Towards a Methodology for Self-Verification

Christoph Lüth*†, Martin Ring*, Rolf Drechsler*†

*Dept. Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz
28359 Bremen, Germany

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

***Abstract*—The exponential growth of the complexity of electronic systems makes their verification increasingly difficult. To address this problem, incremental refinements of existing approaches are insufficient in the long term; new approaches are needed. In this paper, we explore the new approach of *self-verification*, where systems will verify themselves during run time. Self-verification will give system engineers more time, more resources, and more information to successfully finish the verification. We sketch an architecture and methodology for self-verification, which maps system properties to the development phase in which they are verified, and illustrate the approach with a first case study.**

I. Introduction

Embedded and cyber-physical systems have found their way into almost all parts of our lives, sometimes without us noticing. They are at work in everyday devices, from the ubiquitous smartphone to washing machines to cars and trains, and we expect them to work fault-free and reliably, especially when employed in safety-critical application areas such as transportation. This proliferation into different application areas has been made possible by immense progress in the development of microchips, which are at the core of these cyber-physical systems. Microchips have steadily become more powerful over the last forty years, growing in an exponential fashion — the number of transistors in a device is still doubling every 18 months (the well-known Moore's Law) — resulting in systems which today consist of billions of components.

To assure the correctness of embedded and cyber-physical systems, a variety of *verification methods* help designers to check whether the system is free of errors, and whether it meets its specified requirements. Amongst the verification methods in use today are the following:

- *Simulative verification*, which is based on a model of the system. The inputs are explicitly assigned and propagated through the system, and afterwards the outputs are compared to the expected values. This technique is very mature and well supported, but complete coverage of all possible input values is practically impossible to obtain by simulation for contemporary systems.
- *Emulative verification* realizes simulation directly in a prototypical implementation of the desired chip in dedicated hardware. While this allows for an acceleration by several orders of magnitudes, with this method complete coverage can only be achieved in rare cases.
- *Formal verification* considers the problem mathematically and proves that a chip is correct (*i.e.* satisfies certain *properties*). This is currently an intensively researched topic in both academia and industry. Formal verification aims for complete coverage, but scalability remains an issue: formal verification can today only be applied to rather small circuits and systems.

Unfortunately, the progress in manufacturing and developing microchips has far outpaced the progress in developing verification methods. This is known as the *verification gap*: the time needed to verify systems has been growing while simultaneously the time to market has been decreasing.

Current research activities try to address this problem. However, almost all corresponding developments rely on incremental improvements of existing solutions. For example, designers try to lift the respective design and verification tasks to higher levels of abstraction, provided by modeling languages such as SysML or system description languages such as SystemC, which eventually leads to design at the so-called *formal specification level* [1] or the *electronic system level* [2]. But these developments have been unable to close the verification gap comprehensively, due to the exponential nature of Moore's law. It is obvious that the underlying problems cannot be addressed through incremental improvements, but require a fundamental change in the way how circuits and systems are verified today.

Because of the complexity of the underlying theoretical satisfiability problem (it is NP-complete) we cannot reasonably hope to speed up verification methods by the required orders of magnitude. Hence, we need to extend the time available for verification. This can be achieved by *continuing verification*

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

978-1-5090-3012-5/17/$31.00

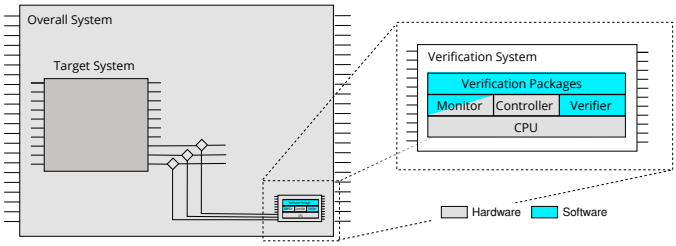

Fig. 1. A basic system architecture for self-verifying systems. Self-verification is performed by a dedicated *verification system* which is added to the target system. The monitor observes the behaviour of the target system and triggers the verifier if it observes input related to unverified properties. The controller decides how to handle successful or unsuccessful verification results.

after production and deployment. In other words, systems continue to verify their correctness at run time. This is the basic idea of *self-verification*. We have put forth this idea in earlier papers [3], [4]; in this paper, we will lay out first steps towards a sustainable methodology of self-verification.

This paper is structured as follows: we first present the basic principles of self-verification, followed by the methodology of how to develop self-verifying systems. We explore a first case study, and finish with conclusions.

II. Self-Verification

By allowing verification to continue after deployment, engineers will gain the following advantages:

- *more time*, as verification is not cut off at deployment, and thus not bound by time-to-market constraints anymore;
- *more resources*, as self-verification can be run on all deployed systems in parallel;
- *more information*, as self-verification can take into account knowledge about the environment the system is deployed in.

The basic architecture for self-verification is sketched in Figure 1. The *target system*, which realizes the intended functionality, is extended with a *verification system*, which performs verification at run time. The latter checks and proves that the system realizes the intended functionality correctly. Using dedicated hardware separate from the target system instead of *e.g.* using one dedicated core of a multi-core processor has the advantage that at least in theory the verification system can be formally verified and thus proven to be error-free, and that we can also reuse the verification system design for many other systems, in particular for custom-built hardware which may not provide an additional core (as in our case study below). Due to limited resources available in the verification system, the tools supporting formal verification have to be adapted to lightweight versions of existing tools, with the goal of providing maximum performance with minimal resources.

This architecture raises a number of interesting research questions, such as which properties are suitable to be verified at run time, which have to be proven at design time, how does a design flow for a self-verifying system look like, and how can we reduce the system state space enough to allow verification at run time under limited resources? We will address these questions in the following, thus constituting a methodology for self-verification.

III. Methodology

To motivate our methodology, we consider a simple methodology exercise from the application domain of smart homes. Suppose we want to implement a smart light controller connected to a light sensor and a light switch. It should turn off the light when it gets bright enough outside, and turn on the light when it gets dark. This system is basic enough that we can understand the behaviour *in toto* and thus are able to envisage the design flow. We emphasize that due to this simplicity it is not a useful case study; we will consider one later.

A. Informal Requirements Specification

We start the development with a non-formal specification of the intended behaviour. A first attempt would be an informal, textual specification (which we call ***RS-1***):

> Let $e(t)$ be the luminosity at time t, and $E_{lo} < E_{hi}$.
> (i) If the luminosity drops below E_{lo} the light should switch on.
> (ii) If the luminosity raises above E_{hi} for T_D seconds the light should switch off.

Note how we implemented a hysteresis with two different thresholds $E_{lo} < E_{hi}$ to avoid a flickering effect when the luminosity jitters around one threshold. We also introduce a delay in switching off the light such that the light stays on if there is a short bright flash (*e.g.* lightning, a passing car) during an otherwise dark night. A first attempt to formalize this specification using differential equations (describing the change of the system behaviour over time) might be the following, which we call ***RS-2***:

$$\mathit{on}(t) \stackrel{\mathit{def}}{=} \begin{cases} 1 & \text{if } e(t) = E_{lo}, \frac{\partial e(t)}{\partial t} < 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$\mathit{off}(t) \stackrel{\mathit{def}}{=} \begin{cases} -1 & \text{if } \frac{\partial e(t-T_D)}{\partial t} > 0 \text{ and} \\ & \forall s.\, t - T_D \leq s \leq t \Longrightarrow e(s) \geq E_{hi} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$\mathit{light}(t) \stackrel{\mathit{def}}{=} \int_0^t \mathit{on}(t) + \mathit{off}(t)\ dt \tag{3}$$

where $\mathit{light}(t)$ is the state of the light at time t, and $\mathit{on}(t), \mathit{off}(t)$ auxiliary functions which model turning the light on and off.

B. Formal Requirements Specification

The continuous mathematics used in ***RS-2***, although well-known to engineers, is not well suited for systems development, as digital system are by their nature discrete.[1] A discrete system uses a discrete clock, which is measured in

[1]Having said that, there are numerous attempts to describe the behaviour of these so-called hybrid systems (*e.g.* [5], [6]). However, all of these serve to bridge the gap in expressiveness between continuous and discrete mathematics, so we end up with discrete descriptions in any case.

ticks of length ΔT.[2] The discrete luminosity measurement is $E(n) = e(n \cdot \Delta T)$, and the discrete equivalent of specification ***RS-2*** is the following specification ***FS-1***:

$$\mathit{on}(n) \stackrel{\mathit{def}}{=} E(n) < E_{lo} \tag{4}$$

$$\mathit{off}(n) \stackrel{\mathit{def}}{=} E(n) > E_{hi} \tag{5}$$

$$\mathit{light}(n) \stackrel{\mathit{def}}{=} \begin{cases} 0 & n = 0 \\ 1 & \text{if } \mathit{on}(n) \\ 0 & \text{if } \forall m.\, n - D \leq m \leq n \Longrightarrow \mathit{off}(m) \\ \mathit{light}(n-1) & \text{otherwise} \end{cases}$$

where $D \stackrel{\mathit{def}}{=} \ulcorner \frac{T_D}{\Delta T} \urcorner$ is the number of ticks in the time span T_D. The system state at tick n depends on $\mathit{off}(n-D), \ldots, \mathit{off}(n-1)$. This is not desirable, because the underlying system model is a finite state machine with transitions based on the current state and the observed input; in other words, we do not keep track of past states. Specifications at this level need to be *relational* in the sense that they only relate two states, a pre- and a post-state, thereby defining all possible state transitions.

To achieve such a relational specification, we internalize the counting into the system state by defining a function *cnt* as follows in specification ***FS-2***, where $\mathit{on}(n)$ and $\mathit{off}(n)$ are given by (4), (5):

$$\mathit{cnt}(n) \stackrel{\mathit{def}}{=} \begin{cases} 0 & \text{if } n = 0 \\ 0 & \text{if } n > 0 \text{ and } \neg\, \mathit{off}(n) \\ 1 + \mathit{cnt}(n-1) & \text{if } n > 0 \text{ and } \mathit{off}(n) \end{cases} \tag{6}$$

$$\mathit{light}(n) \stackrel{\mathit{def}}{=} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } \mathit{on}(n) \\ 0 & \text{if } \mathit{cnt}(n) \geq D \\ \mathit{light}(n-1) & \text{otherwise} \end{cases} \tag{7}$$

The system state $\Sigma(n)$ at tick n is $\Sigma(n) \stackrel{\mathit{def}}{=} \langle \mathit{cnt}(n), \mathit{light}(n) \rangle$, and the system behaviour is a state transition function $E(n) \times \Sigma(n) \rightarrow \Sigma(n+1)$, consisting of the two functions *cnt* and *light* as defined in (6), (7).

C. Formal Specification Level

It is not straightforward how to implement the system specified by ***FS-2***. The equations do not specify which arguments are inputs and what constitutes the output of the system. Thus, we need a more concrete model of the system which specifies the components and datatypes in more detail. We give this model in SysML/OCL, where SysML is used to model the data, and OCL is used to specify the state transitions.

SysML [7], [8] is a modeling language, closely related to UML, which describes the structure and behaviour of a system by nine different diagram types, such as block definition diagrams, state machine diagrams, or sequence diagrams. SysML diagrams can be formal or informal; we concentrate on the former here, and in particular use block definition diagrams to describe the structure of our system.

[2]The length of ΔT is determined by the band width, *i.e.* the minimal time width T_{min} for which a change in system behaviour is detectable. With the Nyquist-Shannon sampling theorem, we get a maximum time between ticks needed to sample the system accurately as $\Delta T = \frac{1}{2} T_{min}$.

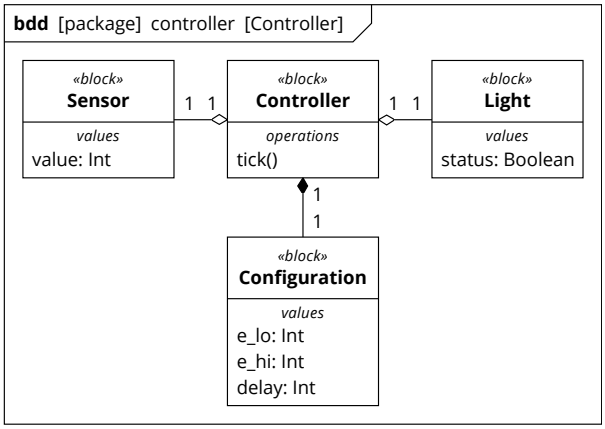

Fig. 2. SysML specification (***FS-3***)

Figure 2 shows a SysML specification of the system design. It decomposes the system into *blocks*. Specifically, we have decomposed the system into the main controller, which is connected to the light sensor $sensor$ and the switch $light$. It has a *configuration*, which we split into a separate block, detailing the lower and upper luminosity bound, and the delay (in ticks) for switching off the light.

Figure 2 does not specify the actual state transitions from equations (6) and (7). This is done in an OCL specification; OCL [9] is a language using which we can *constrain* the models described by SysML diagrams, by annotating preconditions, postconditions or invariants. Thus, OCL is a language to denote relational specifications such as ***FS-2***. Translated into SysML/OCL, ***FS-2*** looks as follows, where tick () is the state transition operation:

```
context Controller
  def e: sensor.value
  def off: e > config.e_hi
  def on: e < config.e_lo
  def off_s: cnt ≥ config.delay

context Controller::tick() post:
  (not off implies cnt = 0) and
  (off implies cnt = cnt@pre+ 1) and
  (on implies light.status) and
  (off_s implies not light.status) and
  (not (on or off_s) implies
     light.status = light.status@pre)
```

D. Electronic System Level

We are now arriving at a formal model of the system which we can implement in either software or hardware; we call this the electronic system level. We are more interested in hardware, so we give an implementation in the hardware modeling language Chisel [10] (another alternative would be the more popular, but more verbose, SystemC). To go from the Chisel model to actual hardware, we can *e.g.* generate Verilog, and implement it on an FPGA.

Spec.	Contents	Formalism	When to show proof obligations
RS-1	Requirements	Natural language	No proof required
RS-2	Requirements	Informal mathematics	No proof required
FS-1	Formal requirements	Temporal and first-order logic	Design time
FS-2	Formal requirements	Relational first-order logic	No proof required
FS-3	System structure, formal requirements	SysML/OCL	Run time
FM-1	Executable system model	Chisel	

Fig. 3. Specification levels, with their contents, their formalism, and the time in the development process when the properties have to be proven.

```
class LightController(e_low: Int, e_high: Int) extends
    Module {
  val io = new Bundle {
    val e = UInt(INPUT,8)
    val light = Bool(OUTPUT)
  }

  val x = Reg(Bool(false))
  val cnt = Reg(UInt(8))

  cnt := io.e > UInt(e_low, 8) && cnt+1 || io.e ≥
      UInt(e_low) && 0
  x := cnt > d || !(io.e > UInt(e_high,8)) && x

  io.light := x
}
```

To sum up the development, we have developed a concrete hardware implementation of a light switch controller from an initial informal requirements specification in natural language. The question is whether this development is correct, *i.e.* whether the final result satisfies the initial requirements. Can we prove the correctness properties?

E. Proving Verification Properties

The development here is of course simple enough to prove correctness comprehensively at design time, but we want to explore the question from the point of self-verification; in particular, we want to know which correctness properties arise, which properties we need to prove at design-time, and which properties we can feasibly push into run time verification.

First, note that the step from the informal natural language specification ***RS-1*** to the continuous specification ***RS-2*** can by its nature not be proven formally,[3] but we only trace the requirements [11]. The second step from ***RS-2*** to ***FS-1*** could be proven mathematically, but not many tools support proofs at this level, so we view the continuous specification ***RS-2*** as more rigorous but informal too.

In the third step, the relational specification ***FS-2*** was developed. Here, we can actually prove correctness (that ***FS-1*** implies ***FS-2***) by induction on the states. This proof has to be done at design time, since it talks about all system states, and at run time we can only refer to the current and possibly previous state.

In the fourth step, we go from the relational specification ***FS-2*** to the SysML/OCL model ***FS-3***. Part of this step is just a change of notation, rephrasing relational statements as OCL constraints, part of this step specifies the system structure which was implicit or unspecified before. All in all, there are no proof obligations here.

Finally, there is the step from SysML/OCL to Chisel. In this step, we need to prove that the invariants and pre/postconditions are satisfied by the operations in the implementation. This is a typical verification activity as mentioned in Section I, and can be achieved by a variety of means, such as testing, model checking, or theorem proving (in increasing order of completeness and complexity). However, these proofs can be pushed into run time, since they only relate the current and previous state.

Figure 3 gives an overview over the specification levels and formalisms involved, and when properties are to be proven. To sum up, we have found three different kinds of properties: *informal* ones, formulated in natural language, which cannot be proven; *global* ones which talk about all system states, which we can formulate *e.g.* in linear temporal logic (LTL, [12]); and *local* ones which talk about the current and next system state. At run time, we can only prove local properties, as we do not have access to all system states. Hence, the formal system model given in SysML/OCL (***FS-3***) should form the basis of the self-verification.

F. Efficient Self-Verification

Now that we have ascertained which properties we can verify at run time, we can investigate how to verify them. At run time, we can only use fully automatic proof procedures with a light memory footprint, such as lightweight versions of SAT checkers or SMT provers. In general, the memory requirements of these tools grow exponentially with the state space of the system; this is known as *state explosion*. Hence, we need to reduce the state space as much as possible.

[3]Note that when we speak about "formal proof" we mean proofs conducted on and checked by a computer, using tools such as SAT checkers, SMT provers, or interactive theorem provers.

In our development, the state space of the final system comprises the variables e, light, d, e_low and e_hi. The first two have 8 and 1 bit, respectively; assuming the latter three to be of 8 bits width, we have a total state space of $2^{8+1+3\cdot 8} = 2^{33}$. However, d, e_low and e_hi do not change very often,[4] so to reduce the state space we can treat them as parameters. More precisely, each time the values of d, e_low or e_hi change, the invariant is proven for the *current* values of d, e_low or e_hi and *all possible* values of e and light — a state space of 2^9, or a reduction by a factor of 16 millions. We call attributes such as d, e_low or e_hi *quasi-static*.

This is the reason why we have modeled the configuration as a separate block in Figure 2. By annotating this block with a suitable stereotype the engineer can mark it as a rarely changing configuration, which is treated as quasi-static. This is a small, conservative extension to SysML which all tools can handle, and which at the same time reduces the state space for run time verification dramatically.

IV. Case Study

In order to explore the scalability of our approach, we extend the initial light controller example into a more realistic application. This controller supports multiple light sources and luminosity sensors, as well as manual switches and presence detectors. Its informal specification is given as follows:

> Sensors, switches and presence detectors are assigned to a light source via a configuration.
>
> (R-1) If the arithmetic average of all sensors connected to a light source is below E_{lo}, then this light is switched on. Conversely, if the arithmetic average of all sensors connected to a light source is above E_{hi}, then this light is switched off.
>
> (R-2) Every time the state of a manual switch connected to a light source changes, the state of the light source is negated. After such a manual override, the automatic control of (R-1) is disabled as long as a person is detected by at least one of the presence detectors.
>
> (R-3) Once no person is detected, the light is switched off.

The assignments of luminosity sensors to lights is stored in the controller. Two operations, connect and disconnect, are triggered whenever sensors are (dis)connected:

```
context Controller::connect(sensor: Sensor, light: Light)
  pre: not sensors.contains(sensor)
  post: sensors.contains(sensor) and
        config.at(light.id).sensors.contains(sensor.id)
```

When a sensor is disconnected from the controller, the assignment is also removed from the configuration:

```
context Controller::disconnect(sensor: Sensor)
  pre: sensors.contains(sensor)
  post:
    not sensors.contains(sensor) and
    not config.at(light.id).sensors.contains(sensor.id)
```

[4]In fact, the system model does not specify how they can change at all; the case study below specifies how a configuration can be changed.

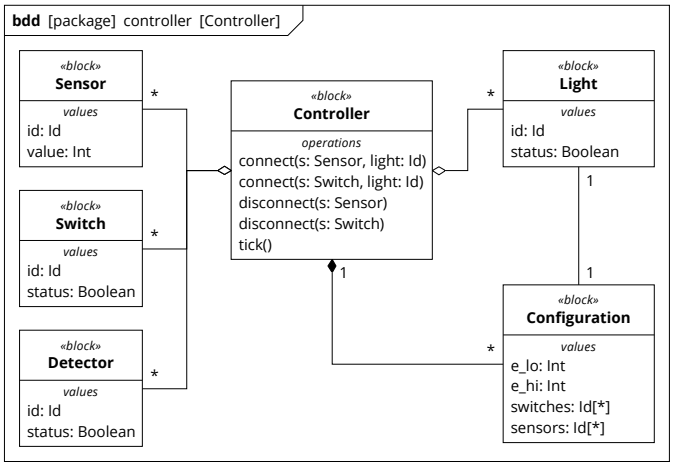

Fig. 4. SysML specification of the case study. The sensors (on the left) include luminosity sensors, manual switches and presence detectors; actors (on the right, top) light switches. The configuration (on the right, bottom) specifies how switches and sensors are connected.

The following definition specifies the average of all luminosity sensors switch a light or off (requirement R-1):

```
context Light
  def sensors: controller.config.at(id).sensors
  def e: sensors.collect( self.value ) / sensors.size()

  def off: e > controller.config.at(id).e_hi
  def on: e < controller.config.at(id).e_lo
```

Requirements (R-2) and (R-3) are specified as the postcondition of the tick operation, which models the state transition:

```
context Controller::tick() post:
  lights.forAll (light |
    if light.detectors.exists ( self.value ) then
      if light.switches.exists(switch |
          switch ≠ switch@pre) then
        light.override and
        light.status = not light.status@pre
      else if light.override@pre then
        light.status = light.status@pre
      else
        not light.override and
        (on implies light.status) and
        (off implies not light.status) and
        (not (on or off) implies
            light.status = light.status@pre)
      endif
    else not light.status and not light.override
    endif)
```

For space reasons, we do not give the Chisel model here,

as our main focus is the verification of properties of the SysML/OCL model. What makes the verification of this system particularly challenging is the dynamic nature of the infrastructure. All the components can be dynamically added to and removed from the system, and every sensor, switch and detector can register for a light source. Dynamic aspects of this kind yield unmanageably large search spaces when attempting a full verification. If we assume identifiers to be only one byte for the components allowing for a maximum of just 256 components of every kind, this already yields $(2^8)^{2^8} = 2^{2048}$ possible layouts for every sensor type and $(2^{2048})^3 = 2^{6144}$ for the system, multiplied by $(2^{2048})^2$ configuration states for e_lo and e_hi respectively. This search space of $2^{6144+4096} = 2^{10240}$ for the system configuration states is multiplied by the other inputs variables, *viz.* 8 bits for each sensor, one for each switch, presence detector, and light, giving a maximum search space of $2^{256\cdot(8+1+1+1)} = 2^{2816}$. The total search space of $2^{10240+2816} = 2^{13056}$ can obviously only be handled by a symbolic proof. By considering the infrastructure configuration (all identifiers attributes, and the configuration) as quasi-static we can reduce the search space dramatically to 2^{2816}; this represents the absolute maximum with all possible sensors, detectors, switches and lights connected. This further shows that for realistic examples simple enumeration of all states is not enough, and more advanced techniques (such as a lightweight SAT checker) are needed at run-time; self-verification is more than just checking property instances.

Of course the complexity of a proof can not be measured by the number of variables involved alone. Consider the arithmetic average function which is used to combine several sensor values into one. When implementing this in hardware one might chain multiple additions with a multiplication. While hardware adders are relatively easy to check, optimized hardware multipliers are very hard to verify since the usual branch cutting approaches of model checkers and solvers fail to reduce the search space for those particular circuits. Even a simple 32 bit multiplier already yields a 64 bit search space, requiring very elaborate proof methods to reduce this search space [13], [14]. In the example, we have to multiply the sum of the sensor values with the reciprocal of the number of sensors. Obviously the sum of the sensor values is changing frequently, but the number of sensors assigned to a light source only changes when the system is reconfigured and can thus be considered as quasi static. By verifying the multiplier with one particular factor considered static every time the configuration changes we can reduce the search space from 2^{64} to 2^{32}.

V. Conclusions

In this paper, we have proposed a basic methodology to develop *self-verifying systems*, which continue their verification at run time after deployment. This gives engineers more time, more resources and more information to successfully finish the verification. Self-verification goes beyond self-testing (which addresses failures in the production process), towards proving functional correctness.

For the properties to be verified, we distinguish *global* properties, concerning all system states, and *relational* properties, relating a pre- and a post-state. The latter suggests the use of the SysML modeling language together with OCL as a specification language for self-verifying systems.

Proof at run time has to be automatic, and at the same time has to have a small memory footprint. To reduce the search space of automatic proof methods, we have introduced the concept of *quasi-static* attributes, which are those system which change infrequently, so instead of trying to prove verification properties for all values of these quasi-static attributes, we only prove it for fixed values whenever they are set; a typical application example of quasi-static data are configurations.

We have demonstrated our methodology with a small first case study: we have shown how to model the system behaviour in SysML/OCL, and how by treating configuration data as quasi-static we could reduce the state space by a factor of 2^{10240} even in this very simple scenario.

These are but small first steps, and need to be followed by more work to make self-verification applicable, yet we believe that self-verifying systems could provide the key to closing the verification gap, and make the cyber-physical systems which surround us every day more correct, and hence more safe.

References

[1] 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.

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

[3] 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, pp. 1–4.

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

[5] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," *Hybrid Systems*, pp. 209–229, 1993.

[6] C. Zhou and M. Hansen, *Duration Calculus: A formal approach to real-time systems*. Springer, 2013.

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

[8] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers, 2007.

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

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.

[11] O. C. Z. Gotel and A. C. W. Finkelstein, "An analysis of the requirements traceability problem," in *IEEE International Conference on Requirements Engineering (ICRE 94)*, 1994, pp. 94–101.

[12] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[13] R. Kaivola and N. Narasimhan, "Formal verification of the pentium/-sup/spl reg//4 floating-point multiplier," in *Design, Automation & Test in Europe (DATE), 2002*. IEEE, 2002, pp. 20–27.

[14] 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 *Design, Automation & Test in Europe (DATE), 2016*. IEEE, 2016, pp. 1048–1053.