Automatic Refinement Checking for Formal System Models

Julia Seiter[1] Robert Wille[1] Ulrich Kühne[1] Rolf Drechsler[1,2]

[1]Institute of Computer Science, University of Bremen, D-28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, D-28359 Bremen, Germany
{jseiter,rwille,ulrichk,drechsle}@informatik.uni-bremen.de

***Abstract*—For the design of complex systems, formal modeling languages such as UML or SysML find significant attention. The typical model-driven design flow assumes thereby an initial (abstract) model which is iteratively refined to a more precise description. During this process, new errors and inconsistencies might be introduced. In this paper, we propose an automatic method for verifying the consistency of refinements in UML or SysML. For this purpose, a theoretical foundation is considered from which the corresponding proof obligations are determined. Afterwards, they are encoded as an instance of *Satisfiability Modulo Theories (SMT)* and solved using proper solving engines. The practical use of the proposed method is demonstrated and compared to a previously proposed approach.**

I. Introduction

Due to the increasing complexity of today's systems and, caused by this, the steady strive of designers and researchers towards higher levels of abstractions, modeling languages such as the *Unified Modeling Language* (UML) [1] or the *Systems Modeling Language* (SysML) [2] together with textual constraints e.g. provided by the *Object Constraint Language* (OCL) [3] received significant attention in computer-aided design. They allow for a *formal* specification of a system prior to the implementation. Such an initial blueprint can be iteratively refined to a final model to be implemented. The actual implementation is then carried out manually, by using automatic code generation, or a mix of both.

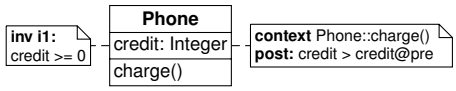
An advantage of using formal descriptions like UML or SysML is that the initial system models can already be subject to (automatic) correctness and plausibility checks. By this, inconsistencies and/or errors in the specification can be detected even before a single line of code has been written. For this purpose, several approaches have been introduced [4]–[8]. They tackle verification questions such as *"Does the conjunction of all constraints still allow the instantiation of a legal system state?"* or *"Is it possible to reach certain bad states, good states, or deadlocks?"*. These verification tasks are typically categorized in terms such as consistency, reachability, or independence [9].

However, these verification techniques are usually carried out on a single model and, hence, are not sufficient for the typical model-driven design flow in which an abstract model is generated first and iteratively refined to a more precise description. Indeed, they enable the detection of errors and inconsistencies in one iteration, but they need to be re-applied in the succeeding iteration even for minor changes. Instead, it is desirable to check whether a refined model is still consistent to the original abstract model. In this way, verification results from abstract models will also be valid for later refined models.

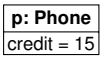
For the creation of software systems, such a refinement process has already been established. Here, frameworks such as the *B-method* [10], *Event-B* [11] and *Z* [12] exist. These methods rely on a rigorous modeling using first-order logic. Extensions e.g. of *Event-B* to the UML-like *UML-B* or translations of UML to *B* models, are available in [13] and [14], respectively. But since the proof obligations for a correct refinement in these frameworks are undecidable in general, usually manual or interactive proofs must be conducted – a time-consuming process which additionally requires a thorough mathematical background.

Hence, *automatic* proof techniques are desired. For this purpose, existing solutions proposed in the context of hardware verification and the design and modeling of reactive systems may be applied. Here, the relation of an implementation and its specification (comparable to a refined and an abstract system) is traditionally described by *simulation relations* on finite state systems (see e.g. [15]–[18]). There exist algorithms for computing such relations [19], [20]. However, since these algorithms operate on explicit state graphs, they do require the consideration of all possible system states and operation calls – infeasible for larger designs. A similar difficulty occurs when attempting to automatize the verification process proposed by the B-method. In [21], an extension to the tool ProB has been proposed which automatically solves all proof obligations created in the refinement process. However, according to their evaluation, the run-time for the verification grows exponentially.

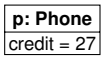
As a consequence, an alternative solution is proposed in this work which exploits the recent accomplishments in the domain of model-based verification (i.e. approaches like [4]–[8]) using a symbolic state representation. Based on a theoretical foundation of refinement, we can prove the preservation of safety properties from an abstract model to a more detailed model. In contrast to the existing approaches like in [22], this also includes *non-atomic* refinements, where an abstract operation is replaced by a sequence of refined operations. By this, the consistency of a refined model against an original (abstract) model can be checked automatically.



(a) Given model



(b) State σ_0



(c) State σ_1

Fig. 1. Example of a model and its instantiation

The remainder of the paper is structured as follows. The paper starts with a brief review on models and their notation in Section II. Section III describes the addressed problem which, afterwards, is formalized in Section IV. The proposed solution is introduced in Section V and its usefulness is demonstrated in Section VI where it is applied to several examples and compared to the results from [21]. The paper is concluded in Section VII.

II. Models and Their Notation

Modeling languages provide a description of a system to be realized, i.e. proper description means to *formally* define the structure and the behavior of a system. At the same time, implementation details which are not of interest in the early design/specification state remain hidden. In the following, we briefly review the respective description by means of UML and OCL. The approaches proposed in this work can be applied to similar modeling languages (e.g. such as SysML) as well.

Definition 1: A *model* is a tuple $m = (C, R)$ composed of a set of classes C and a set of associations R. A *class* $c = (A, O, I) \in C$ of a model m is a tuple composed of attributes A, operations O, and invariants I. An n-ary *association* $r \in R$ of a model m is a tuple $r = (r_{\text{ends}}, r_{\text{mult}})$ with *association ends* $r_{\text{ends}} \in C^n$ for a given set of classes C and *multiplicities* $r_{\text{mult}} \in (\mathbb{N}_0 \times \mathbb{N})^n$ that is defined as a range with a *lower bound* and an *upper bound*.

Example 1: Fig. 1(a) shows a model composed of the class Phone which itself is composed of the attributes $A = \{\mathsf{credit}\}$, the operations $O = \{\mathsf{charge}\}$, and the invariant $I = \{\mathsf{i1}\}$.

Invariants in the model describe additional constraints which have to be satisfied by each instantiation of the model. For this purpose, textual descriptions provided in OCL can be applied. OCL also allows the specification of the behavior of operations.

Definition 2: OCL expressions Φ are textual constraints over a set of *variables* $V \supseteq A \times R$ composed of the attributes A of the respective classes, but also further (auxiliary) variables. An OCL condition $\varphi \in \Phi$ is defined as a function $\varphi : V \to \mathbb{B}$. They can be applied to specify the invariants of a class as well as the pre- and post-condition of an operation, i.e. $I \subseteq \Phi$. An *operation* $o \in O$ is defined as a tuple $o = (\lhd, \rhd)$ with pre-condition $\lhd \in \Phi$ and post-condition $\rhd \in \Phi$, respectively. The valid initial assignments of a class are described by a predicate $\text{init} \in \Phi$.

Example 2: In the model from Fig. 1(a), the invariant i1 states that credit always has to be greater or equal to 0. The post-condition of the operation charge ensures that after invoking the operation, credit is increased.

Any instance of a model is called a *system state* and is visualized by an object diagram.

Definition 3: Object diagrams represent precise system states in a model. A *system state* is denoted by σ and is composed of *objects*, i.e. instantiations of classes. The attributes of the objects are derived from the classes and assigned precise values. Associations are instantiated as precise *links* between objects.

In order to evaluate a model, it is crucial to particularly consider whether system states are valid or sequences of system states represent valid behavior. This requires the evaluation of the given OCL expressions.

Definition 4: For a system state σ and an OCL expression φ, the evaluation of φ in σ is denoted by $\varphi(\sigma)$. A system state σ for a model $m = (C, R)$ is called *valid* iff it satisfies all invariants of m, i.e. iff $\bigwedge_{c \in C} I_c(\sigma)$. An operation call is valid iff it transforms a system state σ_t satisfying the pre-condition to a succeeding system state σ_{t+1} satisfying the post-condition[1], i.e. iff $\lhd(\sigma_t)$ and $\rhd(\sigma_t, \sigma_{t+1})$. A sequence of system states is called valid, if all operation calls are valid.

Example 3: Fig. 1(b) and Fig. 1(c) show two valid system states (in terms of object diagrams) for the model from Fig. 1(a). This is a valid sequence of system states which can be created by calling the operation charge.

III. Refinement of Models

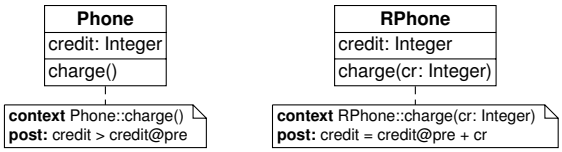
Using the description means reviewed in the previous section allows for a *formal* specification of a system to be implemented. By this, precise blueprints are available already in the early stages of the design. A rough initial model is thereby created first which covers the most important core functionality. Afterwards, a *refinement* process is conducted in which a more precise model of the respective components and operations is created. This refinement process may include

- the addition of new components and relations (i.e. classes and their associations),
- the extension of classes by new attributes,
- the extension of the behavioral description (i.e. the addition of new operations as well as pre- and post-conditions and the strengthening of existing pre- and post-conditions), and
- the extension of the constraints (i.e. the addition of new and the strengthening of existing invariants).

Example 4: Consider the model from Fig. 2(a) representing a simple phone application. It consists of a phone with a credit which can be charged by a corresponding operation. A possible refinement of this model is depicted in Fig. 2(b). Here, the post-condition of the operation *charge* has been rendered more precise, i.e. a parameter defining the amount of credits to be charged has been added.

Remark. Up to this point, we do not consider the refinement of associations and operation parameters. This includes the type of association and the multiplicities of the associations

[1]The post-condition is a binary predicate, since it can also depend on the source state, which is expressed using @pre in OCL.



(a) Abstract model (b) Refined model

Fig. 2. Refinement

ends. However, this is not due to a technical limitation of our approach which can easily be extended to further description means. Here, we decided to focus on the refinement of attributes and operations, considering in particular non-atomic refinement, as these are the most important modeling elements in formal system specifications. Other kinds of refinement, e.g. for operation parameters, can be conducted analogously.

In the following, we denote the *abstract model* by m^a and the *refined model* by m^r. A refinement is described by a refinement relation defined as follows:

Definition 5: A refinement relation is a pair $\text{Ref} = (\text{Ref}_\Sigma, \text{Ref}_\Omega)$ with

- Ref_Σ describing the refinement of the states, i.e. Ref_Σ^{-1} is a function mapping a refined state σ^r to its corresponding abstract state σ^a, and
- Ref_Ω describing the refinement of operations, i.e. Ref_Ω is a function mapping an abstract operation o^a to a sequence $o_1^r \cdot o_2^r \cdot \dots \cdot o_k^r \in (O^r)^+$ of refined operations.

Example 5: The refinement from the model in Fig. 2(a) to the model in Fig. 2(b) is described by the relation Ref=$(\text{Ref}_\Sigma, \text{Ref}_\Omega)$. That is, each state σ^r in the refined model (composed of objects from class RPhone) has one corresponding state $\text{Ref}_\Sigma^{-1}(\sigma^r) = \sigma^a$ in the abstract model (composed of objects from class Phone) such that RPhone.credit = Phone.credit. Furthermore, the operation Phone::charge() is refined so that Ref_Ω(Phone::charge()) = RPhone::charge(cr), i.e. a corresponding operation with an additional parameter.

Adding details step by step – like in the above example – is common practice in model-driven design using UML or SysML. Nevertheless, during this manual process, new errors might be introduced, leading to a refined model that is not consistent with the abstract model anymore. In fact, the refinement sketched above contains a serious flaw.

Example 6: The refined model in Fig. 2(b) allows for a behavior that is not specified by the abstract model. It is possible to assign a value equal to or less than 0 to the parameter cr, so that after calling the operation charge, the value of credit does not change at all or even decreases. This contradicts the behavior described in the abstract model which only allows for a strict increase of that attribute. As a possible repair of this inconsistency, the precondition pre: cr > 0 could be added to the operation RPhone::charge(cr).

In order to identify and fix inconsistencies of the refinement, designers have to intensely check the refined model against the abstract original – often a complicated and cumbersome task which results in a manual and time-consuming procedure. In the worst case, all components, constraints, and possible executions have to be inspected. While this might be feasible for the simple model discussed above, it becomes highly inefficient for larger models. Hence, in the remainder of this paper we consider the question "How to automatically check whether a refined model m^r is consistent with respect to the originally given abstract model m^a?"

IV. Theoretical Foundation

This section formalizes the problem sketched above. For this purpose, we exploit the theoretical foundation of Kripke structures and their concepts of simulation relations. We show how these concepts can be applied for the refinement of system models provided e.g. in UML or SysML. This provides the basis for the proposed solution which is described afterwards in Section V.

Since we are considering models mostly in the context of software and hardware systems, we assume bounded data types and a bounded number of instances in the following[2]. Based on these assumptions, the behavior of a model can be described as a finite state machine, e.g. a *Kripke structure*.

Definition 6: A Kripke structure is a tuple $\mathcal{K} = (S, S_0, AP, \mathcal{L}, \rightarrow)$ with a finite set of states S, initial states $S_0 \subset S$, a set of atomic propositions AP, a labeling function $\mathcal{L} : S \rightarrow 2^{|AP|}$, and a (left-total) transition relation $\rightarrow \subseteq S \times S$.

Using this formalism, we can define the behavior of a UML or SysML model and its operations as follows:

Definition 7: A model $m = (C, R)$ induces a Kripke structure $\mathcal{K}_m = (S, S_0, AP, \mathcal{L}, \rightarrow)$ with

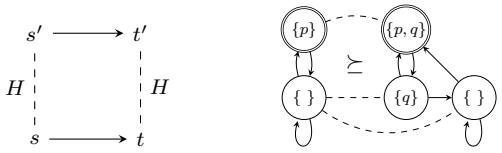
- S being the set of all valid system states of $m = (C, R)$,
- S_0 being the set of initial states defined by the predicate init (cf. Def. 2), i.e. $S_0 = \{\sigma \in S \mid \text{init}(\sigma)\}$,
- $\rightarrow$ being the transition relation including the identity (i.e. $\sigma \rightarrow \sigma$) as well as all transitions caused by executing operations $o = (\lhd, \rhd) \in O$ of the model (i.e. $\sigma_1 \rightarrow \sigma_2$ with $\lhd(\sigma_1)$ and $\rhd(\sigma_1, \sigma_2)$), and
- AP and $\mathcal{L}$ are defined s.t. $\mathcal{L}$ can be used to retrieve the values of the attributes of σ in the usual bit-vector encoding.

We will write $\sigma_1 \overset{o}{\rightarrow} \sigma_2$ to make clear that an operation o transforms a state σ_1 to a state σ_2.

With this formalization, we can make use of known results for finite and reactive systems. To describe refinements in this domain, *simulation relations* are usually applied for this purpose (see e.g. [15]–[20]). In this work, we adapt this concept for the considered formal models. This leads to the following definition of a simulation relation.

Definition 8: Let $\mathcal{A} = (S_\mathcal{A}, S_{\mathcal{A}0}, AP_\mathcal{A}, \mathcal{L}_\mathcal{A}, \rightarrow_\mathcal{A})$ be a Kripke structure of an abstract model and $\mathcal{R} = (S_\mathcal{R}, S_{\mathcal{R}0}, AP_\mathcal{R}, \mathcal{L}_\mathcal{R}, \rightarrow_\mathcal{R})$ be a Kripke structure of a refined model with $AP_\mathcal{R} \supseteq AP_\mathcal{A}$. Then, a relation $H \subseteq S_\mathcal{R} \times S_\mathcal{A}$ is a simulation relation iff

[2]This restriction is common in many approaches (e.g. [4]–[8]) and also justified by the fact that, eventually, the implemented system will be realized by bounded physical devices anyway.



(a) Correspondence of states (b) Example for simulation

Fig. 3. Simulation relation

1) all initial states in the refined model have a corresponding initial state in the abstract model, i.e. $\forall s_0 \in S_{\mathcal{R}0} \exists s'_0 \in S_{\mathcal{A}0}$ with $H(s_0, s'_0)$,
2) all states in the refined model are constrained by at least the same propositions as their corresponding abstract state, i.e. $\forall s, s' : H(s, s') \Rightarrow \mathcal{L}_{\mathcal{R}}(s) \cap AP_{\mathcal{A}} = \mathcal{L}_{\mathcal{A}}(s')$, and
3) all possible transitions in the refined model have a corresponding transition in the abstract model leading to a corresponding succeeding state, i.e. $\forall s, s' : H(s, s') \Rightarrow s \rightarrow_{\mathcal{R}} t \Rightarrow \exists t' \in S_{\mathcal{A}}$ s.t. $s' \rightarrow_{\mathcal{A}} t'$ and $H(t, t')$.

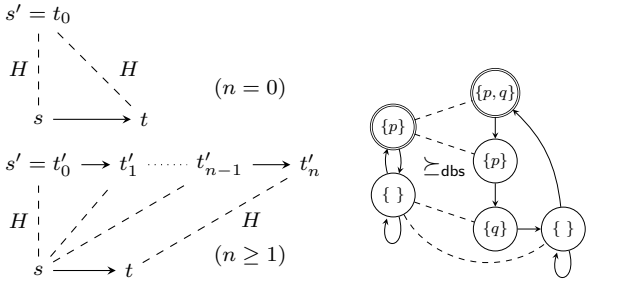
We say that $\mathcal{R}$ is simulated by $\mathcal{A}$ (written as $\mathcal{R} \preceq \mathcal{A}$), if there exists a simulation relation.

Example 7: As an illustration of the above definition, Fig. 3(a) shows the general scheme of a transition between states from a refined model (denoted by s and t) and a corresponding transition in an abstract model (from s' to t'). The simulation relation H is indicated by dashed lines. Figure 3(b) on the right shows an example for two Kripke structures. The abstract model is the one on the left hand side and simulates the refined model on the right hand side. Initial states are marked by a double outline. While all corresponding states agree on the atomic proposition p, the refined model has an additional proposition q. It can easily be checked that for each refined transition, there is a corresponding abstract one.

The simulation relation ensures that a refined model is consistent to an abstract system, i.e. whatever the refined system does must be allowed by the abstract system. Besides that, there might be more behavior allowed in the abstract system than implemented. If we have $\mathcal{R} \preceq \mathcal{A}$, then the traces of $\mathcal{R}$ are contained in those of $\mathcal{A}$. This also means that globally valid properties of $\mathcal{A}$ carry over to $\mathcal{R}$, as for example the non-reachability of bad states. Hence, by proving that the applied refinement Ref (cf. Def. 5) satisfies the properties of a simulation relation H, the consistency of a refined model can be verified.

However, determining a simulation relation requires a strict step-wise correspondence between the transition in the refined model and in the abstract one. But refinements of UML or SysML models often include the replacement of a single abstract operation by a sequence of refined operations (also known as *non-atomic refinement* [23]). In order to formalize this, we need a more flexible relation. This is provided by the notion of *divergence-blind stuttering simulation* (dbs-simulation).

Definition 9: Given two Kripke structures $\mathcal{R}$ and $\mathcal{A}$ with $AP_{\mathcal{R}} \supseteq AP_{\mathcal{A}}$, a relation $H \subseteq S_{\mathcal{R}} \times S_{\mathcal{A}}$ is a divergence blind stuttering simulation (dbs-simulation) iff



(a) Correspondence of states (b) Example for dbs-simulation

Fig. 4. dbs-simulation relation

1) $\forall s_0 \in S_{\mathcal{R}0} \exists s'_0 \in S_{\mathcal{A}0}$ with $H(s_0, s'_0)$,
2) $\forall s, s' : H(s, s') \Rightarrow \mathcal{L}_{\mathcal{R}}(s) \cap AP_{\mathcal{A}} = \mathcal{L}_{\mathcal{A}}(s')$, and
3) each possible transition in the refined model corresponds to a sequence of 0 or more abstract transitions, i.e. $\forall s, s' : H(s, s')$ and $s \rightarrow_{\mathcal{R}} t$, then there exist $t'_0, t'_1 \ldots t'_n$ $(n \geq 0)$ such that $s' = t'_0$ and $\forall i < n : t'_i \rightarrow_{\mathcal{A}} t'_{i+1} \wedge H(s, t'_i)$ and $H(s', t'_n)$.

We say that $\mathcal{R}$ is dbs-simulated by $\mathcal{A}$, written as $\mathcal{R} \preceq_{\mathsf{dbs}} \mathcal{A}$, if there exists a dbs-simulation.

Compared to the original simulation relation, this definition is less precise with respect to the *duration* of specific operations. But, it still guarantees that the functional behavior of the refined model is consistent with the behavior of the abstract model – even in the absence of a (step-wise) one-to-one correspondence of the transitions. In particular, if the properties of the dbs-simulation are satisfied, a bad state unreachable in $\mathcal{A}$ is also unreachable in $\mathcal{R}$.

Example 8: In Figure 4, the dbs-simulation relation is illustrated. The general scheme of corresponding states and transitions is shown in Figure 4(a). In Figure 4(b), the abstract model on the left hand side dbs-simulates the refined model on the right hand side. Note that the transition from the initial state of the refined model is a *stuttering* transition, since it corresponds to an empty sequence of transitions in the abstract model.

The above definitions provide the formal foundation for consistency checks of refinements. By referring to dbs-simulation, we can preserve safety properties from an abstract model to a refined model. Hence, by proving that the actually applied refinement Ref (c.f. Def. 5) indeed satisfies the properties of a dbs-simulation H (cf. Def. 9), the consistency of the refinement is shown. In the next section, we describe how the refinement for UML or SysML models can efficiently be checked.

V. Proposed Solution

In this section, we present the proposed solution to automatically check the refinement of the model m^a to the model m^r. As outlined above, we particularly require that the applied refinement Ref satisfies the properties of a dbs-simulation H. For this purpose, *all* (valid) system states as well as *all* possible operation calls in those states need to be considered. Naive schemes e.g. relying on enumerating all possible scenarios are clearly infeasible for this purpose. Hence, we propose an

approach that maps the problem to an instance of *Satisfiability Modulo Theories* (SMT) and, afterwards, exploits the efficiency of corresponding solving techniques (such as [24]).

To this end, we represent arbitrary system states and transistions for the abstract model m^a as well as the refined model m^r together with their invariants and the refinement relation Ref in terms of bit-vectors and bit-vector constraints. In the same way, the verification objectives proving that the applied refinement Ref indeed ensures dbs-simulation are encoded and checked automatically. In the following, the resulting verification objectives are briefly sketched. Then, we illustrate how to encode these in SMT.

A. Verification Objectives

As motivated in Section III, we are interested in the relation between abstract operations and their possibly non-atomic refinements. These operation refinements are given as operation sequences according to Def. 5. By this, the refinement check is reduced to the question of whether there is a sequence of operation calls in the refined model that corresponds to a single call in the abstract model (according to the given refinement relation), but violates the requirements of the abstract operation. Unsatisfiability of such an instance shows that no such sequence exists and, hence, the refinement is correct. Otherwise, a counterexample showing the inconsistency is provided.

Based on this intuitive notion of refinement, we derive three verification objectives that prove the correspondence of an abstract operation and its refined operations and are sufficient to prove dbs-simulation. By this, the preservation of safety properties is guaranteed and the refinement is proven consistent. The three objectives read as follows:

1) Check whether all initial states in the refined model indeed correspond to the respective initial states in the abstract model, i.e.

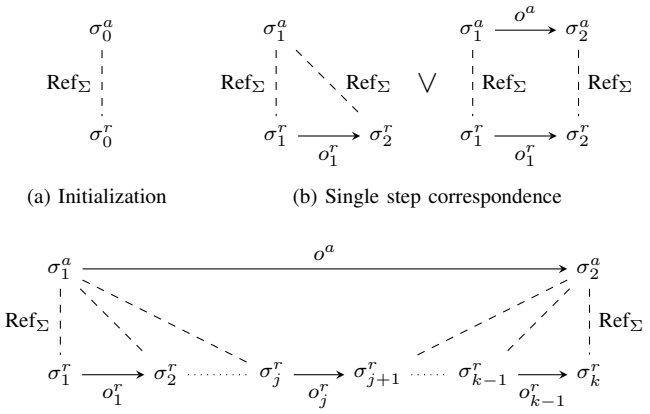
$$\forall \sigma_0^r : \mathrm{init}(\sigma_0^r) \Rightarrow \mathrm{init}(\mathrm{Ref}_\Sigma^{-1}(\sigma_0^r)).$$

This check is illustrated in Fig. 5(a).

2) For each step o_j^r of the refined operation which transforms a refined state σ_1^r, check whether this step does not lead to a succeeding state o_2^r which is inconsistent to its corresponding abstract states. In fact, the succeeding state σ_2^r either has to correspond to the unchanged abstract state or to its abstract state which results after applying the corresponding abstract operation o^a, i.e. for each step o_j^r

$$\begin{aligned}\forall \sigma^a, \sigma_1^r, \sigma_2^r : \mathrm{Ref}_\Sigma(\sigma^a, \sigma_1^r) \wedge \sigma_1^r \xrightarrow{o_j^r} \sigma_2^r \\ \Rightarrow \big(\mathrm{Ref}_\Sigma^{-1}(\sigma_2^r) = \sigma^a \\ \vee (\lhd_{o^a}(\sigma^a) \wedge \rhd_{o^a}(\sigma^a, \mathrm{Ref}_\Sigma^{-1}(\sigma_2^r)))\big)\end{aligned}$$

is checked. This check is illustrated in Fig. 5(b). These two objectives are already sufficient to prove dbs-simulation. Nevertheless, a third objective is additionally checked.



(a) Initialization (b) Single step correspondence

(c) Chaining of refined steps

Fig. 5. Verification objectives

3) Check whether the joint effect of the refined operation sequence adheres exactly to the specification of the abstract operation. That is, for each operation o^a and its refinement $o_1^r \dots o_k^r$

$$\begin{aligned}\forall \sigma_1^a, \sigma_1^r, \sigma_2^r \dots \sigma_{k+1}^r : \\ \mathrm{Ref}_\Sigma(\sigma_1^a, \sigma_1^r) \wedge \sigma_1^r \xrightarrow{o_1^r} \sigma_2^r \dots \sigma_k^r \xrightarrow{o_k^r} \sigma_{k+1}^r \\ \Rightarrow \big(\lhd_{o^a}(\sigma_1^a) \wedge \rhd_{o^a}(\sigma_1^a, \mathrm{Ref}_\Sigma^{-1}(\sigma_{k+1}^r))\big)\end{aligned}$$

is checked. This check is illustrated in Fig. 5(c). This check particularly considers the common UML or SysML refinement which often refines a single abstract operation into a sequence of refined operations.

Together, these three objectives represent the verification tasks to be solved by the respective solving engine. Next, we illustrate how they are encoded as an SMT instance.

B. Basic Encoding

In order to represent arbitrary system states and transitions in an SMT instance, we use an encoding similar to the ones previously presented e.g. in [4]–[7] and particularly in [8]. Here, systems states (basically defined by the values of their attributes) and links are represented by corresponding bit-vector variables. Invariants are represented by corresponding SMT constraints. By this, it is ensured that the solving engine only considers systems states σ composed of objects satisfying all invariants of the underlying class, i.e. $I_c(\sigma)$.

In order to encode transitions caused by operation calls, bit vectors $\vec{\omega}_i \in \mathbb{B}^{\lceil \mathrm{ld}(|O|)\rceil}$ are created for each step i in the refined model. Depending on the assignment to $\vec{\omega}$, the respective pre-conditions and post-conditions have to be enforced. This can be realized by a constraint

$$\vec{\omega}_i = \mathrm{enc}(o) \Rightarrow \lhd_o(\sigma_i^r) \wedge \rhd_o(\sigma_i^r, \sigma_{i+1}^r),$$

where $\mathrm{enc}(o)$ represents a unique binary representation of the operation o, i.e. a number from 0 to $|O_r|$ with $\mathrm{enc}(\mathrm{id}) = 0$. Furthermore, to ensure that only legal values can be assigned to a vector $\vec{\omega}$, we use a constraint $\vec{\omega} \leq |O_r|$

We further introduce auxiliary predicates that reflect the relationship between an abstract operation and its refined steps. For this purpose, the operation refinement Ref_Ω is utilized:

$$\text{step}_i(o^a, o^r_j) \Leftrightarrow \text{Ref}_\Omega(o^a) = o_1 \cdot o_2 \ldots o_k \wedge o_i = o_j$$

$$\text{step}(o^a, o^r_j) \Leftrightarrow \bigvee_{i=1}^{|\text{Ref}_\Omega(o^a)|} step_i(o^a, o^r_j)$$

Here, $step_i(o^a, o^r_j)$ evaluates to true iff the refined operation o^r_j is the i-th step in the refinement of o^a, while $step(o^a, o^r_j)$ reflects that o^r_j occurs in any position in the refinement of o^a.

In order to encode the chaining of the refined operation steps according to the scheme in Fig. 5(c), we define the predicate chain:

$$\text{chain}(o^a) \Leftrightarrow \bigwedge_{i=1}^{l} \big(\text{step}_i(o^a, o^r_i) \wedge \vec{\omega}_i = \text{enc}(o^r_i) \vee i > |\text{Ref}_\Omega(o^a)| \wedge \vec{\omega}_i = \text{enc}(\text{id})\big)$$

In the above formula, in order to cover all abstract operations in one instance, the refined operation sequences are brought to the same maximal length l by filling up the sequence with the identity function for operations where $|\text{Ref}_\Omega(o)| < l$. We thereby make use of the maximum number of steps according to Ref_Ω, i.e. $l = \max\{|\text{Ref}_\Omega(o^a)| \mid o^a \in O^a\}$. Next, the above "ingredients" are put together in order to encode the verification objectives of a refinement.

C. Encoding the Verification Objectives

While the encodings from above ensure a proper representation of the models, system states, and execution of operations in an SMT instance, finally the verification objectives from Section V-A are encoded. In order to prove (1), we encode its negation and check for unsatisfiability, i.e.

$$\exists \sigma^r_0, \sigma^a_0 : \text{Ref}_\Sigma(\sigma^a_0, \sigma^r_0) \wedge \text{init}(\sigma^r_0) \wedge \neg\, \text{init}(\sigma^a_0). \quad (1)$$

To check (2), we try to determine a refined operation call that cannot be matched with one of the schemes in Fig. 5(b). Hence, instead of encoding (2) for each individual refined operation, we let the solving engine choose a refined step that violates the requirements, i.e.

$$\exists \sigma^a_1, \sigma^a_2, \sigma^r_1, \sigma^r_2, o^a, o^r : \text{Ref}_\Sigma(\sigma^a_1, \sigma^r_1) \wedge \vec{\omega}_1 = \text{enc}(o^r) \wedge \text{step}(o^a, o^r) \wedge \text{Ref}_\Sigma(\sigma^a_2, \sigma^r_2) \wedge \neg\,(\sigma^a_1 = \sigma^a_2 \vee \lhd_{o^a}(\sigma^a_1) \wedge \rhd_{o^a}(\sigma^a_1, \sigma^a_2)). \quad (2)$$

That is, we check that, given a pair of corresponding states and an operation call in the refined state, whether it is possible that the reached refined state neither corresponds to the original abstract state nor does it satisfy the specification of the abstract operation. In case this instance is unsatisfiable, objective (2) has been proven.

Finally, for (3) we need to check whether we can determine an instantiated sequence of refined operation calls, such that their joint effect does not adhere to the specification of the respective abstract operation. For this purpose, we use the chain predicate as defined in the previous section to construct the *unrolled* operation sequence, i.e.

$$\exists \sigma^a_1, \sigma^a_2, \sigma^r_1 \ldots \sigma^r_{l+1}, o^a, o^r_1 \ldots o^r_l : \text{Ref}_\Sigma(\sigma^a_1, \sigma^r_1) \wedge \text{chain}(o^a) \wedge \text{Ref}_\Sigma(\sigma^a_1, \sigma^r_{l+1}) \wedge \neg\left(\lhd_{o^a}(\sigma^a_1) \wedge \rhd_{o^a}(\sigma^a_1, \sigma^a_{l+1})\right). \quad (3)$$

That is, we are searching for a chain of $l+1$ refined states and connected by l operation calls such that there are no corresponding abstract states which satisfy the pre- and post-conditions of the respective abstract operation. Unsatisfiability proves that no such chain exists and, hence, objective (3) holds.

TABLE I
SIZE OF EXAMPLES

Model	#Classes	#Attributes	#Operations	#Constraints
AC0	2	3	1	3
AC1	2	4	2	6
AC2	2	5	3	10
MPC0	2	2	4	12
MPC1	2	6	12	40
MPC2	2	8	16	52
MPC3	2	8	16	60

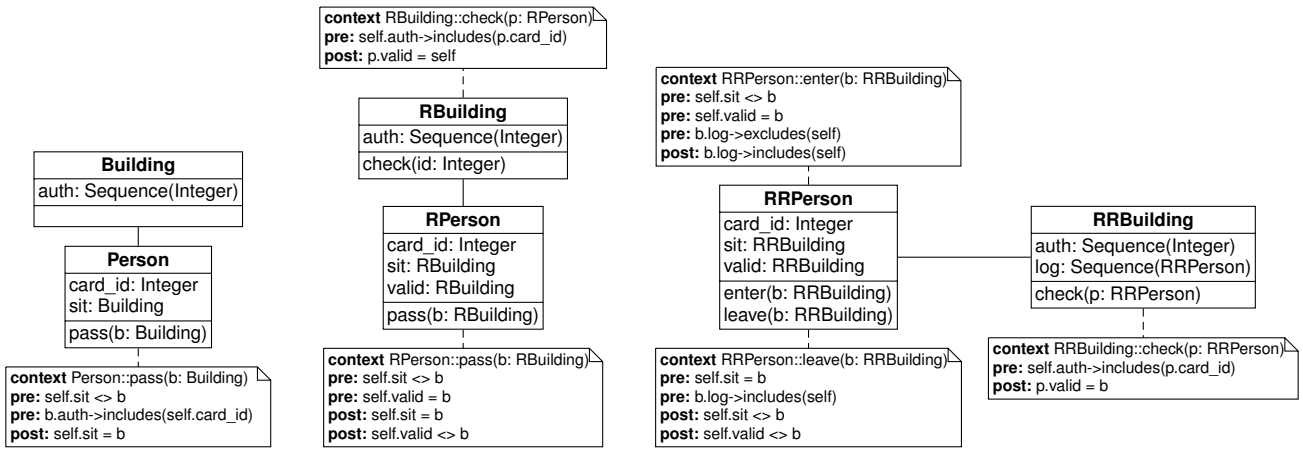
VI. EVALUATION

The approach presented in this work has been implemented in Java, using the SMT solver *Boolector* [24] as underlying solving engine. In order to evaluate the applicability and scalability of our approach, we have applied it to two systems based on examples presented in [10]. For the sake of comparison, these examples have additionally been verified using the previously proposed B method following a manual as well as an automatic scheme [21].

The first example describes an access control system (AC) which is employed to grant access to a building when presented with an authorized ID by a user. Two refinement steps have been modeled, a correct and an erroneous one, which are depicted in Figure 6 together with the abstract model. All types of refinement presented in this work have been applied to this model, i.e. attribute refinement as well as atomic and non-atomic operation refinement.

Table I provides the sizes of the three models (denoted by AC0, AC1, and AC2), i.e. the number of classes, attributes, operations, and OCL constraints are listed. As can be seen, the abstract model (AC0) and the two refined models (AC1, AC2) are relatively small regarding the number of UML elements. Only the number of OCL constraints increases slightly as the added and refined operations are extended.

In order to compare our work to the traditional B approach, we re-modeled this example in B and verified the refinement manually, using the event-B tool *Rodin*. The first refinement step led to a total of 14 proof obligations that had to be discharged. While five of them could be proven fully automatically and some further proofs needed only minor effort, the remaining ones required rather complex interactions like manually entered hypotheses or case splitting. Furthermore, the event-B model required additional invariants. This became particularly crucial for the second refinement which, due to the



(a) Abstract model (b) First refinement (c) Second refinement

Fig. 6. Access control system

non-atomic nature of the refinement conducted here, could not be modeled in a straight forward fashion in event-B.

In contrast, both steps could be automatically verified in negligible time by the approach proposed in this work. The non-atomic refinement did not lead to an increased run-time in this case.

The second example is a mechanical press controller (MPC), which has also been used to evaluate the automatic verification approach in [21] with the tool ProB. It describes a mechanical press with a motor, a clutch and a door which interact in such a way as to guarantee a safe use. As in [21], we have modeled the first three refinement steps in UML and verified them with the proposed SMT-based approach. Here, the refinement contains the introduction of new attributes and constraints as well as atomic operation refinement. All three refinement steps have been proven to be correct.

Again, the size of the abstract model and its refinements is shown in Table I (denoted by MPC0, MPC1, MPC2, and MPC3). In contrast to the first example, the amount of attributes and operations as well as the number of OCL constraints increases. Especially the growing number of operations is important, since, for the SMT-based approach, each of these operations has to be verified according to the criteria presented earlier.

Table II shows the run-times of our experiments compared to those of ProB. The first two columns indicate which models have been verified against each other. The third and fourth column contain the run-times of ProB without and with XSB Prolog taken from [21]. In [21], the ProB tool has already been compared to an automatic refinement verification approach based on CSP (namely [25]) which was clearly outperformed by ProB.

Again, the proposed approach proved the correctness of all three refinement steps in negligible time whereas the run-time of ProB was much larger. Also, with and without XSB Prolog, ProB's run-time increased drastically with every step in the refinement process. A similar development has not been observed for the SMT-based method so far.

TABLE II
EXPERIMENTAL RESULTS

		Run-time		
Abstract	Refined	ProB	ProB+XSB	SMT-based
AC0	AC1	-	-	< 0.01 s
AC1	AC2	-	-	< 0.01 s
MPC0	MPC1	6.28 s	2.85 s	< 0.01 s
MPC1	MPC2	70.57 s	26.66 s	< 0.01 s
MPC2	MPC3	333.85 s	136.12 s	< 0.01 s

These experiments confirm that our approach is robust in such a way that it is applicable to various types of models and refinements. Neither errors in the refinement process nor the type of operation refinement – atomic or non-atomic – have a significant influence on the run-time.

VII. CONCLUSIONS

In this work, we proposed an automatic approach which proves refinements of UML or SysML class diagrams. By this, we are considering the typical model-driven design flow which usually assumes an initial (abstract) model that is iteratively refined to a more precise representation. Based on a theoretical foundation, we introduced an SMT encoding checking whether the respective refinement relation represents a dbs-simulation and, hence, preserves (safety) properties from the abstract model to the refined model. We compared our approach to the tool ProB, which performs automatic refinement verification on B models. An experimental evaluation has shown that the SMT-based technique can verify refinements much faster and scales better than the B-based method.

For future work, we plan to extend our approach in order to support more modeling elements such as refinement of associations or parameters. Also, since the refinement relation still has to be provided manually, we intend to develop a method which automatically extracts a correct refinement relation from two class diagrams.

ACKNOWLEDGMENTS

This work was supported by the Graduate School SyDe (funded by the German Excellence Initiative within the University of Bremen's institutional strategy), 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.

REFERENCES

[1] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Essex, UK: Addison-Wesley Longman, Jan. 1999.

[2] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Feb. 2008.

[3] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Boston, MA, USA: Addison-Wesley Longman, Mar. 1999.

[4] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," in *IEEE Int'l. Conf. on Software Testing Verification and Validation Workshop*, Apr. 2008, pp. 73–80.

[5] M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini, "Finite Model Reasoning on UML Class Diagrams Via Constraint Programming," in *AI*IA*, ser. Lecture Notes in Computer Science, R. Basili and M. T. Pazienza, Eds., vol. 4733. Springer, 2007, pp. 36–47.

[6] 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.

[7] 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*. IEEE Computer Society, Mar. 2010, pp. 1341–1344.

[8] M. Soeken, R. Wille, and R. Drechsler, "Verifying dynamic aspects of UML models," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, 2011, p. 1–6.

[9] M. Gogolla, M. Kuhlmann, and L. Hamann, "Consistency, Independence and Consequences in UML and OCL Models," in *Tests and Proofs*. Springer, Jul. 2009, pp. 90–104.

[10] J.-R. Abrial, *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996.

[11] ——, *Modeling in Event-B: System and Software Engineering*, 1st ed. New York, NY, USA: Cambridge University Press, 2010.

[12] J. Woodcock and J. Davies, *Using Z: specification, refinement, and proof*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[13] C. Snook and M. Butler, "UML-B: formal modeling and design aided by UML," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 1, p. 92–122, 2006.

[14] B. Ben Ammar, M. T. Bhiri, and J. Souquières, "Incremental development of UML specifications using operation refinements," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 259–266, Aug. 2008.

[15] R. Glabbeek, "The linear time - branching time spectrum," in *CONCUR '90 Theories of Concurrency: Unification and Extension*, ser. Lecture Notes in Computer Science, J. Baeten and J. Klop, Eds. Springer Berlin Heidelberg, 1990, vol. 458, pp. 278–297.

[16] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, and D. Probst, "Property preserving abstractions for the verification of concurrent systems," *Formal Methods in System Design*, vol. 6, no. 1, p. 11–44, 1995.

[17] S. Nejati, A. Gurfinkel, and M. Chechik, "Stuttering abstraction for model checking," in *Software Engineering and Formal Methods*, 2005, p. 311–320.

[18] C. Braunstein and E. Encrenaz, "CTL-property transformations along an incremental design process," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 1, pp. 77–88, Jun. 2006.

[19] P. Bulychev, I. V. Konnov, and V. A. Zakharov, "Computing (bi)simulation relations preserving CTL^*_X for ordinary and fair kripke structures," in *Mathemathical Methods and Algorithms, ISP RAS*, vol. 12, 2006, pp. 59–76.

[20] F. Ranzato and F. Tapparo, "Computing stuttering simulations," in *Concurrency Theory (CONCUR)*, ser. LNCS. Springer, 2009, vol. 5710, p. 542–556.

[21] M. Leuschel and M. Butler, "Automatic Refinement Checking for B," in *International Conference on Formal Engineering Methods*, 2005.

[22] C. Pons and D. Garcia, "Practical verification strategy for refinement conditions in UML models," in *Advanced Software Engineering: Expanding the Frontiers of Software Technology*, ser. IFIP International Federation for Information Processing. Springer, 2006, vol. 219, pp. 47–61.

[23] E. A. Boiten, "Introducing extra operations in refinement," *Formal Aspects of Computing*, pp. 1–13, Oct 2012.

[24] R. Brummayer and A. Biere, "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays," in *Tools and Algorithms for Construction and Analysis of Systems*. Springer, Mar. 2009, pp. 174–177.

[25] M. Goldsmith, B. Roscoe, and P. Armstrong, *Failures-Divergence Refinement - FDR2 User Manual*, 2005.