Challenges Engaging Formal CBSE in Industrial Applications

Yi Li[1] and Meng Sun[2(✉)]

[1] Huawei
liyi.fm@foxmail.com
[2] School of Mathematical Sciences, Peking University
sunm@pku.edu.cn

Abstract. Component-based software engineering (CBSE) is a widely used software development paradigm. With software systems becoming increasingly sophisticated, CBSE provides an effective approach to construct reusable, extensible, and maintainable software systems. Formal verification provides a rigorous and systematic approach to validate the correctness of software systems by mathematically proving properties or checking them exhaustively against specified requirements. Using formal verification techniques in component-based development can further enhance the correctness of the development process. However, the adoption of component-based development supported by formal methods is hardly widespread in the industry. It serves to a limited extent in domains with stringent requirements for safety and reliability. In this paper, we aim to analyze the successful application scenarios of formal methods in component-based development, identify the challenges faced during their application, and explore methods to further broaden their adoption.

Keywords: Formal Methods · Component-based Software Engineering.

1 Introduction

The ever-increasing demand for more sophisticated and efficient software systems has necessitated the exploration of novel development methodologies. Among these methodologies, Component-Based Software Engineering (CBSE) [23, 39] has emerged as a promising approach that focuses on the development of software systems by assembling pre-existing, self-contained software components. This paradigm shifting from traditional monolithic software development to a component-based approach offers numerous advantages in terms of flexibility, reusability, and modularity.

CBSE does address the issue of code reusability to some extent, and the repeated use and validation of the same components across different projects does improve their reliability. However, it does not fundamentally solve the problem of the correctness of the components and the systems themselves. In recent years, complex software has become increasingly involved in and deeply integrated into people's daily lives, such as autonomous vehicles [13], smart cities [41], and smart homes [17]. The correctness of such systems has a significant impact on the safety of human life and property. Formal methods provide a rigorous and systematic approach to software development, ensuring correctness, reliability, and robustness of the system. Therefore, introducing formal methods as an aid in CBSE is of great importance in enhancing software quality and correctness.

A list of impressive research on formal methods in CBSE has been proposed [3, 5, 23, 27, 37]. However, in industrial practice, we find that such application is relatively niche. And its usage scenarios are also somewhat limited. In this paper we present the main obstacles, as observed by the authors, that hinder the application of formal methods in the field of component-based software development.

The paper is organized as follows. Section 2 briefly introduces some necessary background knowledge. Section 3 shows some popular industrial examples where formal methods are engaged in CBSE. Section 4 lists major challenges when scaling formal methods to more scenarios. And Section 5 discusses and proposes some ideas to tackle the challenges.

2 Background

In this section we introduce some background knowledge to provide readers with a better understanding of the challenges that will be discussed in the following sections.

2.1 Software Development Process

Neither component-based software development nor formal verification is a silver bullet. They are not powerful enough to take over the software development processes [43] widely used in the field today. In typical industrial applications, CBSE is only applied to part of the steps in the whole software development process.

Therefore, for researchers who are interested in applying formal verification techniques or component-based software development approaches in practical applications, it is crucial to understand the actual software development process. Only then can we determine which part of the process our designed methods and developed tools can be applied to and whether they might have any negative impact on the remaining parts of the software development process.

Popular software development processes include the V-model, waterfall model, spiral model, agile development model, and others. Different models are usually associated with different business scenarios. In actual development processes, development teams often customize these processes to align with their specific needs. Therefore, instead of focusing on a specific development model, here we only focus on some common steps in these development models, as shown in Fig. 1.

In the software development process, the steps are all linked with one another. As a result, when we want to use a tool or a methodology to "optimize" some steps, it is extremely important to evaluate its side-effect.

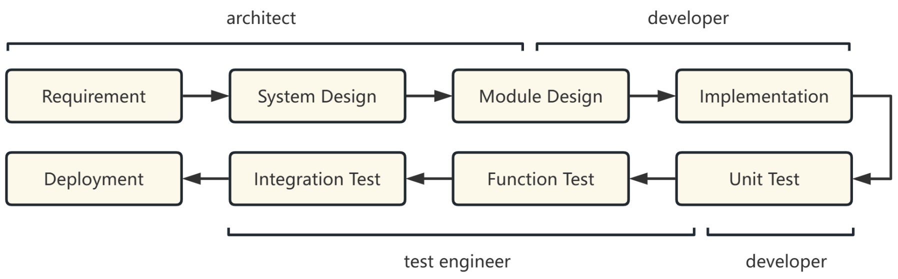

Fig. 1. Software Development Process

For example, if we use a code generator to synthesize an implementation from the module design, then the developers have to read the generated code to write the unit test cases. In this case, the readability of the generated code could become a major obstacle. Similarly, using formal models to represent system designs and module designs makes it harder for test engineers to write integration test cases and function test cases. Usually extra documentation is needed to make it work.

2.2 CBSE, MDE and Others

Component-based software engineering (CBSE) and model-driven engineering (MDE) are two distinct methodologies from a design perspective. However, in practical applications, these two approaches have a close relationship and can easily be confused. Given that this relationship is derived from the involvement of formal methods (the topic discussed in this article), we provide a brief explanation based on our view.

From a design perspective, CBSE emphasizes the relationship between the whole and its parts, with the goal of maximizing reusability. On the other hand, MDE focuses on the relationship between abstraction and concreteness (e.g., requirements and design, domain-specific and generic), with the aim of building tool-chains for analysis and verification, and reducing the complexity of software development.

In the field of model-driven development, models typically have strict formal semantics. Additionally, since abstract models often do not need to address specific details, they naturally exhibit better reusability.

Taking programming languages as an example, interfaces (in object-oriented programming languages) and function declarations (in procedure-oriented programming languages) themselves are abstractions, yet their presence allows for flexible and interchangeable implementations that serve as reusable components.

Therefore, model-driven development and component-based development are highly compatible. Conversely, once there is the involvement of strict formal semantics in the paradigm of CBSE, it naturally introduces the paradigm of model-driven development, as the relationship between components and the system itself corresponds to that of concreteness and abstraction.

Although CBSE and MDE are two different development paradigms, when formal semantics are introduced, they often converge in engineering practice. Therefore, while we discuss the challenges of component-based development, there will also be many concepts derived from the field of model-driven development in the subsequent sections.

3 Applications of Formal CBSE

The CBSE methodology has gained wide attention and application in the industry. However, the integration of formal methods and CBSE (referred as *Formal CBSE* in the following) is still limited to very specific domains. Some examples are introduced in this section.

3.1 Avionics and Railway Software

The avionics and railway software are representative sectors for safety-critical software systems. Software of this type are characterized by long development cycles, high cost of upgrades, and severe consequences in case of failures. Similar industries also include aerospace software, nuclear power control software, etc., but they will not be separately mentioned here due to their lower level of commercialization.

Avionics software. Various tools such as SCADE (Safety Critical Application Development Environment) [7] with its model checking capabilities, Simulink and Event-B have been applied in the development of the aircraft control and display systems. For example, [18] used SCADE and its formal verification component, the Design Verifier, to assess the design correctness of a sensor voter algorithm. The algorithm is representative of embedded software used in many aircraft systems.

Railway software. B-Method [6] and Event-B [42] have been successfully applied to modeling and code generation for subway systems. The work in [26] combines Event-B with a component-based reuse strategy realized with session types, and ensures global safety of railway interlocking systems by the local verification of entities. [32] used a subset of the SysML language for modeling of railway systems, automatically transformed the models to Event-B and finally imported the models into the RODIN platform [1] for formal verification.

3.2 Automobile Software

Vehicle software is a domain that requires a high demand for both component-based development and formal verification.

On one hand, due to the complex architecture and numerous devices (chips, peripherals, etc.) involved in the automotive industry, which have relatively stable accompanying software, it makes the field highly suitable for component-based development. The emergence of the AUTOSAR standard [21] further promotes the application of component-based development in the domain of vehicle software. The AUTOSAR standard divides vehicle software into three layers: Application Layer, Runtime Environment, and Basic Software. Each layer is further divided into several abstract components, with constraints defined in the standard. For example, the Basic Software is further divided into different layers such as Services, ECU Abstraction, Microcontroller Abstraction and Complex Drivers, and these layers are further divided into functional groups. Examples of Services include System, Memory and Communication Services. Such a layered architecture allows software and hardware suppliers to design components according to AUTOSAR specifications, ensuring that these components can be integrated together to form a complete system.

On the other hand, the automotive industry has stringent requirements for safety and reliability, creating ample opportunities for formal methods. In the functional safety standard ISO 26262, formal verification is considered an important added advantage. Currently, widely used CBSE tools that integrate formal verification capabilities include model-driven development tools like SCADE [7] and Ptolemy [10], which are based on synchronous data flow. Some modern tools targeting at automatic end-to-end compositional verification of automotive software system properties have been developed as well, such as EVA [14].

3.3 Industrial Manufacturing Software

Advanced industrial manufacturing, particularly in the modeling and analysis of control algorithms, is a typical application area for component-based development. Among this area, Matlab Simulink [19] and LabVIEW [9] are among the most general-purpose component-based development tools.

In this domain, formal methods find application mainly in the following two areas:

1. *Verification of control logics.* Large-scale industrial manufacturing processes often involve multiple concurrent control logics with complex interaction behaviors. Methods such as model checking can be employed to mitigate issues like deadlock and reduce failure rates, thereby enhancing overall stability of the manufacturing assembly line [47].
2. *Semantical analysis of control logics and algorithms.* This includes evaluating and analyzing the execution time [34] of control algorithms, stability analysis of controlled physical systems, optimization and synthesis of controllers [36], etc. By analyzing the results, algorithm optimization can be guided to improve efficiency and yield rates of the manufacturing assembly line.

It is worth noting that although Simulink has integrated basic formal verification capabilities such as Simulink Design Verifier [30] for quite some time, it is rare to see developers using them in practical applications. More often, developers choose domain-specific formal analysis tools based on the specific requirements of their scenarios, such as Coco Platform [15] and LSAT [35]. (Besides, there are much more domain-specific or proprietary formal analysis tools developed by large companies, but many of them are not publicly available.)

4 Challenges

After finishing the previous sections, attentive readers may now raise the following question:

> "The domains that have been listed above are quite familiar to the CBSE researchers. So, how to evaluate the applications of CBSE, especially equipped with formal methods, in the broader software industry?"

Objectively speaking, the CBSE methodology has already had a significant and far-reaching impact on the software industry. Derivatives such as cloud computing and micro-services architecture have emerged based on this idea [22, 28]. However, the widespread application of component-based software engineering combined with formal verification remains challenging. This section primarily focuses on the challenges in this aspect.

4.1 Hard to Keep Consistency Between Implementations and Models

Large-scale industrial software is typically developed through collaborative efforts, resulting in a fast-paced development cycle. If formal CBSE methods fail to cover the entire process ranging from design to code implementation (as mentioned in Section 2.1), potential inconsistencies could exist between the formal models and their corresponding implementations. For example,

- As the software evolves, new features can not be formalized using the formal model. This usually happens when the developers used a simpler formal model in order to simplify the verification.
- The benefits of formal verification have been claimed at the first time when the software is released to the customers. After that, continuous investment on formal verification does not produce comparable commercial value. Consequently, some teams choose to use formal methods in the prototype development and use traditional methods (or CBSE but without formal methods) to develop the released versions.

While the initial design models undergo comprehensive formal verification, ensuring the correctness of the software, we have observed that, as development speed accelerates, most development teams do not consistently invest manpower in model construction and verification. As a result, this eventually leads to disparities between the released software and the verified models in terms of formal semantics.

Unfortunately, this challenge is not exclusive to the application of formal verification in component-based software engineering. To the best of our knowledge, in most formal verification scenarios we are suffering from the same situation.

4.2 Lack of Life-cycle Maintainability

When aiming to address the aforementioned issue of semantic inconsistencies, it is natural to consider the introduction of code generation tools to directly generate executable code from formally verified models. Currently, many component-based engineering tools, whether or not equipped with formal proof capabilities, support this functionality.

However, it is unfortunate that additional engineering maintainability issues are introduced by code generation. For example,

- *Poor readability of generated implementations.* Automatically generated code often lacks readability. For developers such generated codes are hard to read and comprehend. The problem becomes even worse without automated test-case generation techniques.
- *Increased integration complexity.* Industrial software projects often involve the incorporation of third-party components. It is typically difficult to invoke these third-party components from the code generated by CBSE tools. In practice, developers often need to develop compatibility layers to invoke third-party components. Conversely, the introduction of such compatibility layers and third-party libraries significantly increases the risk of inconsistencies between the models and their implementations.
- *Lengthened error-fixing cycles.* If the code is automatically generated by a tool, it implies that developers cannot manually modify the code itself (as any changes would be overwritten in the next model modification). Consequently, when clients discover bugs in the software, even minor issues require modifications starting from the models, leading to a longer overall repair process and an inability to achieve prompt customer response.

On the other hand, if any bug is found in the code generation methodology itself, it is usually hard to motivate the tool developers (community or commercial entities) to fix the bug. For developers, an unavoidable fatal error may impede the entire CBSE development process, posing significant risks to commercial software.

4.3 Fragmented Requirements from Developers

One interesting aspect of component-based development is the introduction of various development roles during the development process, each with different tool requirements. For example, in the field of control algorithms, developers of specific components (such as PID controllers, various filters, etc.) are often more concerned about their runtime efficiency and may optimize them for specific hardware mechanisms. On the other hand, the algorithm designers, who combine these components, focus more on the performance of the algorithm itself, such as convergence speed and other parameters. They also want to validate the component's parameters to prevent potential configuration errors. Developers working on higher-level business logic focus on the correctness and stability of the entire business chain, such as the presence of deadlocks or the yield rate of the production line.

Fragmented requirements often lead to two possible outcomes. Either the blind accumulation of numerous features increases the learning cost of the tool significantly, diminishing its performance and usability, or different domains start customizing the tool themselves, leading to changes in data structures due to the addition of functionalities, resulting in siloed deliveries and a fragmented tool ecosystem. Both outcomes are detrimental to the promotion of component-based development.

4.4 Extra Learning Cost for Various Tools

Introducing new tools requires extra learning cost, and due to the inherent knowledge requirements of formal methods, the learning cost can be relatively high. What's worse is that fragmented domain demands often introduce a plethora of different tools. These tools may have different input languages, output formats, and user interfaces that may vary significantly in terms of user experience. In such cases, the learning cost incurred may offset the benefits.

5 Discussion

Based on the our academic research and practical experience in component-based development in the past decades, we present some open ideas in this section and hope that they can facilitate the broader application of component-based development methods based on the formal theories in the software development domain.

5.1 LLM-Aided Explanation of Codes and Exceptions

The idea aims to tackle the challenges in Section 4.1 and Section 4.2.

To the best of our knowledge, it is difficult to find mature solutions to maintain consistency between specification models and their implementations. One typical approach in industry is to extract models from black-box implementations through model extraction, e.g. using active learning [24,33]. However, such model extraction algorithms usually fail on complex implementations.

Program synthesis is the process of automatically generating a program or code snippet that is consistent with a given model, a formal specification or a natural language description. It involves a family of techniques such as deep reinforcement learning [44], constraint solving [16] and symbolic execution [38]. The goal of program synthesis is to minimize human intervention in the coding process, reduce program errors and complexity, and improve productivity.

Recently, the rapid evolution of Large-Language Models (LLMs [46]) brings us a new chance. Large-language models, literally, are artificial intelligence models that relies on huge parameters sets to achieve general-purpose language tasks,

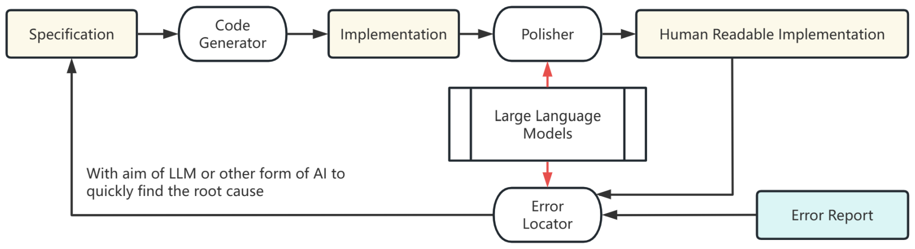
Specification
Code Generator
Implementation
Polisher
Human Readable Implementation
Large Language Models
With aim of LLM or other form of AI to quickly find the root cause
Error Locator
Error Report

Fig. 2. Integrating LLM in CBSE Workflow

such as language understanding and generation. Modern LLMs (e.g. ChatGPT, Llama2 [40], etc.) has been fine-tuned to perform program-related tasks [40] like code generation and explanation.

On one direction, the Github Copilot [4], built by Microsoft, has already proved its ability to understand and synthesize programs. On the other direction, there are already promising research on using LLMs in reversed engineering [31]. With the help of LLMs, it would be possible to increase the readability of generated code and help the developers to quickly locate the root cause in the models when bugs are reported. A possible integration of LLM into the CBSE workflow is shown in Fig. 2.

In this example the LLM is mainly used to help with:

– *Code polishing.* Automatically generated or synthesized code are usually lack of readability. However, with the background knowledge given (domain knowledge and the algorithm of code generation, etc.), the LLM is able to understand the behavior of the code, and then perform semantically equivalent code transformation.
– *Error locating.* When errors happen in generated programs, with the background knowledge (the same as above) a LLM could be able to help locating the root cause of the error in higher level specifications. So that the developer would be able to easily fix the error by updating the specification (instead of directly updating the generated code). This is helpful to keep a short iteration cycle even in software developed by formal CBSE methodology.

5.2 Decoupling Formal Specification from Verification

This idea aims to tackle the challenges in Section 4.1 and Section 4.2. Its core motivation is to enrich the formal model so that it can cover the specifications in various domains and abstraction levels.

Formal methods are mathematically rigorous techniques for the specification, development, analysis, and verification of software and hardware systems [11] where formal specification serves as the fundamental model for the remaining parts.

To formalize complex large-scale systems, we need an expressive formal model to capture their specifications. However, expressive models are usually harder to validate or verify (sometimes even impossible to verify). Consequently, some research tends to restrict the expressive power of their models to ensure the feasibility of formal verification. However, the limitation in model expressive power significantly reduces the usability of the tools, which often forces developers to resort to hacks like using Single-State State Machines (SSSM [45]) as a workaround. This compromises the maintainability of the model.

Decoupling formal specification from formal verification means using highly expressive formal semantics in component-based development tools and integrating them throughout the entire development life-cycle. For safety-critical and reliability-critical components, we can require developers to use a verifiable subset, while allowing more flexible and complex models in other parts. For other components, we can engage other light-weighted methods such as bounded model checking, model-driven testing, and static analysis to enhance software reliability.

5.3 Layered Modeling Through Domain-Specific Languages

This idea aims to tackle the challenge in Section 4.3. It also provides a technical basis for the idea discussed in Section 5.4.

Hiding unnecessary complexity can significantly improve development efficiency and reduce potential errors. This is an undeniable truth in the field of software development. Principles such as the Dependency Inversion Principle in software architecture are aimed at hiding unnecessary complexity. Considering the example of different roles mentioned in the previous section, if we hide formal semantics, analysis, and verification features that are irrelevant to each role during component-based development, it can alleviate the issues of fragmented domain requirements and high learning costs.

To achieve this goal, introducing Domain-Specific Languages (DSLs) is undoubtedly a good approach. Compared to general-purpose programming languages, DSLs are designed specifically for certain domains and provide simple and intuitive programming languages that are only used to describe domain-specific data objects or program behavior. For example, JSON and SQL# can be considered typical DSLs for data modeling and querying [25,29], respectively.

By using DSLs, we can layer the tool framework for component-based development, including a foundational model layer and a domain model layer. The foundational model layer consists of a more expressive formal semantic model, on which various common capabilities can be built, such as analysis, verification, and code generation. On top of the foundational model, there can be multiple modeling domains, each based on a DSL, including model transformation algorithms and a set of domain-specific functionalities. Through these algorithms, the domain models can be transformed into the foundational model, thereby reusing the capabilities of the foundational model, while also enabling the construction of domain-specific analysis capabilities. The foundational model is invisible to ordinary users. In this way, by sinking common capabilities and data structures as much as possible, this architecture can hide complexity while ensuring that fragmented domain requirements are met, as illustrated in Fig. 3.

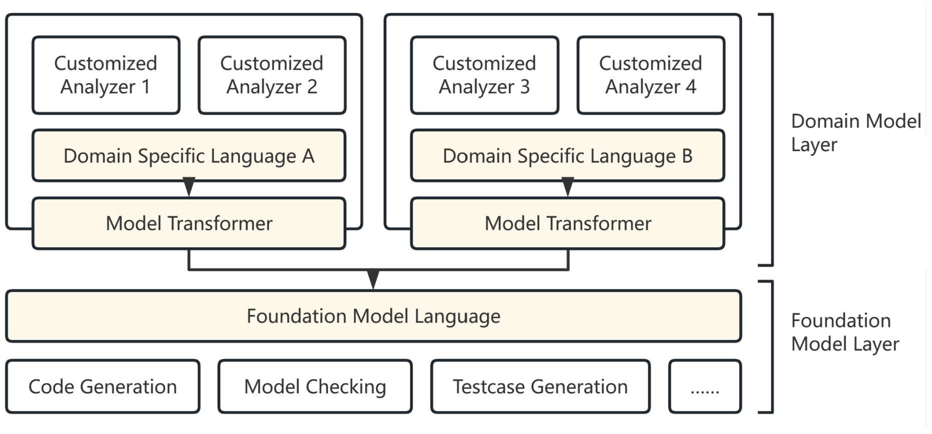

Fig. 3. Domain-specific Layered Modeling

There are quite a few previous works where domain-specific languages are applied to component-based development. For example, AADL (Architecture Analysis and Design Language) [20] supports *an annex mechanism* which can be used to extend its semantics [2]. However, such examples often focus on using DSLs to enhance the extensibility of the framework rather than hiding unnecessary complexity.

The primary purpose of using DSLs in these cases is to provide a specialized language that captures the domain-specific concepts and features required for component-based development. This allows developers to express their ideas and intentions in a more concise and natural way, improving productivity and reducing errors. DSLs can also provide higher-level abstractions that hide low-level details and technical complexities, making the development process more accessible to domain experts.

5.4 Providing Multi-Domain Integrated Development Environment

This idea aims to tackle the challenge in Section 4.4.

To reduce the extra learning effort required for users, a possible approach is to provide similar user experiences and logical workflows across different tools.

Taking Matlab as an example, we can use Simulink to build data flow models and Stateflow to construct state machines for controlling the data flow. This approach reduces the learning curve for users while enhancing their overall experience. SCADE takes it a step further by integrating requirements management (which can be considered as another domain, but not necessarily formal) and UI design into the same tool platform. This allows users to complete end-to-end design and implementation in certain scenarios using SCADE.

To make Formal CBSE tools applicable to a wider range of scenarios, we need a highly extensible framework that allows the introduction of various tool plug-ins through secondary development based on the foundation modeling language and domain-specific languages mentioned earlier. Here the foundation modeling language is a general (compared with the domain-specific languages above) and expressive modeling language, upon which we can build different domains.

There are already many open-source initiatives working in this direction, such as Jetbrains' Meta Programming System (MPS [12]) and Eclipse Xtext [8]. However, a common issue in practice is that these platforms focus too much on generality, resulting in a high learning curve for secondary development. In this case, a more promising solution to build specialized platforms with a limited level of scalability, which in turn makes it possible to add more domain-specific features to make it easier to use. The SCADE Suite also employs this idea where the instrument and physical simulation system give an intuitive view for controller designers.

6 Conclusion

This paper is not exactly a scientific research paper but rather resembles list of open topics to the researchers in the area of component-based software engineering, especially formal aspects of component software. We outline several typical scenarios in the industry where formal methods are used in conjunction with component-based development. It summarizes some of the challenges faced in the practical application of component-based development with formal methods and presents open viewpoints that may aid in the promotion of component-based development methods. The intention is for these challenges and perspectives to provide researchers and tool developers with insights into the current state of component-based development in industrial applications, helping them identify valuable research directions.

Acknowledgements This research was sponsored by the National Natural Science Foundation of China under Grant No. 62172019, and CCF-Huawei Populus Grove Fund.

References

1. Abrial, J., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-b. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
2. Ahmad, E., Dong, Y., Wang, S., Zhan, N., Zou, L.: Adding formal meanings to AADL with hybrid annex. In: Lanese, I., Madelaine, E. (eds.) Proceedings of FACS 2014. LNCS, vol. 8997, pp. 228–247. Springer (2014)

3. Arbab, F.: Coordination for component composition. In: Liu, Z., Barbosa, L.S. (eds.) Proceedings of the International Workshop on Formal Aspects of Component Software, FACS 2005, Macao, October 24-25, 2005. Electronic Notes in Theoretical Computer Science, vol. 160, pp. 15–40. Elsevier (2005)
4. Barke, S., James, M.B., Polikarpova, N.: Grounded copilot: How programmers interact with code-generating models. Proc. ACM Program. Lang. **7**(OOPSLA1), 85–111 (2023)
5. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
6. Behm, P., Benoit, P., Faivre, A., Meynadier, J.: Météor: A successful application of B in a large project. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) Proceedings of FM 1999. LNCS, vol. 1708, pp. 369–387. Springer (1999)
7. Berry, G.: Synchronous design and verification of critical embedded systems using SCADE and esterel. In: Leue, S., Merino, P. (eds.) Proceedings of FMICS 2007. LNCS, vol. 4916, p. 2. Springer (2007)
8. Bettini, L.: Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd (2016)
9. Bitter, R., Mohiuddin, T., Nawrocki, M.: LabVIEW: Advanced programming techniques. Crc Press (2006)
10. Buck, J.T., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: A framework for simulating and prototyping heterogenous systems. Int. J. Comput. Simul. **4**(2) (1994)
11. Butler, R.W.: What is formal methods? NASA LaRC Formal Methods Program (2001)
12. Campagne, F.: The MPS language workbench: volume I, vol. 1. Fabien Campagne (2014)
13. Chouali, S., Boukerche, A., Mostefaoui, A., Merzoug, M.A.: Ensuring the compatibility of autonomous electric vehicles components through a formal approach based on interaction protocols. IEEE Trans. Veh. Technol. **72**(2), 1530–1544 (2023)
14. Cimatti, A., Cristoforetti, L., Griggio, A., Tonetta, S., Corfini, S., Natale, M.D., Barrau, F.: EVA: a tool for the compositional verification of AUTOSAR models. In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 3–10. Springer (2023)
15. Cocotec.io: Cocotec: All systems go, https://cocotec.io/
16. Colón, M.: Schema-guided synthesis of imperative programs by constraint solving. In: Etalle, S. (ed.) Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3573, pp. 166–181. Springer (2004)
17. Criado, J., Asensio, J.A., Padilla, N., Iribarne, L.: Integrating cyber-physical systems in a component-based approach for smart homes. Sensors **18**(7), 2156 (2018)
18. Dajani-Brown, S., Cofer, D.D., Bouali, A.: Formal verification of an avionics sensor voter using SCADE. In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004,

Grenoble, France, September 22-24, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3253, pp. 5–20. Springer (2004)

19. Documentation, S.: Simulation and model-based design (2020), https://www.mathworks.com/products/simulink.html
20. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. SEI series in software engineering, Addison-Wesley (2012)
21. Fürst, S., Bechter, M.: Autosar for connected and autonomous vehicles: The autosar adaptive platform. In: Proceedings of DSN-w 2016. pp. 215–217. IEEE (2016)
22. Giacomo, G.D., Lenzerini, M., Leotta, F., Mecella, M.: From component-based architectures to microservices: A 25-years-long journey in designing and realizing service-based systems. In: Aiello, M., Bouguettaya, A., Tamburri, D.A., van den Heuvel, W. (eds.) Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future - Essays Dedicated to Michael Papazoglou on the Occasion of His 65th Birthday and His Retirement. Lecture Notes in Computer Science, vol. 12521, pp. 3–15. Springer (2021)
23. He, J., Li, X., Liu, Z.: Component-based software engineering. In: Hung, D.V., Wirsing, M. (eds.) Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3722, pp. 70–95. Springer (2005)
24. Hendriks, D., Aslam, K.: A systematic approach for interfacing component-based software with an active automata learning tool. In: Margaria, T., Steffen, B. (eds.) Proceedings of ISoLA 2022. LNCS, vol. 13702, pp. 216–236. Springer (2022)
25. Hu, Y., Jiang, H., Tang, H., Lin, X., Hu, Z.: Sql#: A language for maintainable and debuggable database queries. Int. J. Softw. Eng. Knowl. Eng. **33**(5), 619–649 (2023)
26. Kiss, T., Janosi-Rancz, K.T.: Developing railway interlocking systems with session types and event-b. In: 11th IEEE International Symposium on Applied Computational Intelligence and Informatics, SACI 2016, Timisoara, Romania, May 12-14, 2016. pp. 93–98. IEEE (2016)
27. Li, Y., Sun, M.: Component-based modeling in mediator. In: Proença, J., Lumpe, M. (eds.) Formal Aspects of Component Software - 14th International Conference, FACS 2017, Braga, Portugal, October 10-13, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10487, pp. 1–19. Springer (2017)
28. Liu, C., Yu, Q., Zhang, T., Guo, Z.: Component-based cloud computing service architecture for measurement system. In: 2013 IEEE International Conference on Green Computing and Communications (GreenCom) and IEEE Internet of Things (iThings) and IEEE Cyber, Physical and Social Computing (CPSCom), Beijing, China, August 20-23, 2013. pp. 1650–1655. IEEE (2013)
29. McNutt, A.M.: No grammar to rule them all: A survey of json-style dsls for visualization. IEEE Trans. Vis. Comput. Graph. **29**(1), 160–170 (2023)
30. Miranda, B., Masini, H., Reis, R.: Using simulink design verifier for automatic generation of requirements-based tests. In: Bjørner, N.S., de Boer, F.S. (eds.) Proceedings of FM 2015. LNCS, vol. 9109, pp. 601–604. Springer (2015)
31. Pearce, H., Tan, B., Krishnamurthy, P., Khorrami, F., Karri, R., Dolan-Gavitt, B.: Pop quiz! can a large language model help with reverse engineering? CoRR **abs/2202.01142** (2022), https://arxiv.org/abs/2202.01142
32. Salunkhe, S., Berglehner, R., Rasheeq, A.: Automatic transformation of sysml model to event-b model for railway CCS application. In: Raschke, A., Méry, D. (eds.) Rigorous State-Based Methods - 8th International Conference, ABZ 2021,

Ulm, Germany, June 9-11, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12709, pp. 143–149. Springer (2021)

33. Sanchez, L., Groote, J.F., Schiffelers, R.R.H.: Active learning of industrial software with data. In: Hojjat, H., Massink, M. (eds.) Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers. LNCS, vol. 11761, pp. 95–110. Springer (2019)
34. van der Sanden, B., Bastos, J., Voeten, J., Geilen, M., Reniers, M.A., Basten, T., Jacobs, J., Schiffelers, R.R.H.: Compositional specification of functionality and timing of manufacturing systems. In: Drechsler, R., Wille, R. (eds.) Proceedings of FDL 2016. pp. 1–8. IEEE (2016)
35. van der Sanden, B., Blankenstein, Y., Schiffelers, R.R.H., Voeten, J.: LSAT: specification and analysis of product logistics in flexible manufacturing systems. In: Proceedings of CASE 2021. pp. 1–8. IEEE (2021)
36. van der Sanden, B., Geilen, M., Reniers, M.A., Basten, T.: Partial-order reduction for supervisory controller synthesis. IEEE Trans. Autom. Control. **67**(2), 870–885 (2022)
37. Sifakis, J.: Component-based construction of real-time systems in BIP. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 33–34. Springer (2009)
38. Ströder, T.: Symbolic execution and program synthesis: a general methodology for software verification. Ph.D. thesis, RWTH Aachen University, Germany (2019)
39. Szyperski, C., Gruntz, D., Murer, S.: Component Software – Beyond Object-Oriented Programming, 2nd edition. Publishing House of Electronics Industry (2003)
40. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T.: Llama 2: Open foundation and fine-tuned chat models. CoRR **abs/2307.09288** (2023), https://doi.org/10.48550/arXiv.2307.09288
41. Trivedi, P., Zulkernine, F.H.: Componentry analysis of intelligent transportation systems in smart cities towards a connected future. In: 22nd IEEE International Conference on High Performance Computing and Communications; 18th IEEE International Conference on Smart City; 6th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2020, Yanuca Island, Cuvu, Fiji, December 14-16, 2020. pp. 1073–1079. IEEE (2020)
42. Wakrime, A.A., Ayed, R.B., Dutilleul, S.C., Ledru, Y., Idani, A.: Formalizing railway signaling system ERTMS/ETCS using uml/event-b. In: Abdelwahed, E.H., Bellatreche, L., Golfarelli, M., Méry, D., Ordonez, C. (eds.) Proceedings of MEDI 2018. LNCS, vol. 11163, pp. 321–330. Springer (2018)
43. Whitten, J.L., Bentley, L.D., Ho, T.I.: Systems analysis & design methods. Times Mirror/Mosby College Publishing (1986)

44. Yang, M., Zhang, D.: Deep reinforcement learning guided decision tree learning for program synthesis. In: Zhang, T., Xia, X., Novielli, N. (eds.) IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023. pp. 925–932. IEEE (2023)
45. Yang, N., Cuijpers, P.J.L., Schiffelers, R.R.H., Lukkien, J., Serebrenik, A.: Single-state state machines in model-driven software engineering: an exploratory study. Empir. Softw. Eng. **26**(6), 124 (2021)
46. Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J., Wen, J.: A survey of large language models. CoRR **abs/2303.18223** (2023)
47. Zheng, Z., Tian, J., Zhao, T.: Refining operation guidelines with model-checking-aided FRAM to improve manufacturing processes: a case study for aeroengine blade forging. Cogn. Technol. Work. **18**(4), 777–791 (2016)