Better Late Than Never
Verification of Embedded Systems After Deployment

Martin Ring*, Fritjof Bornebusch*, Christoph Lüth*†, Robert Wille*‡, Rolf Drechsler*†
*Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
†Mathematics and Computer Science, University of Bremen, Germany
‡Institute for Integrated Circuits, Johannes Kepler University Linz, Austria

***Abstract*—This paper investigates the benefits of verifying embedded systems *after* deployment. We argue that one reason for the huge state spaces of contemporary embedded and cyber-physical systems is the large variety of operating contexts, which are unknown during design. Once the system is deployed, these contexts become observable, confining several variables. By this, the search space is dramatically reduced, making verification possible even on the limited resources of a deployed system. In this paper, we propose a design and verification flow which exploits this observation. We show how specifications are transferred to the deployed system and verified there. Evaluations on a number of case studies demonstrate the reduction of the search space, and we sketch how the proposed approach can be employed in practice.**

I. Introduction

In the past decades, the verification of embedded and cyber-physical systems has become a pressing, complex and elaborate problem for which a number of high-end tools are available [1], [2], [3], [4]. Designers and verification engineers have access to an enormous amount of computational power, *e.g.* in terms of high-end design and compute servers. However, time-to-market constraints are putting increasing pressure on releasing products earlier and, hence, most of today's systems get deployed without being fully verified.

This is obviously caused by the exponential complexity of the problem. Each year, more complex systems are being designed and need to be verified. Iterative improvements have been proposed in the past years, *e.g.* the introduction of higher levels of abstractions for design such as the *Formal Specification Level* [5] and the *Electronic System Level* [6], or the lifting of SAT solvers to solvers for *SAT Modulo Theory* (SMT) [7], [8], [9], [10], [11], but these cannot and will not be able to cope with the complexity. The consequences are evident today: While several years back, the actual implementation process was the core activity in any design flow, verification dominates today. In fact, more than 40% of the time and costs within the design are devoted to prove the correctness of a system [12].

Because of this situation, we are convinced that verification cannot solely be addressed by incremental improvements of existing approaches anymore, but rather a shift in the existing verification paradigm. In this work, we are proposing a methodology towards such a paradigm shift. To this end, we start with the observation that contemporary systems are designed to operate in a variety of operating contexts. In order to do so, *configurations* are used, *i.e.* parameters which are set post-deployment by the particular environment of the individual system. While these parameters may not change frequently, they are not fixed and hence verification, which is conducted prior to deployment, has to consider all possible configurations. Consequently, designers and verification engineers are faced with verifying systems with huge possible search spaces, while after deployment just a fraction is used.

Research supported by BMBF grant SELFIE, grant no. 01IW16001.

Motivated by this observation, this work proposes a design and verification methodology which conducts verification after deployment, *i.e.* in the field and once the actual configuration is observable. By this, this paper proposes a first realization of the concept of *self-verification* envisioned in [13]. Even though it results in continuous verification tasks as the environment keeps changing, the drastic reduction of the search space outweigh this. As a result, embedded systems can be verified even on a much weaker machine and with much less sophisticated tools, while prior to deployment verification failed due to the exponential complexity. Our approach of post-deployment verification guarantees safety with maximum availability — the system will never refuse to operate when it is actually safe to run.

In order to assess the feasibility of the proposed methodology, we have implemented the proposed design and verification flow and used a lightweight version of the SAT solver MiniSat [7], [14] to solve the resulting verification conditions after deployment. The evaluation of a number of case studies showed that, following the proposed methodology, verification problems which failed prior to deployment (using high-end verification tools and machines) could be completed after deployment using the lightweight solver on reduced hardware.

In the following, we present the proposed methodology as follows: First, we motivate and illustrate the proposed verification methodology using a small case study. Section III then describes the implementation of this methodology using the same case study and, by this, provides a detailed description for the entire flow. Section IV summarizes the results of evaluating further, more sophisticated case studies conducted by us — confirming the applicability and benefits of the proposed methodology. Besides that, this section also discusses the consequences of the proposed methodology to the established design flow. Finally, Section V summarizes and concludes this work.

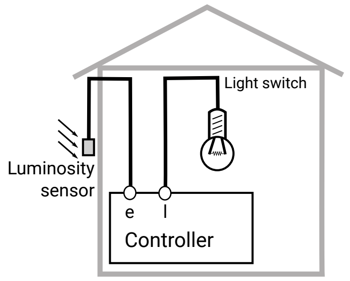

Fig. 1. Bringing light into darkness: The light controller is connected to a luminosity sensor, and switches a light on or off when it becomes too dark or bright.

II. Proposed Idea: Verification after Deployment

The key idea of the proposed approach presented here is to defer part of the verification until *after* deployment. At first sight, this seems like a rather strange idea. A system deployed in the field is likely to have far less computational power, memory and network resources available than a design server. However, it has a main advantage which, we argue, outweigh theses deficiencies: after deployment, there is generally *more information* about the operating context available.

In order to enjoy this benefit, the design needs to be geared towards verification after deployment. At an abstract level, the general idea is to partition the system state space into one part which changes frequently post-deployment and thus has to be explored symbolically, and one part (preferably as large as possible) which only changes infrequently. This part is called the *configuration*. Marking a variable as a configuration variable means that its value rarely changes, and entails that we can substitute actual values before verification post-deployment. By marking variables of n bits as configuration variables, we reduce the search space we need to explore for verification by 2^n — turning the exponential growth into an exponential reduction. The idea and its benefits are illustrated by the following (running) example[1].

Example 1. *The simple light controller system sketched in Fig. 1 works as the running example in the following. This system connects a controller to a luminosity sensor and a light switch. The controller should turn on the light if the sensor e drops below a given level e_{lo}, and turn it off if it exceeds a given level e_{hi}. To avoid a flickering effect when the luminosity varies close to a given threshold, the lower and upper threshold levels are not equal (hysteresis), and the system should switch off the light only with a certain delay d. The threshold levels e_{lo}, e_{hi} and the delay d are configuration variables, and can be changed post-deployment.*

Systems like these are designed in a flexible fashion, so that they can be applied in various contexts. For the light controller, the threshold levels and delay are not fixed at design or production time but will be set post-deployment. Hence, in order to verify the correctness of the system, we need to take into account *all* possible configurations, which increases the search space exponentially. It also means that a lot of possible configurations are checked during verification which may never be applied during the system's lifetime. Hence, if we instantiate the configuration variables after deployment and keep only the variables of the system which change frequently arbitrary, we get a much smaller search space to explore.

[1]Note that the example has been deliberately kept simple – both for expository and space reasons.

Example 1 (Continued)**.** *Consider again the running example. If we assume a width of 8 bit for the input values (the luminosity sensor and subsequently for the upper and lower bounds) and the time delay, and one bit for the light switch status (these are lower bounds for a realistic system), we get the following search space (where cnt is a variable counting up to delay):*

$$\underbrace{\begin{array}{ccc} e_{lo} & e_{hi} & d \\ 8 & 8 & 8 \end{array}}_{\text{configuration}} \begin{array}{cccccc} e & cnt & status & & total \\ 8 & 8 & 1 & = & 41 \end{array} \tag{1}$$

Thus, we need to check an overall search space of 2^{41} states to verify the system, a huge search space for a very simple example.

In contrast, once the system is deployed and applied in the field, the values for e_{lo}, e_{hi} and d rarely change (once when the system is deployed, and afterwards only if the user actively changes the configuration), as opposed to the values of e, cnt and $status$ which vary constantly. Thus, we can mark e_{lo}, e_{hi} and d as configuration variables, and verify the system only when the configuration is changed. By keeping the values of e_{lo}, e_{hi} and d fixed for the verification, the search space reduces to 2^{17} states.

The reduced search space can be handled comfortably by a lightweight solver after deployment, even under the prevailing conditions of limited computational resources. But note that this verification is only valid for the particular configuration (*i.e.*, the supplied values for e_{lo}, e_{hi} and d) and, thus, can principally not be done prior to deployment without severely reducing the flexibility and versatility of the system.

III. Implementing the Proposed Approach

The previous section illustrated the potential of conducting verification after deployment. Based on that, we now describe in detail a possible implementation of this methodology. We first describe the design process in more detail, and then demonstrate it at work with a formal development of the running example considered in the previous section.

A. The Design Process

The design flow starts with a *modelling phase*, where the structure and behaviour of the system is modelled at an abstract level without referring to any implementation details (see Fig. 2). In our case, we use SysML [15] and OCL [16] to specify the structure and formalize constraints on its behaviour as well as the functional hardware description language CLaSH [17] for a uniform, executable and synthesizeable model of the system.[2]

[2]The actual specification and implementation languages are of no particular relevance and could be replaced by others (*e.g.* we could use UML instead of SysML, or SystemC instead of CLaSH), but serve here to point out the level of abstraction in the corresponding part of the design process.

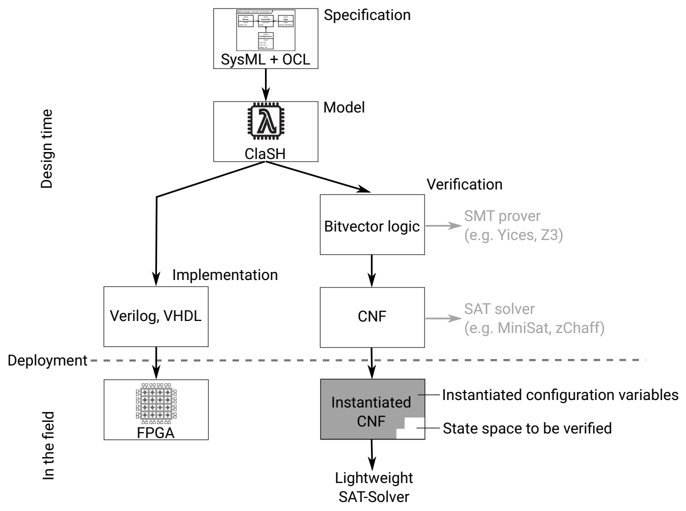

Fig. 2. Design flow for verification after deployment. We start with modelling the system behaviour, then derive an implementation and verification conditions. By proving the verification conditions we make sure the system behaves as specified. Due to the large search space, the proofs are not possible pre-deployment. But instantiation of the configuration variables reduces the size of the search space significantly and makes proofs possible post-deployment.

From the model, we can synthesize an *implementation* of the system by generating a representation in a low-level hardware modelling language such as VHDL or Verilog, which is used to program an FPGA – constituting the actual implementation. Moreover, we want to *verify* that the generated system behaves as specified. In order to do so, we generate a list of *verification conditions* from the executable system model and the specification which have to be shown in order to guarantee this. Specifically, we translate both the CLaSH model and the constraints from the OCL specification into *bit-vector logic* (*i.e.* first-order logic with bit-vectors). Trying to show these in an SMT prover such as Yices [10] or Z3 [11] fails for non-trivial examples, as does trying to show the properties translated into *conjunctive normal form* (CNF) with a SAT solver such as MiniSat. This is where verification usually fails.

However, post-deployment after we have instantiated the configuration variables, the search space is small enough to allow verification of the corresponding properties even by a lightweight solver [14]. By this, verification of all properties becomes possible. Recall that this instantiation cannot be done at the design time, because at that point the instantiating values are still unknown. Therefore, the proofs must be rerun if the values of the configuration variables are changed.

B. The Design Process At Work

Specification (top of Fig. 2): The specification of the system is provided in terms of a SysML block definition diagram as shown in Fig. 3. The structure is composed of the controller as the central block, with one luminosity sensor, and one light switch (actuator) connected. The variables specifying the lower and upper threshold of luminosity and the delay when switching off are in a separate block marking them as configuration variables.

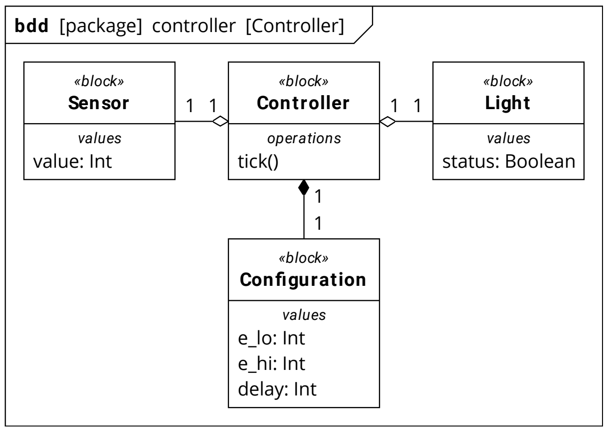

Fig. 3. SysML specification of the light controller

context Controller
def e: sensor.value
def off: e $>$ config.e_hi
def on: e $<$ config.e_lo
def off_s: cnt $\geq$ config.delay

context Controller::tick()
 post a1: **not** off **implies** cnt $=$ 0
 post a2: off **implies** cnt $=$ cnt@pre+ 1
 post a3: on **implies** light.status
 post a4: off_s **implies not** light.status
 post a5: **not** (on **or** off_s) **implies**
  light.status $=$ light.status@pre

Fig. 4. OCL specification of the behaviour of the light controller

The behaviour is provided in OCL as shown in Fig. 4. We model state transitions by an explicit operation tick(); The pre- and postcondition of the state transition are denoted as pre- and postconditions of this operation.

Model (middle of Fig. 2): Based on the specification, a CLaSH model is derived. CLaSH is a strongly typed domain-specific language to model hardware. It is embedded into the functional programming language Haskell, and describes the hardware as functions of the language. The strong type system guarantees that everything we can describe in CLaSH is still synthesizeable, and allows us to model the hardware at an abstract but still executable level. The model describes the hardware by combinators (higher-order functions), building up complicated circuits by composing elementary ones. Fig. 5 shows a brief excerpt of the model, essentially a finite-state machine (a Mealy automaton) with the luminosity values (*Unsigned 8*) and the configuration as input, the light switch (*Bool*) as output, and an internal state (*ControllerState*) which keeps track of the light switch and a counter to implement the delay when switching off. The function *controllerT* (definition omitted for brevity) is the state transition function of the automation, taking the state and the input, and returning a tuple of new state and output.

```
type ControllerState = (Bool,Unsigned 8)

controllerT :: ControllerState
→ (Configuration, Unsigned 8)
→ (ControllerState, Bool)

controller :: Signal (Configuration, Unsigned 8)
→ Signal Bool
controller = mealy controllerT (False,0)
```

Fig. 5. CLaSH model of the light controller (excerpt)

Implementation (left-hand side of Fig. 2): From the CLaSH model, we generate Verilog, which is then compiled onto the FPGA by the proprietary tool chain of the FPGA vendor (in our case, Xilinx). Thus, the CLaSH model is the foundation of the verification after deployment.

Verification (right-hand side of Fig. 2): To prove the verification conditions, we translate them into CNF, which is suitable as input for reasoning engines such as SAT solvers. This translation proceeds in two steps. We first translate both the CLaSH model and the specification into bit-vector logic, which in the second step can be translated into CNF by Yices. The translation from CLaSH is done with an extension of the CLaSH compiler we have developed for this work; the translation of OCL is done by a tool provided in previous work [18].

Fig. 6 shows a small excerpt of the bit-vector representation of the model from Fig. 5. We are modelling the state transition explicitly, so for each state variable (*e.g. switch*, *cnt*) we have a variable to model the pre-state (here, *preSwitch*, *preCnt*). Fig. 6 asserts that the state switches to *true* if the luminosity value drops below *e_lo* and it switches to *false* if the luminosity is above the threshold and *cnt* is larger or equal to the configured *delay*.

To verify the implementation, we translate the specification from OCL into bit-vector logic; for example, the two clauses *a4* and *a5* from Fig. 4 become:

```
(=> off_s (not switch))))
(=> (not (or on off_s)) (= switch preSwitch))))
```

We generate a CNF formula from the negated conjunction of all five clauses (and the invariants) in Fig. 4, together with the model from Fig. 6. This formula is satisfiable iff the specification is violated (because we assert the negated specification). Because we explore the complete search space (there is no state abstraction involved), this procedure is not only sound but also complete; if we cannot find a counter-example, the verification condition holds.

```
(define preSwitch :: bool)  ; light switch before
(define switch :: bool)     ; light switch after
(assert
  (= switch
     (ite (bv-lt e e_lo)
       true
       (ite (and (bv-gt e e_hi) (bv-ge preCnt delay))
         false
         preSwitch
) )   ) )
```

Fig. 6. Implementation modelled in bit-vector logic (excerpt)

Instantiation after Deployment (bottom of Fig. 2): Finally, the configuration variables are instantiated in order to reduce the search space. This is directly conducted in the obtained CNF. In order to give an impression of the generated CNF, we just consider the very simple assertion $e \leq e\_hi$, which translates into bit-vector logic as the assertion:

```
(assert (not (bv-lt e e_hi))).
```

Using only two bits for e and e_{hi}, Yices generates a CNF that represents these bit-vectors as variables, which corresponds to the formula[3]:

$$\begin{aligned}&(\neg e_1 \vee x) \wedge (e_2 \vee \neg e_{hi,2}) \wedge (e_2 \vee x) \wedge \\ &(e_{hi,1} \vee x) \wedge (\neg e_{hi,2} \vee x) \wedge (e_1 \vee \neg e_{hi,1} \vee \neg x).\end{aligned} \tag{2}$$

Yices keeps track of the encoding of the variables, *i.e.* to instantiate the configuration variables corresponding unit clauses are added.

The instantiations now significantly reduce the search space. This can be exploited to solve the resulting instance after deployment using a lightweight solver.

IV. Evaluation and Discussion

So far, the proposed methodology has been illustrated by means of an intentionally rather limited example. Moving on from that, we have applied the idea of verification after deployment, and the proposed verification as described in Section III, to more sophisticated home automation controller in order to demonstrate its applicability. The home controller has been realized on top of a ZedBoard, which comprises an ARMv7 core running Linux to control a Xilinx FPGA, and which for the purposes of verification has been equipped with a lightweight SAT solver [14]. The obtained results are summarized in this section. Furthermore, we also discuss possible ramifications which have to be considered when utilizing the proposed methodology in practice.

A. Evaluation

The proposed methodology has been evaluated on a set of systems which are natural extensions of the light controller considered above to highly versatile home automation controllers as follows:

- *simple*: The simple light controller with one light and one luminosity sensor (as considered in the running example).
- *average*: An extended version of the controller which includes up to 16 sensors to be connected and controls one actuator by averaging the values obtained by those sensors. Input and output are generic, *i.e.* we can control any kind of actuator and read from any kind of sensor as long as it gives us integer values.
- *weighted_avg*: A similar version with 32 sensors that allows to add a configurable weight to each sensor when computing the average.

[3]Here, x is an auxiliary variable. e_1 and e_2 denotes the first respectively the second bit of the bit-vector e. The same notation applies to e_{hi}.

TABLE I
EVALUATION RESULTS

	Established Verification Flow (on Intel Xeon (E3-1270 v3) compute server)				Proposed Verification Flow (on ARMv7 target system)			
System	Search space	Variables	Clauses	Time	Search space	Variables	Clauses	Time
simple	2^{41}	161	539	$< 0.1\ s$	2^{17}	131	255	$< 0.1\ s$
average	2^{177}	11807	40086	$131.0\ s$	2^{137}	8181	13010	$1.4\ s$
weighted_avg	2^{545}	43569	146642	$> 24\ h^*$	2^{265}	31374	37559	$28.5\ s$
smart	2^{9504}	1421153	4761633	$> 24\ h^*$	2^{544}	1421153	2704606	$1.5\ s$
multiplier	2^{32}	1177	6096	$> 24\ h^*$	2^{16}	809	2467	$418.0\ s$

* = timeout

This table compares the established verification flow (verifying all properties in full generality at design time, left) with the proposed verification flow (verifying instantiated properties after deployment, right).

- *smart*: A smart home controller, which allows up to 32 sensor inputs to be connected to up to 32 actuator outputs. Each input can be connected with each output, making the controller very versatile and resulting in a huge search space. The smart home controller can be used *e.g.* to control lights, heating and blinds for a number of rooms in an office setting.
- *multiplier*: A 16 bit multiplier component, used to apply the weights in *weighted_avg* and *smart*. Can be verified with a constant factor once the configuration is set.

For all these systems, we have specified their intended behaviour in OCL, similar to the specification of the simple light controller in Fig. 4, and have verified that the implementation satisfies this specification. Table I lists the results. Column *System* gives the name of the considered system. The remaining columns summarize the results in two groups: the first group for verification according to the established verification flow (*i.e.* verifying all properties at design time) and the second group for the verification methodology proposed here (using the lightweight solver on the target system). For each group, we give the size of the search space (*i.e.* the number of possible solutions to be checked); the number of variables; the number of clauses of the resulting CNF; and the runtime (in seconds). The runtime is measured on systems which would typically be used for verification, so they are directly comparable: for the established verification flow, a compute server (Intel Xeon E3-1270 v3, eight cores, 16 GB memory) and, for the proposed verification flow, the ZedBoard (ARMv7, 1GB memory).

The obtained results clearly show the benefits of the proposed approach. Typical embedded systems (as the ones considered here) allow for a huge variety of configurations. As shown in Table I, this results in a rather large search space and SAT instance for the verification, which takes a significant amount of time to solve (in some cases, the corresponding verification task could not be solved within the given time-limit of one day). In contrast, after deployment, configuration variables can be instantiated with their actual values, as discussed in Section II. This substantially reduces the search space and allows to solve the verification task even on the limited resources of an embedded system. Of course, the search space is only one complexity indicator: as the multiplier system shows, even a comparatively small search space may require a long time to be verified, because of its inherent complexity. However, the proposed verification flow reduces the runtime significantly in this example as well, and thus allows us to verify a system which was previously out of reach for established tools.

B. Practical Exploitation

Our approach may be applied in various ways. In the following we illustrate a possible practical application to the design of a smart home controller as described above.

Requirements and properties are established during design time, and checked with contemporary verification tools. All properties which cannot be automatically checked during design time are prepared for self-verification using our approach.

In the deployed system, a verification controller is constantly watching the values of the configuration variables and triggers a proof if a value change is requested. For example, if a light is connected to the smart home controller, the configuration is updated and the proofs have to be re-run. Since the system would now be in an unverified state, it will either stop operating or defer the value change until the proofs have successfully finished; this way, it continues operating with guaranteed safety. (If the risk is considered acceptable, the system might instantly change the value and continue to operate while the proofs are running.)

If a proof fails for the resulting configuration, the system informs the user about the failed proof. The user can disconnect the sensor again or try a different configuration until the proof succeeds and the change result in a safe state. This especially means that the system can still operate safely even though some functionality is missing. Furthermore, the manufacturer is informed about the failed configuration, and can use this information to take appropriate measures.

C. Discussion

The results obtained by the conducted cases studies summarized above clearly show the promises of the proposed verification methodology. However, some obvious ramifications have to be discussed when evaluating the general applicability of this methodology.

The proposed methodology obviously requires the embedded system to be equipped with on-board verification tools to conduct the verification tasks. Since the considered systems are substantially less powerful than usual desktop systems or verification servers, this requires lightweight but still efficient versions of those tools. Here, recent developments on lightweight methods [14], [19] as well as endeavours towards efficient hardware solvers [20], [21] provide promising platforms for this purpose. Besides, the proposed verification methodology yields an exponential reduction in the search space, so even less powerful verification tools might be able to cope.

Our approach differs from *runtime verification*, which is concerned with "checking whether a *run* of a system under scrutiny satisfies or violates a given correctness property" [22]. The central notion of runtime verification is the trace (or run) of a system, and central questions are how to derive monitors checking a concrete run against an abstract specification. The logics employed are typically temporal or modal logics. In our work, we are not concerned with monitoring the system at all, we instead *specialize* given variables in an abstract specifications if they do not change often.

V. Conclusions

In this paper, we presented a novel verification methodology based on verification *after* deployment. We partition the system search space into one part which changes infrequently (the *configuration variables*), and one which does. By conducting the verification with the configuration variables set to their actual values, the search space is reduced drastically, making verification feasible even on the limited resources of an embedded system.

We have used specific modelling languages (SysML, OCL, CLaSH) and prover tools (Yices, MiniSat) in our evaluation, but the basic idea is independent of these. As long as we can translate the verification conditions into a format where we can track the variables to be instantiated, and which is suitable to automatic proof (CNF in our case), the approach is viable and competitive, because of the exponential reduction of the search space. For example, on a more powerful system, we might be able to prove the instantiated verification conditions in bit-vector logic rather than CNF. This will enable the verification of even more sophisticated systems.

Note that the verification flow proposed here does not *replace* the existing verification flow, it *enhances* it. We may use all well-known powerful tools at design time to prove verification conditions as before, and still use verification after deployment to tackle the verification conditions we could not prove during design — giving us the best of both worlds.

Overall, the evaluations and discussions show that, following the proposed idea, allows for a novel verification methodology which does not rely on incremental improvement of existing tools and methods but tackles complexity from a completely different, and more successful, angle.

References

[1] J. Yuan, C. Pixley, and A. Aziz, *Constraint-Based Verification*. Springer, 2006.

[2] R. Wille, D. Große, F. Haedicke, and R. Drechsler, "SMT-based stimuli generation in the SystemC verification library," in *Forum on Specification and Design Languages (FDL)*. IEEE, 2009, pp. 1–6.

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[4] A. Koczor, L. Matoga, P. Penkala, and A. Pawlak, "Verification approach based on emulation technology," in *Int. Symp. on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, 2016, pp. 169–174.

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

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

[7] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing (SAT)*, ser. Lecture Notes in Computer Science (LNCS), vol. 2919. Springer, 2003, pp. 502–518.

[8] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani, "The MathSAT 3 System," in *International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Computer Science (LNCS), R. Niewenhuis, Ed., vol. 3632. Springer, 2005.

[9] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler, "Sword: A SAT like prover using word level information," in *IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC)*. IEEE, 2007.

[10] B. Dutertre, "Yices 2.2," in *International Conference on Computer-Aided Verification (CAV)*, ser. Lecture Notes in Computer Science (LNCS), vol. 8559. Springer, July 2014, pp. 737–744.

[11] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science (LNCS), C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008, pp. 337–340.

[12] H. Foster, "Why the design productivity gap never happened," in *Int'l Conf. on Computer-Aided Design*, 2013, pp. 581–584.

[13] R. Drechsler, M. Fränzle, and R. Wille, "Envisioning Self-Verification of Electronic Systems," in *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2015.

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

[15] Object Management Group, "OMG Systems Modeling Language (OMG SysML)," OMG, Tech. Rep. formal/2015-06-04, 2015.

[16] M. Richters and M. Gogolla, "OCL: Syntax, Semantics, and Tools," in *Object Modeling with the OCL*, ser. LNCS, T. Clark and J. Warmer, Eds. Springer, 2002, vol. 2263, pp. 42–68.

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

[18] M. Ring, J. U. Stoppe, C. Lüth, and R. Drechsler, "Change impact analysis for hardware designs," in *Forum on Specification and Design Languages (FDL)*. IEEE, 2016.

[19] A. Balint and U. Schöning, "Engineering a lightweight and efficient local search SAT solver," in *Algorithm Engineering — Selected Results and Surveys*, ser. Lecture Notes in Computer Science (LNCS). Springer, 2016, vol. 9220, pp. 1–18.

[20] T. Ivan and E. M. Aboulhamid, "An efficient hardware implementation of a SAT problem solver on FPGA," in *Euromicro Conference on Digital System Design (DSD)*. IEEE, 2013, pp. 209–216.

[21] B. Ustaoglu, S. Huhn, D. Große, and R. Drechsler, "SAT-lancer: A hardware SAT-solver for self-verification," in *28th ACM Great Lakes Symposium on VLSI (GLVLSI)*. ACM, 2018.

[22] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, May 2009.