Lecture Notes in Electrical Engineering 361

Frank Oppenheimer
Julio Luis Medina Pasaje
Editors

Languages, Design Methods, and Tools for Electronic System Design

Selected Contributions from FDL 2014

Springer

Lecture Notes in Electrical Engineering

Volume 361

About this Series

"Lecture Notes in Electrical Engineering (LNEE)" is a book series which reports the latest research and developments in Electrical Engineering, namely:

- Communication, Networks, and Information Theory
- Computer Engineering
- Signal, Image, Speech and Information Processing
- Circuits and Systems
- Bioengineering

LNEE publishes authored monographs and contributed volumes which present cutting edge research information as well as new perspectives on classical fields, while maintaining Springer's high standards of academic excellence. Also considered for publication are lecture materials, proceedings, and other related materials of exceptionally high quality and interest. The subject matter should be original and timely, reporting the latest research and developments in all areas of electrical engineering.

The audience for the books in LNEE consists of advanced level students, researchers, and industry professionals working at the forefront of their fields. Much like Springer's other Lecture Notes series, LNEE will be distributed through Springer's print and electronic publishing channels.

More information about this series at http://www.springer.com/series/7818

Frank Oppenheimer • Julio Luis Medina Pasaje
Editors

Languages, Design Methods, and Tools for Electronic System Design

Selected Contributions from FDL 2014

Springer

Editors
Frank Oppenheimer
Transportation Division
OFFIS e.V., Oldenburg, Germany

Julio Luis Medina Pasaje
Universidad de Cantabria
Santander, Spain

ISSN 1876-1100 ISSN 1876-1119 (electronic)
Lecture Notes in Electrical Engineering
ISBN 978-3-319-24455-6 ISBN 978-3-319-24457-0 (eBook)
DOI 10.1007/978-3-319-24457-0

Library of Congress Control Number: 2015957253

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Contents

Part I
Formal Models and Verification and Predictability

Chapter 1
Automatic Refinement Checking for Formal System Models

Julia Seiter, Robert Wille, Ulrich Kühne, and Rolf Drechsler

Abstract For the design of complex systems, formal modelling 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 chapter, 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.

1.1 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, modelling languages such as the *unified modeling language* (UML) [28] or the *Systems Modeling Language* (SysML) [33] together with textual constraints, e.g., provided by the *object constraint language* (OCL) [32] 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

J. Seiter (✉) • U. Kühne
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
e-mail: jseiter@informatik.uni-bremen.de; ulrichk@informatik.uni-bremen.de

R. Wille • R. Drechsler
Institute of Computer Science, University of Bremen and Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
e-mail: rwille@informatik.uni-bremen.de; drechsle@informatik.uni-bremen.de

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_1

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 [3, 11, 12, 30, 31]. 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 [17].

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* [1], *Event-B* [2], and *Z* [34] exist. These methods rely on a rigorous modelling 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 [5, 29], 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 modelling 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., [7, 16, 21, 23]). There exist algorithms for computing such relations [10, 25]. 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 [20], 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 chapter which exploits the recent accomplishments in the domain of model-based verification (i.e. approaches like [3, 11, 12, 30, 31]) 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 [24], 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.

The remainder of the chapter is structured as follows. The chapter starts with a brief review on models and their notation in Sect. 1.2. Section 1.3 describes the addressed problem which, afterwards, is formalized in Sect. 1.4. The proposed solution is introduced in Sect. 1.5 and its usefulness is demonstrated in Sect. 1.6 where it is applied to several examples and compared to the results from [20]. The chapter is concluded in Sect. 1.8.

1.2 Models and Their Notation

Modelling languages provide a description of a system to be realized, i.e. proper description means to *formally* define the structure and the behaviour 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 chapter can be applied to similar modelling 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. Figure 1.1a shows a model composed of the class Phone which itself is composed of the attributes $A = \{\text{credit}\}$, the operations $O = \{\text{charge}\}$, and the invariant $I = \{\text{i1}\}$.

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 behaviour of operations.

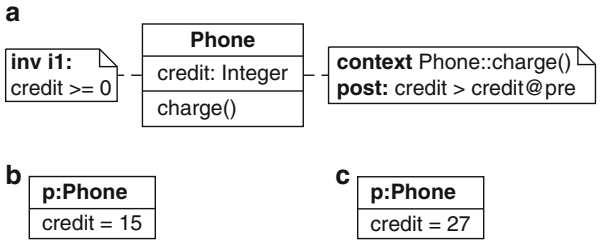

Fig. 1.1 Example of a model and its instantiation. (**a**) Given model; (**b**) State σ_0; (**c**) State σ_1

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 init $\in \Phi$.

Example 2. In the model from Fig. 1.1a, 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 behaviour. 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. Figures 1.1b and c show two valid system states (in terms of object diagrams) for the model from Fig. 1.1a. This is a valid sequence of system states which can be created by calling the operation charge.

1.3 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

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

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 behavioural 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. 1.2a 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. 1.2b. 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 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 modelling 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 $o_1^r \cdot o_2^r \cdot \cdots \cdot o_k^r \in (O^r)^+$ of refined operations.

Example 5. The refinement from the model in Fig. 1.2a to the model in Fig. 1.2b is described by the relation $\text{Ref}=(\text{Ref}_\Sigma, \text{Ref}_\Omega)$. That is, each state σ^r in the

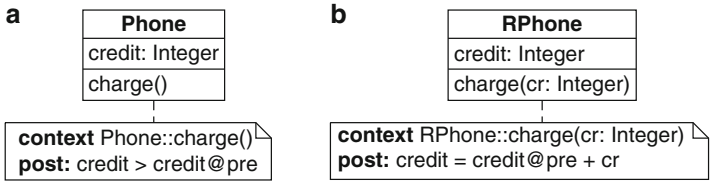

Fig. 1.2 Refinement step. (**a**) Abstract model; (**b**) Refined model

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 any more. In fact, the refinement sketched above contains a serious flaw.

Example 6. The refined model in Fig. 1.2b allows for a behaviour 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 behaviour 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 chapter 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?"

1.4 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 Sect. 1.5.

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 behaviour of a model can be described as a finite state machine, e.g. a *Kripke structure*.

[2]This restriction is common in many approaches (e.g. [3, 11, 12, 30, 31]) and also justified by the fact that, eventually, the implemented system will be realized by bounded physical devices anyway.

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 labelling 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 behaviour 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. Definition 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 $o = (\triangleleft, \triangleright) \in O$ of the model (i.e. $\sigma_1 \rightarrow \sigma_2$ with $\triangleleft(\sigma_1)$ and $\triangleright(\sigma_1, \sigma_2)$), 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., [7, 10, 16, 21, 23, 25]). In this chapter, we adapt this concept for the considered formal models. This leads to the following definition of a simulation relation.

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

1. all initial states in the refined model have a corresponding initial state in the abstract model, i.e. $\forall s_0 \in S_{\mathscr{R}0} \exists s'_0 \in S_{\mathscr{A}0}$ 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. $\forall s, s' : H(s, s') \Rightarrow \mathcal{L}_\mathscr{R}(s) \cap AP_\mathscr{A} = \mathcal{L}_\mathscr{A}(s')$, 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. $\forall s, s' : H(s, s') \Rightarrow s \rightarrow_\mathscr{R} t \Rightarrow \exists t' \in S_\mathscr{A}$ s.t. $s' \rightarrow_\mathscr{A} t'$ and $H(t, t')$.

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

Example 7. As an illustration of the above definition, Fig. 1.3a 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 1.3b on the right shows an example

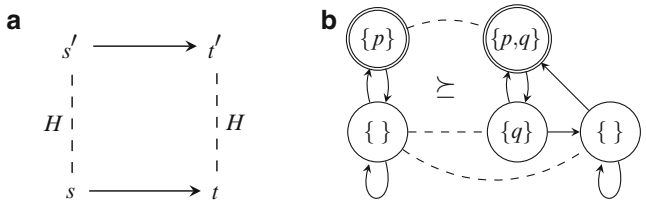

Fig. 1.3 Simulation relation. (**a**) Correspondence of states; (**b**) Example for simulation

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 behaviour allowed in the abstract system than implemented. If we have $\mathscr{R} \preceq \mathscr{A}$, then the traces of $\mathscr{R}$ are contained in those of $\mathscr{A}$. This also means that globally valid properties of $\mathscr{A}$ carry over to $\mathscr{R}$, as, for example, the non-reachability of bad states. Hence, by proving that the applied refinement Ref (cf. Definition 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* [6]). 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 $\mathscr{R}$ and $\mathscr{A}$ with $AP_{\mathscr{R}} \supseteq AP_{\mathscr{A}}$, a relation $H \subseteq S_{\mathscr{R}} \times S_{\mathscr{A}}$ is a divergence blind stuttering simulation (dbs-simulation) iff

1. $\forall s_0 \in S_{\mathscr{R}0} \exists s'_0 \in S_{\mathscr{A}0}$ with $H(s_0, s'_0)$,
2. $\forall s, s' : H(s, s') \Rightarrow \mathscr{L}_{\mathscr{R}}(s) \cap AP_{\mathscr{A}} = \mathscr{L}_{\mathscr{A}}(s')$, 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 $s \rightarrow_{\mathscr{R}} t$, then there exist $t'_0, t'_1 \ldots t'_n$ $(n \geq 0)$ such that $s' = t'_0$ and $\forall i < n : t'_i \rightarrow_{\mathscr{A}} t'_{i+1} \wedge H(s, t'_i)$ and $H(s', t'_n)$.

We say that $\mathscr{R}$ is dbs-simulated by $\mathscr{A}$, written as $\mathscr{R} \preceq_{\mathsf{dbs}} \mathscr{A}$, 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 behaviour of the refined model is consistent with the behaviour of the

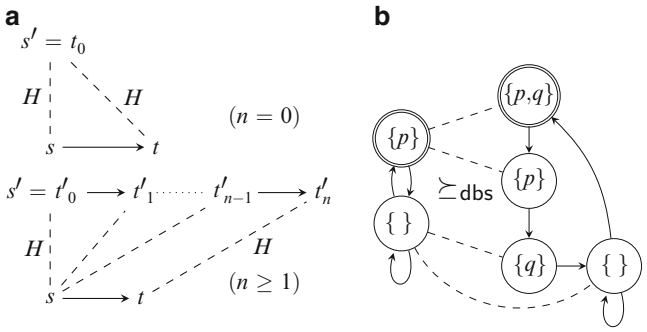

Fig. 1.4 dbs—simulation relation. (**a**) Correspondence of states; (**b**) Example for dbs-simulation

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 $\mathscr{A}$ is also unreachable in $\mathscr{R}$.

Example 8. In Fig. 1.4, the dbs-simulation relation is illustrated. The general scheme of corresponding states and transitions is shown in Fig. 1.4a. In Fig. 1.4b, 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. Definition 5) indeed satisfies the properties of a dbs-simulation H (cf. Definition 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.

1.5 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 [9]).

To this end, we represent arbitrary system states and transitions 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.

1.5.1 Verification Objectives

As motivated in Sect. 1.3, 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 Definition 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.

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

   This check is illustrated in Fig. 1.5a.
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

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

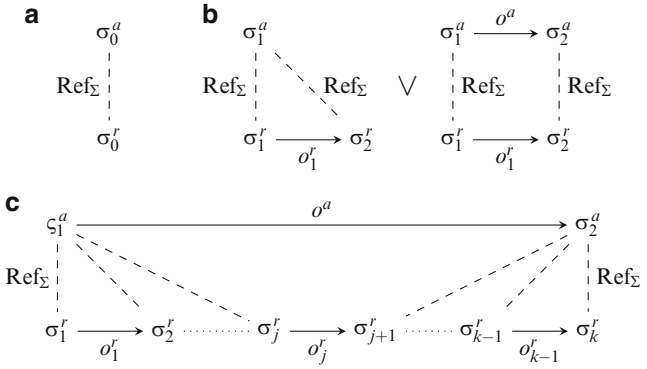

Fig. 1.5 Verification objectives. (**a**) Initialization; (**b**) Single step correspondence; (**c**) Chaining of refined steps

is checked. This check is illustrated in Fig. 1.5b. These two objectives are already sufficient to prove dbs-simulation. Nevertheless, a third objective is additionally checked.

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 $o^r_1 \dots o^r_k$

$$\forall \sigma^a_1, \sigma^r_1, \sigma^r_2 \dots \sigma^r_{k+1} : \mathrm{Ref}_\Sigma(\sigma^a_1, \sigma^r_1) \wedge \sigma^r_1 \xrightarrow{o^r_1} \sigma^r_2 \dots \sigma^r_k \xrightarrow{o^r_k} \sigma^r_{k+1}$$
$$\Rightarrow \left(\lhd_{o^a}(\sigma^a_1) \wedge \rhd_{o^a}(\sigma^a_1, \mathrm{Ref}^{-1}_\Sigma(\sigma^r_{k+1})) \right)$$

is checked. This check is illustrated in Fig. 1.5c. 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.

1.5.2 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 [3, 11, 12, 30] and particularly in [31]. 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 $\boldsymbol{\omega}_i \in \mathbb{B}^{\lceil \mathrm{ld}(|O|)\rceil}$ are created for each step i in the refined model. Depending on the assignment to $\boldsymbol{\omega}$, the respective pre-conditions and post-conditions have to be enforced. This can be realized by a constraint

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

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 $\boldsymbol{\omega}$, we use a constraint $\boldsymbol{\omega} \leq |O_r|$.

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:

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

$$\mathrm{step}(o^a, o_j^r) \Leftrightarrow \bigvee_{i=1}^{|\mathrm{Ref}_\Omega(o^a)|} \mathrm{step}_i(o^a, o_j^r).$$

Here, $\mathrm{step}_i(o^a, o_j^r)$ evaluates to true iff the refined operation o_j^r is the ith step in the refinement of o^a, while $\mathrm{step}(o^a, o_j^r)$ reflects that o_j^r 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. 1.5c, we define the predicate chain:

$$\mathrm{chain}(o^a) \Leftrightarrow \bigwedge_{i=1}^{l} \big(\mathrm{step}_i(o^a, o_i^r) \wedge \boldsymbol{\omega}_i = \mathrm{enc}(o_i^r) \vee i > |\mathrm{Ref}_\Omega(o^a)| \wedge \boldsymbol{\omega}_i = \mathrm{enc}(\mathrm{id})\big).$$

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 $|\mathrm{Ref}_\Omega(o)| < l$. We thereby make use of the maximum number of steps according to Ref_Ω, i.e. $l = \max\{|\mathrm{Ref}_\Omega(o^a)| \mid o^a \in O^a\}$. Next, the above "ingredients" are put together in order to encode the verification objectives of a refinement.

1.5.3 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 Sect. 1.5.1 are encoded. In order to prove (1), we encode its negation