Self-Verification as the Key Technology for Next Generation Electronic Systems

Rolf Drechsler[1,2] Hoang M. Le[1] Mathias Soeken[1,2]
[1] Department of Mathematics and Computer Science, University of Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{drechsle,hle,msoeken}@informatik.uni-bremen.de

***Abstract*—Most safety critical systems today cannot be completely verified by state-of-the-art verification approaches before their deployment to the real world. The rapidly growing complexity of these systems is amplifying the strong demand of a disruptive innovation in verification technology. In this invited paper, we propose the concept of *self-verification* — a fundamental change to the way how verification is approached by employing it as a post-deployment process. This enables a new generation of safety critical systems that are capable of verifying themselves. Essential for the realization of this idea is the design of a core system carrying self-verification capacities. We outline a possible architecture of the core system and demonstrate two application scenarios of how self-verification could be realized. The first one targets the verification of evolving systems whereas the second one allows the seamless integration of partially unverified components in safety-critical applications.**

I. Introduction and Background

Embedded systems, as they occur e.g. in microchips in their simplest forms, have dramatically changed our life in recent years. Most of the time, we do not even realize with how many electronic systems and microchips we are getting involved with each day, since they are usually integrated unobtrusively. They are installed in our phones, tablets, coffee machines, tooth brushes, washing machines, and many more. Defects in those devices usually do not lead to serious consequences but only to undesired maloperation. However, we also lay 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 safety critical systems, we are usually assuming a fault-free behavior. But severe consequences are to be expected if the underlying microchips have even slightest defects [1]. Since a simple defect in a safety critical system can cause deaths in the worst case, assuring the correctness of embedded systems is of uttermost importance.

On top of this, the complexity of systems has steadily grown over the last 40 years in an exponential manner according to Moore's law resulting in systems with over a billion components. These are among the most complex systems that mankind has ever built while being integrated in circuits on a few square centimeters.

In order to check whether these complex systems are free of any error, verification methods are applied which check whether the system meets its specified requirements. Current industry practice applies the following verification techniques in the design flow:

- *Simulative verification*, in which based on a model of the circuit the inputs are explicitly assigned, propagated through the circuit, and the outputs are compared to the expected values. This technique is very mature and well-supported by all major EDA companies.[1]
- *Emulative verification*, which realizes simulation directly in hardware thereby achieving an acceleration of some orders of magnitudes. EDA vendors also provide good support in this area.[2]
- *Formal verification*, which considers the problem mathematically and proves that a chip is correct. This is currently an intensively considered topic in both academic research and industry with commercial tools available.[3]

To ensure the correctness of a system, all techniques can be considered. However, exhaustive application of input patterns by simulation or emulation is practically intractable and thus only applicable to small scale circuits. This is caused by the so-called exponential explosion: the problem space which contains all possible combinations is doubled whenever a single input is added to the circuit. Already today, these techniques do not scale and therefore cannot offer sufficient quality.

Often, formal verification methods come to help in these situations since they can efficiently traverse large parts of the search space by applying clever implications during the proof. Formal verification techniques are, however, much more complex both in terms of implementation and in use compared to their simulative counterparts. Further, although formal methods often provide a good alternative to simulative and emulative methods, they are far from being applicable to embedded systems with billions of gates — dimensions which are state-of-the-art in today's designs.

In summary, current approaches are unable to completely verify complex systems and thus cannot guarantee the correctness of embedded systems of modern scale before their deployment.

II. Self-Verification

The verification challenge of next generation system design requires a fundamental change in the way how verification is applied. We propose *self-verification* as the new key technology. Next generation systems should be able to verify themselves at run-time and ensure their complete correctness after the deployment.[4]

Essential for the realization of self-verification is the integration of state-of-the-art verification engines into the system under consideration. This allows to apply current pre-deployment verification approaches to the system while it is in operation. This post-deployment verification process provides

[1]e.g. http://www.mentor.com/products/fv/questa/

[2]e.g. http://www.cadence.com/products/sd/palladium_xp/Pages/default.aspx

[3]e.g. http://www.onespin-solutions.com or http://www.jasper-da.com

[4]Self-verification is well-known as an abstract concept, but to the best of our knowledge there exists no concrete approach in the context of hardware/software systems. For example in [2], the author outlines a vision of future systems, which can optimize themselves and are capable to verify these optimizations after deployment. However, no concrete ideas for a realization are given.

two major benefits. First, it allows the system to extend its functionality in various ways, that are not known a priori, while still maintaining its functional correctness. The second benefit is a much better scalability of verification. This is due to the fact that during the design phase no assumption can be made about the input stimuli thereby leading to a tremendous search space, while realistic input assumptions can be mined when the system is in operation. These benefits are demonstrated in two major scenarios in the following:

1) *Evolutionary Scenario:* We start with a completely verified system of small size. This system is extended during operation and these extensions are also verified at run-time. Once checked completely, they become part of the verified system.
2) *Partial Scenario:* We start with a partially verified system, where the parts that could not be checked prior to production are known. These parts are verified during normal operation.

Both scenarios will be addressed in the following. We start with a description of the *Evolutionary Scenario* and also provide a first experimental evaluation. Then, we address the *Partial Scenario*.

A. Evolutionary Scenario

We envision the architecture of self-verifying systems as follows:

- A *core system* builds the basis which is just as large such that conventional verification methods can be applied in order to guarantee 100% functional correctness. It is a full-fledged embedded system which can contain both hardware and software components. This can for example be a small 32-bit processor with a simple operating system.
- The core system is equipped with *additional hardware* which can be reconfigured during run-time to integrate more functionality — the evolution step. Therefore, the system size is not fixed at the beginning.
- In order to modify the system at run-time, the core system contains further components: a *monitor*, an *analyzer*, a *synthesizer*, and a *verifier*. The monitor observes the control and data flow for reoccurring patterns. Once such patterns are recognized, the analyzer tries to generalize a new or modified functionality and the synthesizer implements the necessary logic on additional reconfigurable hardware. Finally, the verifier proves that the new implementation indeed works as expected and is fully correct.[5]

As formal verification techniques we target both equivalence checking (e.g. [3]) and property checking (e.g. [4]). Equivalence checking proves that two circuits realize the same functionality, e.g. that an optimized version is still equivalent to the original one. Property checking instead proves whether a circuit meets a given property. A property is usually expressed as a temporal formula and can also be synthesized as hardware [5].

Employing such an architecture, the self-verifying system can evolve and thus become more powerful due to the use of its additional components. As a result, functionality that cannot be verified by the initial system can be accessed in a later evolution phase. Hence, self-verifying systems in this scenario grow incrementally and as a result improve and inherit their verification capabilities.

1) Core Components of the Architecture: This section describes the design of the core components. Each component comes along with individual challenges. The *monitor* must be able to detect patterns on different levels of abstraction. That is, it is often not sufficient to only observe data on bit-level but also observe associated bits as chunks on the word-level, e.g. sequences of instructions. The *analyzer* must be able to interpret the data given by the monitor in order to derive new information. Machine learning algorithms provide a good basis for the analyzer component. The *synthesizer* needs to be able to synthesize space efficient realizations in a reasonable amount of time. Since these two metrics are typically inverse-proportional, the synthesis algorithm and particularly its underlying data structures have a significant influence on the whole system. This is to be addressed by evaluating methodologies based on different graph-based function representations such as *Binary Decision Diagrams* (BDDs, [6]) or AND-inverter graphs (AIGs, [7]). Finally, the *verifier* shows the correctness of new components in the system. Since verification is also the most complex task among the core components, the simultaneous integration of several complementary techniques is required. The core system can be designed to have its own operating system on which verification tools can run as software. This is possible for both equivalence and property checking. Moreover, these verification techniques can also be employed using hardware by means of emulation thereby enabling real-time simulation.

Based on the overall architecture we now come to a more detailed description of an application of this concept followed by some initial experimental evaluation.

2) System Verification: In this section, an application scenario of self-verification for verifying evolving systems based on equivalence checking is described. The most challenging part of the implementation are the special purpose modules of the core system such as the analyzer, the synthesizer, and the verifier. In particular the verifier requires adapting state-of-the-art equivalence checking techniques, which are currently implemented in large software systems, in such a way that they are available on the fabricated system. However, although the actual implementation of the verification method in an embedded system is complex, we expect tremendous benefits from it.

There exist many interesting scenarios which can be implemented as an initial prototype for self-adaptive systems. We can make use of a simple processor such as OpenRISC[6] or the processor presented in [8] as the verified core system with an instruction set architecture consisting of a small set of elementary operations. After some time in operation, new functionality may be required, which can be realized by adding new or optimized instructions to the processor. These are detected by a monitor, realized by the analyzer and the synthesizer, and finally verified by the verifier component.

The scenario is illustrated by means of Fig. 1. In this example, the monitor observes the instruction stream for reoccurring patterns, i.e. sequences of instructions (step 1). It can be seen that a multiplication (**`MULT`**) followed by a left-shift by a constant (**`SLU`**) occurs regularly among the instructions (step 2). In case such patterns are detected, the analyzer determines the functionality and suggests a new instruction by obtaining

[5]Although formal verification techniques are desired for this step, in this scenario also emulative verification can be employed for providing a real-time simulation. This is especially applicable if the additional state space is of reasonable size. Otherwise, hybrid approaches can be used, as is studied below.

[6]http://opencores.org/or1k/Main_Page

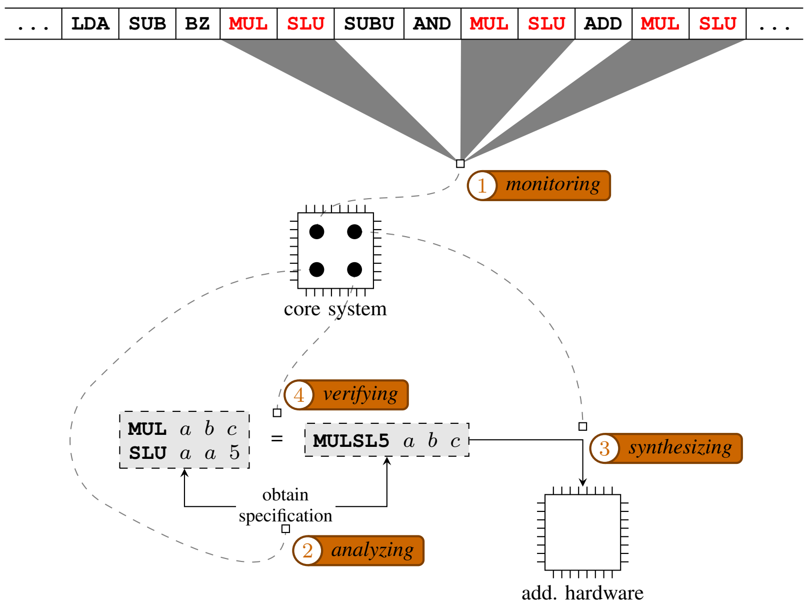

Fig. 1. Self-verification based on equivalence checking

the specification from both extracted instructions. From this the synthesizer module builds a new hardware component that realizes the same functionality but more efficiently (step 3). This new component is verified (step 4) and can then be used in order to enhance the processor's performance. Note that steps 2 (analyzing) to 4 (verifying) may be executed repeatedly until the verification process is successful. From the counter-examples of the verifier, the analyzer can then get additional information to design the new component.

3) Preliminary Evaluations: In order to perform formal verification on the self-verifying system we need formal engines such as SAT solvers or BDD packages that run on the core system. We did some preliminary evaluations with the BDD11 package from Donald E. Knuth[7] that he implemented while preparing the content for Volume 4A of *The Art of Computer Programming* [9]. BDD11 is a very basic BDD package with simple operations for BDD manipulation and no support for additional functionality such as variable reordering or complement edges. Since it has an own memory implementation and no dynamic allocation it is very suitable for running on the core system. We were able to compile the BDD package for the OpenRISC platform with no further adjustment. The BDD package is implemented in only 1850 lines of C code which result in 18555 instructions for the OpenRISC platform after compilation. Consequently, the overhead of having a proof engine on the chip is very low.

A possible testcase is that a system contains a newly generated complex arithmetic component, e.g. the combined multiplier with shifting from Fig. 1 that cannot be fully verified, even though the initial shift operation and the multiplier were proven to be 100% correct. For performing the equivalence check a hybrid technique can be used by representing parts of the structure symbolically, while enumerating all the remaining solutions. Since we are not using simulation of the RTL model, but the real clock speed of the designed circuit, these instances become manageable.

To demonstrate the idea, as an experiment we implemented

[7]http://www-cs-faculty.stanford.edu/~uno/programs/bdd11.w

TABLE I
RESULTS OF THE ADDER EXPERIMENT

Bitwidth		Instructions		
Explicit	Symbolic	Explicit	Symbolic	Total
0	8	0	14 507 523	14 507 523
1	7	3 038	13 838 723	13 841 761
2	6	16 408	13 222 571	13 238 979
3	5	92 352	12 663 247	12 755 599
4	4	486 104	12 159 014	**12 645 118**
5	3	2 421 256	11 713 922	14 135 178
6	2	11 602 920	11 323 487	22 926 407
7	1	54 094 760	10 990 683	65 085 443
8	0	247 124 776	0	247 124 776

a hybrid verification routine for an 8-bit adder that computes the n least significant bits of the adder symbolically using the BDD package and the remaining $8-n$ most significant bits explicitly using circuit simulation. The results are presented in Table I in which the number of executed instructions are listed. We have obtained these numbers using the OpenRISC simulator. As can be seen the hybrid architecture is useful as the best performance, i.e. the minimal number of instructions, is achieved when simulating half of the bits symbolically and half of them explicitly.[8]

B. Partial Scenario

While the previous scenario was based on the idea of starting with a fully verified core that is extended stepwise, the concept of self-verification can also be applied in the case where a priori known components are integrated that are not fully verified yet. The verification process continues while the system is in use.

More formally, this application scenario of self-verification considers the integration of an *unverified component* C_{unv} into a safety-critical system. The idea is that C_{unv} is too complex for complete verification. This might e.g. result from the complex structure of the component itself or from the fact

[8]Note that a very similar experimental setup can also be applied in the scenario discussed below.

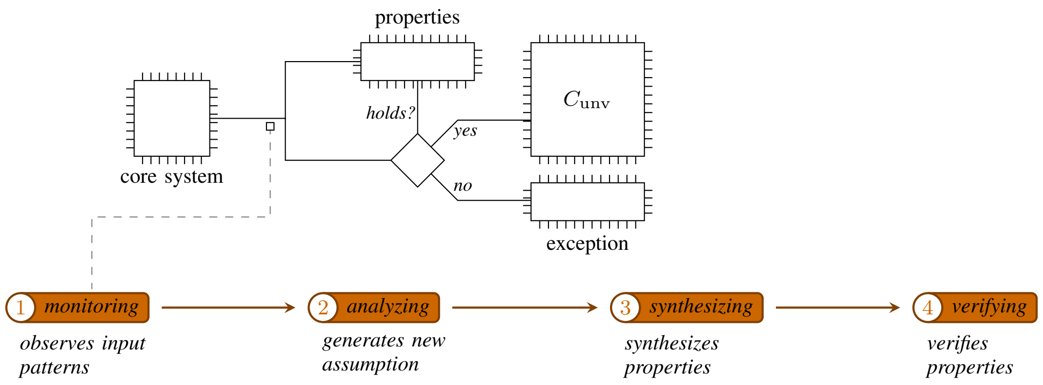

Fig. 2. Self-verification based on property checking

that during the design phase no assumption can be made about the input stimuli thereby leading to a tremendous search space. These assumptions are usually given in terms of a model of the environment.

In the following we make use of the same four components introduced above, although they are used in slightly different contexts.

The *monitor* can be used to capture this model of the environment, which is continuously updated during run-time. Then, C_{unv} can be verified post-deployment by exploiting better assumptions on the actual input assignments. The scenario is illustrated by means of Fig. 2. We propose to insert the unverified component C_{unv} into the circuit together with the *properties* that could not be verified in advance. Additionally, a special component realizing an *exception routine* is added that should be executed if the properties fail. This guarantees a controlled handling in unexpected situations. The overall flow works as follows: Instead of passing the signals to C_{unv} directly, first the properties are checked for this particular pattern. If the properties hold, the component is executed and otherwise, the exception routine is triggered. As one consequence, this significantly slows down the execution time. However, the self-verification architecture enables a solution to this problem: The monitor observes the input patterns which are sent to C_{unv} (step 1 in Fig. 2). As mentioned above, no (adequate) model of the environment was given when designing the circuit which made verification impossible. However, now during operation it is possible to see which inputs are actually assigned. After observing some (probably reoccurring) input patterns, the analyzer generates new assumptions of the environment from which a model is obtained and (re-)synthesized (step 2). The verifier tries to solve the properties with these assumptions and in case of success the control logic around C_{unv} can be optimized in the following way: All input patterns that are contained in the assumption generated by the analyzer can directly be passed to the component. All other patterns still require the additional check through the synthesized properties (steps 3–4). Afterwards, the whole process can be repeated for the new input patterns that are assigned to the component. Eventually more properties are derived and verified enabling 100% correctness for the component at a high execution speed in the safety-critical system.

Finally, note that the same techniques presented in Section II-A3 can be applied in the *Partial Scenario* to verify the components either by a lightweight formal tool or by a hybrid approach.

III. Conclusions

In this paper we presented the concept of *self-verification* and described a possible architecture for its realization. Two scenarios have been outlined, namely the *Evolutionary Scenario* and the *Partial Scenario*. Based on this concept verification of next generation complex systems becomes manageable, since it allows for complete verification and hence implements *Completeness-Driven Development* (CDD) [10].

As a first experimental study outlined, it is possible to have not only testing equipment on board – such as it is standard today using *Build In Self Test* (BIST) – but also verification engines that allow formal proofs while the system is in operation.

Acknowledgements

The ideas in this paper have emerged from many discussions over different verification flows within the context of the Reinhart Koselleck project supported by the *German Research Foundation* (DFG) under contract no. DR 287/23-1. The authors would like to thank all project members for these fruitful discussions. Furthermore, the authors like to thank Robert Wille for several helpful comments on an early draft of this paper.

References

[1] J. C. Knight, “Safety critical systems: challenges and directions,” in *Int’l Conf. on Software Engineering*, vol. 22, 2002, pp. 547–550.
[2] W. Luk, *The Future of Computing, essay in memory of Stamatis Vassiliadis*, Delft, The Netherlands, September 28, 2007, ch. Self-optimizing and self-verifying design: a vision, pp. 69–79.
[3] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, “Data-driven equivalence checking,” in *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 27, 2013.
[4] J. Baumgartner, A. Kuehlmann, and J. A. Abraham, “Property checking via structural analysis,” in *Int’l Conf. on Computer Aided Verification*, vol. 14, 2002, pp. 151–165.
[5] M. Boule and Z. Zilic, “Automata-based assertion-checker synthesis of PSL properties,” *ACM Trans. Design Autom. Electr. Syst.*, vol. 13, no. 1, 2008.
[6] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.
[7] A. Mishchenko, N. Eén, R. K. Brayton, M. L. Case, P. Chauhan, and N. Sharma, “A semi-canonical form for sequential AIGs,” in *Design, Automation and Test in Europe*, vol. 16, 2013, pp. 797–802.
[8] D. Große, U. Kühne, and R. Drechsler, “HW/SW co-verification of embedded systems using bounded model checking,” in *ACM Great Lakes Symposium on VLSI*, vol. 16, 2006, pp. 43–48.
[9] D. E. Knuth, *The Art of Computer Programming*. Upper Saddle River, New Jersey: Addison-Wesley, 2011, vol. 4A.
[10] 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.