software engineering education can help advance theory and practice of software development and computing. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
5. Hatcliff, J., Belt, J., Robby, Carpenter, T.: HAMR: An AADL multi-platform code generation toolset. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
6. Havelund, K., Bocchino, R.: Integrated modeling and development of component-based embedded software in Scala. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
7. Lee, E.A., Lohstroh, M.: Time for all programs, not just real-time programs. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
8. Lethbridge, T.C.: Low-code is often high-code, so we must design low-code platforms to enable proper software engineering. In: Proc. of the 9th Int. Symp. on

Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
9. Margaria, T., Chaudhary, H.A.A., Guevara, I., Ryan, S., Schieweck, A.: The interoperability challenge: Building a model-driven digital thread platform for CPS. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
10. Mosses, P.D.: Fundamental constructs in programming languages. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
11. Robby, Hatcliff, J.: Slang: The Sireum programming language. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
12. Selić, B., Pierantonio, A.: Fixing classification: A viewpoint-based approach. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
13. Stevens, P.: The future of programming and modelling: a vision. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
14. Voelter, M.: Programming vs. that thing subject matter experts do. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
15. Wadler, P.: GATE: Gradual effect types. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume
16. Zweihoff, P., Tegeler, T., Schürmann, J., Bainczyk, A., Steffen, B.: Aligned, purpose-driven cooperation: The future way of system development. In: Proc. of the 9th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2021). LNCS, Springer (2021), in this volume