Model-based Specification and Refinement for Cyber-Physical Systems*

Rolf Drechsler[1,2], Serge Autexier[1], and Christoph Lüth[1,2]

Abstract *Cyber-physical systems* are small yet powerful systems which are embedded into their environment, adapting to its changes and at the same controlling it, and often operating autonomously. These systems have reached a level of complexity that opens up new application areas, but at the same time strains the existing design flows in system development. To ameliorate this problem, we propose a novel design flow for cyber-physical systems by adapting model-based specification and refinement methods known from software development. The design flow allows to start with a system specification and its essential properties at a high level of abstraction, and gradually refines it down to an electronic system level. Properties of higher levels can be inherited during refinements to lower levels by relying on local proof obligations only, which results in a design flow capable to keep up with the increasing complexity of cyber-physical systems.

1 Introduction

Embedded systems have become powerful devices which play an increasingly important role in many areas such as production, transport, medicine and logistics: they control autonomous vehicles, aeroplanes, trains and other transport systems, they run production systems up to whole industrial plants, or they can be found in medical implants. We call these *cyber-physical systems*: embedded systems which are connected to the internet and thus merge the

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

* Research supported by BMBF grant 01IW13001 (SPECifIC) and by the German Research Foundation (DFG) within the Reinhart Koselleck project grant DR 287/23-1.

boundaries of the virtual and physical world, which are autonomous, and adapt to and control their environment.

Cyber-physical systems are often employed in safety-critical situations, where failure is not an option and their correct functioning is of paramount importance; however, their complexity strains the currently existing design flows in system development, and makes this correctness hard to guarantee. This paper presents first steps towards a novel design flow for cyber-physical systems by applying methods from model-based software engineering.

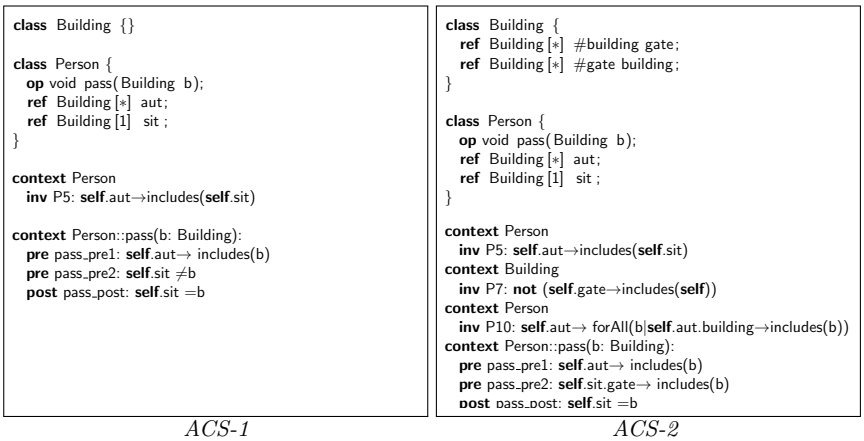
Existing design flows model the system on the so-called Electronic System Level (ESL) using languages such as SystemC [14] or System Verilog. These system level descriptions hide details of the precise realisation in hardware and software while still allowing the execution and simulation of the design. They are executable, concrete models which do not allow to state the desired properties of the system abstractly and formally tractable. However, the initial system specification is mostly given informally in natural language which is typically far away from the ESL. To bridge this gap in expressiveness, the Formal Specification Level (FSL) has been proposed [4, 11]. Its aim is to close this gap by employing formal descriptions means such as the UML to give an abstract specification of the system. It has been used for error-detection in early design phases [9] and to provide a notion of refinement for operations at the FSL [5] .

The contributions of this paper are twofold: first, to develop a semantics of the FSL which was lacking so far and enables to formulate system properties at the FSL. Second, it provides provably correct notions of refinement guaranteeing that system properties are preserved under refinement.

This paper is structured as follows: Illustrated by a simple access control system example, Sec. 2 introduces the specification formalisms and semantics for the central Formal Specification Level (FSL) as well as the kind of correctness properties that can be expressed and proven on that basis. Sec. 3 defines the methods to refine an abstract system specification to more concrete specifications. The refinements are defined such that more detailed specifications inherit properties already established at higher levels of abstraction and are illustrated by the running example. They allow to gradually add more details to the specification until it eventually contains all details of an executable implementation at the Electronic System Level (Sec. 4). Sec. 5 compares the contributions with related work and concludes the paper.

2 Introducing the FSL

As a running example, we consider an access control system which governs the access of people to buildings connected by gates, originally modelled by Abrial [1] using the B method [2]. We start with very high-level specifications describing the overall behaviour.



ACS-1 *ACS-2*

Fig. 1 Initial specification (*ACS-1*), and first refinement step (*ACS-2*), of our running example. For UML class diagrams we use the textual EMFatic [13] notion from the Eclipse Modelling Framework (EMF).

An FSL model consists of classes and operations. The operations can be restricted by OCL constraints, which are either invariants constraining all operations of the class, or pre-/postconditions for specific operations.

In the initial specification (*ACS-1*), the model contains only persons and buildings (given by the two classes). Each person is authorised to enter a number of buildings (attribute aut), and will always be in exactly one building (attribute sit). The class invariant P5 models the requirement specification that a person must only be in a building where they are allowed to be. The pre- and postcondition on the enter operation specify that to enter a building, a person has to be authorised for the building, and that he is not allowed to enter the building he is currently in.

In a first refinement (*ACS-2*), still on a very abstract level we introduce connections between buildings (given by a UML association). This constrains the pass operation: persons can only go from one building in which they are into another if they are authorised to do so, and if the buildings are connected.

Fig. 1 shows the first two steps of our example. It gives a good feel on how system development can start at a very abstract level. At this level, it is clear that the central safety invariant is adequate. We cannot prove it yet, but it gives a good idea on how it could be proven: if we can show that all operations preserve it, and that it holds in the initial states.

2.1 Semantics

Technically, an FSL specification is given by a tuple

$$SP = \langle \mathcal{M}, init, \text{OPN}, inv, pre, post, st \rangle$$

where $\mathcal{M}$ are the classes (specifically, the *object model* [7]), *init* is a specification of the initial states of the system, OPN the operations, *inv* the class invariants, *pre* the preconditions and *post* the postconditions of the operations, and *st* the state diagrams. We do not consider state diagrams in their full UML generality (in particularly, we do not allow hierarchical states and concurrent regions). Instead, we allow state diagrams which can be encoded into pre- and postconditions on the class operations: if the operation f of some class corresponds to a transition from state d to a state r (denoted $f : d \to r$) in the state diagram associated to that class, we encode this in the pre- and post-conditions using an additional attribute m_state as follows: pre s: **self**.m_state =d and **post** ps: **self**.m_state =r.

The semantics of SP consists of the states given by the object model $\mathcal{M}$, and state transitions given by the operations. We add a special state $STOP$, which corresponds to the system state from which no further transitions are possible, and thus models deadlock. This amounts to a *Kripke structure* $[\![SP]\!] = \langle S, I, \to \rangle$ where S is the set of states and I the initial set of states that satisfy *init* and the invariants *inv*. The transition relation $\to = \{\to_o\}_{o \in \text{OPN}}$ is a family of transitions labelled by operations; there is a transition $\sigma_1 \to_o \sigma_2$ iff

(i) all invariants hold in σ_1 and σ_2,
(ii) the preconditions of o are satisfied in σ_1, and
(iii) the postconditions of o are satisfied in σ_1 and σ_2;
(iv) if there is no outgoing transition from σ_1 by the previous clauses for any σ_2, we add a transition $\sigma_1 \to STOP$.

The invariants and pre-/postconditions of an operation are the *constraints* of o and denoted by $cons(o)$. For a specification SP, an execution trace is a infinite sequence $s = \langle s_i \rangle_{i \in \mathbb{N}}$ of states such that $s_i \to s_{i+1}$.

2.2 Verification Properties

Given a specification such as above, we want to exhibit certain properties of the modelled system. For example, the invariant P5 is a property we want the system to have. We call these properties *verification properties*. In its simplest form, a verification property is an OCL property, which is required to hold in all states of the execution trace of a system. This corresponds to the temporal $\Box$ operator; we typically specify safety properties this way ('Something bad never happens'). We might require that other properties only hold

at some point in the future, corresponding to the temporal $\Diamond$ operator; these are typically liveness properties ('Something good will eventually happen'). Generalising slightly further, we define verification properties as follows:

Definition 1 (Verification Property). The set of *verification properties* is defined as follows. Let ϕ be an OCL formula without the **@pre** postfix, then
(i) ϕ is a verification property,
(ii) $\Box\phi$ is a verification property ('safety'),
(iii) $\Diamond\phi$ is a verification property ('liveness'),
(iv) $\Box\Diamond\phi$ is a verification property ('fairness'),
(v) $\Diamond\Box\phi$ is a verification property ('persistence').

The two additional verification properties are 'fairness' ($\Box\Diamond\phi$), which specifies that at every point in the system, ϕ will eventually hold, and 'persistence' ($\Diamond\Box\phi$), which requires that ϕ will at some point start to hold.

Example 1. Using our running example, we illustrate below the different kinds of verification properties. ∀b:Building . ∃b':Building . b.gate→includes(b') is a verification property requiring for the initial state, that each building is connected to at least one other building. To express that each person eventually enters each building for which he has access is achieved by ∀p:Person . ◇(p.aut→includes(p.sit). To express that in each building there are always at least two persons is achieved by ∀b:Building.□∃p,p':Person. p!=p' **and** p.sit=b **and** p'.sit=b.

Example 2. In the *STOP* state, every OCL expression (including True) evaluates to *False*. Thus, the verification formula □True holds for all systems which do not reach the *STOP* state, *i.e.* it expresses freedom from deadlock.

3 Refinement in the FSL

In general terms, refinement is a property-preserving mapping from an 'abstract' model to a more 'concrete' one. Semantically, a refinement should restrict the possible implementations of the specification. As properties are defined in terms of traces in the Kripke structure, to preserve properties for each trace in the concrete model there needs to be a trace in the abstract one.

3.1 Data Refinement

Data refinement allows to change the system state of our models. It can be constructed by mapping all classes of operations of the abstract model to the

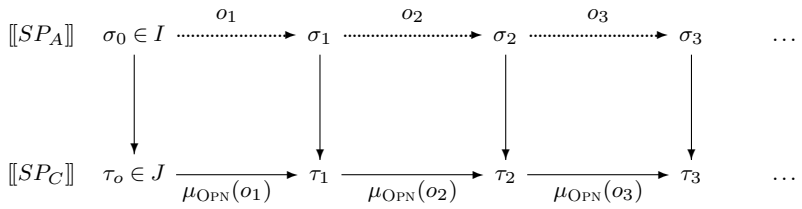


Fig. 2 Data refinement: for each trace in $[\![SP_C]\!]$, we can construct one in $[\![SP_A]\!]$.

concrete one. Such a mapping is given by a *model morphism*. Given two object models $\mathcal{M}$, $\mathcal{M}'$, a model morphism is a tuple of maps $\mu = \langle \mu_C, \mu_A, \mu_O, \mu_{\text{ASSOC}} \rangle$ with $\mu_C, \mu_A, \mu_O, \mu_{\text{ASSOC}}$ maps between the class names, attribute names and association names respectively, which preserve the type hierarchy, the types of attributes and operations, and the associations and their cardinality. Given such a model morphism, the *homomorphic extension* $\mu^\#$ maps OCL expressions ϕ in $\mathcal{M}$ to OCL expressions in $\mathcal{M}'$, by replacing all classes, attributes and associations in ϕ with their image under μ. A model morphism becomes a map between FSL specifications if it also preserves the initial states and, crucially, the invariants on states as well as the pre- and postconditions of the operations in the abstract model (translated appropriately) are implied by the ones in the concrete model:

Definition 2 (Specification Morphism). Given two FSL specifications $SP_A = \langle \mathcal{A}, I_A, \text{OPN}_A, inv_A, pre_A, post_A, st_A \rangle$ and $SP_C = \langle \mathcal{C}, I_C, \text{OPN}_C, inv_C, pre_C, post_C, st_C \rangle$, a *specification morphism* is a tuple of maps $\mu = \langle \mu_M, \mu_I, \mu_{\text{OPN}} \rangle$ where μ_M is a model morphism, and μ_I and μ_{OPN} are maps from the initial states and operations of SP_A to those of SP_C, satisfying:

$$\tau \in I_A \Longrightarrow \mu_I(\tau) \in I_C \tag{1}$$

$$cons_C(\mu_{\text{OPN}}(o)) \Longrightarrow {\mu_{\mathcal{M}}}^\#(cons_A(o)) \tag{2}$$

If all initial states and operations in SP_C are in the image of μ, and the model morphism is injective on the state, we have a *data refinement* from SP_A to SP_C. The following lemma shows why data refinements are useful tools: they preserve all verification properties (in fact, all LTL properties).

Lemma 1. *Given two FSL specifications* $SP_A = \langle \mathcal{A}, I_A, \text{OPN}_A, inv_A, pre_A, post_A, st_A \rangle$ *and* $SP_C = \langle \mathcal{C}, I_C, \text{OPN}_C, inv_C, pre_C, post_C, st_C \rangle$, *and a specification morphism* $\mu : SP_A \to SP_C$. *If* μ_M *is injective, and* μ_I *and* μ_{OPN} *are surjective, then all verification properties which hold in* SP_A *hold in* SP_C *as well.*

Proof. We first show that for any execution trace t in $[\![SP_C]\!]$, the Kripke structure induced by SP_C, there is a trace in $[\![SP_A]\!]$. This corresponds to the fact that for any transition $\sigma \to_o \sigma'$ in $[\![SP_C]\!]$ there is a transition $\tau \to_o \tau'$

in $[\![SP_A]\!]$. There is a transition $\sigma \to_o \sigma'$ in $[\![SP_C]\!]$ only if $cons_C(o')$ holds in σ and σ' for some o'. Because μ_{OPN} is surjective, $o' = \mu_{\text{OPN}}(o)$ and by (2), ${\mu_{\mathcal{M}}}^{\#}(cons_A(o))$ holds in σ and σ'. Hence there are states τ, τ' such that $cons_A(o)$ holds, and there is a transition $\tau \to_o \tau'$ as required.

Now given any LTL property ϕ, ϕ holds for SP_C if it holds for all traces in $[\![SP_C]\!]$. Assume ϕ holds for SP_A. To show ϕ holds for SP_C, we have to show ϕ holds for all traces in SP_C. But for any such trace we have shown there exists a trace in $[\![SP_A]\!]$, for which ϕ holds by assumption. □

Example 3 (Running Example). Continuing our running example, Fig. 1 shows a simple example of data refinement. The model morphism is the injection from *ACS-1* to *ACS-2*, and thus by construction injective; it is easy to see that it is surjective on the operations. It remains to show that the model morphism is a specification morphism to be able to apply Lemma 1. For demonstration, we will show here in detail how syntactic proof obligations are derived, and how they are proven.

From Def. 2 it follows that we need to show that $cons_{ACS\text{-}1}(o) \implies cons_{ACS\text{-}2}(o))$ for all operations. Since there is only one operation pass, we have to show[2]

ACS2.P5 **and** ACS2.P7 **and** ACS2.P10 **and**
ACS2.pass_pre1 **and** ACS2.pass_pre2 **and** ACS2.pass_post
implies ACS1.P5 **and** ACS1.pass_pre1 **and** ACS1.pass_pre2 **and** ACS1.pass_post

To show this proof obligation, we break it down into four obligations using conjunction introduction. We note that except for pass_pre2 all the other axioms are the same in *ACS-1* and *ACS-2*, hence three of the resulting proof obligations are trivial, like this:

P5 **and** P7 **and** P10 **and** pass_pre1 **and** ACS2.pass_pre2 **and** pass_post
implies P5

The remaining one we need to show is ACS1.pass_pre2. To do so, we need to unfold the axioms P7 and ACS2.pass_pre2. (In order to avoid overcrowding, we drop the unneeded axioms P5, P10, ACS2.pass_pre1 and pass_post).[3]

∀**self@pre**: Building. **not** (**self@pre**.gate→ includes(**self@pre**))
and self@pre.sit.gate→ includes(b)
implies self@pre.sit.gate ≠b

Note that the invariant is universally quantified over all buildings, because that is its context (we omit the outer universal quantifiers in the conclusion). Eliminating the quantifier in the premise and instantiating **self@pre** with b results in

[2] In this ad-hoc notation we replace axioms by their name, and use qualified notation s.a to refer to axiom a from specification s.

[3] For readability, we use the notation ∀a:C. p instead of the correct allInstances(C)→forAll(a|p).

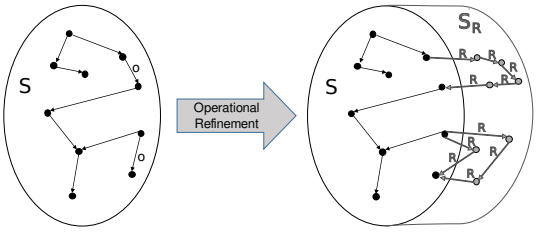


Fig. 3 Operation Refinement of the Kripke Structure $\mathcal{K}_A$ to $\mathcal{K}_C$: in each trace in $\mathcal{K}_C$, sequences of steps starting in I_R, going through S'_R and ending in F_R can be mapped to one step in S.

not (b.gate→ includes(b)) **and self@pre**.sit.gate→ includes(b)
implies self@pre.sit.gate ≠b

This is proven by reductio ad absurdum: assume **self@pre**.sit.gate=b, then the second premise reduces to b.gate→includes(b), and together with the first premise we can derive False, thus proving **self@pre**.sit.gate ≠b.

3.2 Operation Refinement

Data refinement allows changes in the state space; the typical use cases are introduction of new classes or attributes. If we want to replace an abstract operation by more concrete ones, or if we want to move an operation from one class to another, the restrictions of Def. 2 are too prohibitive. In this case, we need *operation refinement*.

Operation refinement decomposes the state space of the refining model SP_C such that the additional states are used for the refining operations (Fig. 3). This requires some restrictions on the refining operations, in particular in how they are allowed to be composed. We express these restrictions as a particular form of UML state diagrams.

When refining an abstract operation $f : d \to r$ of an FSL specification, the following LTL properties that must be proven about the refining state machine $\mathcal{S}$:

(A) the invariants *inv* of the FSL specification must also hold for $\mathcal{S}$, hence we need to prove $\Box\, inv$ for $\mathcal{S}$ (see also the condition of case (ii) in Lemma 2);
(B) the preconditions $pre(f)$ must hold for the initial states;
(C) the postconditions $post(f)$ relate values in the state before and the state after f. This can be reformulated as an LTL property for $\mathcal{S}$ by introducing variables in a straight forward, though technical manner. Instead of providing a formal definition, we illustrate the construction by the following example postcondition:

self@pre.p < **self@pre**.q **and self@pre**.q =**self**.q ∗ 2

This postcondition can be reformulated to the LTL formula

$$\forall x{:}\mathsf{Int} \;.\; (x = \textbf{self}.q \;\textbf{and}\; \textbf{self}.p < \textbf{self}.q) \;\textbf{implies}\; \Diamond\Box\, x = \textbf{self}.q * 2$$

Requiring $\mathcal{S}$ to have that property transfers the postconditions to the refining state machine. Essentially, it means the refining state machine $\mathcal{S}$ must at some point make the postcondition true such that from that on it holds until $\mathcal{S}$ terminates.

We denote by $cons_{\Box}(f)$ the conjunction of the above LTL properties (A)-(C). Additionally, the refining state machine $\mathcal{S}$ must be always terminating, which can be formulated as a proof obligation about $\mathcal{S}$ and be proven by respective methods from termination analysis. Finally, all traces in $\mathcal{S}$ must start in d and terminate in r, which we formulate as an additional requirement to be proven.

Definition 3 (Operation Refinement). Given an FSL specification $SP_A = \langle \mathcal{M}, I, \{f : d \to r\} \uplus \text{OPN}_A, inv_A, pre_A, post_A, st_A \rangle$ an *operation refinement* of SP_A wrt. f is given by a FSL specification $SP_C = \langle \mathcal{M}, I, \text{OPN}_c, inv_C, pre_C, post_C, st_C \rangle$, new operations OPN_{C0} over the states T_{C0} such that:

(i) The states T_{C0} are disjoint from the states of SP_A,
(ii) $\text{OPN}_C = \text{OPN}_{C0} \uplus \text{OPN}_A$, and for all $o \in \text{OPN}_A$, $cons_C(o) = cons_A(o)$
(iii) the state machine $\mathcal{S}_{\text{OPN}_{C0}}$ induced by OPN_{C0} is with initial state d terminating with final state r, and $\mathcal{S} \models cons_{\Box}(f)$

Remark 1. This includes a specific case where an abstract operation f is refined by one concrete operation g which does not preserve the types and hence cannot be expressed by specification morphisms. In that case the proof obligation simplifies to $cons_C(g) \implies cons_A(f)$.

Lemma 2. *Let SP_C be an operational refinement of an operation o of an FSL specification SP_A using the operations* OPN *and P, Q OCL-formulas without* **@pre**. *The following verification properties of SP_A are preserved by operation refinement to SP_C:*

(i) $\Diamond P$
(ii) $\Box P$ *if, and only if,* $\mathcal{S}_{\text{OPN}} \models \Box P$, *i.e. the refining structure satisfies* $\Box P$
(iii) $\Box\Diamond P$
(iv) $\Diamond\Box P$ *if* $\mathcal{S}_{\text{OPN}} \models P \Rightarrow \Box P$ *(note that this is not an equivalence).*

Proof. First we observe, that for each trace of τ_C of the concrete FSL specification SP_C we can construct a trace of SP_A by replacing all maximal finite traces of the refining state machine $\mathcal{S}_{\text{OPN}}$ which occur in τ_C by a single transition $\to_o$ of the refined operation o of SP_A. As a consequence all states in τ_A occur in the same order in τ_C, except that they may be interspersed with states from $\mathcal{S}_{\text{OPN}}$. Thus, if $\Diamond P$ or $\Box\Diamond P$ hold for τ_A, they also hold for τ_C, which proves (i) and (iii). Assume $\Box P$ holds for τ_A we need to know that P also holds for all inserted states from $\mathcal{S}_{\text{OPN}}$, which is ensured by the proof

obligation $\Box P$ for the $\mathcal{S}_{\text{OPN}}$, proving (ii). Finally, assume $\Diamond\Box P$ holds for τ_A; this means that there is an infinite suffix of the trace τ_A for which $\Box P$ holds. Now consider a state s in τ_C at which a trace from $\mathcal{S}_{\text{OPN}}$ starts: either it is before the suffix, then $\Box P$ will be reached later in the trace; or it is in the suffix, then P holds for s and because $P \Rightarrow \Box P$ holds for $\mathcal{S}_{\text{OPN}}$, the rest of the sequence in $\mathcal{S}_{\text{OPN}}$ satisfies $\Box P$, hence $\Box P$ holds, proving (iv). □

Example 4 (Running Example). We now consider the refinement step from *ACS-2* to *ACS-3* (see Fig. 4), which consists of moving the pass operation from the class Person to the class Door, where it is called enter. This refinement cannot be expressed with a specification morphism, because the refinement step actually consists of two refinements: first, a data refinement which introduces the class Door, and then an operational refinement which moves the operation pass from the class Person to the operation enter in Door. It is thus an example of a simple operational refinement mentioned in Remark 1. For this refinement, we have to show that the pre- and postconditions on enter imply those on pass. This means we have to prove that

(∀**self**: Door ∀p: Person. p@pre.aut→ includes(**self@pre**.dest) **and**
p@pre.sit= **self@pre**.org **and** p.sit= **self**.dest)
implies (∀**self**: Person ∀b: Building. **self@pre**.aut→ includes(b@pre) **and**
self@pre.sit.gate→includes(b@pre) **and**
self.sit= b)

This proof is a bit more delicate. We start by expanding the universal quantification of the goal by substituting **self**: Person and b: Building with indefinite constants p0 and b0. Conversely, in the premise the universally quantified variable **self**: Door is instantiated with b0.dest_dom and p:Person with p0[4] to obtain the three subgoals:

(a) p0@pre.aut→includes(b0@pre.dest_dom.dest)
and p0@pre.sit=b0@pre.dest_dom.org **and** p0.sit=b0.dest_dom.dest)
implies p0@pre.aut→includes(b0)
(b) p0@pre.aut→includes(b0@pre.dest_dom.dest)
and p0@pre.sit=b0@pre.dest_dom.org **and** p0.sit=b0.dest_dom.dest)
implies p0@pre.sit.gate→includes(b0)
(c) p0@pre.aut→includes(b0@pre.dest_dom.dest)
and p0@pre.sit=b0@pre.dest_dom.org **and** p0.sit=b0.dest_dom.dest)
implies p0.sit=b0

To show the first conjoint (a), knowing that b0@pre =b0, p0@pre.aut=p0.aut, b0@pre.dest_dom=b0.dest_dom, and b0.dest_dom.dest=b0 this simplifies to

(p0.aut→ includes(b0) **and** p0@pre.sit= b0.dest_dom.org **and** p0.sit= b0)
implies p0.aut→ includes(b0)

[4] Note that substituting **self** by b0.dest_dom in **self@pre**.dest moves the suffix **@pre** inside to result in b0@pre.dest_dom.dest.

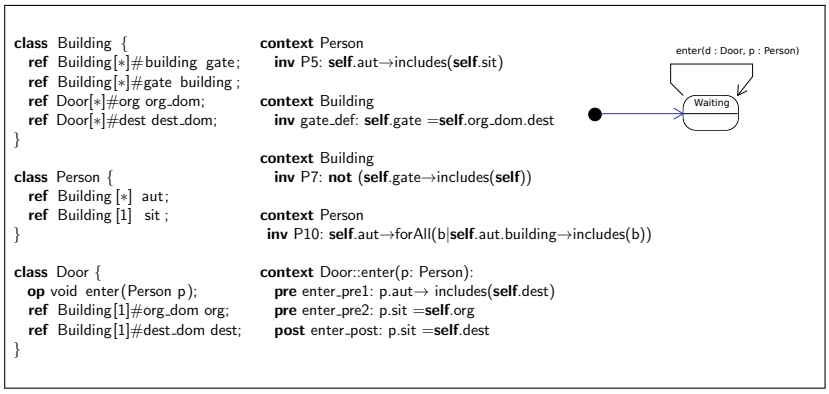


Fig. 4 Third refinement: the class Door is introduced to connect buildings, and the pass operation is implemented by the enter operation from the new class.

which holds trivially.

Applying the same simplifications to (b) results in

(p0.aut→ includes(b0) **and** p0@pre.sit= b0.dest_dom.org **and** p0.sit= b0)
implies p0@pre.sit.gate→includes(b0)

By applying p0@pre.sit=b0.dest_dom.org we obtain the goal

(p0.aut→ includes(b0) **and** p0@pre.sit= b0.dest_dom.org **and** p0.sit= b0)
implies b0.dest_dom.org.gate→includes(b0)

Now using the invariant gate_def we can further transform the goal to

(p0.aut→ includes(b0) **and** p0@pre.sit= b0.dest_dom.org **and** p0.sit= b0)
implies b0.dest_dom.org.org_dom.dest→includes(b0)

This follows because org is inverse to org_dom and further dest is inverse to dest_dom, and some further technical reasoning about the OCL semantics of collections.[5]

Finally, to show the third conjoint (c) we also apply the same simplifications to obtain

(p0.aut→ includes(b0) **and** p0@pre.sit= b0.dest_dom.org **and** p0.sit= b0)
implies p0.sit= b0

which holds trivially.

The final step within the FSL towards the ESL as in Fig. 6 is to refine the enter operation in the Door class. If the person entering the door is allowed to pass, a green light should indicate this; after the person has passed, or

[5] The relevant properties are ∀d:Door . d.org.org_dom→includes(d) and ∀b:Building . b.dest_dom.dest→includes(b).

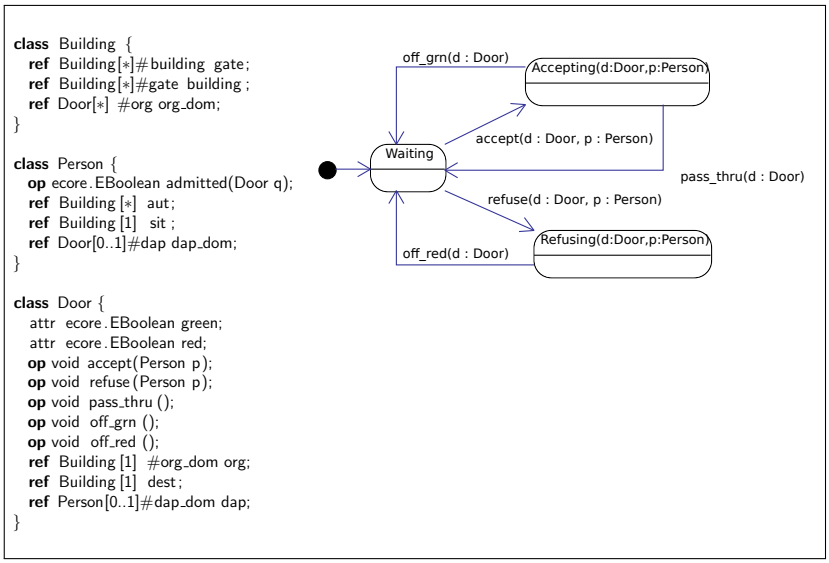


Fig. 5 In this step, we refine the enter operation by introducing several new operations, and describe their behaviour in a UML state machine (right).

at most after 30 seconds, the green light should be off again. If the person entering is not allowed to pass, a red light should be lit for two seconds, and the door should stay locked.

To model the relevant aspects of this behaviour (we do not model the timing aspects here), we need to add attributes for the green and red lights, and introduce operations such as accept, pass_thru, green_off which model the desired state transitions. The state diagram modelling the desired behaviour is on the right of Fig. 5. As new state names, we introduce Waiting, Refusing and Accepting. We incur three proof obligations, corresponding to the pre- and postconditions of the enter operation which have to be derived from the conjunction of pre- and postconditions of the accept and refuse operations and the invariants. The proofs are more elaborate than the ones we have seen so far, but mathematically routine. We further have to prove that the refining operations induce a state machine, which starting in state Waiting always terminates in state Waiting.

4 Refinement from the FSL to the ESL

The ESL language we are using here is SystemC, a C++ class library which allows a cycle-accurate model of the hardware. It provides classes to simulate

```
class Users {
private:
  static std::unordered_set<Building>
         aut[NUM_PERSONS];
  static Building sit[NUM_PERSONS];

public:
  static void init()
  {
    for (Person p= 0;
         p< NUM_PERSONS; p++)
      {
        aut[p]=
          std::unordered_set<Building>();
        sit[p]= DEFAULT_BUILDING;
      }
  }

  static bool admitted(Person p,
                       Building b)
  {
    return (aut[p].count(b) > 0);
  }
};
```

```
SC_MODULE(Door)
{
  sc_in<Person> card;
  sc_out<bool> green;
  sc_out<bool> red;

public:
  Building org;
  Building dest;

 private:
  Person dap= NO_PERSON;

 public:
  SC_CTOR(Door)
  {
    dap= NO_PERSON;
    SC_THREAD(operate);
  }
```

```
void operate()
{
  while (true) {
    wait(card.value_changed_event());
    dap= card.read();
    if (Users::admitted(dap, dest))
      {
        accept();
        wait(sc_time(TIMEOUT_GREEN, TIME_UNIT),
                     passed.posedge_event());
        if (passed.read())
          pass_thru();
        else
          off_grn();
      }
    else
      {
        refuse();
        wait(TIMEOUT_RED, TIME_UNIT);
        off_red();
      }
  }
}
```

Fig. 6 Implementation of the example in the ESL. We show the relevant excerpts, the actual SystemC implementation about 360 loc.

the hardware constructs; these can be mixed in with usual C++ to allow hardware-software co-design.

In brief, the class sc_module models the basic building blocks of the hardware. The signals going in and out of such a block are modelled by generic datatypes sc_in<T> and sc_out<T>.

To map FSL specifications to the ESL, we map classes in the FSL to either instances of the sc_module class (if they are implemented in hardware), or to usual C++ classes (if they are implemented in software). For the former, public attributes are mapped to signals, corresponding to the fact that they can be read or written. Methods are mapped to function members of type **void** f(); all parameter and result passing must be performed by reading and writing to attributes (*i.e.* signals). An instance of sc_module must have a main thread, which must implement the state diagram of the FSL specification.[6]

Example 5 (Running Example: Finale). Fig. 6 shows an excerpt of the ESL implementation. The operation admitted is implemented in software, by the Users class (left of Fig. 6). The class Door is implemented by the Door instance of sc_module; Fig. 6 shows the declaration of the instance, and the implementation of the main method operate.

At this point, we do not aim to verify formally that the SystemC code satisfies the pre- and postconditions and preserves invariants, as formal verification of C++ is still very much a research problem in its own right. There are various methods by which we can *validate* the pre- and postconditions and invariant preservation to a certain degree, *i.e.* check for obvious violations or contradictions [6, 12].

[6] As we currently do not consider concurrency, there can be only one state diagram, and hence only one main thread.

5 Conclusions and Outlook

We have presented the first steps towards a new design flow for cyber-physical systems in order to keep up with the rapidly increasing complexity of these systems. It applies methods from model-based software engineering and allows to start from an abstract specification which states the essential properties down to an implementation in SystemC by using refinement steps which have been proven correct with respect to a comprehensive semantics based on Kripke structures. It thus bridges the gap in expressiveness between system-level modelling languages such as SystemC and initial specifications in natural language leveraging the advantages of the UML and its existing tools without forcing system engineers to give up their existing design flow.

We have developed two refinement operations in Sec. 3, but these should be seen as representative. There are many more possible refinement operations (*e.g.* an obvious one that comes to mind is to remove attributes), which may be uncovered by further case studies. Also, the current approach does not allow to handle concurrency, which is another major area for future work.

In related work, there are a number of so-called wide-spectrum languages which cover the whole of the design flow. For example, our running example was originally conceived by Abrial for the B language [2]. Atelier B, the tool supporting B, covers the whole design flow (and there is a connection to VHDL called BHDL), but it does not easily allow designers to keep their existing work flow, and has a steeper learning curve than our UML-based approach. The advantage of using UML is that engineers can start with a light-weight modelling, using just a few UML diagrams (class diagrams with OCL pre- and postconditions and invariants as in our example). Another relevant language is Event-B, an extension of B with events, which is supported by the Rodin tool chain [3]. There is work on UML-B [10], a UML front-end for Event-B which even supports a notion of refinement [8], which essentially has the same aim as our approach, namely allowing the engineer to use well-known UML concepts rather than having to learn a new specification language, except it uses Event-B as the semantic 'back-end'; our use of Kripke structures makes it easier to connect to a completely separate ESL language such as SystemC.

We have implemented a prototypical tool support for our approach using tools from the Eclipse Modelling Framework (EMF) for the UML, and the LLVM toolset (clang) for SystemC. Our tool can automatically find the obvious mappings between the refinement steps, but has no proof support for proof obligations arising in the development. Its aim is also to be able to track the impact of changes in the development *e.g.* if we find during validation that the initial specification is not adequate.

In closing, we are confident this work will bring the benefits of model-based software engineering into the systems development community by combining the advantages of existing industrial-scale system design flows with well-known model-based specification and refinement concepts.

References

[1] Abrial JR (1999) System study: Method and example. http://atelierb.eu/ressources/PORTES/Texte/porte.anglais.ps.gz

[2] Abrial JR, Abrial JR, Hoare A (2005) The B-book: assigning programs to meanings. Cambridge University Press

[3] Abrial JR, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. International journal on software tools for technology transfer 12(6):447–466

[4] Drechsler R, Soeken M, Wille R (2012) Formal Specification Level: Towards verification-driven design based on natural language processing. In: Forum on Specification and Design Languages (FDL) 2012, IEEE, pp 53–58

[5] Przigoda N, Stoppe J, Seiter J, Wille R, Drechsler R (2015) Verification-driven Design Across Abstraction Levels – A Case Study. In: Euromicro Conference on Digital System Design (DSD)

[6] Przigoda N, Wille R, Drechsler R (2015) Contradiction Analysis for Inconsistent Formal Models. In: International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)

[7] Richters M, Gogolla M (2002) OCL: Syntax, Semantics, and Tools. In: Clark T, Warmer J (eds) Object Modeling with the OCL, no. 2263 in Lecture Notes in Computer Science, Springer, pp 42–68

[8] Said MY, Butler M, Snook C (2009) Class and state machine refinement in UML-B. In: Proc. Workshop on Integration of Model-based Formal Methods and Tools (associated with iFM 2009)

[9] Seiter J, Wille R, Kühne U, Drechsler R (2014) Automatic Refinement Checking for Formal System Models. In: Forum on Specification and Design Languages (FDL), pp 1–8

[10] Snook C, Butler M (2006) UML-B: Formal modeling and design aided by UML. ACM Trans Softw Eng Methodol 15(1):92–122

[11] Soeken M, Drechsler R (2015) Formal Specification Level. Springer

[12] Stoppe J, Wille R, Drechsler R (2014) Validating SystemC Implementations Against Their Formal Specifications. In: Symposium on Integrated Circuits and System Design (SBCCI)

[13] The Eclipse Foundation (2012) Emfatic: A textual syntax for EMF Ecore (meta-)models. URL http://www.eclipse.org/emfatic

[14] The IEEE Computer Society (2011) Standard SystemC Language Reference Manual. IEEE Standard 1666 – 2011