Towards Self-Explaining Digital Systems: A Design Methodology for the Next Generation

Rolf Drechsler[*†], Christoph Lüth[*†], Goerschwin Fey[‡], Tim Güneysu[*§]

[*]Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI)
Dept. Cyber-Physical Systems
28359 Bremen, Germany

[†]University of Bremen
Dept. Mathematics and Computer Science
28359 Bremen, Germany

[‡]Hamburg University of Technology (TUHH)
Institute of Embedded Systems
21071 Hamburg, Germany

[§]Ruhr-Universität Bochum
Dept. Electrical Engineering & Information Technology
44780 Bochum, Germany

***Abstract*—As digital systems get ever more complex, their behaviour may at times appear unfathomable. Users will only be prepared to accept this if they are convinced that the system does indeed work correctly. Thus, we argue the need for *self-explaining systems*: systems that are able to explain their behaviour, and the reasons for it. In this paper, we propose first steps towards a design methodology for such systems, and argue that beyond user acceptance, self-explanation also has other applications such as self-verification and reconfiguration. We propose a conceptual framework for self-explaining systems, discuss how to achieve completeness, and consider implementation aspects.**

I. Introduction

Every day we are surrounded by embedded and cyber-physical systems, some of them as visible as smartphones and desktop computers, some of them invisible in cars or trains. We have come to rely on them to work flawlessly, and get at the very least mildly annoyed if they cease to function as expected. But not every unexpected behaviour is due to a flaw in the system, sometimes users expectations can be wrong as well. In order for the user to accept the system behaviour, it is therefore crucial that the system is designed in such a way that the correct functioning is guaranteed, and moreover that the system can explain why it behaves the way it does.

Hence, user acceptance is a matter of correctness. Users will only be prepared to use the system — in particular, a safety-critical system — if they are convinced that the system has been designed correctly. A mathematical correctness proof supports this case, but is usually so complex as to require intensive study before it is believed. What we argue here is that in addition to proofs of correctness, systems should be able to *explain* why they work correctly, the extent to which they work correctly, and when they may not work correctly. This way, users can build an understanding of the system, and gain confidence in its correctness.

Research supported by BMBF grants SELFIE, (grant no. 01IW16001) and SecRec (grant no. 16K1S0606K).

How would such a *self-explaining system* work? As an example, take a service robot which is supposed to be working in close cooperation with a human worker. Figure 1 shows an example of such a setting: a human worker cooperating with a robot to produce gear boxes. In this setting, it is critical that the robot's two manipulation arms do not hit the human, and equally that the system assures the human worker that this is the case. Technically, this is solved by calculating a safety zone for the robot arms, LIDAR (light detection and ranging) and time-of-flight sensors capturing the posture of the worker and stopping the arms if the worker invades the safety zones. There is an element of self-explanation here — the system visualizes the safety zones (lower right in the picture), and displays them on a monitor representing a view of the algorithm to the human worker — but the system is not fully self-explaining yet. For that, it would *e.g.* have to be able to explain why the robot arm has stopped.

In our view, a basic prerequisite for self-explanation is that the system design is based on an abstract model of the system in the first place, otherwise the system operates on an ad-hoc basis and will not be able to give convincing explanations of its behaviour. To this end, we need to formalize the properties that we expect the system to have, and a model of the system itself. In the example above, the system model includes a three-dimensional model of the robot's arms and their movement, and based on that the safety zones, which are a basic element of the self-explanation in that setting.

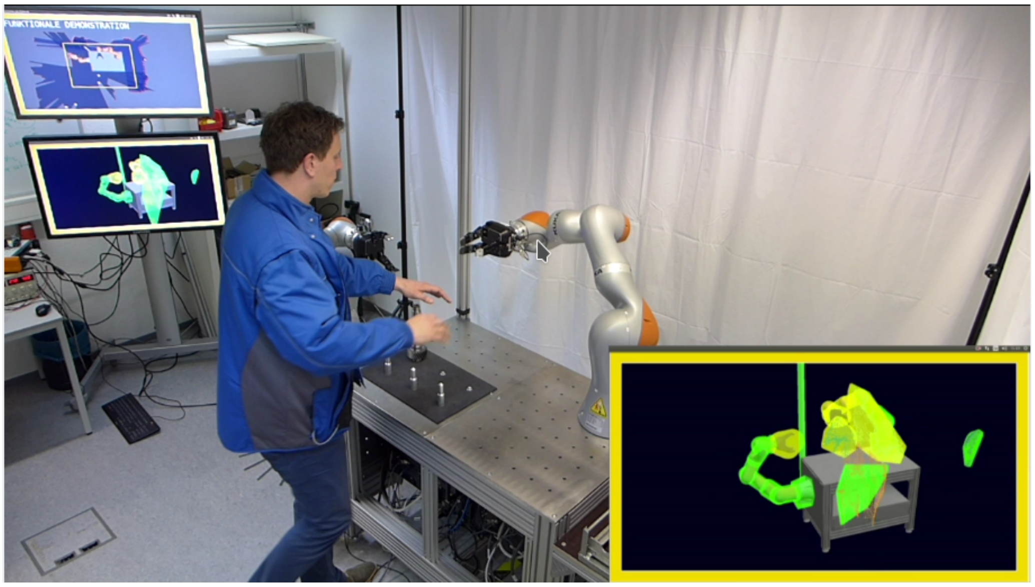
Fig. 1. Example of a self-explaining system: a human worker cooperating with a robot in a factory setting.

But self-explaining systems can do more than explain their actions: they may also verify their own correctness, or reconfigure themselves. Our vision are systems which verify their correctness at run-time in their respective deployment contexts, and are able to explain to users why or why not they are functioning the way they do — systems which are both functionally correct and user-adequate.

A. Related Work

Self-awareness [1] is another recent development making systems inherently adaptable — and thus, more difficult to understand. A self-aware system is one that has an internal representation of its own state, and can thus for example adapt its power consumption according to external and internal stimuli [2].

When the user's expectation of a system's behaviour differs from the system's actual behaviour because the user has a different internal model of the system, this is called mode confusion. There has been a lot of work on mode confusion, of which most related to our approach is work on using model checking to detect mode confusion [3].

For AI systems, *i.e.* systems based on techniques of machine learning and neural nets, verification and self-explanation is already a research trend [4]. Because of the subsymbolic nature of these techniques, obtaining a reconstructible explanation of their decisions is a challenge in itself, as is the verification of their correctness [5]. Both machine learning and neural nets work by heuristic algorithms where configurations (classifiers, weights of the net) are set by training, that is we configure the system until its output is conformant to our expectations, but we do not construct the weights of the net or the classifier from a mathematical model of the specification of the algorithm. But if we cannot really reconstruct why a neural net or classifier works, how can we prove it? And how can we explain its output?

Proof carrying code [6] is a technique where programs are annotated with a proof of the desired safety or security properties, such that the proof can be checked easily before the program is run. There are various approaches to proof-carrying code, most of which hinge on the syntax and semantics of the proof language (we can use formats based on type theory, propositional, first-order or higher-order logic, or proof tactics — the simpler the language, the easier it is to write a proof checker, but the lengthier the proofs).

Decision procedures are typically very complex algorithms that are used to certify the integrity of systems, *e.g.* [7], [8]. This pairing of complexity and certification early stimulated the search for understanding the verdict provided by a decision procedure. Typically, this verdict either yields a feasible solution to some task, *e.g.* a satisfying assignment in case of Boolean satisfiability (SAT) solver, or denies the existence of any solution at all, *e.g.*, unsatisfiability in case of SAT solving. A feasible solution can easily be checked as a recipe to solve some task. However, understanding why some task cannot be solved is more difficult. Here, proofs provide a natural explanation why something is not possible that can efficiently be produced for SAT solvers [9], [10].

B. Our Contribution

The major contribution of this paper is to sketch the requirements for a design flow for self-explaining systems, by adapting the usual design flow of digital systems such that information needed at runtime does not get discarded but is preserved. We will show two major applications of this paradigm which we have developed in our current work, namely self-verifying systems and self-reconfiguring systems. Self-verification aims to improve the functional safety of

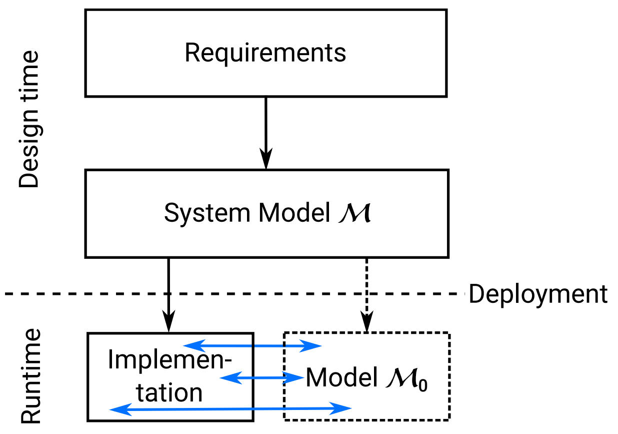

Fig. 2. Design flow for self-explaining systems: the system at runtime needs a representation $\mathcal{M}_0$ of the system model $\mathcal{M}$.

the system by showing its correctness at runtime; and self-reconfiguration aims to improve the security of a system by making it a moving target for side-channel attacks.

Following on, we propose a conceptual framework for self-explanation that yields layered explanations providing details where necessary, but keeping explanations understandable at the same time. We discuss how to achieve completeness for self-explanation, and how self-explanation can support verification.

This paper is structured as follows: we first discuss the requirements for the new design flow, then consider self-verification and reconfiguration, before discussing self-explanation in detail.

II. Design for Self-Verification and Self-Explanation

The usual design flow starts with analyzing the requirements of the system, typically first in natural language, and then formalized in a specification language such as SysML/OCL [11], [12], [13]. At this level, we describe the structure of the system, and the functional and non-functional requirements it needs to satisfy *in toto* (for example, safety requirements). The next step is an executable system model (in languages such as SystemC [14], [15], [16] or CLaSH [17]) which we can use for emulation, proof, or generating an implementation. Each of the design steps from the initial specification to the final implementation is complemented by a corresponding verification activity [18], using any of the available verification techniques such as testing, static analysis, theorem proving or model checking [19]. However, once the system is developed and deployed, the model is discarded; it is not represented anymore during runtime.

For self-explaining systems, we propose a design flow which carries a representation of the abstract model and verification conditions into runtime, to allow referring back to it. Figure 2 sketches this design flow. We have the same first steps as in traditional design flows: start with abstract requirements, formalize them, then find or derive an abstract system model $\mathcal{M}$. However, we carry an abstraction $\mathcal{M}_0$ of the model into runtime, in such a way that its elements (actions or properties of the system) can be linked to specific functionalities of the running system. This allows us, at runtime, to elucidate on the system's behaviour — to explain why a functionality is safe, or why a functionality is not available. It also allows us, at a more basic level, to verify the safety at run-time. However, it is important that we construct an *abstraction* of the model at runtime, as the full model with proofs would simply be too large.

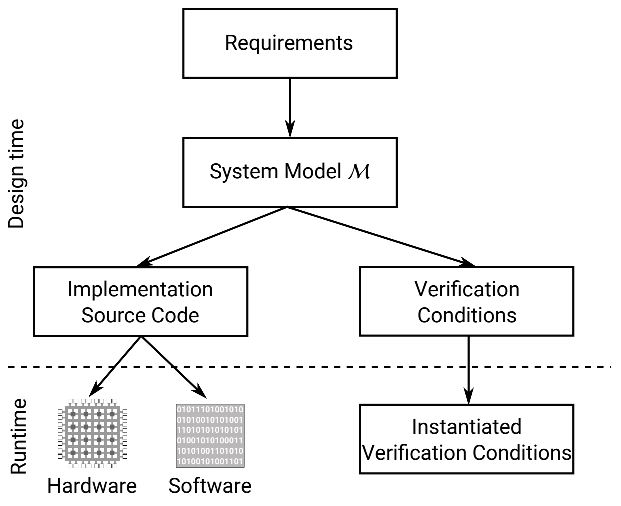
Requirements
System Model $\mathcal{M}$
Design time
Implementation Source Code
Verification Conditions
Runtime
Hardware
Software
Instantiated Verification Conditions

Fig. 3. Design flow for self-verification: at runtime, many parameters of the verification conditions can be instantiated, making proving them easier.

III. Safety by Self-Verification

In the usual development models verification has to end before the system is deployed. Hence, due to time-to-market constraints verification often has to be terminated before it can be complete — leading to potentially faulty systems. By keeping a suitable representation of the formal system model in the deployed system, we can continue verification after deployment. This has three main benefits: we have more time, because verification does not have to be finished by the time the system is deployed; we have more resources, because the computing power of all deployed systems can be harnessed; and we have more information, because once deployed and operational, verification can take its context into account, reducing the system state space drastically [20], [21].

Figure 3 sketches an instantiation of the design flow from Figure 2 for self-verification. The aspect of the formal model of the system which is needed at run time are the verification conditions, the proofs that the implementation does indeed satisfy the formalized requirements. For self-verification, we *instantiate* some of the parameters of the system which are unknown at design time, but which become known and change only rarely or not at all during runtime. This reduces the state space of the system, and makes verification viable, even under the resource constraints of an embedded or cyber-physical system.

We have realized such a design flow in a prototypical system design [22], which we will describe in the following. The design starts with a formal model of the system, which is given in SysML (specifically, SysML block definition diagrams) with OCL constraints. The OCL constraints can be invariants or pre/postconditions for operations declared in the block definition diagrams. To formally verify them, we convert them into a formula in conjunctive normal form (CNF), which can be proven using a SAT checker.

The deployed system will usually have less computing power and memory than the servers typically used for design, so the question is how much of the design process we can move into the deployed system. We decided to move the final step — the SAT checking of the CNF formula — into runtime. SAT checking is where the exponential blowup comes from (SAT checking is famously NP-complete). As mentioned above we can instantiate certain variables in the CNF once they are known at runtime, allowing an exponential *reduction* of the system state space.

To elaborate on this: modern electronic systems are usually designed to be very versatile so they can be configured to a variety of deployment contexts. For example, a smart home controller could be configured to have a number of sensor inputs (such as brightness, temperature or presence sensors), connected to a number of outputs (controlling lights, heating, doors, or blinds) in a highly configurable number of ways (*e.g.* if the brightness sensor outside drops below a certain value turn on lights in all rooms where the presence sensors indicate inhabitants are present). In this setting, all configurations are unknown at design time, but can be considered as fixed at run-time, in contrast to the actual sensor (and actuator) values, which may change rapidly. Hence, if we instantiate the configurations by replacing the corresponding variables in the CNF with constants, the resulting CNF can be checked by a light-weight version of MiniSAT [23] at run-time. In a simple experiment with one light sensor and one light controller, we achieve a reduction of the system space from 2^{33} to 2^{9} (a reduction by a factor of 2^{24}; more realistic applications achieve reductions in the order of 2^{13056} to 2^{2816} (factor of 2^{10240}) [22]. In further experiments, specifications which were not provable in full generality at design time could be proven at runtime, on a realistic embedded system, in times ranging between less than one second up to seven minutes.

IV. Security by Reconfiguration

In addition to functional correctness of self-explanatory systems, modern applications and use-cases often come with strong security requirements that need to integrate efficient countermeasures to thwart a large number of directed attacks. Usually these requirements involve the inclusion of cryptographic schemes and powerful security subsystems. With the increasing complexity of modern digital systems, however, it becomes a major challenge for the designer to build a secure system that does not yield options for attacks. Due to the asymmetry of the attacking and defending party, a single failure in the security subsystem usually enables the attacker to completely break the system by extracting secrets or to disabling the security system.

However, an attacker still needs to perform extensive profiling of the system in order to identify and exploit the vulnerabilities, *e.g.*, by manipulating the physical execution environment of the system. In this regard, common attack vectors for an attacker with physical access to the device are known to be side-channel analysis (SCA) [24] or fault-injection attacks (FIA) [25]. Passive SCA attacks exploit the leakage of secret information that is emitted during the execution of a, *e.g.*, cryptographic operation. Possible sources for leakage are power traces, electro-magnetic emanations or acoustic and photonic emissions that can be easily observed if the attacker has physical access to the device. With the active injection of faults during the security-critical operation, the attacker manipulates the operating environment or the device itself so that the device either cannot finish the security-critical operation correctly or reveals critical information about the secret from the faulty output. Besides permanent faults (*e.g.*, by specifically destroying cryptographic features or parts of it), the injection of transient faults disturbs the operation of the device for a short moment only.

It is still a subject of research which combination of countermeasures is optimal to achieve protection against all types of such physical attacks. Nevertheless, a fundamental problem for the security designer is the basic limitation of *static* hardware that can easily be analyzed by an attacker by a static profiling process. Generally this assumption is inevitable since all gates and routes are assigned to immovable locations on the chip. The capabilities of reconfigurable platforms that can modify and swap circuits in place, as with field-programmable gate arrays (FPGA), provide the potential for new powerful countermeasures against static profiling attacks on the static implementation. Any type of physical correlation for leaked information that is gathered by an attacker will become significantly more difficult when the hardware dynamically evolves during or between security-critical operations. In this context, the continuous dynamic reconfiguration of a security hardware implementation can generate powerful countermeasures against side-channel analysis and fault injection attacks. Likewise, reverse-engineering attacks will become more complex due to the self-updating circuits and components that require a runtime analysis of the hardware components.

Technically speaking, the integration of dynamically evolving circuits for improved hardware security involves the following major challenges:

- Identification of algorithmic components in security systems that can be efficiently placed (and re-mapped) on different elements using local and partial reconfiguration.
- Specification of efficient randomized isomorphic mappings between the identified reconfigurable hardware elements.
- Verification and protection of the function-invariant reconfiguration process of dynamic security components during runtime (by a reconfiguration controller).

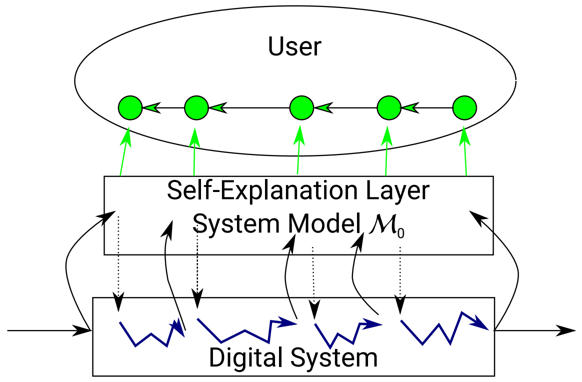

Fig. 4. Self-explanation at work: the self-explanation layer uses the abstract system model $\mathcal{M}_0$ to link user-observable actions to the state transitions, and thus construct explanations of their causes and effects.

Note that the protection of the dynamic reconfiguration process is of utmost importance since an exploitable vulnerability in the reconfiguration controller might render the dynamic reconfiguration process unusable. This, however, is likely to lead to a static behaviour of the security circuit. Hence, additional tamper protection needs to be in place to guarantee the correct operation of the reconfiguration process at any time.

V. Towards Self-Explaining Systems

Self-verification of a system relies on a certain level of self-introspection. Reconfiguration makes a system and the actions executed by a system quite intransparent to users and even designers. Self-explanation is able to provide self-introspection for a system and also helps to understand why the system behaves the way it does. Depending on whether the system uses a model of its environment, *e.g.*, a robot that expects an increase in light intensity when moving towards a light source, or not, *e.g.*, the robot simply moves towards the light source, the system can even explain differences from expectations.

In general explaining observations is difficult as causes for events are not necessarily unambiguous [26] and even restoring a system-wide time reference causes some overhead [27]. However, during runtime every detail of the system is determined and can be used for a precise unambiguous explanation of the actual behaviour.

A. A Technical Approach

We propose the approach illustrated by Figure 4. The system itself is extended by a layer for self-explanation that supports users such as end-users or designers in understanding the actions executed by the system. The layer for self-explanation includes the abstracted model $\mathcal{M}_0$ of the actual system. Here abstraction means that the system's data has a simplified representation, *e.g.*, the "motor turns forward with high speed" instead of storing the exact values of all signals driving a motor. This reduces the cost for explanation and makes explanations easier understandable.

In more detail, we view at the system executing a series of events, where an event denotes a certain action at a specific point in time. An action may be internal (not externally observable) when updating the state of the system, or externally observable, *e.g.*, by using an actuator. Typically, a system processes input, updates its internal state, and produces output, as indicated by the blue zig-zag arrows sketching the data path in Figure 4. The self-explanation layer observes these events, indicated by black bold arrows, and provides an explanation with each event. Such an explanation may refer to previous events that triggered the current one, and to the specification of the system that requires certain behaviour. In order to do so, the self-explanation layer constructs and stores cause-effect chains for events from the abstracted system model $\mathcal{M}_0$. This way, the self-explanation layer references events and their explanation to causes of subsequent events, as indicated by dotted black arrows. Users can access the cause-effect chains stored in the self-explanation layer for understanding the causes of actions they observe (indicated by green dots and arrows).

In principle, any action of the deterministic system must be explainable in terms of the input data and specification. However, a limited memory capacity in the self-explanation layer limits the historical data to the most recent history. Including an abstraction of the system's state into the system model $\mathcal{M}_0$ for the self-explanation layer alleviates the effect of limited historical data. Cause-effect chains can be cut, removing the explanation how the system arrived at a certain state but showing which state was responsible.

B. Different Perspectives and Usage Scenarios

The precise small-step execution of a program or hardware unit can in principle be considered as an explanation for the actions taken at the end of the execution. However, this precision in explanation definitely overloads users with irrelevant details.

We propose to consider several types of explanations supporting different perspectives and various levels of granularity:

- *User-understandable* explanations refer to input data and the user-visible conceptual state that triggers an action.
- *Specification-defined* explanations remove all references to input data and the system state to reduce the explanation to the specification relevant for certain actions.
- *Architectural* explanations show which modules contributed to triggering an action.
- *Program-level* explanations justify the execution paths taken by a program or hardware units.

At least user-understandable explanations must be available during runtime, and be generated by the system itself. Other types of explanations, *e.g.* program-level explanations that are explaining low-level details, may only be needed during design. However, consistency between the explanations and the system must be guaranteed.

C. Completeness

Whether explanations are complete must be assessed in the design of the system. Observable actions must be specified precisely. Whether any observable action is associated to an explanation and whether the related causes are appropriately justified can be analyzed automatically. This also serves as a basis for verification. Instead of referring to low-level aspects of the system, verification can be lifted to the abstract system model of the self-explanation layer, provided that an appropriate mapping between this model and the actual system has been guaranteed. Such completeness can be checked with approaches similar to [28]. Inferring causes for events is similar to extracting behaviour from a given design, *e.g.*, using an approach similar to [29].

VI. Conclusions and Outlook

In this paper, we have proposed first steps towards a methodology for designing self-explaining systems. We argue that the complexity of cyber-physical systems can make their behaviour rather intransparent to the user, impeding user acceptance and endangering safety. By giving systems the ability to explain their actions, users will gain confidence in these systems' correctness.

Self-explanation requires a formal specification and model of the system, an abstraction of which is needed at run-time to construct explanations for the systems behaviour, based on cause-effect chains of actions constructed from the system specification. We have shown that in addition, the self-referential approach needed for self-explanation can also have other applications, such as self-verification (where the system shows its own correctness at runtime) or reconfiguration (where the system becomes a moving target to specific types of external attacks).

We have only sketched first steps here, which require more work as described above. Adding cognitive abilities to the system, and functionalities based on subsymbolic AI techniques such as neural nets and machine learning, will provide additional challenges that we will have to master. However, we are confident that ultimately systems without self-explanation capabilities will not be accepted by users any more.

References

[1] A. Jantsch, N. Dutt, and A. M. Rahmani, "Self-awareness in systems on chip — a survey," *IEEE Design & Test*, vol. 34, no. 6, pp. 8– 26, November/December 2017.

[2] J. J. Davis, J. M. Levine, E. A. Stott, E. Hung, P. Y. K. Chung, and G. A. Constantinides, "KOCL: Power self-awareness for arbitrary FPGA-SoC-accelerated OpenCL applications," *IEEE Design & Test*, vol. 34, no. 6, pp. 36– 45, November/December 2017.

[3] J. Bredereke and A. Lankenau, "A rigorous view of mode confusion," in *Computer Safety, Reliability and Security*, S. Anderson, M. Felici, and S. Bologna, Eds. Springer, 2002, pp. 19–31.

[4] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks," *Digital Signal Processing*, vol. 73, pp. 1 – 15, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1051200417302385

[5] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *Computer Aided Verification CAV 2017*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10426. Springer, 2017, pp. 3–29. [Online]. Available: https://doi.org/10.1007/978-3-319-63387-9_1

[6] G. C. Necula and P. Lee, "Safe, untrusted agents using proof-carrying code," in *Mobile Agents and Security*, G. Vigna, Ed. Springer Verlag, 1998, pp. 61–91. [Online]. Available: https://doi.org/10.1007/3-540-68671-1_5

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

[8] R. Drechsler, S. Eggersglüß, G. Fey, A. Glowatz, F. Hapke, J. Schloeffel, and D. Tille, "On acceleration of SAT-based ATPG for industrial designs," *IEEE Trans. on CAD*, vol. 27, no. 7, pp. 1329–1333, 2008.

[9] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Design, Automation and Test in Europe*, 2003, pp. 886–891.

[10] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications," in *Design, Automation and Test in Europe*, 2003, pp. 880–885.

[11] Object Management Group, "Object Constraint Language," OMG, Tech. Rep. formal/2014-02-03, 2012.

[12] ——, "OMG Systems Modeling Language (OMG SysML)," OMG, Tech. Rep. formal/2015-06-04, 2015.

[13] R. Drechsler, M. Soeken, and R. Wille, "Formal Specification Level: Towards verification-driven design based on natural language processing," in *Forum on Specfication and Design Languages*, 2012, pp. 53–58.

[14] "IEEE Standard SystemC Language Reference Manual," IEEE Computer Society, 2006.

[15] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[16] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Dordrecht, Heidelberg, London, New York: Springer, Dec. 2009.

[17] C. Baaij, M. Kooijman, J. Kuper, W. Boeijink, and M. Gerards, *CLaSH: Structural Descriptions of Synchronous Hardware using Haskell*. IEEE Computer Society, 9 2010, pp. 714–721, eemcs-eprint-18376.

[18] H. M. Le, D. Große, V. Herdt, and R. Drechsler, "Verifying SystemC using an intermediate verification language and symbolic simulation," in *Design Automation Conference*, vol. 50, 2013, pp. 116:1–116:6.

[19] N. Przigoda, M. Soeken, R. Wille, and R. Drechsler, "Verifying the structure and behavior in UML/OCL models using satisfiability solvers," *IET Cyper-Phys. Syst.: Theory & Appl.*, vol. 1, no. 1, pp. 49–59, 2016. [Online]. Available: https://doi.org/10.1049/iet-cps.2016.0022

[20] R. Drechsler, H. M. Le, and M. Soeken, "Self-verification as the key technology for next generation electronic systems," in *Symposium on Integrated Circuits and System Design*, 2014.

[21] R. Drechsler, M. Fränzle, and R. Wille, "Envisioning self-verification of electronic systems," in *Int'l Symp. on Reconfigurable Communication-centric Systems-on-Chip*, 2015, pp. 1–6.

[22] C. Lüth, M. Ring, and R. Drechsler, "Towards a methodology for self-verification," in *6th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO 2017)*, 2017.

[23] F. Bornebusch, R. Wille, and R. Drechsler, "Towards lightweight satisfiability solvers for self-verification," in *2017 7th International Symposium on Embedded Computing and System Design (ISED)*, 2017, pp. 1–5.

[24] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, ser. Lecture Notes in Computer Science, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 388–397. [Online]. Available: https://doi.org/10.1007/3-540-48405-1_25

[25] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, ser. Lecture Notes in Computer Science, B. S. K. Jr., Ed., vol. 1294. Springer, 1997, pp. 513–525. [Online]. Available: https://doi.org/10.1007/BFb0052259

[26] D. Lewis, "Causation," *Journal of Philosophy*, vol. 70, no. 17, pp. 556–567, 1973.

[27] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[28] D. Große, U. Kühne, and R. Drechsler, "Analyzing functional coverage in bounded model checking," *IEEE Trans. on CAD*, vol. 27, no. 7, pp. 1305–1314, 2008.

[29] J. Malburg, T. Flenker, and G. Fey, "Property mining using dynamic dependency graphs," in *ASP Design Automation Conf.*, 2017, pp. 244–250.