Let's Prove it Later — Verification at Different Points in Time

Martin Ring[1] and Christoph Lüth[1,2]

[1] Deutsches Forschungszentrum für Künstliche Intelligenz, Bremen, Germany
[2] FB 3 — Mathematics and Computer Science, Universität Bremen, Germany

Abstract. The vast majority of cyber-physical and embedded systems today is deployed without being fully formally verified during their design. Postponing verification until after deployment is a possible way to cope with this, as the verification process can benefit from instantiating operating parameters which were unknown at design time. But there exist many interesting alternatives between early verification (at design time) and late verification (at runtime). Moreover, this decision also has an impact on the specification style. Using a case study of the safety properties of an access control system, this paper explores the implications of different points in time chosen for verification, and points out the respective benefits and trade-offs. Further, we sketch some general rules to govern the decision when to verify a system.

1 Introduction

Contemporary embedded and cyber-physical systems have become so commonplace that we, almost unconsciously, rely on their correct functioning — we just expect our smartphone to work. This is contrary to the fact that these systems have reached a complexity where the verification of their correct behaviour becomes prohibitively expensive. Subsequently, a full correctness proof is only ever done for the most safety-critical systems. For all other devices, errors during the design process may remain undetected in the final product. This is due to the way these systems are currently designed.

The current design flow for embedded and cyber-physical systems is (idealized) as follows: we first *specify* the system's intended behaviour, then construct a *model* of the system and finally an executable *implementation*. Some of these steps may be conflated or missing; e.g. in model-based specification, the specification is the model, or one may generate an implementation from the model. In this design flow, *verification* refers to all activities which show that the implementation of the system satisfies its specification [6].

Current verification techniques such as theorem proving, model checking, static analysis or testing are conducted at design time and finished before deployment, for two reasons: firstly, we want to make sure the system has no errors before putting it into operation, and secondly, it is not entirely clear how to conduct verification at runtime. But this approach has the drawback that the time for verification is limited; errors which are not caught by the time the system is going into operation will remain undetected and may later on have unintended, unpleasant, or even catastrophic consequences.

On the other hand, verification does not necessarily need to terminate with the end of the development. In *runtime verification*, we check whether a particular run of the system satisfies desired properties. This has the advantage that we do not need to stop verification if we deploy the system, and checking whether a specific run of the system satisfies the desired property is of lower complexity compared to model-checking [7]. The drawbacks are that it may be costly to continuously monitor the behaviour of the system at runtime, and once we find an error, it may be too late to do anything about it. This is particularly true for hardware, and systems where the split between hardware and software is decided rather late in the development process.

The idea of *self-verification* is to investigate the middle ground in between: verify properties of the system as soon as practically possible, but as late as necessary. In other words, verification does not terminate with deployment, but is also not kept until the last moment. The present paper investigates the idea of self-verification as proposed in [8, 11] further. The key contribution is to examine the implication of self-verification on the development process. We do so by means of a case study, an access control system, which is simple to understand yet offers subtle effects and is easy to visualize.

The paper is structured as follows. Section 2 introduces the basic concepts of self verification, which are elaborated more concretely in Section 3 using the case study, an access control system. Section 4 shows our approach to realizing self-verifying systems, and Section 5 concludes with a general discussion of the wider applicability.

2 Self-Verification

Modern cyber-physical systems are designed to be versatile, such that they are able to handle numerous operating contexts and operate in many different environments. Thus, they have a large number of parameters which become instantiated at runtime. The key advantage of self-verification is hat after deployment, the concrete values of these parameters may become known for verification. Some may be instantiated early on after deployment, and not change after that at all, or only very infrequently; others may change, but not that often; and even others may be sensor data which are read in small intervals, but where the rate of change may be limited. All of this information may be utilized at runtime for more efficient verification.

This observation hinges on the fact that proving a property ϕ depends, *inter alia*, on the number of free variables in ϕ, and that parameters as mentioned above usually occur as free (or universally quantified) variables in ϕ. Then, proving $\phi\left[\frac{t}{x}\right]$ with a ground term t instantiated for x is typically orders of magnitude easier than proving ϕ.

Self-verification provides some challenges. At runtime, we do not have as many resources in terms of memory and computing power as at design time, and

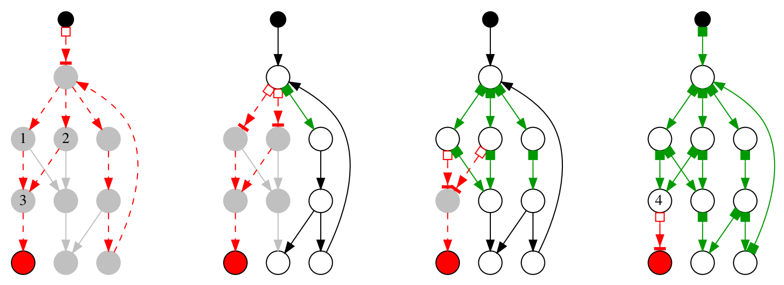

Fig. 1. Four different points in time chosen for verification, from design time (leftmost) to runtime (rightmost). Trigger transitions are marked with small boxes; they trigger verification tasks which show that every possible path through the state space which does not include other trigger transitions is safe. Green boxes mark successful verification, and red boxes mark failed verification tasks. The solid red state is unsafe; it violates the safety property ϕ. Grayed-out states are not reachable, because they come after a failed verification (open red box). Design time verification (on the left) would identify the system as erroneous and prohibit its execution. Second to left, the system is verified early after deployment and thus is allowed to execute only a small fraction (6 transitions) of the system, blocking two transitions and leaving 6 transitions unreachable. Third to left, most of the system is executable (11 transitions) but two transitions are blocked and one transition is not reachable. The rightmost example allows all but one transition. Note that in the last example the system gets deadlocked in state 4 when taking the leftmost path.

we need to transport the proof obligations derived from the specification into the runtime environment. So, self-verification needs a design flow to support it: a format and logic in which to encode the properties at design time, and lightweight proof engines which run under the resource constraints of an embedded system. We will show in Section 4 how such a design flow can be implemented.

However, the focus of the present paper is to investigate the effects of self-verification on the development. That is, we want to explore *when* to prove properties and which ones, and we want to investigate how self-verification interacts with the development process.

Comparing self-verification to runtime and *a priori* design time verification on a more abstract level, we consider specific runs of the system $\langle\sigma_i\rangle_{i\in\mathbb{N}}$, consisting of states σ_i, and a safety property ϕ. Usual design time verification proves the general property that for all runs, $\forall i.\, \phi(\sigma_i)$, i.e. the safety property holds for all states. In OCL and related formalisms, this is achieved by an inductive argument, showing that we start in a safe state, $\phi(\sigma_0)$, and that from a safe state we can only get to a safe state, $\phi(\sigma_i)$ implies $\phi(\sigma_{i+1})$. Runtime verification, on the other hand, considers whether a specific run satisfies $\forall i.\, \phi(\sigma_i)$ and does not restrict the transitions of the system; unsafe states can be reached, but this is always detected. In self-verification, instead of restricting transitions, we classify them into trigger transitions and ordinary transitions. The idea is that when the system goes through a trigger transition $\sigma_i \rightarrow \sigma_{i+1}$, self-verification shows that all states σ_k reachable with ordinary transitions from σ_{i+1} are safe, i.e. $\phi(\sigma_k)$. If another trigger transition is reached, the self-verification is run again. Note that the classification of trigger transitions and ordinary transitions depends on the particular ϕ, and is a design decision (see Section 3 below). *A priori* and runtime verification can be seen as extreme cases of self-verification: in design time verification only one transition (the one leading to the initial state of the system) is classified as a trigger transition, while in runtime verification every transition is a trigger transition. Figure 1 illustrates the effect of different sets of trigger transitions for one system. Because the effort to state and prove ϕ increases with the number of states we want to cover, self-verification allows us to strike a balance: we may prove ϕ with little effort for a small number of states, and so have to reprove it more often, or we may prove ϕ for more states, but with more effort.

When we specify the desired behaviour of the system with design time verification, we need to state the required preconditions very precisely — they need to be strong enough to be able to actually show that the system globally satisfies the specified properties, and to preclude unwanted behaviour, but weak enough to still allow all desired implementation. If we move verification into runtime, we can relax preconditions at design time, allowing for more readable specifications and speeding up the development process. Consider Figure 1 again: to make the system usable as well as correct, one would have to, e.g. refine the specification (or the implementation) to exclude the transitions from states 1 and 2 to 3. With self-verification, we can allow a more liberal specification or implementation and still remain safe, making the development process easier.

Thus, in essence specification becomes easier and faster to write, and moreover we are liberated from having to prove everything *a priori*, and can instead adapt the proving strategy to the problem at hand.

3 Case Study

In the following, we will demonstrate our methodology in a case study (building loosely on Abrial [1]). The case study is simple enough to be easily understood, yet complex enough to show the subtle effects of verification at different points in time.

3.1 Informal Description

To motivate our case study, think of a building where fine-grained access control is needed for security or safety reasons, e.g. a nuclear power plant, but which also needs to be able to be evacuated very fast in the case of an emergency. In that case, we want to be able to eliminate access control (to allow fast evacuation) and just open some of the doors in such a way that all users are able to get out, but no user gains access to a room where they are not allowed to enter.

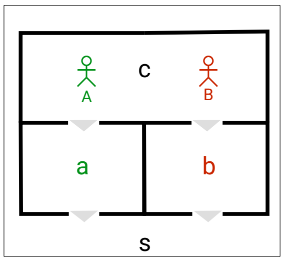

Fig. 2. Example of a very simple building. The user with card A is authorized for room a, user B is authorized for room b, both are authorized for rooms c and s. Room s is the only safe room (it is the outside). The situation shown violates the safety property.

More precisely, we have a *building* consisting of several *rooms*. The rooms are connected by *doors*, which are unidirectional (think of turnstiles; normal two-way doors are an obvious generalization). Thus, doors lead from one room to another one, which is equivalent to each room having a set of entries and exits.

Users are represented in the system by *cards* which regulate the access to rooms. (In the following, we use cards and users interchangeably; the formal specification only has cards.) Each card authorizes access to a set of rooms, by restricting passage through the doors. The access control system operates in two modes: in normal mode, a door may only be passed (using a card) if the card authorizes access to the room the door is leading to. However, we can declare an emergency for the whole building; in that modus, some doors are opened, allowing anyone to pass through.

Opening doors in an emergency is subject to two *safety properties*: firstly, it should allow any user (card) to eventually arrive in a safe room, and secondly, it should not allow any user to enter a room they are not authorized to. A subset of rooms is considered to be *safe*; in the simplest case, this can just be the outside modelled as a room. As an example for the necessity of the safety properties, take the nuclear power plant: even in case of an emergency, one would not want anybody to exit through the reactor core.

This rather innocuous specification allows some subtle effects. Consider the simple building in Figure 2; the depicted situation violates the safety property, as in case of an emergency, we cannot disable access control and open the doors in such a fashion that neither user A or user B are allowed to access rooms they are not authorized to (rooms b and a, respectively), and both are able to get to a safe room (s).

Hence, we need to prevent a situation like this from happening. This could be done by

- either restricting the layout of the building in such a way that situations like this do not happen (this is what is usually done, with layouts were corridors are the default escape route, and users do not have to traverse long sequences of rooms);

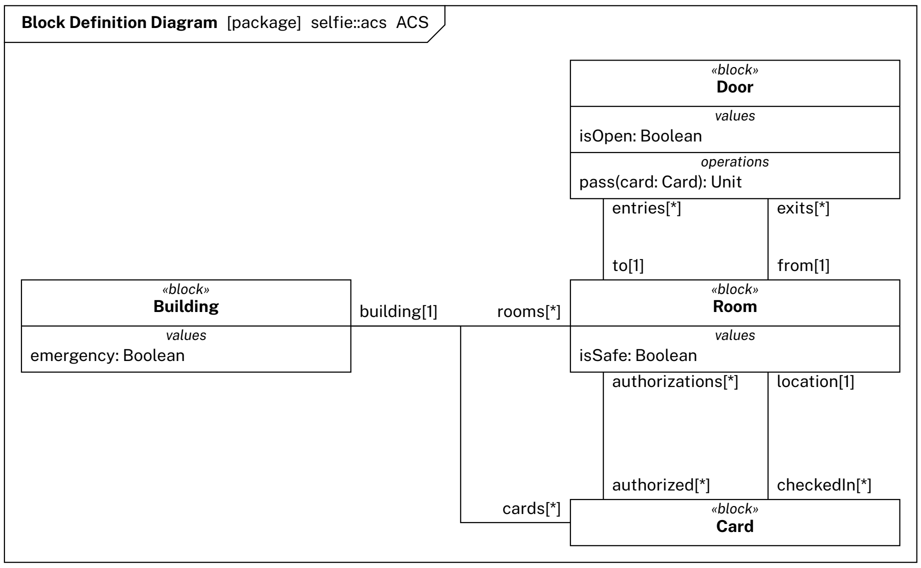

Fig. 3. Formal specification of an access control system.

- or by restricting the authorizations of the cards in such a way that a situation like above does not happen;
- or by checking that *before* a users enters a room no situation violating the safety property like above is created.

3.2 Formal Specification

We can now give a formal specification of our access control system. We will use a subset of SysML[10] and OCL[9], where block definition diagrams (BDDs, the SysML equivalent to UML class diagrams) model the structure of the system, and OCL constrains the dynamic behaviour.

In Figure 3, we can see blocks modelling the building, doors, rooms and cards respectively. The building has a Boolean attribute *emergency*. A door leads from exactly one to another room, but a room may have many (or no) entries and exits. A door may only connect rooms which are part of the same building:

```
context Door
  inv: from.building = to.building
```

Furthermore cards are also associated to buildings and may only authorize access to rooms which belong to the same building:

```
context Card
  inv: authorizations→forall(r| r.building = self.building)
```

Cards have a set of *authorizations* (rooms which the holder of the card is allowed to enter) and exactly one *location*, which determines the current location of the card, and which must always be contained in the set of authorizations. On the other hand, rooms have a set of *authorized* cards (those cards which have the room in their set of authorizations), and a set of *checkedIn* cards (the set of cards whose location is this room).

```
context Room
  inv: checkedIn→forall(p |authorized→contains(p))

context Card
  inv: location→forall(r |authorizations→contains(r))
```

Rooms have a Boolean attribute *isSafe* which determines whether the room is safe during an emergency. A door has a method *pass*, which determines whether a given card is allowed to pass. This is the case if either the door is open (see immediately below), or if the card is in the room this door is opening from, and the card is authorized for the room the door is opening to. We have encapsulated this precondition as an OCL function *mayPass* in order to reuse it later. The postcondition of the *pass* method is that the *location* of the card has changed to the room the door is opening to. Doors are only allowed to be opened in case of an emergency.

```
context Door
  def: mayPass(card: Card): Boolean =
    isOpen or from.building.emergency
    and card.authorizations→contains(to)
  inv: isOpen implies from.building.emergency

context Door::pass(card: Card):
  pre: mayPass(card) and card.location = from
  post: card.location = to
```

We now want to formalize the safety property: in an emergency, users can always reach a safe room, yet no user has access to a room they are not authorized to. To formalize a user being able to reach a room, we formalize the notion of recursive *access*, which models the traversal along a sequence of connected rooms: users have access to the room they are currently in, and recursively to all rooms which can be reached through doors which may be passed (i.e. rooms which have an entry from an accessable room that this card has access to). We formulate this notion as an OCL function *hasAccess* which for a given room determines whether a given card has access to this room. Since OCL does not allow non-terminating functions we pass the set of already traversed rooms to the helper function *hasAccess$* such that we do not traverse cycles:

```
context Room
  def: hasAccess(card: Card): Boolean = hasAccess$(card,Set{})
  def: hasAccess$(card: Card, visited: Set(Room)): Boolean =
    card.location = self or
    visited.excludes(self) and entries→exists(e |
```

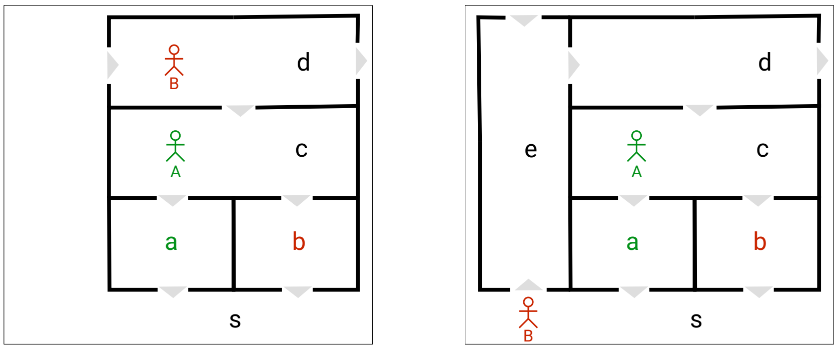

Fig. 4. Situations which are safe. On the left, user B cannot enter room c until user A has left. On the right, a similar situation, but the user B may have taken the long path through room e and d quite unnecessarily before not being able to proceed further.

```
      e.mayPass(card) and
      e.from.hasAccess$(card, visited→including(self)))
```

We can now specify the safety properties: firstly, that users can always reach a safe room, and secondly, that users only have access to rooms they are authorized for:

```
context Card:
  inv safe1: building.rooms→exists(r |
    r.isSafe and r.hasAccess(self))
  inv safe2: building.rooms→forall(r |
    not r.authorized→contains(self) implies not r.hasAccess(self)))
```

3.3 When to Verify

In order to preclude an unsafe situation as in Figure 2, we have to show our system satisfies the safety property. Of course, in full generality — universally quantified over all buildings and all authorizations — the safety property does not hold; we can easily find counterexamples (such as Figure 2). If we want to show the safety property at design time, we have to formalize conditions which are sufficient for the safety property (i.e. preclude unsafe buildings).

With self-verification, we can show the safety property after deployment, at different points in time:

(a) right after deployment to a specific building, for all possible cards, authorizations and allocations of users to rooms; or
(b) after authorization has changed, for a specific building, but for all possible allocations of users to rooms; or
(c) when a user requests access to a different room: if the new configuration of the user in this different room is unsafe, access is not granted.

In case (a), we would either need an explicit and sufficient characterization of "every user always has a safe exit route", or we need to search a lot of instances (all paths for all users from all rooms). For most buildings, we will be able to find counterexamples of unsafe configurations of users and access rights, but we may be able to restrict access rights in such a way that we can prove the safety property. If we can prove the safety property at this point, we are done, but this may not always be possible.

The other extreme case is (c); this is fairly straightforward to verify, but might be inconvenient to the user. (Thus, this is an example of making a system safe by restricting its availability.) Consider the situation in Figure 4 with the same authorizations as in Figure 2. On the left, user B cannot enter room c until user A has left, because otherwise we would have the situation from Figure 2 which is not safe. This might result in situations like on the right of Figure 4, where user B might take a long tour through room e to room d only to find they cannot proceed any further.

A good compromise is case (b): we verify the safety property each time the authorizations change, for a specific building and specific authorizations. In most cases, this should be reasonably efficient — the search space is through all possible allocations of users to rooms — but still precludes unsafe allocations.

Note how self-verification allows us to relax the development process: because we can prove the safety property at runtime, we do not need to specify all its preconditions at design time (here, we do not need to characterize the preconditions to make buildings and authorizations safe). This makes the development process more *agile* without compromising safety.

4 Realization

4.1 A Design Flow for Self-Verification

Our design flow targets hardware-software co-design for embedded and cyber-physical systems. As demonstrated in Section 3, we use a subset of SysML (block definition diagrams and state machine diagrams[3]) together with OCL as a specification formalism. Block definition diagrams and state machine diagrams can be given a formal semantics (which is not the case for all SysML diagrams), so our specifications have a mathematically well-defined, formal meaning. This is indispensable if we want to perform formal correctness proofs. Figure 5 sketches the design flow.

We have developed a textual representation of block definition diagrams and state machine diagrams (in the spirit of USE [4]), which we use in our design flow. Figure 6 shows an excerpt; parts of the corresponding OCL specifications have been shown in Section 3 above. We can also use commercial tools like Astah SysML, but their OCL support tends to be not as sophisticated. Instead, we make use of the OCL implementation of the Eclipse Modelling Foundation.

[3] The case study only uses block definition diagrams.

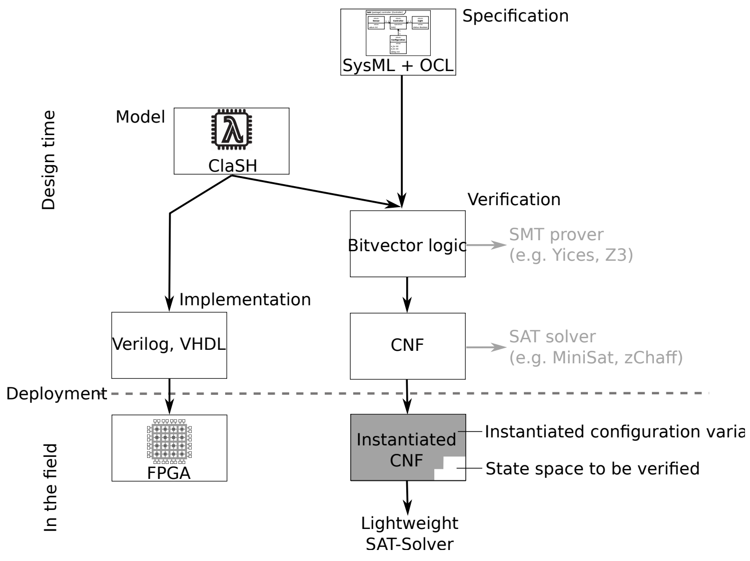

Fig. 5. A design flow for self-verification.

Moreover, our textual representation makes the design flow fairly light-weight, allowing users to employ any editor and versioning system at their disposal.

The implementation is given as an executable *system model*. To stay independent of a specific programming language, we use the functional hardware description language CλaSH [2] as modelling language, since it allows us to simulate the system as well as synthesize an implementation in VHDL or VeriLog. Another possibility with more commercial traction would be SystemC, but that has a less clear semantics and it is embedded in C++, technically a lot more awkward to handle (in CλaSH, adding proof support was merely a question of adding an additional backend; in SystemC, we do not even have an explicit representation of the model to start from).

Our tool chain reads the SysML and OCL specification, performs the appropriate type checks, reads the CλaSH model, and generates the corresponding first-order proof obligations in bitvector format (first-order logic with limited width integers as datatypes). The proof obligations are essentially obtained by taking a representation of the system model in bitvector logic, and showing they satisfy the OCL constraints (pre/postconditions and invariants). They can be either processed at design time by an SMT prover such as Yices or Z3, or transferred to runtime. Proving at runtime is either performed by an SMT prover running on the target system, if the latter is powerful enough, or by converting the proof obligations into conjunctive normal form (e.g. using the Yices prover) before transferring it to the target system, and using a SAT solver at runtime (either as a lightweight software SAT solver [3] or even a hardware SAt solver [12]). We have evaluated this design flow using a ZedBoard (which consists of a Xilinx FPGA controlled by an ARMv7 core), see [11]. Our evaluation has shown that

```
bdd [package] selfie::acs [ACS]
-------------------------------

block Building
  references
    rooms: Room[*] <- building
    cards: Card[*] <- building
  values
    emergency: Boolean
```

Fig. 6. Textual representation of the SysML block definition diagram (bdd). The excerpt shows the bdd for Building.

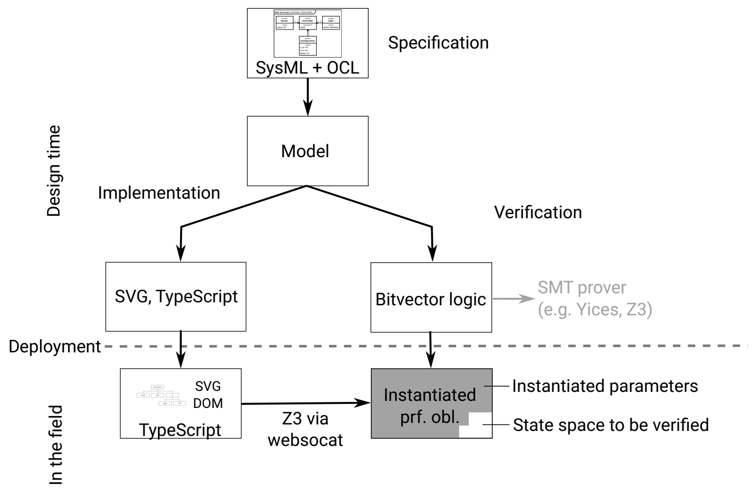

Fig. 7. Design flow adapted to our demonstrator.

verification at runtime can cope with systems where *a priori* verification fails, precisely because of the reduction in search space by instantiating parameters which become known at runtime.

4.2 The Demonstrator

If we implement the case study in our usual design flow, we derive a hardware implementation, e.g. on an FPGA. In order to explore the implications of proving at different points in time, and to demonstrate the effects of self-verification in an easily accessible setting, we implemented the case study as an interactive demonstrator.

Simulating the hardware turned out to be very slow, so instead we chose to adapt our flow: the implementation is an interactive SVG, with the dynamic behaviour implemented in TypeScript. The core of the system is generated as implementation stubs, using an adapted form of our design flow (see Figure 7). We

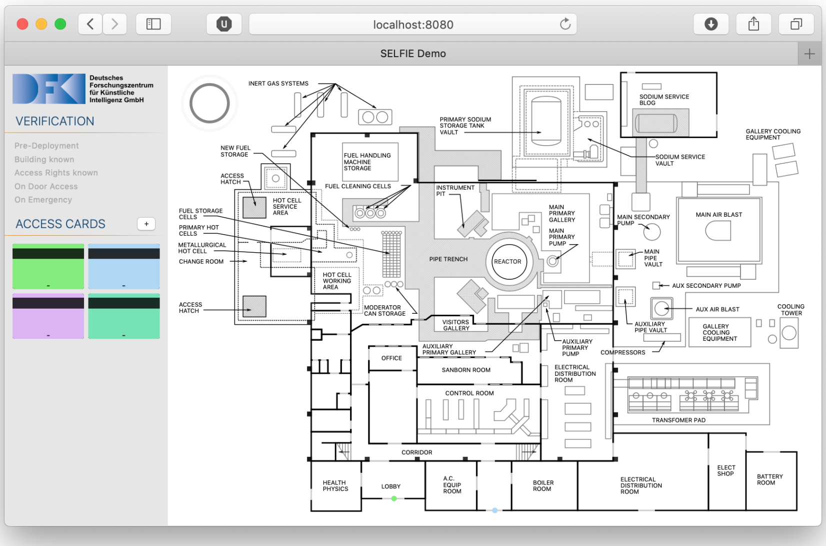

Fig. 8. The demonstrator is implemented as an interactive SVG document, displayed here in a web browser.

have chosen TypeScript [5] as the target language (TypeScript is like JavaScript, but with added type security), because it allows us to dynamically modify the abstract syntax tree (the DOM) of the SVG. This allows the demonstrator to be displayed and run on any recent web browser. In addition to the specified behaviour we manually implemented means to add and remove cards and change their access rights, and reading building topologies from a non-interactive SVG. We have implemented access cards (and implicitly their owners) as automated agents which randomly roam the building. This allows us to observe the implications of the different points in time of the verification; for example, the behaviours mentioned for case (c) in Section 3 above manifest themselves in agents hovering in one place unable to proceed because of the violation of the safety property this would incur.

The generated SMT proof obligations are a general equivalence proof which can be processed by an SMT prover at design time. As mentioned above, the prover quickly finds counter examples since our specification can easily be violated in general. By adding runtime information in the form of assertions, we refine the instance on the fly. This was realized by establishing a WebSocket connection between the SVG and the Z3 prover. For this, we use the *websocat* utility, which wraps a WebSocket server around a command-line program. This allows us to load the general proof and then incrementally send assertions restricting the state space.

Technically, the arbitrarily mutable state of our simulation is in principle not compatible with the monotonous nature of adding assertions: assertions can only add information but not change or remove. Fortunately, SMT-LIB (the common language used by most SMT provers) allows us to use scopes (with the commands *push* and *pop*) for this. In order for this to work, we introduce a fixed order in which information is added, which is based on the order of execution in the system, ideally corresponding to the frequency of change. First we add the general building topology, then the access rights, and after that, the tracked locations of the card holders. Between every assertion, we save the current size of the assertion stack with the *push* command. If any information changes, we remove the assertion with the now outdated information as well as any assertion which came afterwards. Then we only need to add the updated assertions. Depending on the point in time chosen, we can check satisfiability anywhere between.

An interesting feature of our implementation is that we did not implement any algorithm which opens the doors. Instead, we use the prover to give us a model of the existentially quantified safety property, which states that there must be a safe way to exit (i.e. a set of doors to open in case of emergency). Through self-verification not only did we not have to characterize buildings, access rights or safe paths through the building, we even did not have to implement a path finding algorithm at all.

The demonstrator is shown in Figure 8. It connects the implementation to the proof engine running the SMT instance. We can manually choose one of the three different information levels for the proof, which result in different assertions being added as well as different triggers for the proof.

Users can explore the consequences of the different points in time for the self-verification. For example, if they choose to verify early on (after a new card has been added or access rights change) and add a lot of cards, they will notice a considerable slow-down when adding new cards or changing access rights. If they choose to verify late (before a user enters a room), and construct situations like in Figure 4, they will realize how users congregate in front of a room unable to get in. (The demonstrator is intended to be used together with additional interactive explanation, not stand-alone, as situations like this will have to be constructed consciously.)

The source code of the demonstrator is publicly available on GitHub.[4]

5 Discussion and Conclusions

This paper has elaborated on earlier proposals of self-verification — systems which are not verified *a priori*, during the design phase, but where the proof obligations incurred during the development are postponed until after deployment, and are proven at runtime. This makes proofs more easy, as we can instantiate a number of the parameters of the system which are unknown at design

[4] https://github.com/DFKI-CPS/selfie-demo

time, but become known at runtime. This reduces the state space, turning the exponential growth of the state space — the bane of model-checking — into exponential reduction. Self-verification is supported by a tool chain we have developed, which allows specification in SysML/OCL, system modelling in CλaSH, and verification using SMT provers and SAT checkers.

It should be noted that self-verification is in no way intended to *replace* design time verification. If proof obligations can be shown at design time, they should by all means be discharged; however, self-verification offers a different way to tackle proof obligations which can *not* be shown at design time, supplementing design time verification, and offering the designer to pick the best of all possible worlds.

5.1 When to Prove

The focus of the present paper has been to investigate the implications and consequences of the point in time at which the proof of safety properties take place at runtime. Generally, the earlier we can prove, the more general the proven safety property, but the larger the search space is and subsequently the longer it will take. How to pick the right points in time depends on the actual system and is very much a design decision. In future work, we want to further investigate how the designer can be assisted in this decision; in particular, the system should suggest which variables offer the most reduction in proof time when instantiated.

However, we have made a number of observations which can help to assist in finding the right set of trigger transitions. The set of trigger transitions should be large enough such that verification tasks can be completed in a timely manner (again, acceptable verification times depend on the concrete use case), but reduced in a way such that no critical transition is included. Trigger transitions might be prohibited by self-verification in case the specification is violated (fails to verify in the concrete instance), so critical transitions should not be included in the set of trigger transitions: e.g. if we verify the existence of an escape route in case of an emergency it is clearly too late to handle failure. On the other hand administrative operations like changing access rights are far better suited to be included as trigger transitions, since a potential failure is presented to a trained user of the system. Lastly, one should avoid transient states (e.g. a user is inside a security gate) which can only be left through trigger transitions since self-verification may lead to a system dead-locked there, as in Figure 1.

5.2 Conclusions

The vehicle of our investigations was a case study consisting of an access control system, which is parameterized in many dimensions (the building under control, the access rights, the users) that can be instantiated at different points in time. In order to make our results concrete and tangible, we have developed a demonstrator — the access control system implemented as an interactive SVG, which can be viewed and run in any web browser. Users can directly experience the effect of choosing different verification triggers.

The demonstrator also exhibits the general applicability of self-verification and the versatility of our tool chain, which could be adapted to support a different implementation platform (SVG and TypeScript instead of CλaSH) with moderate effort.

This raises the question of the general applicability of the approach. As presented here, some kinds of safety-critical systems could not be addressed adequately, namely fail-safe systems, where there is no default safe state which we can always revert to if self-verification does not succeed. On the other hand, an attractive avenue for further exploration is "just-in-time verification", where one tries to prove properties at run time as they are needed.

References

1. Abrial, J.R.: System study: Method and example (1999), http://atelierb.eu/ressources/PORTES/Texte/porte.anglais.ps.gz
2. Baaij, C., Kooijman, M., Kuper, J., Boeijink, W., Gerards, M.: ClaSH: Structural descriptions of synchronous hardware using haskell. In: Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools. pp. 714–721. IEEE Computer Society, United States (9 2010). https://doi.org/10.1109/DSD.2010.21
3. Bornebusch, F., Wille, R., Drechsler, R.: Towards lightweight satisfiability solvers for self-verification. In: 7th International Symposium on Embedded Computing and System Design (ISED). IEEE (2017)
4. Gogolla, M., Richters, M.: Development of UML Descriptions with USE. In: Shafazand, H., Tjoa, A.M. (eds.) Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002). pp. 228–238. Springer, Berlin, LNCS 2510 (2002)
5. Hejlsberg, A.: Typescript (2012), https://www.typescriptlang.org
6. IEEE: IEEE std 1012-2016, IEEE standard for software verification and validation. Tech. rep., IEEE (2016)
7. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (May 2009). https://doi.org/10.1016/j.jlap.2008.08.004
8. Lüth, C., Ring, M., Drechsler, R.: Towards a methodology for self-verification. In: 2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO). pp. 11–15 (Sep 2017). https://doi.org/10.1109/ICRITO.2017.8342390
9. OMG: Object Constraint Language (OCL), Version 2.4 (February 2014), http://www.omg.org/spec/OCL/2.4/
10. OMG: Systems Modeling Language (SysML), Version 1.5 (May 2017), http://www.omg.org/spec/SysML/1.5/
11. Ring, M., Bornebusch, F., Lüth, C., Wille, R., Drechsler, R.: Better late than never — verification of embedded systems after deployment. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 890–895 (March 2019). https://doi.org/10.23919/DATE.2019.8714967
12. Ustaoglu, B., Huhn, S., Große, D., Drechsler, R.: SAT-lancer: A hardware SAT-solver for self-verification. In: 28th ACM Great Lakes Symposium on VLSI (GLVLSI) (2018)