Verification-driven Design Across Abstraction Levels

A Case Study

Nils Przigoda[1] Jannis Stoppe[2] Julia Seiter[1] Robert Wille[1,2] Rolf Drechsler[1,2]

[1]Group for Computer Architecture, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{przigoda,jstoppe,jseiter,rwille,drechsle}@informatik.uni-bremen.de

***Abstract*—For the development of complex systems – composed of hardware, software, or both – more and more high-level descriptions have been introduced over the past years. Starting from an informal specification, models of the system are created with the help of languages such as UML, SysML, or MARTE. Based on this model, an implementation is generated in a programming language such as C++, Java, etc. for software or SystemC, VHDL, etc. for hardware. Whereas various approaches for the verification of the single levels of abstraction exist, their application to a cross-level design flow is still to be considered. In this work, we evaluate this issue by providing a case study on a verification-driven design across abstraction levels. The results of this case study demonstrate the capabilities of existing methods as well as challenges and open issues to be addressed in future work.**

I. Introduction

With the increasing size and complexity of modern software and hardware, their design becomes more and more complex as well. In order to handle this complexity, higher levels of abstraction are considered. In the domain of software design, high-level languages such as C++, Java, etc. are already established for several decades. Besides that, the use of modeling languages such as UML (the *Unified Modeling Language* [1], [2]) prior to the actual implementation process has become an established step of many of today's design flows. This enables designers to create a formal representation (a *model*) of the intended system (including both structural and behavioral requirements), while, at the same time, implementation details are hided in these early design steps. Based on this "blueprint", an implementation (in C++, Java, etc.) is created as soon as the model is considered to sufficiently represent the original specification of the system.

Similar developments can be observed in the field of hardware design where the design of circuits and systems does no longer begin with a gate netlist. Instead, abstractions are applied which, eventually, have led to the *Register Transfer Level* (RTL) and the *Electronic System Level* (ESL). Nowadays, more and more hardware engineers also employ formal models provided in modeling languages to represent an initial description of the design to be built. For this purpose, profiles of the UML such as SysML (the *Systems Modeling Language*, [3], [4]) or MARTE (the *Modeling and Analysis of Real-Time and Embedded Systems* framework, [5]) are applied, which have been specifically adapted for hardware and system design. Similar to software design, an implementation (here in SystemC, SystemVerilog, etc.) is created as soon as the model has been completed.

Hence, the design flows for both software and hardware employ very similar first steps: The creation of a high-level description by means of modeling languages (UML or SysML, respectively) followed by the initial implementation using very similar programming languages (C++ or SystemC, respectively)[1]. However, the ever increasing complexity of today's systems did not only motivate the emergence of higher levels of abstractions, but also affected the actual design tasks. While a few years ago, the implementation process was the core activity in any design flow, verification – i. e., checking whether the design has been modeled or implemented as intended – dominates the entire process today. In fact, more than 40 % of the time and costs of the design effort are devoted to prove the correctness of a system rather than adding new functionality [6] – with increasing tendency.

As a consequence, engineers and researchers have started to develop methods for (automatic) verification for all abstraction levels. As an example, the *UML-based Specification Environment* (USE) [7] provides well-established methods that can be applied e. g., to automatically generate test cases for the respective UML/OCL models [8]. Besides that, researchers began to exploit formal methods for the verification of UML/OCL models, e. g., PVS [9], HOL-OCL/Isabelle [10], and KeY [11], as well as fully automatic proof engines including methods based on constraint programming (CSP) [12], [13], [14], description logic [15], [16], the modeling language Alloy based on relational logic [17], [18], or Boolean satisfiability (SAT) [19], [20]. While these approaches are usually introduced for consistency checking, i. e., to check if a model free of contradictions or not, also approaches for debugging [21], [22] and behavioural checks have been introduced [23], [24].

For the implementation level, the (verified) models provide a precise blueprint to be realized. While they can even be used to generate corresponding code stubs or initial implementations, a manual refinement is usually required afterwards. Due to this manual process, new errors might be introduced, i. e., an implementation might violate constraints that have been defined before. This motivates approaches which validate an implementation against its formal models (as recently considered e. g., in [25]).

Although impressive progress has been made in this regard, most of the verification approaches only consider single abstraction levels, i. e., either the correctness/consistency of formal models or of (initial) implementations. While those are important contributions, a gap between these abstraction levels remains. In this work, we evaluate this issue. We examine the current state-of-the-art of verification approaches as summarized above and consider how they can adequately be applied within an integrated design flow. Our focus is thereby on a verification-driven methodology which is supposed to detect errors as soon as possible. For this purpose, the initial

[1]For sake of clarity, we focus on UML and C++ in the following, i. e., a view from a pure software design perspective. However, the considerations from this paper can easily be transferred to the respective hardware-counterparts.

steps of the design flow are considered by means of a small practical example: an access control system (inspired by [26]). Starting with a textual specification, we consider both the creation of a formal model as well as an initial implementation – with a constant focus on verification.

Our case study illustrates the capabilities of the respective verification approaches and clearly shows which checks can be conducted best at which level. Furthermore, it demonstrates how an initially given model that may appear correct at first glance can be improved throughout the design process and eventually be realized as a (correct) implementation. By this, the advantages and shortcomings of verification methods solely proposed for models and implementations are discussed in the context of a verification-driven design across abstraction levels. This unveils open issues which shall be addressed in future work.

The remainder of this paper is structured as follows: The next section reviews the specification and provides details on the case study, i. e., the access control system to be implemented in this work. Afterwards, the design of this system on the modeling level and the (higher) implementation level is considered in Section III and Section IV, respectively, with a particular focus on the verification of the respective descriptions. In Section V, we discuss our findings and draw conclusions.

II. Considered Access Control System

We will study the application of verification approaches at the different abstraction levels by means of an access control system which has originally been presented in [26]. The goal of this case study is to develop a control system which grants access to buildings based on a person's authorization. This authorization is based on the ID of a magnetic card each person receives. Each building is equipped with turnstiles and card readers to check the card upon entry or exit.

In addition to this rather general description of the scenario, [26] provides a more detailed list of requirements to be realized which reads as follows.

1) Each person is given permanent authorization to enter certain buildings.
2) Entry and exit must be controlled by the system so that at any point, it is known which person is inside which building.
3) Each person possesses a magnetic card where a unique ID is stored. Access to buildings is controlled based on this ID.
4) At each entrance and exit of a building, a card reader is installed to check a person's magnetic card.
5) Each card reader is equipped with two lights, red and green, which can be turned on and off and are used to signal if the inserted card's owner is authorized to enter/leave the building.
6) Aside from card readers, each entrance/exit is also equipped with a turnstile.
7) Turnstiles are blocked until someone is allowed to pass based on the check of the person's card.
8) As each entrance and exit has its own turnstile, there are no two-way turnstiles to be used for both purposes.

Figure 1 sketches the access protocol to be implemented. A person approaches a blocked turnstile to enter or leave a building. When the card is inserted into the card reader, it is checked for the person's authorizations. If the person is allowed to enter or leave the building, then the green light turns on and the turnstile is open for at most 30 seconds. Upon expiry of the 30 seconds or once someone passes the

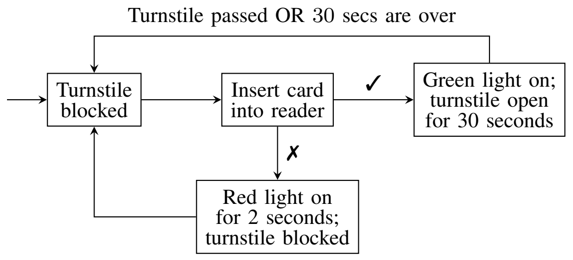

Turnstile passed OR 30 secs are over
Turnstile blocked
Insert card into reader
✓
Green light on; turnstile open for 30 seconds
✗
Red light on for 2 seconds; turnstile blocked

Fig. 1. Access protocol of the control system

turnstile, the green light is switched off and the turnstile is blocked again. If the person is not authorized for this building, then the red light is turned on for 2 seconds and the turnstile remains blocked.

III. Verification-driven Modeling

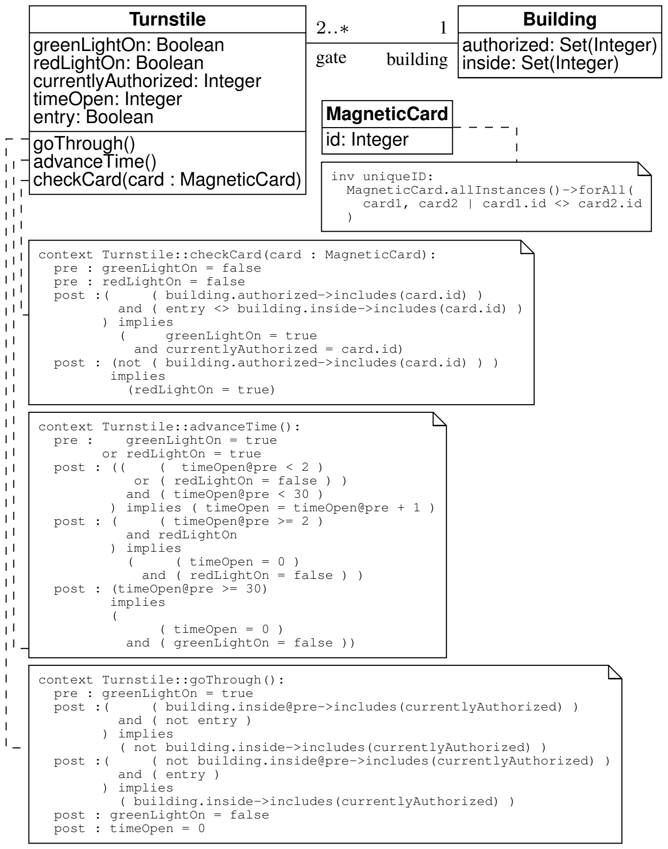
In a first design step, the structure and the behavior of the access control system are modeled using the means of the UML. This section briefly reviews the resulting model. Afterwards, we evaluate which verification tasks can already be conducted at this abstraction level and how this results in a significantly improved model.

A. Resulting Model

The model derived from the specification is shown in Fig. 2. It is composed of three classes which represent its main components, i. e., *Building*s, *MagneticCard*s, and *Turnstile*s. These components are modeled as follows.

- The class *MagneticCard* represents the magnetic card and consists of one integer attribute `id` which represents the unique ID. A magnetic card is possessed and used by a person (cf. Req. 3 Sect. II).
- The class *Building* represents the considered buildings and consists of the two attributes `authorized` and `inside`. The attribute `authorized` is a set of integers storing the IDs of all cards with which the building can be entered or left (cf. Req. 1 Sect. II). The attribute `inside` is a set of integers which is used to store the IDs of all persons currently inside the building (cf. Req. 2 Sect. II).
- The class *Turnstile* represents the entries/exits of the buildings. The card readers mentioned in Req. 4 and 5 in Section II and the turnstile are combined into a single class *Turnstile* as indicated in Req. 6. It consists of five attributes and three operations. While the Boolean attributes `greenLightOn` and `redLightOn` represent the state of the respective lights, the Boolean attribute `entry` states if the corresponding turnstile is an entry port (`entry` set to `true`) or an exit port (`entry` set to `false`). The integer attribute `currentlyAuthorized` stores the ID of the last magnetic card which has been inserted into the reader. The attribute `timeOpen`, also an integer, represents an internal clock to realize the timing constraints of the system.

The classes *Turnstile* and *Building* are connected by an association ensuring that each instance of class *Building* is connected with at least two *Turnstile* instances. A *Turnstile* instance can only be a part of one *Building* object. The invariant `uniqueID` ensures that the IDs of all *MagneticCard*s are unique as defined in Req. 3 in Section II.



```
inv uniqueID:
  MagneticCard.allInstances()->forAll(
    card1, card2 | card1.id <> card2.id
  )
```

```
context Turnstile::checkCard(card : MagneticCard):
  pre : greenLightOn = false
  pre : redLightOn = false
  post :(      ( building.authorized->includes(card.id) )
          and ( entry <> building.inside->includes(card.id) )
        ) implies
          (     greenLightOn = true
            and currentlyAuthorized = card.id)
  post : (not ( building.authorized->includes(card.id) ) )
         implies
           (redLightOn = true)
```

```
context Turnstile::advanceTime():
  pre :    greenLightOn = true
        or redLightOn = true
  post : ((     (  timeOpen@pre < 2 )
            or ( redLightOn = false ) )
           and ( timeOpen@pre < 30 )
         ) implies ( timeOpen = timeOpen@pre + 1 )
  post : (      ( timeOpen@pre >= 2 )
           and redLightOn
         ) implies
           (      ( timeOpen = 0 )
             and ( redLightOn = false ) )
  post : (timeOpen@pre >= 30)
         implies
         (
                ( timeOpen = 0 )
           and ( greenLightOn = false ))
```

```
context Turnstile::goThrough():
  pre : greenLightOn = true
  post :(      ( building.inside@pre->includes(currentlyAuthorized) )
          and ( not entry )
        ) implies
          ( not building.inside->includes(currentlyAuthorized) )
  post :(      ( not building.inside@pre->includes(currentlyAuthorized) )
          and ( entry )
        ) implies
          ( building.inside->includes(currentlyAuthorized) )
  post : greenLightOn = false
  post : timeOpen = 0
```

Fig. 2. Class Diagram of the Access Control System

The behavior of the three operations of the class *Turnstile* is defined by means of pre- and postconditions as follows.

- The operation `checkCard` checks whether a person inserting a *MagneticCard* may pass the respective turnstile. It receives a *MagneticCard* instance as parameter and can only be called if both lights are switched off in the current system state. If the ID of the magnetic card is included in the set of authorized IDs for the associated building, the green light is switched on and the ID of the currently inserted card is stored. Otherwise, the red light is switched on.
- The operation `advanceTime` realizes the timing constraints stated in the specification. It can only be called if one of the lights is switched on and increases the `timeOpen` attribute until its respective limit, depending on the state of the lights, has been reached. Then, the clock is reset and the respective light is switched off.
- The operation `goThrough` allows the currently authorized person to pass the turnstile and, by this, to enter or to leave the associated building. This operation can only be called if the green light is switched on. After execution, the respective ID is added to or removed from the set of IDs currently registered inside the building. Besides that, the green light is switched off and the clock is reset.

The resulting model could be considered to sufficiently represent the original specification from Section II and an implementation engineer could start implementing the given features. However, since this model already provides a formal description, we can use it to determine if any meaningful

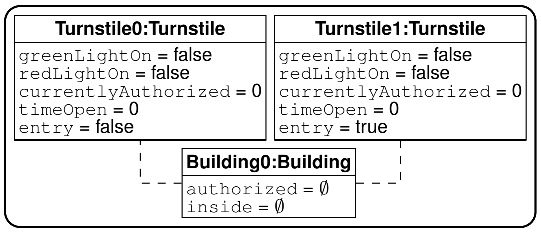


Fig. 3. A valid system state

implementation exists at all or if the model needs to be revised. Hence, before actually starting the implementation, we consider a selection of possible checks as a next design step.

B. Verification

Possible directions for the verification of models usually address two major concerns: Do the given model and all of its constraints allow for a valid instantiation/implementation (also known as *consistency checking*) and does the given model describe the desired behavior (demanding *behavior checking*)? In our case study, we are going to consider both consistency and behavior checking.

These kinds of verification problems, however, are undecidable when applied to an unbounded model which is the case for UML models. Datatypes such as integers have no upper or lower bound, the number of objects is not bounded, etc. To solve this issue, we apply bounded verification methods and restrict the concerned aspects, making the verification tasks decidable. Since the eventual implementation will be bounded as well, these limitations are not too strict.

1) Consistency Checking:

Checking the feasibility of a given model, it is essential to know whether at least one arbitrary system state satisfying all constraints provided by the model can be instantiated. If this is the case, the model is called *consistent*. Otherwise, the model contains contradictory constraints which prevent a valid instantiation – the model is *inconsistent*. In case of any inconsistencies, the model has to be revised prior to the implementation process because no implementation precisely realizing the model's constraints can result in an instantiatable system.

In order to conduct such a consistency check, different approaches have been proposed in the past. As an example, the *UML-based Specification Environment* (USE) [7] provides methods to generate a valid system state or to prove that no such system state exists. Whereas USE internally relies on enumerative methods, other approaches are based on theorem provers, e. g., PVS [9], HOL-OCL/Isabelle [10], KeY [11], and SAT solvers [19], [20]. In principle, all of them are sufficient for our purpose.

To conduct the most basic consistency check, we try to determine a valid system state where at least one class is instantiated (an empty system state with no objects at all does not provide any useful information and, hence, is explicitly not considered). More precisely, we ask the applied consistency checker to determine a system state such that the number of instances per class may vary from 0 to 5. This results in the (valid) system state shown in Fig. 3.

While this system state proves that the considered model is not self-contradictory, it does not provide a sufficient witness for our scenario as no instance of the class *MagneticCard* was created. If we are going to implement an access control system based on this model, we should be certain that there is at least one magnetic card to be used by a person. Hence, we slightly change the configuration of the consistency check and enforce the generation of a system state where all classes are instantiated at least once – including the *MagneticCard*.

This consistency check, however, turns out to be failing, i. e., there is no valid system state in which *all* classes are instantiated at least once. This information is important: Although the model described in Section III-A seemed complete and correct at first glance, it obviously includes a serious flaw. Without conducting consistency checks at this early stage in the design flow, this flaw might have remained undetected until the completion of a first implementation. Applying verification-driven design, which employs correctness checks as early as possible, avoids the resulting large debugging effort and the equally expensive revision of both the model and the implementation.

So far, the reason for the detected flaw is still unknown. But just like for the verification, automated methods may help here. A significant body of research has recently been conducted aiming at supporting engineers in *debugging* inconsistent models (see e. g., [22], [21], [27]). Based on the current configuration of the (failed) consistency check, both approaches automatically generate so-called *error candidates*, i. e., a subset of UML/OCL constraints which might explain the contradiction. Inspecting this subset instead of the whole collection of constraints significantly simplifies the identification of the flaw's origin.

In our case, these approaches return the invariant `uniqueID` as the only error candidate. Having a more detailed look unveils that this invariant indeed is self-contradictory: The iterator of the `forAll`-operation also checks `card1.id <> card2.id` for `card1` and `card2` being the same object. This means that the ID of a magnetic card has to be different from itself which is impossible. With this information, the engineer can easily fix this problem by revising this invariant to

```
context MagneticCard inv uniqueID:
  MagneticCard.allInstances()->forAll( card |
    (self <> card) implies (self.id <> card.id)
  )
```

Performing the consistency check with the revised model now leads to a valid system state including instantiations for all classes. This is a promising indication that the considered model indeed is a proper representation of the structure of the desired system.

2) Behavior Checking:
The checks from above validated that an implementation of the model from Section III-A exists, i. e., an instantiation in the form of a consistent system state of the access control system exists. Now, it remains to be checked whether such an implementation would also realize the desired behavior. For this purpose, automatic approaches have recently been proposed as well (see e. g., [23], [24]). They allow for the consideration of several verification tasks.

To ensure that any kind of dynamic behavior is possible in the model, we aim for checking the executability of the model, i. e., is it possible at all to conduct a sequence of (arbitrary) operation calls on an instantiation. More precisely, we try to determine a sequence of (valid) system states with valid operation calls between two successive system states – i. e., each transition has to include a calling state satisfying the precondition as well as a succeeding state satisfying the postcondition of the respective operation call. This is an important check: if a model does not allow an arbitrary sequence of operation calls, all further checks are rendered meaningless. Using methods such as proposed in [23], [24], we can prove that about a dozen of consecutive arbitrary operation calls indeed can be conducted on the model presented in Section III-A (including the already mentioned corrections).

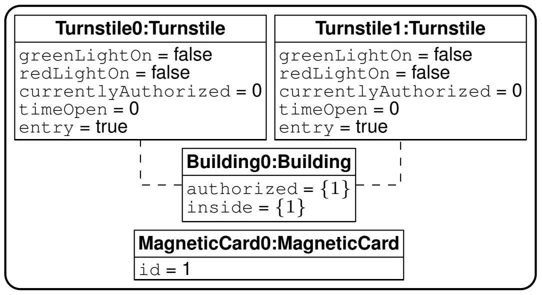


Fig. 4. A deadlock system state

Still, the question remains whether the operation calls are performed as intended or whether certain unwanted states can be reached. Other, more refined, checks focus on the evaluation of these issues. Guaranteeing that the model does not allow for any deadlocks is such an issue. This can be formulated as the query "Does the model allow for a system state in which no further operation calls can be performed due to violated preconditions or an invalid subsequent system state." The approaches mentioned above can handle such queries and, in our case, generate such a system state as illustrated in Fig. 4.

Here, none of the possible operation calls can be performed without violating a constraint. More precisely:

- In order to invoke `advanceTime`, either `self.greenLightOn` or `self.redLightOn` must be set to `true`. For both *Turnstile* instances, both elements are set to `false` in the system state shown in Fig. 4.
- In order to invoke `goThrough`, `self.greenLightOn` must be set to `true`. Again, this is not the case for both *Turnstile* instances in the system state shown in Fig. 4.
- In order to invoke `checkCard` an instance of the class *MagneticCard* is required which is either inside the building in case of an exit turnstile or outside the building in case of an entry turnstile. But since both instances of the class `Turnstile` are set to represent entries (`entry` set to `true`), the *MagneticCard* is identified as inside the building and, thus, the operation can not be invoked due to failed preconditions.

In addition to detecting a deadlock state (as intended by this check), another serious design flaw has been unveiled. While the association between *Turnstile* and *Building* requires that each building has at least two turnstiles, it is not guaranteed that there is always one entry and one exit – even though the relation $2..*$ was intended for this. Having the unwanted system state shown in Fig. 4, a designer can identify this problem manually and resolve it by adding the following invariant to the class *Building*:

```
context Building
  inv atLeastOneEntry :
    self.gate->exists( g | g.entry)
  inv atLeastOneExit :
    self.gate->exists( g | not (g.entry) )
```

This also resolves the deadlock problem.

Knowing that the model is executable and how deadlock states can be detected, another important aspect of behavioral models shall be considered: frame conditions. Frame conditions describe which model elements may change during the transition from one system state to another, triggered by an operation call. They are important because, thus far, only pre- and postconditions restrict the respective calling and succeeding states – allowing for plenty of arbitrary and unwanted changes.

For example, during the execution of `checkCard`, which requires a *MagneticCard* instance as a parameter, the ID of the *MagneticCard* could change although this is not an intended behavior and should be prohibited. To this end, the OCL command **`modifies only`** representing an *invariability clause* has been proposed in [28]. For each operation, this command allows to declare a list of all model elements which might be modified in combination with a scope. All other model elements are assumed to keep their respective value.

For the operation `checkCard`, defining the invariability clauses can be conducted based on the following considerations: The values of the elements `self.greenLightOn` and `self.redLightOn` indicate whether the check of the magnetic card was successful or not. The element `self.currentlyAuthorized` stores the ID of the magnetic card which has just been checked for authorization. Consequently changes of their values do make sense and all three elements have to be added to the set of invariability clauses. In contrast, the element `self.timeOpen`, which is never used in the postconditions, should not be affected by this operation and, hence, not be added to an invariability clause. This applies to all elements which do not occur in a postcondition. The remaining model elements mentioned in the postconditions require further inspection: `self.entry` is used in the postcondition, but a change of its value would change the direction of the turnstile – clearly an undesired behavior. Therefore, `self.entry` is not added to the invariability clause. In the same way, the designer decides that changes of `self.building`, `self.building.authorized`, and `self.building.inside` are not desired and, thus, they are not added to the invariability clause. Frame conditions for all other operations are defined similarly.

Obviously, defining these frame conditions is a highly manual approach thus far. Since a great deal of design understanding has to be considered for this purpose, it is unlikely that fully automatic methods will ever be able to completely handle this task. Nevertheless, first methods aiming for aiding this process are currently under consideration (see, e. g., [29]). Here, automatic analyses are performed which lead to suggestions for the respective classifications of the model elements. While the designer have to make the final decision on the frame conditions, these classifications may already provide a good starting point.

After adding all frame conditions, an operation of the model is clearly more restricted than before and, thus, it is advisable to re-run all the behavioral checks conducted thus far again. Considering the updated model, all these checks passed. As a consequence, we gained a significantly improved model which has been stripped from its flaws and additionally provides a more precise definition of its behavior.

Now, it would be advisable to conduct further, more application-specific checks, e. g., is it somehow possible to access a building with a magnetic card without authorization or is it possible to register one ID of a magnetic card in more than one building at the same time, etc. All these queries can be checked in a similar fashion as done above for executability or deadlocks: The respective desired/bad states and a bounded sequence of arbitrary operation calls are configured. Then, approaches such as [23], [24] can be applied to determine a corresponding sequence or to prove that no such sequence exists. Whenever the model is considered as sufficiently checked, it can be passed to the next stage of the design flow for an implementation.

Overall, the verification-driven design of the model as sketched above leads to a significant improvement in the quality of the resulting model. Although the model as described in Section III-A may have appeared correct at first glance, it included a significant number of flaws and/or imprecisions that would have complicated the implementation process or, in the worst case, would have led to an implementation which would not satisfy the described requirements. Using the methods exemplarily discussed above allowed for avoiding these flaws/imprecisions and has led to a much more mature model. Note that this still is no guarantee that *all* flaws/imprecisions are detected – but chances for these are significantly reduced.

IV. Implementation and Comparison to the Formal Model

After the formal model is considered complete and verified, the implementation process follows, i. e., the generation of a working, executable realization of the given model. As this is mostly conducted manually, this design step provides another source of errors. Hence, after completing the implementation, it is necessary to validate the result against the original formal model. In this section, we consider the corresponding design steps. First, the access control system specified in Section II and modeled in Section III is implemented in C++. Afterwards, approaches are applied which allow for a comparison of the resulting implementation to the model.

A. Implementation of the Model

The implementation derived from the model introduced in Fig. 2 and revised in Section III-B is shown in Fig. 5. More precisely:

- Lines 1–3 specify packages which are referenced in the implementation to keep the example self-contained and working.
- Line 4 contains a preprocessor directive: The `NUM_AUTH` identifier is replaced with the given replacement value (`32` in this case) before compilation. In this case, this leads to a given maximum amount of cards that are allowed to be inside a building. This already means that, in contrast to the model which does not specify any size on the given sets, this implementation has been refined to only allow up to a certain number of values to be added to the according sets.
- Lines 6–11 introduce a common base class for all other classes that gives each of them a name, making it easier to, e. g., have readable feedback from the application later on.
- Lines 13–46 introduce the classes that were specified in Fig. 2. Fields are named accordingly. Due to using C++, methods are declared in the class declaration parts but are implemented later. Line 24 is a forward declaration to enable the `Turnstile` class to reference the `Building` class before the latter is declared below.
- Lines 48–55 implement functions to check whether or not the given sets include certain elements. These functions have not been specified before but they are implicitly part of the OCL constraints that allow checking sets for the inclusion of elements.
- Lines 57–93 finally implement the functions that have been specified to be part of the `Turnstile` class. Note that each function follows the pattern of starting with an if-clause which tests the preconditions given in the formal model. If these conditions are not met, the `else{}` statement at the bottom leaves the program's state untouched while exiting the function. This ensures that a function call with unsatisfied conditions does not

```
#include <iostream>
#include <algorithm>
#include <string>
#define NUM_AUTH 32

class Structure {
public:
        string m_name;
        string name() {return m_name;}
        Structure(string p_name) {this->m_name = p_name;}
};

class MagneticCard: Structure {
public:
  int id;
  static int currentMaxID;
  MagneticCard(string p_name): Structure(p_name) {this->id = currentMaxID++;}
};

int MagneticCard::currentMaxID = 0;

class Building;

class Turnstile: Structure {
public:
  bool greenLightOn;
  bool redLightOn;
  int currentlyAuthorized;
  int timeOpen;
  bool entry;
  Building* building;
  Turnstile (string p_name): Structure(p_name) { }
  void checkCard(MagneticCard* card);
  void advanceTime();
  void goThrough();
};

class Building: Structure {
public:
  int authorized[NUM_AUTH];
  bool inside[NUM_AUTH];
  Turnstile** gates;
  Building (string p_name): Structure(p_name) { }
  bool authorizedIncludes(int value);
  bool insideIncludes(int value);
};

bool Building::authorizedIncludes(int value) {
  int *begin = authorized;
  int *end = authorized + NUM_AUTH;
  if (end == std::find(begin, end, value)) {return false;}
  return true;
}

bool Building::insideIncludes(int value) {return inside[value];}

void Turnstile::checkCard(MagneticCard* card) {
  if (!this->greenLightOn && !this->redLightOn) {
    if (this->building->authorizedIncludes(card->id) &&
      this->entry != this->building->insideIncludes(card->id)) {
      this->greenLightOn = true;
      this->currentlyAuthorized = card->id;
    } else if (!this->building->authorizedIncludes(card->id)) {
      this->redLightOn = true;
    } else { }
  } else { }
}

void Turnstile::advanceTime() {
  if (this->greenLightOn || this->redLightOn) {
    this->timeOpen++;
    if (this->timeOpen >= 2 && redLightOn) {
      this->timeOpen = 0;
      this->redLightOn = false;
    }
    if (this->timeOpen >= 30) {
      this->timeOpen = 0;
      this->greenLightOn = false;
    }
  } else { }
}

void Turnstile::goThrough() {
  if (greenLightOn) {
    if (this->building->insideIncludes(this->currentlyAuthorized) && !entry) {
      this->building->inside[currentlyAuthorized] = false;
    } else if (!this->building->insideIncludes(currentlyAuthorized) && entry) {
      this->building->inside[currentlyAuthorized] = true;
    }
    this->greenLightOn = false;
    this->timeOpen = 0;
  } else { }
}
```

Fig. 5. C++ implementation of model.

result in an unspecified program state. The first block of the `if` clause contains further `if` clauses implementing the various `implies` postconditions from the model.

Having this implementation, the question remains whether or not it actually implements the formal model, no matter how close it *seems* to be to the model.

B. Comparison to the Formal Model

In order to compare the implementation to its model, the program's features need to be available. In particular for languages such as C++, extracting the respectively required information is not straightforward. In fact, C++ has a strong focus on performance. Thus, compilers usually strip as much information as possible from the code when it is translated

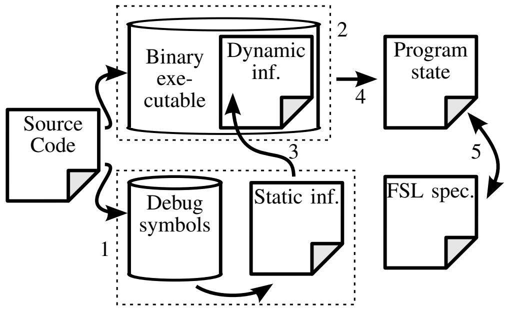


Fig. 6. C++ information extraction

into a (natively) executable file. Additionally, C++ is not easy to parse. Over the last decades, several dialects have emerged which are supersets of the standardized core language – each offering slightly different tools and frameworks. Hence, before an actual comparison can be conducted, we need to extract essential features of the implementation first.

1) Exctracting the Features from the Implementation: Due to the reasons mentioned above, C++ offers no support for reflection and introspection. Hence, the respectively required information needs to be extracted directly during the compilation process. The approach presented in [30] offers a methodology for this purpose, which exploits debugging symbols generated by an off-the-shelf compiler in combination with the existing C++ API[2]. Figure 6 illustrates the respective steps. More precisely:

1) The compiler generates debug-symbols during the compilation. These contain the *static* information which can be gained by analyzing the source code (e. g., which classes are part of the system and which fields and methods are parts of these).
2) The source code is compiled in order to generate an executable. This file can be executed on a system, building the simulated design and simulating it afterwards.
3) While the system is running, instances of certain program states (called *snapshots* in the following) are extracted.
4) The information obtained in the step before can be stored on the disk. At this point, this information is basically the state of the program at the time of the extraction.
5) Both the static structure of the program and (if available) the extracted program state can then be compared to the formal model. Differences between the program state and the formal model may indicate discrepancies. This pinpoints designers to certain parts which may require further inspection or adaptation.

For the implementation from Fig. 5, the static information obtained in Step 1 is shown in Fig. 7. Compared to the original formal model from Fig. 2, this information of course does not contain any OCL constraints and is restricted to the structural information present in the implementation. Moreover, it contains several classes that are not defined in the original formal model (e. g., the `basic_string` class, which is used to name the objects and part of C++'s standard namespace). They are a result of extracting anything the compiler keeps in a given program. Nevertheless, it already provides a good basis for a comparison to the formal model.

Next, behavioral information of the implementation are considered. This can be obtained using the executable generated by the compiler in Step 2. If a set of instances is available to

[2]Note that relying on the C++ API and additional debugging symbols allows for utilizing the approach within different C++ frameworks as well as dialects.

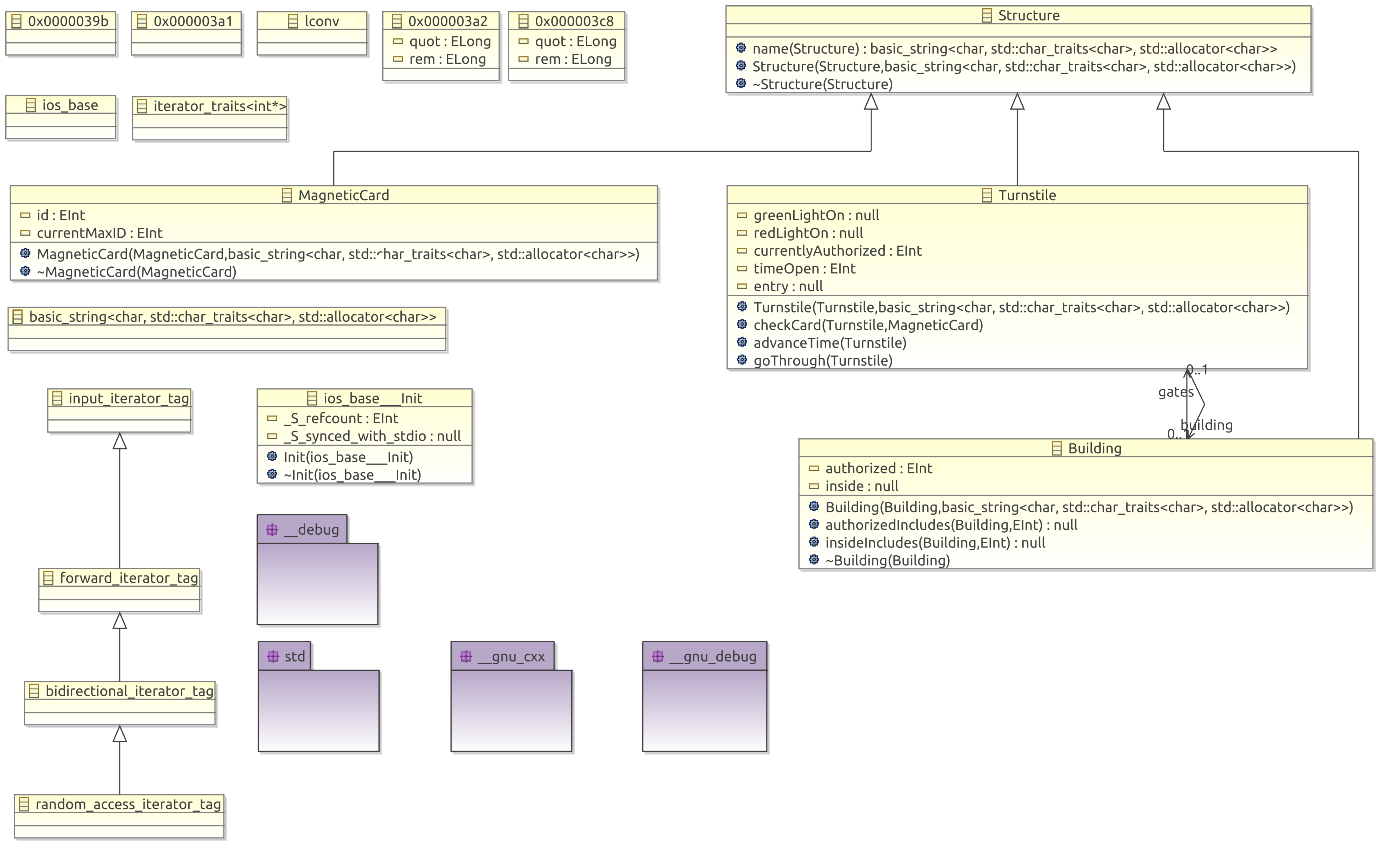

0x0000039b
0x000003a1
lconv
0x000003a2
quot : ELong
rem : ELong
0x000003c8
quot : ELong
rem : ELong
ios_base
iterator_traits<int*>
Structure
name(Structure) : basic_string<char, std::char_traits<char>, std::allocator<char>>
Structure(Structure,basic_string<char, std::char_traits<char>, std::allocator<char>>)
~Structure(Structure)
MagneticCard
id : EInt
currentMaxID : EInt
MagneticCard(MagneticCard,basic_string<char, std::char_traits<char>, std::allocator<char>>)
~MagneticCard(MagneticCard)
Turnstile
greenLightOn : null
redLightOn : null
currentlyAuthorized : EInt
timeOpen : EInt
entry : null
Turnstile(Turnstile,basic_string<char, std::char_traits<char>, std::allocator<char>>)
checkCard(Turnstile,MagneticCard)
advanceTime(Turnstile)
goThrough(Turnstile)
basic_string<char, std::char_traits<char>, std::allocator<char>>
input_iterator_tag
ios_base___Init
_S_refcount : EInt
_S_synced_with_stdio : null
Init(ios_base___Init)
~Init(ios_base___Init)
0..1
gates
building
0..*
Building
authorized : EInt
inside : null
Building(Building,basic_string<char, std::char_traits<char>, std::allocator<char>>)
authorizedIncludes(Building,EInt) : null
insideIncludes(Building,EInt) : null
~Building(Building)
__debug
forward_iterator_tag
bidirectional_iterator_tag
std
__gnu_cxx
__gnu_debug
random_access_iterator_tag

Fig. 7. Retrieved model from the implementation in Fig. 5.

the extraction algorithm, the internal state of an implementation can be extracted by reading the classes' structure information from the debug symbols while the program is running (Step 3). Using this extraction scheme repeatedly extracts a set of program states, i. e., snapshots (Step 4). These states can be seen as the cornerstones of a behavior: if a certain protocol is invoked and the snapshots after each communication step represent the state the program is expected to be in, the communication can be interpreted to be correct. This way, a series of program snapshots serves as a simple way to check the correctness of a system. If no framework is available, using Aspect Oriented Programming techniques [31] can be used to solve this problem by previously refactoring the code to give access to the given structures [32]. Eventually, this leads to structural and behavioral information to be compared to the originally given model.

2) Model Comparison:
With a given formal model and an extracted set of information from the implementation, a comparison of both descriptions can be conducted. Although this would not provide a formal proof, it may lead to a validation of whether the resulting implementation indeed realizes the given model. With the extraction methods still being an active field of research, how to actually perform the comparison remains in its infancy as well.

One solution introduced in [25] is a direct comparison of the two models. While this method suggests that the added complexity (such as the explicitly modeled `std` namespace in the extracted model) can safely be ignored, it otherwise simply compares the two models and locates any differences that go beyond added complexity as potential errors. In this case, the added `Structure` class that forms a base class for all subclasses already represents a structural difference that would be reported, suggesting that either the implementation or the formal model should be altered in order to make them comply with each other's structure.

The comparison from [25] is able to match all elements from the formal model to the elements from the implementation but not vice versa. The algorithm issues warnings for all subpackage records (i.e. the `std`, the `___gnu_cxx`, and `___gnu_debug` namespaces) and the respective sub nodes. It also reports that the operations `Building`, `~Building`, `MagneticCard`, `~MagneticCard` and `Turnstile` (i.e. the constructors and destructors which are required to setup a working C++ implementation) are missing, just as no corresponding `Structure` element can be located.

Besides these warnings (which can easily be ruled out), the approach confirms that the structural features of the formal model are a subset of the structural features of the implementation. That is, the implementation is in line with the notion of the formal model. Comparing various testcases from the formal model to the generated snapshots, similar conclusions can be drawn with respect to the behavior.

V. Discussion and Conclusion

In this work, we provided a case study showcasing the state-of-the-art in verification-driven design of hardware and software. We started with a specification of an access control system in natural language and derived a formal model from it. Before commencing any implementation steps, we showed that the initial model contains flaws and, thus, prevented an implementation phase based on an erroneous model. With the help of various approaches for model verification, we revised the model and eventually implemented it in C++. Finally, the compatibility of the resulting implementation with respect to the formal model has been shown.

By this, we explicitly showed how the combination of all the different verification methods proposed in the past may aid designers not only in detecting flaws as soon as possible, but also in improving the quality of the resulting intermediate steps, i. e., the model and the initial implementation. Besides that, the case study unveiled open gaps to be addressed in future work including, e. g.,

- When an inconsistency in a model is detected, debugging approaches determining reasons for this are available. Thus far, the analysis of inconsistencies has focussed on static behavior only. How to determine reasons for flaws in dynamic behavior, e. g., deadlocks, remains an open question.
- The mentioned approaches for checking the behavior of a model can be applied for a lot of different checks, however, it remains an open question how *coverage*, i. e., the guarantee that the model has *completely* been checked, can be ensured.
- Solutions for the (automatic) determination of frame conditions are still in their infancy and depend on the designers experience. Hence, more elaborated methods which require only a minimum of interaction with the designer are desired. In the best case, the designer directly gets examples and counterexamples for a specific model in order to decide which one is to be added to an invariability clause.
- Checking whether the implementation indeed realizes the model is mainly done by validation, i. e., structural features of an implementation are compared against the originally given model. While this gives a strong indication whether or not an implementation sticks to the features that were previously designed, verification methods which actually prove the correct implementation of operations by, e. g., comparing their internal logic to the original pre- and postconditions are yet to be developed.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001, the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1, and a research project under grant no. WI 3401/5-1, as well as the Siemens AG.

References

[1] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Addison-Wesley Professional, 2004.

[2] Object Management Group, *OMG Unified Modeling Language TM (OMG UML), Infrastructure*, Object Management Group.

[3] G. Martin and W. Müller, *UML for SOC Design*. Springer, 2005.

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

[5] Object Management Group, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Object Management Group, 2011.

[6] H. Foster, "Why the design productivity gap never happened," in *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD'13, San Jose, CA, USA, November 18-21, 2013*, J. Henkel, Ed. IEEE/ACM, 2013, pp. 581–584.

[7] M. Gogolla, F. Büttner, and M. Richters, "USE: A UML-based specification environment for validating UML and OCL," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 27–34, 2007.

[8] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, independence and consequences in UML and OCL models," in *TAP 2009*, 2009, pp. 90–104.

[9] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler, "Formalizing UML Models and OCL Constraints in PVS," *Electronic Notes in Theoretical Computer Science*, vol. 115, pp. 39–47, 2005.

[10] A. D. Brucker and B. Wolff, "A Proposal for a Formal OCL Semantics in Isabelle/HOL," in *TPHOLs*, ser. Lecture Notes in Computer Science, V. Carreño, C. A. Muñoz, and S. Tahar, Eds., vol. 2410. Springer, 2002, pp. 99–114.

[11] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of Object-Oriented Software: The KeY Approach*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[12] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL class diagrams using constraint programming," in *Int'l. Conference on Software Testing Verification and Validation Workshop*. IEEE, 2008, pp. 73–80.

[13] T. Mancini, "Finite satisfiability of UML class diagrams by constraint programming," in *Description Logics*, 2004.

[14] H. Malgouyres and G. Motet, "A UML model consistency verification approach based on meta-modeling formalization," in *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2006, pp. 1804–1809.

[15] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, no. 1, pp. 70–118, 2005.

[16] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers, "Using description logic to maintain consistency between UML models," in *UML*, 2003, pp. 326–340.

[17] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[18] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," in *Int'l Conf. on Model Driven Engineering Languages and Systems*. Springer, Oct. 2007, pp. 436–450.

[19] E. Torlak and D. Jackson, "Kodkod: A Relational Model Finder," in *Tools and Algorithms for Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 4424. Springer, Apr. 2007, pp. 632–647.

[20] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler, "Verifying UML/OCL models using Boolean satisfiability," in *Design, Automation and Test in Europe*, 2010, pp. 1341–1344.

[21] R. Wille, M. Soeken, and R. Drechsler, "Debugging of inconsistent UML/OCL models," in *Design, Automation and Test in Europe*, 2012, pp. 1078–1083.

[22] E. Torlak, F. S.-H. Chang, and D. Jackson, "Finding Minimal Unsatisfiable Cores of Declarative Specifications," in *Int'l Symp. on Formal Methods*, ser. Lecture Notes in Computer Science, J. Cuéllar, T. S. E. Maibaum, and K. Sere, Eds., vol. 5014. Springer, May 2008, pp. 326–341.

[23] M. Soeken, R. Wille, and R. Drechsler, "Verifying Dynamic Aspects of UML Models," in *Design, Automation and Test in Europe*, 2011, pp. 1077–1082.

[24] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, and R. B. France, "From application models to filmstrip models: An approach to automatic validation of model dynamics," in *Modellierung 2014*, ser. LNI, H. Fill, D. Karagiannis, and U. Reimer, Eds., vol. 225. GI, 2014, pp. 273–288.

[25] J. Stoppe, R. Wille, and R. Drechsler, "Validating SystemC Implementations Against Their Formal Specifications," in *Proceedings of the 27th Symposium on Integrated Circuits and Systems Design*. ACM, 2014, p. 13.

[26] J.-R. Abrial. (1999) System study: Method and example. [Online]. Available: http://atelierb.eu/ressources/PORTES/Texte/porte.anglais.ps.gz

[27] N. Przigoda, R. Wille, and R. Drechsler, "Contradiction Analysis For Inconsistent UML/OCL Models," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS) 2015*. IEEE, 2015.

[28] P. Kosiuczenko, "Specification of invariability in OCL - specifying invariable system parts and views," *Software and System Modeling*, vol. 12, no. 2, pp. 415–434, 2013.

[29] P. Niemann, F. Hilken, M. Gogolla, and R. Wille, "Assisted Generation of Frame Conditions for Formal Models," in *Design, Automation and Test in Europe*, 2015.

[30] J. Stoppe, R. Wille, and R. Drechsler, "Data extraction from SystemC designs using debug symbols and the SystemC API," in *VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on*. IEEE, 2013, pp. 26–31.

[31] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," in *International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*. Australian Computer Society, Inc., 2002, pp. 53–60.

[32] J. Stoppe, R. Wille, and R. Drechsler, "Automated Feature Localization for Dynamically Generated SystemC Designs," in *Design, Automation and Test in Europe*, 2015.