Ensuring Correctness of Next Generation Devices: From Reconfigurable to Self-Learning Systems

(Invited Paper)

Rolf Drechsler Daniel Große
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{drechsle, grosse}@informatik.uni-bremen.de

***Abstract*—Nowadays electronic systems are small yet powerful and embedded into their environment. They are adapting to changes and often operate autonomously. These systems have reached a level of complexity that opens up new application areas, like autonomous driving or self-learning robotics, but at the same time strains the existing design flows in system development. For two concrete examples we show the importance of ensuring the correctness: verification of robotic plans, and verified partial reconfiguration as part of a reconfiguration-based countermeasure against side-channel attacks.**

I. Introduction

Based on the steady progress in silicon technology endless possibilities for applying electronic emerged. For instance high-performance computing allows for new applications ranging from entertainment, medicine, robotics, AI to the frontiers of scientific research. Another class of systems are embedded devices which enabled the *Internet of Things* (IoT) and paved the way for the evolution to industry 4.0. Of key importance in this context are two properties for such kinds of systems: *adaptivity* and *autonomy*. In essence, more and more electronic systems will emerge whose actions are not explicitly programmed, but these systems have to make decisions in situations which are previously unknown, and finally they have to learn from each manged situation.

From a design perspective, realizing such next generation devices requires self-learning and (hardware) reconfiguration. However, *verification*, i.e. to ensure the correctness of such devices, as well as *security* is still very challenging.

In this paper, we consider two concrete examples, i.e. the verification of robotic plans (Section II) and verified partial reconfiguration as part of a countermeasure against side-channel attacks (Section III). In addition, we outline pressing verification challenges for industry and academia in Section IV.

II. Verification of Robotic Plans

Advances in technology, artificial intelligence and engineering allowed to build autonomous robots, which are able to perform tasks in unpredictable environments, to learn, and to adjust their behavior. However, enabling a robot to perform complex everyday manipulation activities, like e. g. setting the table or preparing a meal autonomously, poses several challenges to the development of the robot control system. Essentially, robots need to be equipped with *cognitive mechanisms*, which allow them to deduce what kind of action is suitable to achieve a desired task goal. This includes, but is not limited to, applying different grasp types for different objects and positioning themselves spatially to be able to reach out to a location. As a consequence, the control programs of cognition-enabled autonomous robots use high-level behavior specification languages, which allow to infer control decisions instead of requiring pre-programmed decisions. Several specialized high-level behavior specification languages have been developed in the past. Examples are RPL [1], RMPL [2], and CPL [3]. They all share certain attributes like their inherent concurrency and the ability to call perception, navigation, and manipulation tasks.

While non-trivial scenarios of everyday manipulation activities can be mastered today, the complexity of the plans is steadily increasing. At the same time, simulation of these plans and even testing on physical robots reach computational limits. This causes concern about the safety of autonomous service robots; especially those interacting with humans or handling potentially dangerous items. Hence, formal verification techniques are necessary to ensure the safety of involved humans and the robots themselves.

In the following we review the verification approach presented in [4]. It is the first approach for **verifying plans of cognition-enabled autonomous robots that perform everyday manipulation activities in human environments**. We use the toolbox *Cognitive Robot Abstract Machine* (CRAM) [3] for realizing the cognition-enabled robot control program. In particular, CRAM provides the *CRAM Plan Language* (CPL), which captures high-level plans in Common Lisp. For the CPL, we envision a verification methodology based on *Symbolic Execution* (SE) [5], [6] as it has been shown that SE is a highly effective technique for finding deep errors in complex software applications. The foundation of our approach is the new *Intermediate Plan Verification Language* (IPVL) which serves as a formal intermediate representation. Our approach compiles CPL plans down to IPVL and integrates environment models as well as robot belief states into a single IPVL description. We additionally devised the *Symbolic Execution Engine for Cognition-Enabled Robotics* (SEECER), which is tailored for IPVL. SEECER allows to check plan correctness with respect to environment models as well as annotated assumptions and assertions.

A. CRAM Plan Language

We consider control programs for the autonomous robots written as high-level plans in the *CRAM Plan Language* (CPL) [3]. Plans describe desired behavior in terms of hierarchies of goals, rather than fixed sequences of actions that need to be performed. An architectural overview of the CRAM ecosystem, to which CPL belongs, is depicted in Fig. 1. A CPL plan receives additional information from the robots belief state and knowledge base via a query-answer architecture. It also activates *Perception* and *Manipulation & Navigation* modules, which then get information from or act on the *Environment*. These interaction calls happen in the form of *designators*.

Designators are a common concept employed in several reasoning

978-1-7281-2695-1/19/$31.00 © 2019 IEEE

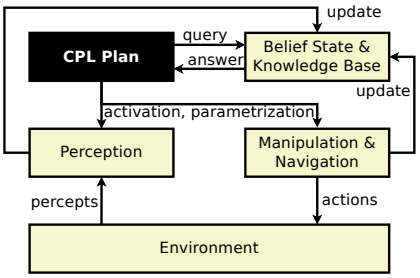

Fig. 1: Overview of the CRAM stack architecture

```
(defun place-object (?target-pose ?arm)
  (par
    (perform (a motion (type looking)
      (target (a location (pose ?target-pose)))))
    (perform (an action (type placing)
      (arm ?arm) (target (a location (pose ?target-
    pose)))))))
```

Listing 1: CPL High-Level Plan

and planning systems. They are often implemented as data types encapsulating high-level descriptions of entities familiar to humans, but abstract to robots. Classes of designators available in CPL are for instance

- *location designators*: physical locations under constraints like reachability, visibility, etc.,
- *object designators*: real world objects on a semantic level like what they are and what they could be used for,
- *human designators*: description of a human entity within an environment, and
- *motion and action designators*: actions that can be performed by a robotic agent.

In CPL, an action designator contains the action type to perform (like perceiving or grabbing) and several parameters. It can be passed to the `perform` function, which breaks it down to sub-tasks and takes care of their execution. The following example illustrates a typical use of different designators.

Example 1. *Listing 1 shows a typical high-level CPL plan using multiple designators. Designators are generated using the `a` keyword. The plan in Listing 1 performs a motion to turn the robot's head to look at a specified target position and places the robot's arm to the same location in parallel. As can be seen, designators may be nested, such as the two location designators used by the action and motion designator.*

B. Formal Verification of CPL Plans

In this section, we present our verification approach for plans of cognition-enabled autonomous robotic agents based on symbolic execution. We start with an overview and the general idea. We then go into detail about how we deal with plan-environment-interaction via an interface. Afterwards, we present our own *Intermediate Representation* (IR) for plan verification and finally close with a detailed description of symbolic execution on that very representation.

1) Overview: Consider Fig. 2 for an overview. Our goal is to formally verify that certain safety constraints on a given CPL plan hold.

We start with compiling the CPL plan into our own IR. For that, we use a language, which we call IPVL and which we describe in Section II-B2.

Additionally, we integrate environment models as well as agent belief states into the IR. Integrating the environment model allows

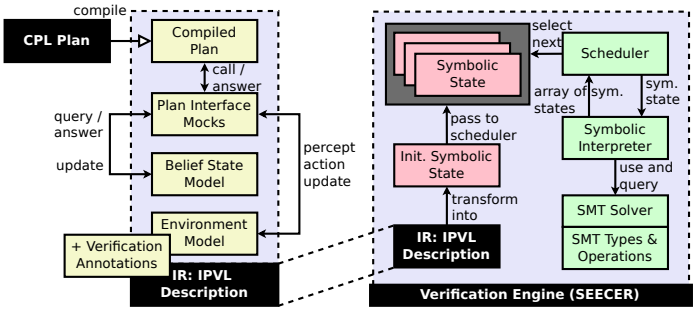

Fig. 2: Overview of proposed plan verification approach

reasoning about the agent's actions. The IR plan accesses these IR models by means of *mocked* functions. Essentially, these *mocks* are models of the corresponding CPL plan interface functions. They enable the IR plan to perform perception, navigation, and manipulation tasks on the environment model and query the belief state model.[1]

For verification purposes, symbolic expressions in combination with assumptions and assertions (verification annotations) are embedded into the IR (see lower left yellow box in Fig. 2). This enables a comprehensive state space exploration. Finally, the combined IR description is passed to a verification engine to check for assertion violations triggered by the plan execution. Our contribution includes (1) the IPVL to act as an IR, and (2) SEECER, which is tailored for IPVL. IPVL is compact, yet powerful enough to capture the simulation semantics of cognition enabled robotic plans in combination with the agent's belief state and environments. SEECER checks plan correctness with respect to the environment model and the specified assumptions and assertions. For modeling plan interface functions and the environment we refer the reader to [4]. Instead, we describe IPVL and SEECER in more detail.

2) Intermediate Plan Verification Language: We define the language features of IPVL in this section. We start with definitions and examples of IPVL's core and then introduce compiling Common Lisp to IPVL by linearization.

Whichever planning language might be used by a robotic system, one only needs to implement a compiler for translating it to IPVL in order to get symbolic execution and verification mechanisms with SEECER on top.

We especially designed the IPVL to make a translation as easy as possible. IPVL is Turing-complete, dynamically typed (like Common Lisp and many other languages in robotics), and incorporates an Assembly-like paradigm.

IPVL code is a sequential list of instructions. Such a representation is general and at the same time much more manageable for a verification back-end (e. g. similar concepts are adopted by LLVM or CBMC). However, it requires to linearize functional languages like Common Lisp.

IPVL uses simple arithmetical, logical, comparison, and conditional instructions. In combination with variable assignments, `goto`s, function calls, and special verification instructions, the whole language can already be described. The instruction set has been designed to be as simple as possible to allow a compact verification backend. On the other hand it should be complex enough to express plans written in higher level robotic languages like CRAM. The IPVL acts as a interface between a verification frontend such as annotated CRAM and a verification backend like SEECER, and allows for both parts to be developed independently. The most

[1] Note that it is possible to exchange the environment model without modifying the plan; hence, to verify the same plan's safety in different environments.

common instruction in IPVL is that of an assignment, where the left-hand-side is a variable name and the right-hand-side is either a constant or an expression. Expressions of arithmetical, logical, and comparison type have at most two parameters.

3) Symbolic Execution for IPVL: In this section, we present our symbolic execution engine SEECER for IPVL, that was mentioned over the previous sections. The right part of Fig. 2 shows an overview of SEECER's architecture. Essentially, SEECER consists of a scheduler and a symbolic interpreter. The scheduler manages a set of symbolic execution states and orchestrates the state space exploration by selecting, which state to consider next. The selected state is passed to the interpreter for symbolic execution. IPVL instructions are interpreted one after another while the symbolic execution state is updated accordingly. The interpreter returns to the scheduler in one of three cases: (1) the end of the IPVL program is reached, (2) an unsatisfiable assumption is reached, or (3) a branch instruction with symbolic condition is executed. In the third case the interpreter will split the symbolic execution state into two independent states and return these two states to the scheduler for further processing. The interpreter employs an SMT solver to check for assertion violations and check feasibility of symbolic branch instructions. Besides user specified assertions, our interpreter also checks for generic execution assertions, e. g. zero divisions.

SEECER starts with a combined IPVL description (which, as described in Section II-B1, integrates the environment model, the belief state model, and the actual plan). The IPVL description is transformed into an initial symbolic execution state, which is then passed to the scheduler. The scheduler performs a *Depth First Search* (DFS). DFS is a common state space exploration strategy that focuses on each path individually and thus is memory efficient (which is important in handling large state spaces). SEECER terminates either after finding a violated assertion or after exploring the whole state space. In the latter case, the plan is shown to be correct with respect to the environment model and the specified assumptions and assertions.

C. Case Study: The Wumpus World

We have implemented our verification approach for plans of cognition-enabled autonomous robotic agents as the symbolic execution tool *SEECER* and the CPL-to-IPVL compiler in C++. As a case study, we consider two CPL plans acting on the well known Wumpus World [7]. Our primary verification objective is to ensure the safety of the plan execution. All experiments are performed on a Linux machine with a 3.5 GHz Intel processor using the Z3 SMT solver [8] (version 4.8.0).

For evaluation, we consider a slalom plan and a column-wise plan as well as their faulty versions, each in combination with square Wumpus Worlds of edge lengths 3 to 10 rooms.

Further, we fixed the number of Wumpus' and gold nuggets to one, but tried multiple numbers of pits (0, 1, and 5). The agent always starts in room $(0, 0)$, while the positions of Wumpus, gold and pits are fully symbolic. This enables a comprehensive plan verification for all possible environment configurations within these boundaries.

We observed, that SEECER has been highly effective in finding the bugs in both faulty plan versions. For each combination of plan and environment setup (i. e. size of the Wumpus World and the number of included pits) SEECER found a counterexample demonstrating the bug on the CPL plan leading to unsafe behavior in less than a second. In the following, we focus on the more interesting results, namely proving safety of the bug-free plans.

TABLE I: SEECER plan verification results

Slalom Plan: safe version									
pits		3×3	4×4	5×5	6×6	7×7	8×8	9×9	10×10
0	T	1s	3s	7s	14s	27s	46s	1m	2m
	#P	10	22	38	58	82	110	142	178
1	T	2s	5s	14s	32s	1m	2m	3m	6m
	#P	13	31	55	85	121	163	211	265
5	T	2s	7s	26s	1m	3m	5m	9m	16m
	#P	4	19	43	73	109	151	199	153
Column-wise Plan: safe version									
pits		3×3	4×4	5×5	6×6	7×7	8×8	9×9	10×10
0	T	1s	4s	11s	26s	54s	2m	3m	4m
	#P	6	13	22	33	46	61	78	97
1	T	5s	40s	3m	12m	34m	1h33m	3h39m	7h56m
	#P	21	102	306	722	1464	2670	4502	7146
5	T	1s	1m	21m	3h49m	TO	TO	TO	TO
	#P	2	115	1319	10357	—	—	—	—

T: execution time (s=seconds, m=minutes, h=hours)
#P: number of symbolic execution paths, **TO**: Timeout (8h)

Table I shows the results for the safe versions of the *Slalom* plan (upper half of Table I) and *Column-wise* plan (lower half of Table I). We report the execution time T and the number of paths $\#P$ for each combination of plan and environment setup. In order to prove desired behavior (i. e. none of the assertion classes is violated), SEECER needs to explore the complete symbolic state space.

It can be observed, that the verification time correlates with the environment complexity. This is to be expected, as the environment model has a direct influence on the state space size. Furthermore, the verification time also depends on the actual plan. While SEECER is able to handle the *Slalom Plan* with increasing environment complexity, it can be observed that the verification runtimes grow exponentially for the *Column-wise Plan*. This can be explained with the significantly larger branching logic in the *Column-wise Plan*, which in turn leads to a much larger number of symbolic execution paths (#P) and SMT solver queries. Symbolic state merging should be a viable technique to increase the scalability of SEECER on such problem instances.

Nonetheless, despite currently missing state-of-the-art optimizations in the symbolic execution engine, the evaluation already demonstrates the applicability and effectiveness of our approach in verifying cognition-enabled robotic plans and indicates that the general approach can be a suitable foundation to deal with larger and more complex environments and plans.

III. Verified Partial Reconfiguration for Security

Partial reconfiguration is a powerful technique to adapt the functionality of *Field Programmable Gate Arrays* (FPGAs) at run-time. When performing partial reconfiguration a dedicated *Intellectual Property* (IP) component of the FPGA vendor, i.e. the *Partial Reconfiguration Controller* (PRC), among a wide range of IP components has to be used. While ensuring the functional safety of FPGA designs is well understood, hardware security is still very challenging. This applies in particular to reconfiguration-based countermeasures which are intensively used to form a moving target for the attacker. However, from the system security perspective a critical component is the above mentioned PRC as noticed by many papers implementing reconfiguration-based countermeasures.

In this section we review [9]. This work **leverages the container principle [10]–[12] – originally proposed as safety mechanism – to the security domain, and by this protect the PRC of an FPGA**. The proposed encapsulation-scheme results in an architecture consisting of so-called *ReCoFuses* (RCFs), each capturing a specific property of interest which has to be fulfilled at any time during PRC

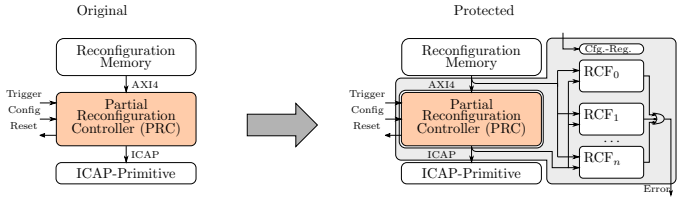

Fig. 3: Original PRC vs Proposed ReCoFuse Container Arch.

operation. The terminology follows the classical electric installation including a *fuse box*. In our scheme we employ formal verification to guarantee the correctness in detecting a security violation. Only after successful verification, the RCFs go live via integration into the *ReCoFuse Container*.

A. ReCoFuse Container

This section presents our encapsulation-scheme for the *Partial Reconfiguration Controller* (PRC) of a FPGA which implements reconfiguration-based countermeasures against physical attacks. The scheme is based on two main components: (1) A "container" encapsulating the PRC, and (2) individual ReCoFuses to monitor and react on untrusted communication with the PRC which would compromise the security of the reconfiguration-based countermeasure.

In the following, we first introduce the overall architecture of the ReCoFuse Container. Then, we detail the interfacing of the PRC and the ReCoFuse Container which hosts the individual ReCoFuses. Finally, the required formal verification of ReCoFuse behavior is described.

1) Architecture of ReCoFuse Container: The left part of Fig. 3 depicts the original unprotected PRC architecture. On the right of Fig. 3 the proposed architecture realizing our encapsulation-scheme for the PRC is shown. As can be seen the ReCoFuse Container has several "slots" for individual ReCoFuses (details see next section). The ReCoFuses are denoted as $\text{RCF}_{0\ldots n}$ in Fig. 3.

Moreover, all outgoing data connections between the main components, i.e. Reconfiguration Memory, PRC and ICAP, are now also fed into the ReCoFuses. Furthermore, a configuration register has been added which allows the user to dynamically enable or disable each RCF.

2) Interfaces and ReCoFuses: Listening on all reconfiguration interfaces allows to monitor the reconfiguration operations requested by the reconfiguration-based countermeasures. In Fig. 3, these are the AXI4 and ICAP interfaces. The ICAP protocol follows a valid/acknowledge scheme where the header of each partial bitfile can be analyzed during data communication. For more details, we refer to the Xilinx 7-Series partial reconfiguration user guide [13].

As can be seen in the architecture, the observed input data is sent to the ReCoFuses. A ReCoFuse essentially implements a *Finite State Machine* (FSM), and hence performs state transitions based on the observed data. Reaching a predefined "good" or "bad" state determines whether the usage of the PRC is considered as trusted or untrusted. The output signal of a ReCoFuse (e.g. in this work an error signal) allows each ReCoFuse to communicate untrusted behavior. As a consequence emergency actions can be executed, for instance to shut down the system. For simplicity, in Fig. 3 on the right we have just ORed all the error signals from each ReCoFuse.

3) Verification of ReCoFuses: To guarantee the correctness of each ReCoFuse, we require formal verification of its behavior. Hence, temporal properties describing the state transitions of a ReCoFuse have to be specified by the user. In other words, these properties are used to prove which PRC communication with the control of the reconfiguration-based counter-measures is untrusted and what will be the resulting action in that case. Typically, a ReCoFuse observes the communication over several clock cycles and finally reaches a "bad" state.

```
1  property advance_timer;
2    disable iff (rst) (
3      t ##0 (cnt!=TIMEOUT and RP_active)
4    implies
5      t ##1 (cnt==$past(cnt)+1)
6    );
7  endproperty
8
9  property detect_error;
10   disable iff (rst) (
11     t ##0 (cnt==TIMEOUT and RP_active)
12   implies
13     t ##1 (error)
14   );
15 endproperty
```

Listing 2: Example properties for timer

B. Case Study

This section demonstrates the proposed encapsulation-scheme for the PRC. As a case study we selected an encryption system using AES. The system implements the moving target principle via reconfiguration by switching between different implementations of the AES. By this, the attacker is not faced with static logic in the FPGA, but permanently changing one, and hence physical attacks become much harder.

In the following, we first describe two major attack vectors. Then, we present the ReCoFuse Container and the two ReCoFuses. Finally, we consider their verification.

1) Attack Vectors: Breaking the reconfiguration-based moving-target characteristic of a cryptographic system, allows attackers to extract secret information via side-channel leakage. In order to attack a specific area (e.g. the PRC) in a FPGA, electromagnetism- and fault-injection-based attacks have both been reported to be effective [14] and are viable methods for disturbing the reconfiguration procedure. We identified two major attack vectors on the PRC:

1) *Time-out attack:* Forcing the PRC to keep the same reconfiguration active for a too long time, would result in no protection. It removes the moving-target characteristic of the design and makes it vulnerable to side-channel attacks.
2) *Replay attack:* Forcing the PRC to chose a single reconfiguration continuously (or more often) removes the moving-target characteristic as well.

This list, however, is not exhaustive and can be extended by the respective adopters needs.

2) ReCoFuse Container: We encapsulated the PRC in a ReCoFuse Container. It instantiates the PRC and provides connection to the configuration memory via AXI and the ICAP primitive as described in Section III-A. ReCoFuses are integrated inside the ReCoFuse Container to achieve countermeasures against the time-out and replay attack. The concrete ReCoFuse are presented in the following two sections.

3) Timeout ReCoFuse: Functionality: The timeout ReCoFuse (RCF_0) basically keeps track of the time between two consecutive reconfigurations. Hence, after a successful reconfiguration, a timer is started. If this timer expires *before* a new reconfiguration procedure is initiated, the timeout ReCoFuse signals an error. Keeping a specific reconfiguration active for an extended period of time – rendering the moving target principle ineffective – can be detected reliably by this ReCoFuse.

Verification: In Listing 2, a subset of the properties for verifying the timeout ReCoFuse RCF_0 are shown. The first property `advance_timer` (Line 1 – Line 7) states that the counter (which realizes the timer) advances with each time step after the previous reconfiguration is done. Here, `TIMEOUT` (Line 3) defines the allowed active duration of one Reconfiguration Module (RM), i.e. a concrete AES implementation. The `RP_active` (Line 3) signal is derived from multiple signals from the reconfiguration infrastructure and captures whether the Reconfigurable Partition (RP) is active, i.e. no reconfiguration is currently performed. In Line 5, the `$past()` statement is used to refer to the previous time-point.

The second verification property `detect_error` (Line 9 – Line 15 in Listing 2) ensures that RCF_0 enters the "bad" state (i.e. raising `error`), when the respective RM was not reconfigured in time (i.e. before `cnt` reaching `TIMEOUT`) (Line 11). For the replay ReCoFuse we refer to [9].

C. Evaluation

All experiments have been conducted on a Xilinx Zync-7000 Series FPGA, more precisely our evaluation platform is a Zedboard featuring a XC7Z020-CLG484-1 FPGA component. More recent FPGA generations feature the same reconfiguration interface, thus our approach maintains applicability in the future. Enhanced capabilities, such as better encryption and authentication however can help to increase the difficulty of attacks further. Our encryption system implements the moving target principle by switching between different implementations of the AES core (based on tiny_AES IP-core from OpenCores.org) via reconfiguration. A dedicated controller in the FPGA (called *MOVECTRL*) initiates the random (i.e. uniform) replacement of a *Reconfiguration Module* (RM), i.e. between the different AES implementations.

To run the experiments, we attacked the reconfiguration process by injecting faults in the encryption system in order to disturb the operation of the PRC. The experiments detailed in [9] showed that the implemented measures – leveraging the proposed scheme – realize an effective and cost efficient protection for reconfiguration-based secured designs. Our flexible architecture allows adding more ReCoFuses (e.g. CRC, additional encryption, hash-based finger printing etc.) easily.

IV. Challenges

In the following we provide a list of challenges in the context of verification. The list is not complete in the sense that all difficulties are covered, but many pressing ones have been identified. By this, a better understanding of the current problems in verification of the next generation electronic systems becomes possible and directions for future research are suggested.

Formal verification: Formal verification is inevitable for the next generation devices to guarantee correctness. If due to the complexity reasons formal verification cannot be applied, simulation should be augmented or guided using formal techniques leading to semi-formal verification. Moreover, formal/semi-formal verification has to be considered at different levels of abstractions, i.e. for hardware up to virtual prototypes which include to execute the software.
Further reading: [15]–[30]

Heterogeneous systems: The separation of a system into analog and digital parts is not sufficient anymore. New integrated verification solutions are necessary in particular due to increasing sensor information available in IoT or CPSs.
Further reading: [31]–[37]

Extra-functional properties: Besides functional behavior also extra-functional behavior, like timing/power but also security, has to be considered during the design of a new system. Often power and timing is controlled by firmware. Hence, these properties have to be captured and verified as early as possible. On the other hand, security was always afterthought and therefore new security verification techniques for hardware and software have to be developed. More recently, the perspective of ensuring morality settings and proving the compliance with ethical guidelines has been considered. In terms of autonomous systems, this will become an important aspect.
Further reading: [38]–[53]

Arithmetic: Proof techniques like BDDs and SAT/SMT suffer from limitations when applied to complex arithmetic. Recently, methods based on *Symbolic Computer Algebra* (SCA) have been used very successfully to prove the correctness of multipliers.
Further reading: [54]–[62]

Self verification: Closing the always widening verification gap is a major problem. A promising approach aims in continuing verification tasks after fabrication of a chip during its lifetime. Several challenges arise here to be solved on the software level as well as on the hardware level.
Further reading: [63]–[68]

Learning systems, especially self-adapting and -learning: Driven by the enormous advances in computing power, learning for instance in the form of neural networks can now be easily integrated into these systems. Verifying their correctness and robustness however is extremely challenging.
Further reading: [69]–[72]

Acknowledgments

The work has (partially) been supported by the German Research Foundation (DFG), as part of Collaborative Research Center (Sonderforschungsbereich) 1320 EASE – Everyday Activity Science and Engineering, University of Bremen (http://www.ease-crc.org/; the research was conducted in subproject P04) by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001, within the project CONFIRM under contract no. 16ES0565, within the project SATiSFy under contract no. 16KIS0821K, within the project CONVERS under contract no. 16ES0656, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

Finally, we would like to thank Vladimir Herdt, Tim Meywerk, Kenneth Schmitz, Buse Ustaoglu, and Marcel Walter for their contributions.

References

[1] B. Drabble, "EXCALIBUR: a program for planning and reasoning with processes," *Artif. Intell.*, vol. 62, no. 1, pp. 1–40, 1993.
[2] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proc. of the IEEE*, vol. 91, no. 1, pp. 212–237, 2003.
[3] M. Beetz, L. Mösenlechner, and M. Tenorth, "Cram—a cognitive robot abstract machine for everyday manipulation in human environments," in *IROS*. IEEE, 2010, pp. 1012–1017.
[4] T. Meywerk, M. Walter, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of plans for cognition-enabled autonomous robotic agents," in *DSD*, 2019.

[5] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[6] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[7] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[8] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340, available at https://github.com/Z3Prover/z3.

[9] K. Schmitz, B. Ustaoglu, D. Große, and R. Drechsler, "(ReCo)Fuse your PRC or lose security: Finally reliable reconfiguration-based countermeasures on FPGAs," in *ARC*, 2019, pp. 112–126.

[10] R. Drechsler and U. Kühne, "Safe ip integration using container modules," in *ISED*, 2014, pp. 1–4.

[11] A. Chandrasekharan, K. Schmitz, U. Kühne, and R. Drechsler, "Ensuring safety and reliability of ip-based system design – a container approach," in *RSP*, 2015, pp. 76–82.

[12] K. Schmitz, A. Chandrasekharan, J. G. Filho, D. Große, and R. Drechsler, "Trust is good, control is better: Hardware-based instruction-replacement for reliable processor-IPs," in *ASP-DAC*, 2017, pp. 57–62.

[13] Xilinx, "User guide – 7 series fpgas configuration," Mar. 2018, https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf.

[14] H. Li, G. Du, C. Shao, L. Dai, G. Xu, and J. Guo, "Heavy-ion microbeam fault injection into sram-based fpga implementations of cryptographic circuits," *IEEE Transactions on Nuclear Science*, vol. 62, no. 3, pp. 1341–1348, 2015.

[15] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999, pp. 193–207.

[16] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, 2000, pp. 108–125.

[17] A. R. Bradley, "SAT-based model checking without unrolling," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011, pp. 70–87.

[18] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[19] D. Große, H. M. Le, and R. Drechsler, "Proving transaction and system-level properties of untimed SystemC TLM designs," in *MEMOCODE*, 2010, pp. 113–122.

[20] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang, "Symbolic model checking on SystemC designs," in *DAC*, 2012, pp. 327–333.

[21] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking SystemC," *TCAD*, vol. 32, no. 5, pp. 774–787, 2013.

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

[23] V. Herdt, H. M. Le, and R. Drechsler, "Verifying SystemC using stateful symbolic simulation," in *DAC*, 2015, pp. 49:1–49:6.

[24] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Boosting sequentialization-based verification of multi-threaded C programs via symbolic pruning of redundant schedules," in *ATVA*, 2015, pp. 228–233.

[25] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models – a case study," in *DATE*, 2016, pp. 1160–1163.

[26] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "ParCoSS: efficient parallelized compiled symbolic simulation," in *CAV*, 2016, pp. 177–183.

[27] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.

[28] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards fully automated TLM-to-RTL property refinement," in *DATE*, 2018, pp. 1508–1511.

[29] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, vol. 38, no. 7, pp. 1359–1372, July 2019.

[30] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019, pp. 188:1–188:6.

[31] S. Steinhorst and L. Hedrich, "Trajectory-directed discrete state space modeling for formal verification of nonlinear analog circuits," in *ICCAD*, 2012, pp. 202–209.

[32] C. Radojicic, C. Grimm, F. Schupfer, and M. Rathmair, "Verification of mixed-signal systems with affine arithmetic assertions," *VLSI Design*, vol. 2013, pp. 239 064:1–239 064:14, 2013.

[33] M. Barnasconi, M. Dietrich, K. Einwich, T. Vörtler, J. Chaput, M. Louërat, F. Pêcheux, Z. Wang, P. Cuenot, I. Neumann, T. Nguyen, R. Lucas, and E. Vaumorin, "UVM-SystemC-AMS framework for system-level verification and validation of automotive use cases," *IEEE Design & Test*, vol. 32, no. 6, pp. 76–86, 2015.

[34] M. Hassan, D. Große, H. M. Le, T. Vörtler, K. Einwich, and R. Drechsler, "Testbench qualification for SystemC-AMS timed data flow models," in *DATE*, 2018, pp. 857–860.

[35] T. Vörtler, K. Einwich, M. Hassan, and D. Große, "Using constraints for SystemC AMS design and verification," in *DVCon Europe*, 2018.

[36] M. Hassan, D. Große, H. M. Le, and R. Drechsler, "Data flow testing for SystemC-AMS timed data flow models," in *DATE*, 2019, pp. 366–371.

[37] M. Hassan, D. Große, T. Vörtler, K. Einwich, and R. Drechsler, "Functional coverage-driven characterization of RF amplifiers," in *FDL*, 2019.

[38] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, C. Ykman-Couvreur, D. Quaglia, F. Ferrero, and R. Valencia, "The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration," *Microprocessors and Microsystems*, vol. 37, no. 8, Part C, pp. 966 – 980, 2013.

[39] G. Onnebrink, R. Leupers, G. Ascheid, and S. Schürmans, "Black box ESL power estimation for loosely-timed TLM models," in *SAMOS*, 2016, pp. 366–371.

[40] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, 2017, pp. 1–8.

[41] D. Lemma, D. Große, and R. Drechsler, "Natural language based power domain partitioning," in *DDECS*, 2018, pp. 101–106.

[42] D. Lemma, M. Goli, D. Große, and R. Drechsler, "Power intent from initial ESL prototypes: Extracting power management parameters," in *NORCAS*, 2018, pp. 1–6.

[43] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Maximizing power state cross coverage in firmware-based power management," in *ASP-DAC*, 2019, pp. 335–340.

[44] M. Schwarz, R. Stahl, D. Müller-Gritschneder, U. Schlichtmann, D. Stoffel, and W. Kunz, "ACCESS: HW/SW co-equivalence checking for firmware optimization," in *DAC*, 2019, p. 187.

[45] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *TCAD*, vol. 30, no. 8, pp. 1128–1140, Aug 2011.

[46] P. Schaumont, M. O'Neill, and T. Güneysu, "Introduction for embedded platforms for cryptography in the coming decade," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, pp. 40:1–40:3, 2015.

[47] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *DATE*, 2016, pp. 337–342.

[48] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *DATE*, 2017, pp. 1691–1696.

[49] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Early SoC security validation by VP-based static information flow analysis," in *ICCAD*, 2017, pp. 400–407.

[50] H. M. Le, D. Große, N. Bruns, and R. Drechsler, "Detection of hardware trojans in SystemC HLS designs via coverage-guided fuzzing," in *DATE*, 2019, pp. 602–605.

[51] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Security validation of VP-based SoCs using dynamic information flow tracking," *it-Information Technology*, vol. 61, no. 1, pp. 45–58, 2019.

[52] M. R. Fadiheh, D. Stoffel, C. W. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *DATE*, 2019, pp. 994–999.

[53] R. Drechsler and C. Lüth, "Code is ethics – formal techniques for a better world," in *DSD*, 2019.

[54] J. Lv, P. Kalla, and F. Enescu, "Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits," *TCAD*, vol. 32, no. 9, pp. 1409–1420, Sept 2013.

[55] F. Farahmandi and B. Alizadeh, "Gröbner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction," *MICPRO*, vol. 39, no. 2, pp. 83–96, 2015.

[56] A. Sayed-Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, "Formal verification of integer multipliers by combining Gröbner basis with logic reduction," in *DATE*, 2016, pp. 1048–1053.

[57] A. Sayed-Ahmed, D. Große, M. Soeken, and R. Drechsler, "Equivalence checking using Gröbner bases," in *FMCAD*, 2016, pp. 169–176.

[58] D. Ritirc, A. Biere, and M. Kauers, "Column-wise verification of multipliers using computer algebra," in *FMCAD*, 2017.

[59] A. Mahzoon, D. Große, and R. Drechsler, "Combining symbolic computer algebra and boolean satisfiability for automatic debugging and fixing of complex multipliers," in *ISVLSI*, 2018, pp. 351–356.

[60] A. Mahzoon, D. Große, and R. Drechsler, "PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers," in *ICCAD*, 2018, pp. 129:1–129:8.

[61] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *DAC*, 2019, pp. 185:1–185:6.

[62] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining SAT and computer algebra," in *FMCAD*, 2019.

[63] 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, invited Talk.

[64] 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.

[65] F. Bornebusch, R. Wille, and R. Drechsler, "Towards lightweight satisfiability solvers for self-verification," in *International Symposium on Embedded Computing and System Design*, 2017.

[66] B. Ustaoglu, S. Huhn, D. Große, and R. Drechsler, "SAT-Lancer: a hardware SAT-solver for self-verification," in *GLSVLSI*, 2018, pp. 479–482.

[67] M. Ring, F. Bornebusch, C. Lüth, R. Wille, and R. Drechsler, "Better late than never : Verification of embedded systems after deployment," in *DATE*, 2019, pp. 890–895.

[68] B. Ustaoglu, S. Huhn, F. S. Torres, D. Große, and R. Drechsler, "SAT-Hard: A learning-based hardware SAT-solver," in *DSD*, 2019.

[69] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *CAV*, 2017, pp. 3–29.

[70] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *CAV*, 2017, pp. 97–117.

[71] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *TACAS*, 2018, pp. 408–426.

[72] W. Ruan, M. Wu, Y. Sun, X. Huang, D. Kroening, and M. Kwiatkowska, "Global robustness evaluation of deep neural networks with provable guarantees for the hamming distance," in *IJCAI*, 2019, pp. 5944–5952.