# Envisioning Self-Verification of Electronic Systems

Rolf Drechsler[1,2] Martin Fränzle[3] Robert Wille[1,2]
[1] Department of Mathematics and Computer Science, University of Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
[3] Carl von Ossietzky Universität Oldenburg, OFFIS GmbH, Oldenburg, Germany
{drechsle,rwille}@informatik.uni-bremen.de martin.fraenzle@offis.de

*Abstract*—**The verification of embedded systems remains to be a challenging task. The ever-increasing complexity as well as time-to-market constraints frequently force designers to terminate the verification process before 100% functional correctness can be ensured. This allows bugs to escape into the final product. All research activities aiming for addressing this problem rely on iterative improvements of existing solutions which remain unable to comprehensively cover the issue. In this paper, a fundamental change in how to approach the verification problem is envisioned. We propose the concept of *self-verification* – a methodology which enables the system itself to eventually complete all the verification tasks that could not be mastered before production. Besides the envisioned methodology, we sketch directions towards its realization and discuss possible application scenarios. By this, we provide a complementary new idea that may have the potential to overcome today's verification crisis.**

## I. Introduction & Motivation

Over the last decades, rapid progress in the development of computing machines has led to ever-more widespread deployment. The resulting *embedded systems* have dramatically changed our life. They are installed in our phones, tablets, coffee machines, tooth brushes, washing machines, and many more. Moreover, we also put our lives into the hands of embedded systems when they are e.g. controlling medical or transportation devices as in implants or airplanes, respectively. As users of these systems, we are usually assuming an error-free behavior. But severe consequences are to be expected if the underlying computing devices expose errors. In particular for safety-critical systems, a simple error can cause deaths in the worst case. Consequently, assuring the correctness of embedded systems is of utmost importance.

For this purpose, *verification methods* are applied before production and deployment. With their help, designers can check (i) whether the system is free of errors, (ii) whether it meets its specified requirements, or (iii) whether it shows some unintended behavior. The following verification methods are applied today:

- *Simulative verification*, in which based on a model of the system the inputs are explicitly assigned and propagated through the system. Afterwards, the outputs are compared to the expected values.
- *Emulative verification*, which realizes simulation directly in a prototypical implementation of the desired chip and thereby exploits the computational power of hardware.
- *Formal verification*, which considers the problem mathematically and proves that a chip is correct. Modulo correctness and completeness of the model, this guarantees 100% functional correctness.

The left part of Fig. 1 illustrates the utilization of these methods in the design and application of today's systems: Simulation and formal verification can be applied as soon as first *models* of the system are available, while emulation can be applied as soon as first (prototypical) *implementations* of the system are available. These methods significantly advanced the state-of-the-art and how verification is conducted in today's design flows. For example, emulation allows for an acceleration of several orders of magnitude compared to simple simulation. Formal verification enabled the verification of a system with full coverage, i.e. it provides a full proof of correctness rather than a validation which may miss important corner-cases.

However, all these methods still do not adequately address the verification problem. An exhaustive application of all possible input patterns is practically intractable and, thus, sufficient coverage cannot be obtained by simulation or emulation. Formal verification suffers from the poor scalability, i.e. it can only be applied to comparatively small circuits and systems. The main reason for these enormous difficulties with verification is the ever increasing complexity according to Moore's Law: The number of transistors in a circuit and, hence, the complexity of those devices, doubles every 18 months.

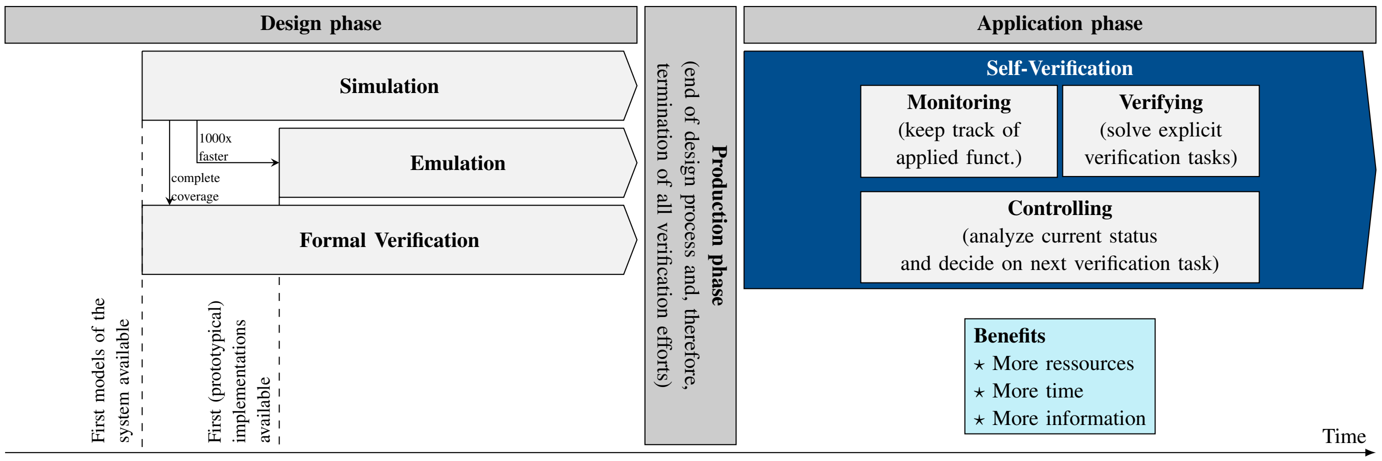
Design phase
Simulation
1000x faster
Emulation
complete coverage
Formal Verification
First models of the system available
First (prototypical) implementations available
Production phase
(end of design process and, therefore, termination of all verification efforts)
Application phase
Self-Verification
Monitoring
(keep track of applied funct.)
Verifying
(solve explicit verification tasks)
Controlling
(analyze current status and decide on next verification task)
Benefits
⋆ More ressources
⋆ More time
⋆ More information
Time

Fig. 1. Verification flow today (left) and with the self-verification methodology (right).

This development has significantly affected how circuits and systems are designed today: While a few years back, the actual implementation process was the core activity in any design flow, verification dominates the entire design process today. Moreover, time-to-market constraints force vendors to terminate the design process and, therefore, also the verification process as soon as possible. Consequently, serious bugs frequently escape into the final product– *complete* verification of correctness (as e.g. envisioned in [1]) is hardly been conducted in practice an verification activities basically focus on important, safety-critical components only.

Current research activities try to address this problem. But almost all corresponding developments rely on incremental improvements of solutions which remain unable to comprehensively cover the issue. For example, designers try to lift the respective design and verification tasks to higher levels of abstraction or apply verification methods which are based on a combination of complementary reasoning techniques. The former development can be seen by the increasing consideration of more abstract system descriptions, e.g. provided by modeling languages such as UML or system description languages such as SystemC, which, eventually, leads to the design at the so-called *Formal Specification Level* (FSL, [2]) or the *Electronic System Level* (ESL, [3]). A representative of combined reasoning techniques can be found e.g. in the domain of so-called solvers for SAT Modulo Theory [4], recently also incorporating efficient techniques from abstract interpretation [5]. But again, these developments will hardly be able to keep pace with the exponential explosion. Eventually, existing and new methods for verification suffer from their limited computational power, while, at the same time, facing an ever-increasing complexity.

In this work, we envision a fundamentally different approach to deal with the verification problem: *self-verification*. The general idea is to realize a system which is capable of completing all the verification tasks that could not be tackled before. To this end, we provide a concept of an according scheme and sketch possible ideas towards its realization in Section II and Section III, respectively. Afterwards, possible application scenarios are discussed in Section IV, before the assumed benefits of the envisioned methodology are discussed in Section V. By this, we provide a complementary new idea that may have the potential to overcome today's verification crisis.

## II. Envisioned Methodology

At a first glance, it might seem natural that verification is only applied in the design phase (as shown on the left-hand side of Fig. 1). However, when a *complete* verification of a system prior to production and deployment is not possible anyway – due to the reasons discussed in the previous section – solutions should be explored how the verification process can be continued and eventually be completed while the system is in operation.

To this end, entirely new design and verification methods are required. Obviously, the main focus is still on the realization of the intended functionality of the system – in the following called the realization of the *target system*. But due to the high effort caused by verification, this aspect has to be considered from the very first moment. Beyond that, new solutions are required to equip the respective systems with additional hardware and software that enables them to perform verification at run-time and to ensure their complete correctness after the deployment.[1] Due to limited resources available in the system, tools supporting formal verification have to be developed accordingly and have to be lightweight versions of existing tools with the goal of providing maximum performance at very low cost. While the principle concept of self-verification may be realized in different fashions and scenarios (which will be discussed in somewhat more detail in the next section), in general the system must be enabled to perform the following three *core tasks*:

1) *Monitoring*, i.e. the observation of the control and data flow which allows the system to keep track of the performed computations in terms of particular patterns or used scenarios. This allows for the recognition of what functionality is usually triggered and what outputs are generated by it.
2) *Verifying*, i.e. checking the correctness of parts of a system, the validity of properties, the complete coverage of a verification results, etc.
3) *Controlling*, i.e. deciding which verification task should be considered next based on an analysis scheme that takes previously obtained information into account, such as properties still left to be verified, frequently occurring patters, application scenarios of the system, etc.

By fulfilling these objectives, self-verification establishes a complementary approach that may be capable of tackling today's verification problems. The biggest obstacle however remains how such a methodology could be implemented. The next section sketches possible realizations.

## III. Possible Realization

In order to realize the envisioned methodology, the embedded system under consideration (which realizes the originally intended target functionality) has to be equipped with additional *core components* which realize the self-verification tasks monitoring, verifying, and controlling. Depending on the respective tasks, they may be realized either in hardware or software. Fig. 2 gives an overview on that partitioning.

More precisely, the basic building block is given by a CPU which should provide special instructions that are used in the software routines (e.g. for triggering a verification process). On top of that, the components realizing monitoring, verifying, and controlling are added. The verifer additionally may take advantage of existing verification software such as reasoning engines which may be provided by so-called verification packages. The resulting sub-system is called the *core system*.

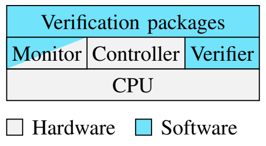

Fig. 2. HW/SW partitioning of core components

In this section, we discuss the main requirements for the core system as well as its components which are important in order to realize self-verification. It may be noted that all core components including the CPU (together forming the *core system* for self-verification) need to be fully verified, so that verification results obtained by them are not spoiled. Furthermore, the core components are supposed to be designed in a flexible manner in order to allow for a manifold integration into different kinds of application scenarios (to be considered in Section IV).

### A. Verification Packages

Verification mainly relies on the availability of powerful reasoning engines. Examples include core verification techniques such as *Binary Decision Diagrams* (BDDs), solvers for *Satisfiability checking* (SAT) and *Satisfiability modullo theory* (SMT), and data structures such as *And-Inverter Graphs* (AIGs). However, for the purpose of self-verification, they have to be provided in terms of lightweight verification packages so that possibly limited resources of the system to be designed are respected.[2]

For this purpose, the software packages should be kept small and the CPU should be optimized with respect to their execution. This could be accomplished e.g. by special instructions derived and implemented into the underlying CPU. In order to guarantee correctness, of course also the software packages need to be verified using software model checkers. Consequently, new lightweight implementations need to be provided that are powerful enough to solve nontrivial verification tasks but not too complex themselves in order to become completely verified. Initial feasibility studies in this direction have already been conducted in [7].

### B. Monitor, Controller & Verifier

Next, hardware and/or software realizations of the core tasks are required. The monitor mainly compares data and, hence, needs optimized circuits for comparison, a fast data path, and direct access to memory. While this calls for a pure hardware realization, scenarios might exists where somewhat more flexibility (and, hence, additional software components) is required.

[1]It should be emphasized that the concepts in the following differ significantly from the aspects of *self-testing*. There, failures in the production process are addressed, while self-verification opens up a way for proving functional correctness.

[2]The BDD11 package from Donald E. Knuth (see [6]) may provide such a lightweight verification solution as it is reduced to simple operations for BDD manipulation only and does not support additional functionality such as variable reordering or complement edges.

The controller needs to perform efficient pattern matching, which best can be realized in hardware. Recent results for regular expression matching [8] can be utilized for this purpose.

Finally, the verifier should be implemented based on the (lightweight) reasoning engines discussed above. This core component is, hence, based on software.

### C. *Resulting Core System*

Combining the components discussed above eventually leads to a core system which is just as large such that conventional verification methods can be applied in order to guarantee 100% functional correctness. Then, full verification of the core system can be applied before deployment, while mission-specific functionality will partially be verified in the field. The resulting system is supposed to be a full-fledged embedded system which can contain both, hardware and software components. It relies on a small 32-bit processor with a simple operating system. But in contrast to established systems, the additional functionality for monitoring, controlling, and verifying has been incorporated.

Furthermore, special machine instructions are part of the instruction set of the architecture to accelerate verification software that is executed on it. As a starting point, a fully verified core system may be based on an OpenRISC architecture from OpenCores.[3] This inherits the advantage of already providing a *gcc*[4] toolchain, so that the focus can be put on the verification of the hardware.

## IV. Possible Application Scenarios

Having the core system, self-verification can be realized in the respective target systems in various fashions. In this section, different scenarios are sketched which illustrate the prospects of this methodology.

### A. *Handling Unverified Behavior*

The main idea of self-verification is to deploy systems even if they have not been fully verified – the remaining verification tasks are supposed to be handled while the system is in its application phase.[5] This obviously bears the risk that errors remain undetected after production. In order to avoid unwanted or even critical behavior of the insufficiently verified system, the core system can be utilized.

[3]www.opencores.org

[4]GNU compiler collection, gcc.gnu.org

[5]In fact, systems are frequently deployed today even if they have not been fully verified. But in contrast to the proposed methodology, this happens without the prospect of completing the verification after deployment.

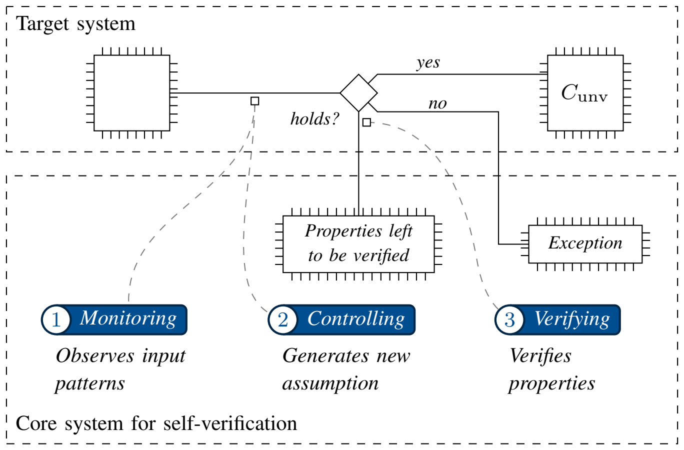

Fig. 3. Handling unverified behavior and performing self-verification

The scenario shown in Fig. 3 sketches a possible solution. The applied architecture allows the explicit consideration of a component $C_{\text{unv}}$ of the target system that is not fully verified at the time of deployment. By monitoring the behavior of the target system, the self-verification system can recognize whenever properties left to be verified (and, hence, affecting $C_{\text{unv}}$) are triggered. More precisely, such a property is e.g. triggered if an input pattern is applied to the target system so that the antecedent of this property evaluates to true. If the pattern violates the property, an exception handling can be invoked (e.g. preventing the execution of erroneous behavior and instead entering a safe mode). Already this provides a significant improvement compared to today's systems, where – although verification with 100% functional correctness could not be performed in most of the cases – an explicit exception handling in cases of errors is often not realized.

### B. *Verification Using Stricter Environment Assumptions*

A main reason why unverified components can not be verified prior to deployment is the fact that they are usually of considerable size and, therefore, are particularly affected by the curse of verification complexity. Using several entities of the product (additionally enriched with lightweight formal verification techniques), a significantly larger computational power is available. But even more importantly, due to *more concise information* on the environment, as provided by the actual user or obtained by monitoring, the actual behavior of the target system can be taken into account: As discussed above, input patterns are constantly monitored. By collecting a lot of them, one will be able to see some common structures, e.g. constant inputs or correlations among them. This common structure can then be used to generate stronger assumptions which in turn ease the verification process.

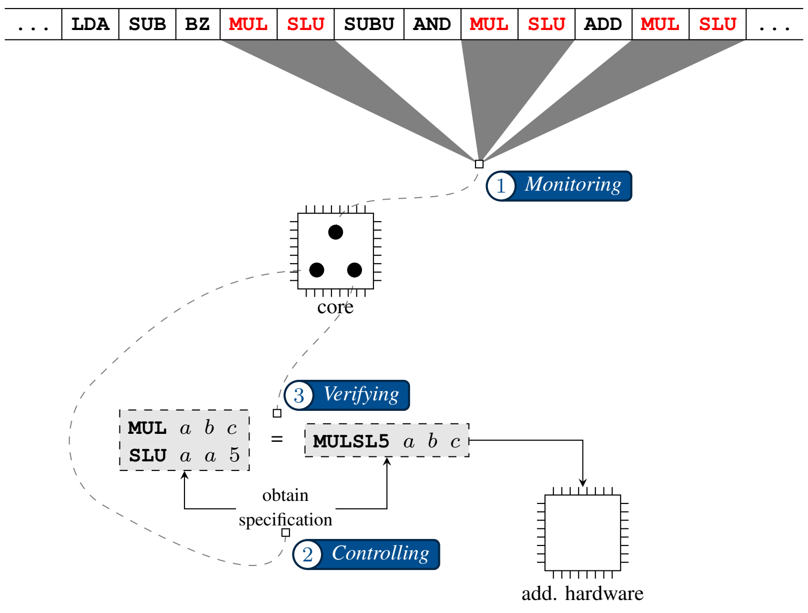

Fig. 4. Self-optimizing architecture

Again this can be illustrated by means of Fig. 3. The blue boxes denote the respective core tasks to be conducted for this purpose which can readily be utilized to implement such a scenario. More precisely, the monitor keeps track of clustering input patterns (step 1) while the controller detects correlations and common structures (step 2). Eventually, the information is used for a stronger assumption to verify a property that has not been verified yet (step 3). If this check is successful, the antecedent can be updated which, in turn, leads to less input pattern to be checked for possible errors. By addressing all verification tasks step by step, a full coverage will eventually be possible within acceptable run-time.

## C. Exploiting Parallelism for Verification

As embedded systems tend to ship in large volumes and thus a large number of produced items usually is available, the concept of self-verification as proposed in this work enables to exploit significantly *more resources* to perform verification.

To this end, the scenario which has been described in Section IV-B for one chip, only has to be extended to multiple chips which share their information. For this purpose, all monitored input patterns are collected at a central place, e.g. some dedicated server. Part of the verification can also take place on this server and even better environmental assumptions can be generated, since the union of all instances is considered. Afterwards, strengthened assumptions or verification results are sent back to all instances.

An alternative approach does not require a central server. Instead, a distributed verification algorithm is developed and concurrently executed on many instances of the item. Recent progress in message passing may provide a basis for the development of such a verification scheme. This particularly allows a good handling of distributing and managing the computation units.

## D. Verification of Changes in the System

This scenario addresses the verification gap in a different manner and especially makes use of having *more time* when verification is applied after production. This enables e.g. verification of self-configurable systems in which changes in the configuration can immediately be verified. Another use case may be the deployment of a simpler, but fully verified system which is optimized during application. The self-verification functionality guarantees the correctness of these optimizations.

A possible case is illustrated in Fig. 4: Here, the monitor observes that *MULT*-commands are frequently followed by *SLU*-commands (step 1). Once the controller detects such a pattern, the determination of a more efficient implementation (e.g. through reconfiguration) may be triggered by the controller (step 2). This eventually leads to the support of *MULSL5*-commands and, hence, a faster implementation. In order to use this new implementation, its correctness is verified (step 3). Following this procedure, simpler realizations of the desired system (for which complete verification was possible) can be deployed, while efficiency is considered during application.

## V. Discussion & Conclusions

The proposed methodology, its possible realization, and discussed application scenarios may result in an extended architecture of embedded systems as shown in the right-hand side of Fig. 1. Beyond the intended target functionality, the resulting system is equipped with methods that allow to perform verification whenever the target application idles.

By this, self-verification explicitly addresses the main reasons of the verification gap, i.e. the limited computational power which, following existing developments, will never keep up with the exponential growth in complexity as well as restrictive time-to-market constraints forcing completion of the verification process well before 100% functional correctness is achieved. Self-verification furthermore provides more situation awareness and thus the prospect of enhanced verification.

Following this paradigm, new prospects emerge that may provide the breakthrough for closing this gap. In fact, verification engineers gain:

1) *More resources*
   Considering it in a naïve way, self-verification can be seen as the continuation of the emulation process where correctness is checked on the actual device. But by combining the verification effort from several instances of the system (out of which usually thousands of them are produced), the efficiency can be increased dramatically. Furthermore, hybrid approaches are applicable, since pure simulation may be complemented by (lightweight) formal proof engines that are available on-board. As a result, the limited computational power – a main reason for the verification gap – is addressed.
2) *More time*
   Since verification is carried out after production, one is not bound to time-to-market constraints anymore. Of course, safety-critical requirements have still to be covered prior to production, as it is the case today. But instead of simply terminating the verification process after that, verification can continue, using idle times of the system in operation, until completeness of the verification task has been achieved.
3) *More information*
   Since a produced circuit or system may be applied in various application domains, barely any knowledge about the environment, application scenarios, frequently used functionality, etc. is known prior to production. Performing verification directly in the system when it is already in operation allows to monitor and exploit such information. Especially for formal verification, adequately modeling the environment and knowing the problem specific invariants significantly simplifies the verification task. This enables a more application-specific consideration and, therefore, does not only improve the effectiveness of the verification but also the quality of the results.

These prospects eventually allow engineers to get a full guarantee that their designs are free from errors – even if this might happen only after deployment. Moreover, also in the worst case, i.e. when the system indeed may comprise an error, the proposed self-verification scheme is beneficial. Once engineers get aware of such a situation, they can accordingly apply countermeasures, e.g. the provision of firmware updates, a restriction of application scenarios, a reconfiguration, etc., which fix or at least circumvent erroneous behavior but still keep the majority of the original functionality intact. If all that does not help, even a call-back might be an option. This is a significant improvement over today's alternatives where, in the worst case, errors are not detected at all or show as hardly reproducible transients, rendering achieving 100% correctness economically infeasible.

## Acknowledgements

This work was supported by the German Research Foundation (DFG) under contract no. DR 287/23-1. The authors like to thank Mathias Soeken for many inspiring discussions during the preparation of this work.

## References

[1] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken, and R. Wille, "Completeness-driven development," in *Int'l Conf. on Graph Transformations*, 2012, pp. 38–50.

[2] 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.

[3] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc., 2007.

[4] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani, "The MathSAT 3 System," in *Int. Conf. on Automated Deduction*, 2005.

[5] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 1, pp. 209–236, 2007.

[6] D. E. Knuth, *The Art of Computer Programming*. Upper Saddle River, New Jersey: Addison-Wesley, 2011, vol. 4A.

[7] 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.

[8] Y. Wakaba, S. Wakabayashi, S. Nagayama, and M. Inagi, "An area efficient regular expression matching engine using partial reconfiguration for quick pattern updating," *IPSJ T. on System LSI Design Methodology*, vol. 7, pp. 110–118, 2014.