FoMSESS-Jahrestreffen 2023

4-5.10.2023

Extended Abstracts

Die Fachgruppe FoMSESS[1] im GI-Fachbereich Sicherheit beschäftigt sich mit der Anwendung von Formalen Methoden und Software Engineering in der Entwicklung sicherer Systeme.

In ihren Jahrestreffen bietet die Fachgruppe die Möglichkeit, über aktuelle Forschungsarbeiten zu berichten und zu diskutieren und sich mit Gleichgesinnten zu vernetzen.

Das Jahrestreffen 2023 wurde online durchgeführt. Auch diesmal gelang es, zwei Nachmittage mit interessanten Vorträgen und lebhaften Diskussionen zu füllen. Die Vortragenden bekamen auch dieses Jahr die Möglichkeit, Extended Abstracts ihrer Beiträge zu verfassen, um diese auf der FoMSESS-Webseite zu veröffentlichen. Das Ergebnis sehen Sie gerade vor sich.

Viel Spaß beim Lesen!

Zoltan Mann, Dezember 2023

[1] https://fg-fomsess.gi.de

Towards the Formal Verification of Neural Networks[⋆]

Extended Abstract

Achim D. Brucker[0000−0002−6355−1200] and Amy Stell[0000−0003−0714−3269]

Department of Computer Science
University of Exeter
Exeter, UK
{a.brucker,as1343}@exeter.ac.uk

1 Introduction

Neural networks (i. e., deep learning) are being used successfully to solve classification problems, e. g., for detecting objects in images. At the same time, it is well known that neural networks are susceptible if small changes applied to their input result in misclassification. Situations in which such a slight input change, often hardly noticeable by a human expert, results in a misclassification are called adversarial attacks. Such adversarial attacks can be life-threatening if, for example, they occur in image classification systems used in autonomous cars or medical diagnosis.

This susceptibility of neural networks to small variations in inputs posses a challenge to the use of neural networks in safety-critical or security-critical applications. This is particularly true for high-integrity systems that need to undergo a formal verification process. Whereas formal verification of traditional programs usually rely on the existence of an implementation whose compliance to a specification can be verified, such an implementation or algorithm that precisely describes the behaviour does, per se, not exist for neural networks.

To address this challenge, we are developing a formal verification approach, using Isabelle/HOL, for neural networks [3, 2]. Our approach is based on a formal embedding of feedforward neural networks into Isabelle/HOL. On top of this formalisation, we are developing a verification approach that allows for the verification of safety and security properties of trained neural networks. We make use of the framework aspect of Isabelle for providing an import mechanism to automate our encoding for neural networks stored in a widely used exchange format, e. g., supported by TensorFlow [1].

Most current works in applying formal methods to neural networks are focusing on ReLU networks [4]. In our choice of using an expressive formalism (i. e., higher-order logic) in an interactive theorem prover, we will be able to also support networks with more complex activation functions. Furthermore, this choice

[⋆] This work was supported by the Engineering and Physical Sciences Research Council [grant number 670002170].

allow for a seamless integration of our approach for verifying neural networks into existing program verification approaches supported by Isabelle. This will, ultimately, allow for the end-to-end verification of complex systems that combine traditional programming with components based on neural networks.

Still, we see our work only as the beginning of a journey towards formally verified safety and correctness guarantees for critical systems employing ML/AI-based components. On a general level, there is further work required to improve the understanding of what it means for a neural network to be safe (and secure), and how to convert this into a formal specification. This discourse will, hopefully, result in further properties that can be used in formal verification, and will allow for a comparison amongst various formal approaches for the verification of neural networks. More specific to our approach, we plan to extend the classes of neural networks supported and also improve the degree of automation and the performance of the automated verification methods.

Availability. A more detailed report has been published in the Proceedings of the Formal Methods conference 2023 [3]. The formalisation and case studies are available to view on Zenodo [2]. The materials include both the Isabelle/HOL implementation and the detailed documentation generated by Isabelle.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., S. Corrado, G., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X.: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. `https://www.tensorflow.org/`.
2. Brucker, A.D., and Stell, A.: *Dataset: Feedforward Neural Network Verification in Isabelle/HOL.* Dec. 2022. DOI: `10.5281/zenodo.7418170`.
3. Brucker, A.D., and Stell, A.: Verifying Feedforward Neural Networks for Classification in Isabelle/HOL. In: Formal Methods (FM 2023). Ed. by M. Chechik, J.-P. Katoen and M. Leucker. Springer-Verlag, Lübeck, Germany (2023)
4. Singh, G., Gehr, T., Püschel, M., and Vechev, M.: Boosting robustness certification of neural networks. In: Learning Representations (2018)

Security Compliance: From Requirements to Runtime

Sven Peldszus

Ruhr University Bochum, Germany, sven.peldszus@rub.de

Abstract. Modern software-intensive systems often operate in a variety of critical environments, such as healthcare or autonomous driving, under constantly changing environmental conditions. In order to perform their diverse tasks under these conditions, such systems must constantly adapt their operating parameters and reconfigure their systems from time to time. Considering the essential security requirements inherent to these systems and the domains in which they operate, this tremendous dynamism exacerbates the problem of verifying the compliance of the implemented systems. In the end, it is inevitable to verify security at runtime. To do this, we must trace security requirements through the entire software development life cycle, from requirements to runtime, and verify compliance at each development phase. Current approaches to this problem rely on extensive manual effort or rigorous processes with detailed low-level traceability links, which are typically not feasible for large and complex systems such as robotic systems. One of the main obstacles is the significant differences in how security is addressed at different stages. To develop effective approaches applicable to complex, potentially distributed systems, we need to bridge this gap to systematically incorporate security checks both statically and dynamically at runtime.

1 Introduction

Today, a growing number of helpful but also security-critical systems, such as autonomous driving systems, are entering our lives. Frequently, new attacks on these systems are discovered, such as white stickers on the road that cause an autonomous vehicle to move into the oncoming lane [1]. Preventing all attacks is non-trivial due to the enormous complexity of modern software-intensive systems. While software-intensive systems have always consisted of parts that are not statically analyzable, such as dynamic class loading or Java reflection, machine learning models are now an integral part of many of them. For example, the Apollo autonomous driving system consists of 28 machine learning models [15]. As a result, developing secure software-intensive systems is becoming more challenging, and well-defined security requirements and proper encapsulation are more important than ever.

An important task in the secure development process is defining trust boundaries [16]. Anything within a trust boundary is assumed to be secure and can arbitrarily access the resources within the trust boundary. In this way, we can conceptually limit our security considerations to communication that crosses a trust boundary. For an autonomous car, such a security architecture is typically designed in an onion style [2]. At the trust boundaries, we need to identify and plan appropriate security features such as access control, secure communication, etc. to counter identified threats [16].

Then developers implement the security design while realizing the intended functionality of the system. During this process, all sorts of errors can occur, from simple bugs to potentially security-critical bugs to actual exploitable vulnerabilities. In the end, it is unavoidable to verify that the intended security design is implemented and that the system does not contain exploitable vulnerabilities. We need to verify that the intended security design is properly implemented and that all security-related assumptions are valid. Unfortunately, such verifications have to be performed mainly manually in the form of some code reviews by security experts. Even when vulnerabilities are discovered, there is often no guarantee that they have been fixed by the time a new product version is released [3].

A particular challenge is that not everyone is a security expert, but any design or programming activity can have security implications. An additional burden is that developers usually work primarily on the concrete implementation, while security experts work on high-level representations for planning the security design [7]. Overall, the different security-related tasks come with different requirements for how to handle them in the most effective way. In particular, at the implementation level, many static analyzers have been developed to help implement more secure code. However, these checks are usually locally restricted and often focus on individual code statements, such as checking for the secure use of a cryptographic API [5] or possible buffer overflows [6]. While these checks are important, a single detected code smell does not necessarily constitute a critical design flaw [12]. Furthermore, the relationship to the security design is missing, although this is essential for assessing exploitability and possible consequences.

Finally, such security checks, be they the static security checks outlined or manual security reviews, can only estimate what will happen at runtime. However, modern software-intensive systems are continuously adapt to their environment, e.g., an autonomous vehicle that reconfigures its sensors and driving behavior according to the weather as well as the driving situation, making it infeasible to verify each individual configuration [13]. Furthermore, this often involves statically not analyzable constructs, such as dynamically loaded classes realizing the different driving behaviors. Therefore, we also need dynamic checks for those properties that cannot be checked statically.

Overall, we need to continuously trace security requirements throughout the software development life cycle (SDLC) and verify that the system meets these security requirements at each phase. In the following, we outline our work on coupling security analysis throughout the different phases of the software development life cycle, and discuss what is needed for effective security compliance checking in the future.

2 Background and Motivation

Following the principle of security by design [4, 7], security requirements and corresponding measures are planned from the very beginning of software development. A common practice is to annotate design-time models with security requirements, aided by tool support, to plan which security requirements need to be met where, and to improve the security design [4, 18].

For example, secure data flow diagrams (SecDFDs) [18] extend data flow diagrams (DFDs), which are commonly used in threat modeling approaches such as STRIDE [16], with concrete security levels for communicated data and allow to plan secure data flow in terms of data processing contracts such as joining, forwarding, or encrypting data [18]. In the implementation, we need to ensure that the implemented data flows match the planned ones and that all data processing contracts are realized. To this end, we propose a semi-automated mapping between the SecDFDs and the corresponding implementation to verify that the implementation reflects the planned structure in terms of convergence, divergence, and absence of planned processes and data flows [14]. Based on this mapping, we can then perform concrete static checks that verify data processing contracts and automate the configuration of taint analyzers [17].

The concrete security design of the system can then be optimized at the system architecture level using UMLsec [4], which allows to define a more detailed security design [9], e.g., concerning the trust boundaries from threat modeling, by annotating UML design models with security requirements. In particular, we investigated the UMLsec secure dependency annotations, which allow a detailed structuring of applications into security levels concerning the confidentiality and integrity of data and services [9]. For the UML class diagrams used, we can create and maintain traceability by tightly coupling the class diagrams to the implementation and co-evolving them using a bi-directional transformation between them [7]. In this way, detailed correspondences between model elements and implementation elements are created and maintained. Based on these correspondences, we were able to statically verify that the implementation is compliant with the planned security levels and even incrementally re-verify this compliance in case of changes [8]. Since programming languages such as Java contain statically unanalyzable concepts such as dynamic class loading and Java reflection, such checks cannot cover all situations that may occur at runtime. To cover these cases, we provide a runtime monitor that dynamically detects violations of the UMLsec secure dependency security annotations at runtime [9]. This monitor even allows to take active countermeasures to protect the system and to adapt the design models to facilitate the investigation of prevented attacks.

While the techniques outlined are successful in verifying the compliance with the security design, they are specifically tailored to concrete design-time security aspects and rely on a detailed coupling between design models and code. These circumstances may limit the application of the techniques to domains where traceability is required by standards, such as the medical or automotive domains, but may be too much overhead for other domains.

3 Enhancing Security Compliance Checks

Our preliminary works have shown the principle feasibility of automated verification of the compliance of a system with its security design throughout its entire development life cycle. However, for a widespread application and further improvement, we have identified three aspects on which future research should focus.

3.1 Lightweight Security Traceability

To enable security compliance reviews, it is essential to establish traceability between artifacts created throughout the SDLC. Manually documenting and maintaining trace links is a huge effort and is therefore usually only done when required by an industry standard or regulatory requirement.

To this end, in our previous work we have used model transformation technologies to automatically co-evolve design models and their implementation [7, 9]. In this way, we are able to abstract the task of creating trace links from the users, while not only ensuring traceability at all times, but even avoiding divergence of the implementation from the models. The applied synchronization technology has proven to be effective for maintaining an analysis model that we used for verifying and executing refactorings [10, 11] and design flaw detection [12], but comes with drawbacks regarding the details of the design models. While the analysis model is close to the implementation it represents, for design models we have more abstraction. As a result, the design model has to grow in detail with the implementation, which may limit its practical applicability to model-based development processes where the growing model is linked to more abstract models. Nevertheless, the continuous tool-supported co-evolution of design models and code even supports agile working in principle.

To enable broader practical application, a more lightweight traceability of security requirements and features is needed. One solution could be to automatically create detailed trace models by monitoring developer activities and recording traces according to their workflow. In this way, a detailed trace model could be built from which semantically meaningful trace links could be derived. To avoid recording trace links altogether, one could try to focus more on the security features themselves, as they are contained in the design models and the implementation. Using design-time security languages such as UMLsec or SecDFD, security features are explicitly modeled at an abstract level, for which the more concrete counterparts must be identified in the implementation. There, many security aspects are explicitly accessible, such as where a cryptographic API is used. Together with code structures and manual feature annotations, this information could be lifted and compared to the security design, providing security traceability without requiring specific development processes.

3.2 Coupling Design-time and Implementation-time Security

After being able to trace security requirements between the various artifacts created throughout the software development lifecycle, we must select appropriate implementation-level security checks to verify the implementation against the security design. To do this, we must bridge a significant gap between design-time security and implementation-level security [7]. While the discussed design-time security techniques rely on coarse-grained specifications of security requirements, current static security analyzers focus on low-level security aspects, such as whether a cryptographic API is used securely. Currently, with the exception of the explicitly coupled checks discussed above, it is unclear which of the many existing security analysis tools can verify which aspects of the security design. We need to find a common understanding of security that is not specific to a single phase of the SDLC.

With such a common understanding, we can leverage the different aspects expressed in different phases to further improve the effectiveness of implementation-level security checks. The planned security design should serve as an additional source of information for tailoring implementation-level checks and obtaining more accurate analysis results. For example, we have shown that we can automatically configure implementation-level taint analysis, particularly with respect to where sensitive information enters and where it is intended to exit the system, based on the information available in design models.

3.3 Integrated Security Compliance Checks throughout the SDLC

For handling undecidable implementation aspects such as dynamic class loading, Java reflection, or today increasingly also the application of machine learning and reconfiguration, the introduction of safe guards that are introduced and checked at runtime, is a frequent proposal. However, our experiences with runtime monitoring show that monitoring everything does not scale well. The simple permission check concerning UMLsec secure dependency, has shown on short-running benchmarks on average a slowdown of factor 3.2 for Java applications [9]. Even on a long-running web application, where the overhead due to class loading has been relativized, the web application still shows 2ms higher response times. Still, dynamic security checks at runtime cannot be avoided entirely.

To keep the overhead of dynamic security checks reasonable, we have to integrate verification in the different phases of the development life cycle. Instead of checking everything dynamically, we have to check as much as possible statically and check dynamically, whether the assumptions of the static verification hold at runtime. Therefore, we have to identify the assumptions from previous phases that we have to verify and what is new in the phase that has to be verified. This way, we avoid expensive runtime verification and are even able to find and fix vulnerabilities early.

4 Conclusion

Verifying that software-intensive systems conform to their security design is essential to ensuring their security. Due to the significant difference in abstraction between design-time security and implementation-level security, this task currently has to be performed mainly manually, which makes it expensive, slow, and error-prone. Here, automation of security compliance checks has the potential to improve all three aspects, allowing for continuous security compliance checks. To this end, our previous work demonstrates the principle feasibility of such automated security compliance checks. However, to realize our vision of continuous automated security compliance checks, we still need to overcome the challenges outlined in this paper. In particular, we need to bridge the huge gap between design-time and implementation-level security. Therefore, we need to extract a common understanding of security that allows coupling these two phases, and develop more lightweight traceability approaches. In addition, emerging technologies such as machine learning and dynamically reconfigurable systems increase the problem of statically unanalyzable implementation aspects and create a higher demand for dynamic security checks. We must strive for a pipeline of integrated security checks that leverage the results of specifications and checks from previous life cycle phases to effectively verify security compliance and detect violations early.

References

1. Experimental Security Research of Tesla Autopilot. Tech. rep., Tencent Keen Security Lab (2019)
2. Chattopadhyay, A., Lam, K., Tavva, Y.: Autonomous Vehicle: Security by Design. Transactions on Intelligent Transportation Systems **22**(11), 7015–7029 (2021). https://doi.org/10.1109/TITS.2020.3000797
3. Gruber, D.: Modern Application Development Security. Tech. rep., Enterprise Strategy Group (2020), https://news.synopsys.com/2020-08-06-DevSecOps-Study-Finds-that-Nearly-Half-of-Organizations-Consciously-Deploy-Vulnerable-Applications-Due-to-Time-Pressures, visited 2023-10-31
4. Jürjens, J.: Secure systems development with UML. Springer (2005). https://doi.org/10.1007/B137706
5. Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., Kamath, R.: CogniCrypt: Supporting Developers in using Cryptography. In: International Conference on Automated Software Engineering (ASE). pp. 931–936 (2017). https://doi.org/10.1109/ASE.2017.8115707
6. Luo, P., Zou, D., Du, Y., Jin, H., Liu, C., Shen, J.: Static Detection of Real-World Buffer Overflow Induced by Loop. Computers & Security **89** (2020). https://doi.org/10.1016/J.COSE.2019.101616
7. Peldszus, S.: Security Compliance in Model-driven Development of Software Systems in Presence of Long-term Evolution and Variants. Ph.D. thesis, University of Koblenz and Landau (2022). https://doi.org/10.1007/978-3-658-37665-9
8. Peldszus, S., Bürger, J., Kehrer, T., Jürjens, J.: Ontology-driven Evolution of Software Security. Data & Knowledge Engineering (DKE) **134** (2021). https://doi.org/10.1016/J.DATAK.2021.101907
9. Peldszus, S., Bürger, J., Jürjens, J.: UMLsecRT: Reactive Security Monitoring of Java Applications with Round-Trip Engineering. Transactions on Software Engineering (TSE) (2023). https://doi.org/10.1109/TSE.2023.3326366
10. Peldszus, S., Kulcsár, G., Lochau, M.: A Solution to the Java Refactoring Case Study using eMoflon. In: Transformation Tool Contest (TTC). pp. 118–122 (2015)
11. Peldszus, S., Kulcsár, G., Lochau, M., Schulze, S.: Incremental Co-Evolution of Java Programs based on Bidirectional Graph Transformation. In: Principles and Practices of Programming on The Java Platform (PPPJ). pp. 138–151 (2015). https://doi.org/10.1145/2807426.2807438
12. Peldszus, S., Kulcsár, G., Lochau, M., Schulze, S.: Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching. In: International Conference on Automated Software Engineering (ASE) (2016). https://doi.org/10.1145/2970276.2970338
13. Peldszus, S., Strüber, D., Jürjens, J.: Model-based Security Analysis of Feature-oriented Software Product Lines. In: International Conference on Generative Programming: Concepts and Experiences (GPCE). pp. 93–106 (2018). https://doi.org/10.1145/3278122.3278126
14. Peldszus, S., Tuma, K., Strüber, D., Jürjens, J., Scandariato, R.: Secure Data-Flow Compliance Checks between Models and Code Based on Automated Mappings. In: International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 23–33 (2019). https://doi.org/10.1109/MODELS.2019.00-18
15. Peng, Z., Yang, J., Chen, T.P., Ma, L.: A First Look at the Integration of Machine Learning Models in Complex Autonomous Driving Systems: A Case Study on Apollo. In: Joint Meeting on the Foundations of Software Engineering (ESEC/FSE). pp. 1240–1250 (2020). https://doi.org/10.1145/3368089.3417063
16. Shostack, A.: Threat Modeling: Designing for Security. Wiley (2014)
17. Tuma, K., Peldszus, S., Strüber, D., Scandariato, R., Jürjens, J.: Checking Security Compliance between Models and Code. Software & Systems Modeling (SoSyM) **22**(1), 273–296 (2023). https://doi.org/10.1007/S10270-022-00991-5
18. Tuma, K., Scandariato, R., Balliu, M.: Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In: International Conference on Software Architecture (ICSA). pp. 191–200 (2019). https://doi.org/10.1109/ICSA.2019.00028

Rigorous Digital Engineering with Formal Methods
—
Extended Abstract for FoMSESS 2023

Frank Zeyda[0009−0009−4251−4740]

Independent Consultant (Galois, Inc.)
`<first name>.<surname>{at}gmail.com`
https://www.linkedin.com/in/frank-zeyda/

Abstract. This extended abstract accompanies a talk given by the author at the 2023 annual meeting of the *Fachgruppe Formale Methoden und Software Engineering für sichere Systeme*[1] (FoMSESS) in Germany. It is a reflection of the author's personal experience and views that have been shaped working as an independent consultant for safety/security-critical and trustworthy systems for Galois in the U.S. In particular, it defines the notion of *Rigorous Digital Engineering* (RDE) — a process and methodology that has been been shaped, adopted and refined by Galois over dozens of projects to design and produce hardware and software systems where the highest possible levels of assurance and trustworthiness are required.

1 What is RDE?

Rigorous Digital Engineer (RDE) is a methodology and process to create trustworthy, safe and secure hardware and software systems 'from the ground up'. It permeates the entire lifestyle of traditional hardware, software and firmware engineering, from domain engineering and requirements elicitation to models of product lines, architectural specifications, component and unit design, implementation, all the way down to assurance and maintenance. Each lifestyle stage is accompanied by meaningful models that support formal analysis, refinement, validation and verification activities, and ultimately aid and increase our understanding of the system while providing tangle evidence, e.g., that all relevant requirements have been captured, architectures satisfy safety and security properties, and implementations are correct with respect to their contractual specifications. RDE moreover ensures that that models at different levels of abstraction remain tightly connected to each other via refinement, both informally/semiformally and also in a strong mathematical sense where possible.

RDE has formed the basis of many projects at Galois, Inc. that require the highest degrees of assurance, targeting embedded encryption devices, avionics, space and defense, and secure voting systems — just to name a few. Despite this, the methodology and process of RDE has only recently been clearly articulated at Galois due to Joe Kiniry's vision and work *as a process in its own right* that is moreover adaptable and can cater for a large class of heterogeneous systems, from pure software to pure hardware and anything in between. The author has worked closely with Joe Kiniry over the last year to produce a set of training and teaching materials for RDE that we expect to be made available to a larger public audience in 2024, via a series of recorded video lectures. Irrespective of those plans, RDE in various incarnations and evolutionary forms has been trialed and tested at Galois over two decades now, and over and over again proving its value in producing systems that exhibited zero bugs or security issues while in operation. Potential issues are often spotted and eradicated early on in the lifecycle and design, thus resulting in substantial cost savings if those issues had to be addressed and patched late during development or even post deployment [9].

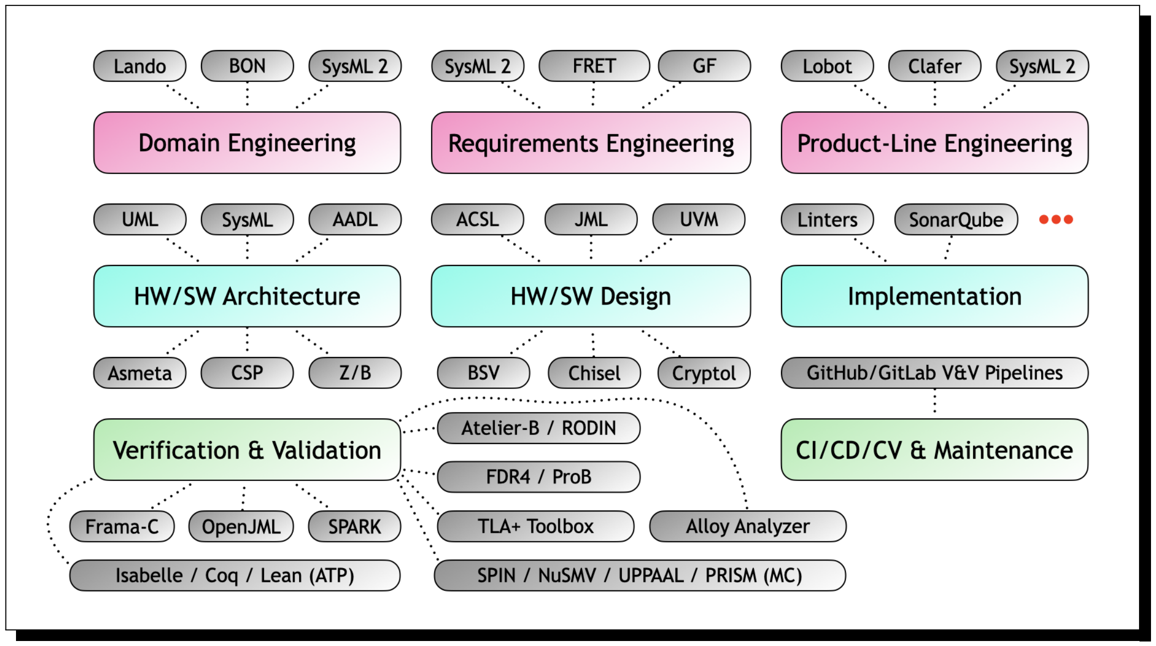

Fig. 1. A subset of common RDE languages and their analysis tools.

2 RDE and Model-Based Engineering

The key ingredients of the RDE approach and philosophy are models. But we do not admin just any model that comes or way, but have some requirements in terms of the utility and suitability of models. These are in summary:

– the use of (preferably executable) models (with preferably known fidelity) to
– rigorously, authentically describe things
– at various levels of abstraction
– such that the models relate to each other in well understood ways
– and the models refine to *bits* or *atoms*
– and thus all of this connects to software, hardware, and systems engineering
– and we use the models to provide assurance of various kinds for the product line or product or platform or system.

Fig. 1 presents an indicative subset of languages, modeling notations, and tools that are often employed in an RDE setting. Note that this diagram is not intended to be complete, and the particular notations and analysis techniques adopted in a particular project depend on the nature of the system, requirements of the client, desired level of assurance, and whether the project is software or hardware centric, or situated in between. Moreover, some of the mentioned languages and tools such as Lando, Lobot and Cryptol are Galois languages. Most of these are, however, available through open-sourced projects and licenses [1,3,5]. All models and assurance techniques in Fig. 1 are strongly couple with each other, through semiformal and formal notions of refinement where applicable.

The technology stacks supported thus far by the RDE methodology is far-reaching and extensive, and not restricted to tools and languages for 'hardcore' formal methods. For instance, it includes:

– many different kinds of programming languages (procedural, object-oriented, functional, hardware, logic, and mixed-model, such as C, C++, C#, Rust, Haskell, Java, Scala, Kotlin, Eiffel, Chisel, Bluespec SystemVerilog (BSV), SystemVerilog, and VHDL);

[1] Working group for formal methods and software engineering for safe systems

- specification and modeling languages (such as F*, ACSL, JML, Code Contracts, Alloy, Z, VDM, classical and Event-B, RAISE);
- architecture specification tools and languages (such as Cameo, Rhapsody, MagicDraw, OSATE, Visual Paradigm and UML, SysML, and AADL, respectively)
- integrated development environments (such as Eclipse, Visual Studio, Visual Studio Code, and IntelliJ IDEA)
- formal modeling and reasoning tools (such as Alloy, PVS, Coq, Isabelle, Lean, UPPAAL, CZT, Overture, Rodin, Frama-C, Logika, SAW, Ivy, TLA Toolbox, FDR4, NuSMV, BLAST, and SPIN)
- operating systems (RTOS, UNIX variants, seL4, and so on)
- spans systems, hardware (ASIC and FPGA-based), firmware, and software

One of the key challenges for RDE is indeed to find appropriate ways to use modeling, design and implementation languages in combination to extract the *maximum value* out of tools for formal analysis, validation and static/run-time verification. Issues that are often neglected in pure formal methods are given special attention in RDE, such as setting up a collaborative development environment (CDE) with CI/CD/CV linked to formal analysis tools early on in the project, performing domain engineering, and *consequently using* domain models as the basis for subsequent requirements specification, product line engineering (PLE) and architectural design. The RDE notion of PLE shall not just include concepts related to final software and hardware artifacts, but also security and threat models, assurance activities, and elements/artifacts of model-based static and dynamic architectures.

3 RDE and Formal Methods

Applying formal methods in RDE is about the practical application of formal methods to *all stages* of a system's lifecycle: process, methodology, domain, requirements, design, refinement, development, assurance, maintenance, and evolution. Moreover, we hold no bias in choice of formal method, tool, or technology: just choose the right tool for the job. RDE, especially in conjunction with industrial clients, often focuses on finding key places where small changes to the lifecycle have large impact, and nearly always hides formalism from the typical user à la Secret Ninja Formal Methods (SNFM) [8].

To give an example of the scope of RDE assurance, we consider the High Assurance Rigorous Digital Engineering for Nuclear Safety (HARDENS) [4] project which is publicly available on GitHub. The project considers the development of an Digital Instrumentation and Control (DI&C) system for Nuclear Power Plants, namely of a Reactor Trip System (RTS). The following assurance evidence has been produced for the final system implementation, see [7] for more details:

- critical components of the RTS are automatically synthesized from the Cryptol [3] model into both formally verifiable C implementations and formally verifiable System Verilog implementations;
- automatically generated C code and handwritten implementations of the same models are used to fulfill safety-critical redundancy and fault-tolerance requirements, and all of those implementations are formally verified both against their model, and verified against each other as being equivalent, using Frama-C and Galois's SAW [5] tool;
- the RISC-V CPU is formally verified against the RISC-V ISA specification using the Yosys open source verification tool;
- the RISC-V-based SoC is rigorously assured against the automatically generated end-to-end test bench;
- the formal requirements specified in FRET are formally verified for consistency, completeness, and realizability using SAT and SMT solvers;
- the refinement of these requirements into Cryptol properties are used as model validation theorems to rigorously check and formally verifying that the Cryptol model conforms to the requirements, the Cryptol model is used to automatically generate a component-level and
- end-to-end test bench (in C) for the entire system, and that test bench is executed on all digital twins and (soon) the full hardware implementation as well, and

– all models and assurance artifacts are traceable and sit in a semiformal refinement hierarchy that spans semiformal system specification written in precise natural language all of the way down for formally assured source code (in verifiable C), binaries, and hardware designs in System Verilog and Bluespec System Verilog (BSV).

Similar RDE example projects are available via the `https://github.com/GaloisInc` organization on GitLab, as well as implementations of RDE-specific languages and tools that those projects benefit from. Another notable one is the BESSPIN project (`https://github.com/GaloisInc/BESSPIN`) [2] funded by DARPA's SSITH program. BESSPIN aimed to develop (i) a set of quantitative metrics that would support practical measurement of security property compliance, enabling objective trade-offs between security and other system properties; (ii) a framework in which security architectures and their properties could be expressed and reasoned about, both at the abstract (model) level and the concrete (product) level; and (iii) a methodology in which metrics would drive decision making during the design of secure systems, particularly with regard to making informed, evidence-based hardware and firmware design trade-offs among security and other characteristics such as performance, power, and area. To achieve these goals, BESSPIN, like HARDENS relied heavily on RDE techniques and includes a product line of formally verified RISC-V soft-core CPUs developed to high assurance standards.

4 Conclusion

While RDE is built on strong principles, it is not a rigid and fixed methodology and is currently being evolved into several new directions, including the use of generative A.I. and large language models (LLMs) in order to ease the burden of writing, for instance, domain models or identifying security threats. Rather than using a single formal method to rule them all, RDE recommends making most of existing techniques that are available, and finding sound and meaningful ways to use these techniques in combination. This does raise some challenges and concerns in relating models and artifacts at different levels of abstraction throughout the lifecycle. We consider the use of Hoare and He's Unifying Theories of Programming [6] a key enabler to alleviate this issue in the future and provide a shared semantic foundation in which all the languages and modeling notations in Fig. 1 can live in harmony.

References

1. Galois, Inc. The BESSPIN Lando System Specification Sublanguage. GitHub project repository, July 2019. Available at: `https://github.com/GaloisInc/BESSPIN-Lando`.
2. Galois, Inc. Balancing the Evaluation of System Security Properties with Industrial Needs (BESSPIN). GitHub project repository, May 2021. Available at: `https://github.com/GaloisInc/BESSPIN`.
3. Galois, Inc. Cryptol: The Language of Cryptography (version 3). GitHub project repository, June 2023. Available at: `https://github.com/GaloisInc/cryptol`.
4. Galois, Inc. High Assurance Rigorous Digital Engineering for Nuclear Safety (HARDENS). GitHub project repository, February 2023. Available at: `https://github.com/GaloisInc/HARDENS`.
5. Galois, Inc. The Software Analysis Workbench (v1.0). GitHub project repository, June 2023. Available at: `https://github.com/GaloisInc/saw-script`.
6. Tony Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall Series in Computer Science. Prentice Hall, Upper Saddle River, NJ, USA, April 1998. Out of print now. But the book content is freely available at `http://www.unifyingtheories.org/`.
7. Joseph Kiniry, Alexander Bakst, Simon Hansen, Michal Podhradsky, and Andrew Bivin. The HARDENS Final Report. Technical report, Galois, Inc., January 2023. See: `https://github.com/GaloisInc/HARDENS/blob/54ac1d87267dff1311111603fa2925bc0197eb1d/docs/HARDENS_Final_Report_Jan_2023.pdf`.
8. Joseph R. Kiniry and Daniel M. Zimmerman. Secret Ninja Formal Methods. In *FM 2008: Formal Methods, Proceedings of the 15th International Symposium on Formal Methods*, volume 5014 of *LNCS*, pages 214–228. Springer, May 2008. DOI: 10.1007/978-3-540-68237-0_16.
9. Colin O'Halloran. Where Is the Value in a Program Verifier? In *Verified Software: Theories, Tools, Experiments, Proceedings of VSTTE 2008*, volume 5295 of *LNCS*, pages 255–262. Springer, October 2008. DOI: 10.1007/978-3-540-87873-5_21.