Object-Oriented Component-Based Design using Behavioral Contracts: Application to Railway Systems

Sebti Mouelhi, Khalid Agrou, Samir Chouali, Hassan Mountassir

Object-Oriented Component-based Design using Behavioral Contracts: Application to Railway Systems

Sebti Mouelhi
SafeRiver
92120, Montrouge, France
sebti.mouelhi@safe-river.com

Khalid Agrou
UPMC Pro
75005, Paris, France
khalid.agrou@free.fr

Samir Chouali
FEMTO-ST, UMR CNRS 6174
25030, Besançon, France
schouali@femto-st.fr

Hassan Mountassir
FEMTO-ST, UMR CNRS 6174
25030, Besançon, France
hmountas@femto-st.fr

ABSTRACT

In this paper, we propose a formal approach for the design of object-oriented component-based systems using *behavioral contracts*. This formalism merges interface automata describing communication protocols of components with the semantics of their operations. On grounds of consistency with the object-oriented paradigms, we revisit the notions of *incremental design* and *independent implementability* of interface automata by novel definitions of components compatibility, composition, and refinement. Our work is illustrated by a design case study of CBTC railway systems.

Keywords

Object-oriented components, Behavioral contracts, Interface automata, Method semantics, Refinement, Railway systems.

1. INTRODUCTION

Component-based development approaches aim to reduce the cost of complex systems design by reusing prefabricated components. A software component is a black box unit of a third-party composition and deployment, with explicit dependencies to its environment [28]. It is exclusively reusable via its interface behavioral specification without disclosing implementation details. However, the design by composition often raises mismatches. A safe interoperability between components should fulfill two main properties: (1) their interactions do not lead to undesirable situations, and (2) the substitution of a component with a new one does not alter the compound system. Commonly, the functional interoperability of components is usually checked at the levels of their operations *signatures* (names and argument types), *semantics* (pre/postconditions and invariants), and their *communication protocols*. A communication protocol regards the temporal scheduling of assumptions on the environment inputs to a component, its output behavior, and its local operations. Component protocols can be modeled naturally by *interface automata* [13] obedient to an *optimistic* approach of composition closely related the the object-oriented context: if they communicate within an environment allowing them to avoid deadlocks, they can be used without changes. In the industrial context, this approach allows errors detection during the design phase, and hence taking the appropriate decision: either keeping components as they was received from their manufacturer, or requesting their modification.

The first contribution of this paper is to demonstrate how object-oriented component-based design (OOCBD) is more rigorous by means of *behavioral contracts* merging interface automata with the semantics of methods. The optimistic approach of interface automata composition is accordingly adapted to fulfill the interaction aspects of object-oriented components. The composition of two interface automata is computed by removing from their synchronized product all states from which the environment cannot prevent deadlock states (arising from semantic and protocol mismatches) by enabling *controllable* or *autonomous* actions [13, 5]. We define the concept of autonomous actions differently by reclassifying them into *method*, *return*, and *exception* actions.

The second is about the study of components *refinement* using behavioral contracts, intended to ensure an independent implementability of components. We present refinement as an *expanding simulation* between interface automata allowing (i) the introduction, in a component refinement, more details about common provided services with the abstraction, and (ii) providing more services than the abstraction. These features lead to consider the refinement relation as *covariance* on input and output events of a component: refinement issues (resp. provides) more outputs (resp. inputs) than the abstraction. A concrete version C' of a component refines an abstract one C if each input, output, or local event of C is simulated at least by the same event in C'. The *alternating simulation* [7], originally proposed in [13], to refine interface automata, requires *contravariance* on input and output events. It is not quite consistent, from our angle, with the object-oriented context.

All through the paper, we justify the relevance of our approach for checking design integrity of railway systems. We propose a case study of trains protection functions in modern railway CBTC control systems to track the evolution of safety standards such as the European Norm EN 50128 [1],

and to give a new industrial perspective for the design of such critical systems using an object-oriented approach.

The paper results appeared partially in preliminary formulations and other contexts in [12, 23]. In Section 2, we start by introducing informally our case study to avoid cluttering the contribution sections. It is nevertheless recalled gradually to validate the various introduced formal concepts. In sections 3 and 4, we proceed with the study of behavioral contracts, and our approach of components compatibility and composition. Section 5 is devoted to the study of refinement of behavioral contracts. Discussions, perspectives, and related works are presented in Section 6.

2. RAILWAY CASE STUDY

We introduce a simplified case study of trains protection functions in CBTC (*Communications-Based Train Control*) systems [2] (cf. Figure 1). These systems allow benefits such as high traffic densities, automatic anti-collision processing, adoption of automated trains, etc. A CBTC system is an automatic train controller independent of track circuits. It determines continuously precise locations of trains, and sends them back control signals by means of bidirectional train-to-wayside data exchange. It has train-borne and wayside devices instrumenting automatic train protection (ATP) functions, as well as automatic train operation (ATO), and automatic train supervision (ATS) functions. ATP functions ensure safety-critical requirements (speed control and braking). ATO functions cover non-safety-related requirements. ATS functions cover traffic managements [2]. ATO and ATS functions do not play a significant role on safety, and they are not considered in this paper.

We consider trains control based on *moving block* regime. The positions of a train and its velocity are continuously computed, based on its kinetic and potential energy, and communicated via wireless to wayside equipments. Thus, a *protected area of circulation* is established for each train up to the next nearest obstacle. The train is consequently able to adapt its speed and braking curves in order to not overcome the limit of this area, called the *danger point* [27].

The *On-Board Device* (OBD) of each train computes two fictional locations: the *tail* and *head external locations* (TEL and HEL). The track fragment between them covers the whole train. Usually, this choice is caught on grounds of safety to keep a safe distance between trains in case of system malfunction. Locations are coordinates on the *trains path* composed of *segments* and set in a given direction according to the *railroad switches* positions. A segment is identified by a number, a length, and a beginning coordinate. In Figure 1, the switch p_1 is positioned on the segment s_3, and the train path is the sequence s_1, s_2, s_3, s_4, s_5, s_6, etc.

The OBDs of T1 and T2 initiate the protection process by asking if they are visible to a *Movement Control Unit* (MCU). There are several MCUs covering the entire line, with overlapping coverage sections allowing safe information handover between them. Only one is represented in our case study. The trains locations are sent by wireless to the nearest *Base Transmission Station* (BTS). The latter converts radio signals to digital data and transmits them to the *Data Exchange Unit* (DEU), which in turn transfers them to MCU (event 1). MCU determines whether the zone between TEL and HEL is completely or partially included within its coverage area, and responds T1 and T2. In Figure 1, T1 and T2 are both visible to MCU (event 2).

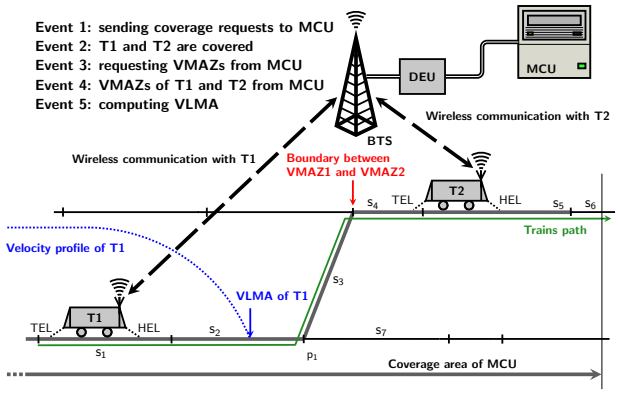


Figure 1: Simplified trains protection functions.

Next, each train asks from its covering MCUs the *Vital Movement Authority Zone* (VMAZ): the area (sequence of segments) in which the train can safely circulate (event 3). In Figure 1, MCU sends to T1 a VMAZ limited by the beginning of s_1 (containing its TEL and HEL) and the end of s_3, and sends to T2 a VMAZ limited by the beginning of s_4 (containing its TEL) and the end of s_5, the last segment covered by MCU (event 4). MCU ensures that VMAZs of successive trains never overlap to avoid collisions. VMAZs are computed by chaining segments according to the route informations. Chaining may be interrupted up to the nearest obstacle on the train trajectory: the end of MCU coverage area, an uncontrolled switch, or the beginning of the segment containing TEL of the next train, etc. This function is covered by a separate wayside component managing persistent informations (segment and switch locations) and variant ones (switch positions) of the route during the traffic.

Finally, the train computes the danger point, called *Vital Limit of Movement Authority* (VLMA), within the boundaries of the received VMAZ. To locate VLMA, OBD takes a fixed safety margin beforehand the limit of its VMAZ. The train velocity is gradually reduced to reach zero when HEL reaches VLMA (event 5).

3. BEHAVIORAL CONTRACTS

In this section, we present behavioral contracts of object-oriented components combining interface automata with the semantics of their methods. We start by interface automata.

3.1 Interface automata

Interface automata [13, 5] model the communication protocols of software components in terms of temporal scheduling of their *input*, *output*, and *hidden actions*. In OOCBD, input actions may represent the component public provided methods, the assignment of return values of their calls, and catching their exceptions. Output actions may represent method calls, and return or exception events. Private methods are implicit and not specified by actions. However, their calls, the assignment of their return values, and the catching of their thrown exceptions are modeled by hidden actions.

Definition 1. *A* interface automaton A *is a tuple* $(\Upsilon_A, \imath_A, \Sigma^{\mathrm{I}}_A, \Sigma^{\mathrm{O}}_A, \Sigma^{\mathrm{H}}_A, \delta_A)$ *where:* Υ_A *is a finite set of states;* $\imath_A \in \Upsilon_A$ *is the initial state;* Σ^{I}_A, Σ^{O}_A, *and* Σ^{H}_A *are resp. the sets of input, output, and hidden actions;* $\delta_A \subseteq \Upsilon_A \times \Sigma_A \times \Upsilon_A$ *is the set of transitions.* A *is* empty *iff* $\Upsilon_A = \emptyset$.

The alphabet of A consists of "$a?$" for $a \in \Sigma_A^{\mathrm{I}}$, "$a!$" for $a \in \Sigma_A^{\mathrm{O}}$, and "$a;$" for $a \in \Sigma_A^{\mathrm{H}}$. The sets $\Sigma_A^{\mathrm{Im}} \subseteq \Sigma_A^{\mathrm{I}}$, $\Sigma_A^{\mathrm{Om}} \subseteq \Sigma_A^{\mathrm{O}}$, and $\Sigma_A^{\mathrm{Hm}} \subseteq \Sigma_A^{\mathrm{H}}$, are resp. actions of public provided methods, call of environment public methods, and calls of private methods. The set Σ_A^{m} of *method actions* of A is $\Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Om}} \cup \Sigma_A^{\mathrm{Hm}}$. Given a set of variables V, we define by $\mathbb{T}[v]$ the type of $v \in V$ *i.e.* $v{:}\mathbb{T}[v]$, and by $\mathbb{T}[\![V]\!] = \prod_{v \in V} \mathbb{T}[v]$ the type of V (cartesian product of $\mathbb{T}[v]$ for all $v \in V$). The *signature* of a method action $a \in \Sigma_A^{\mathrm{m}}$ is $a(i_1{:}\mathbb{T}[i_1], ..., i_k{:}\mathbb{T}[i_k]) \rightarrow o{:}\mathbb{T}[o] \sharp e$. The set of input parameters of a is $\Psi_A^{\mathrm{i}}(a) = \{i_1, ..., i_k\}$. The set of return parameters $\Psi_A^{\mathrm{o}}(a)$ of a is the singleton $\{o\}$. We define $\mathsf{R}_A(a) = o$ the *return action* of a, and $\mathsf{E}_A(a) = e$ the *exception action* of a. The set of attributes used by a is denoted by $\Lambda_A(a)$ if $a \in \Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Hm}}$. The absence of parameters, attributes, or exceptions is represented by a void. If $\mathsf{R}_A(a)$ and $\mathsf{E}_A(a)$ are defined, we set Σ_A^{r} and Σ_A^{e} resp. to $\{\mathsf{R}_A(a) \mid a \in \Sigma_A^{\mathrm{m}}\}$ and $\{\mathsf{E}_A(a) \mid a \in \Sigma_A^{\mathrm{m}}\}$. We denote, by $\Sigma_A^{*\mathrm{r}}$ and $\Sigma_A^{*\mathrm{e}}$, resp. the sets $\Sigma_A^{\mathrm{r}} \cap \Sigma_A^*$ and $\Sigma_A^{\mathrm{e}} \cap \Sigma_A^*$ where $* \in \{\mathrm{I}, \mathrm{O}, \mathrm{H}\}$. It is worth to mention here that $\Sigma_A = \Sigma_A^{\mathrm{m}} \cup \Sigma_A^{\mathrm{r}} \cup \Sigma_A^{\mathrm{e}}$. We set $Succ_A(s, a) = t$ such that $(s, a, t) \in \delta_A$. A *run* σ of A is a finite alternated sequence $s_0[a_0]...[a_{n-1}]s_n$ of states and actions where $(s_k, a_k, s_{k+1}) \in \delta_A$ for all $k \in \mathbb{N}_{<n}$. We set $\Sigma_A\langle\sigma\rangle = \{a_k \in \Sigma_A \mid k \in \mathbb{N}_{<n}\}$ and $\Upsilon_A\langle\sigma\rangle = \{s_k \in \Upsilon_A \mid k \in \mathbb{N}_{\leq n}\}$. We denote, by $\Theta_A(s)$, the set of runs reaching $s \in \Upsilon_A$ from $\imath_A$. A state $s \in \Upsilon_A$ is *reachable* in A if $\Theta_A(s) \neq \emptyset$.

Assumptions: Interface automata are deterministic, *i.e.* for all $(s, a, s_1), (s, a, s_2) \in \delta_A$, $s_1 = s_2$. All states $s \in \Upsilon_A$ are reachable in A. Consider an action $a \in \Sigma_A^{\mathrm{m}}$ where $\mathsf{R}_A(a)$ and $\mathsf{E}_A(a)$ are defined. If $a \in \Sigma_A^{\mathrm{Im}}$ (resp. Σ_A^{Om} and Σ_A^{Hm}), then $\mathsf{E}_A(a) \in \Sigma_A^{\mathrm{O}} \setminus \Sigma_A^{\mathrm{m}}$ (resp. $\Sigma_A^{\mathrm{I}} \setminus \Sigma_A^{\mathrm{m}}$ and $\Sigma_A^{\mathrm{H}} \setminus \Sigma_A^{\mathrm{m}}$): a component providing or requiring a knows its exception. If $a \in \Sigma_A^{\mathrm{Im}}$, then $\mathsf{R}_A(a) \in \Sigma_A^{\mathrm{O}} \setminus \Sigma_A^{\mathrm{m}}$: the method a must output its return value. If $a \in \Sigma_A^{\mathrm{Om}} \cup \Sigma_A^{\mathrm{Hm}}$, then $\mathsf{R}_A(a)$ may belong or not to $(\Sigma_A^{\mathrm{I}} \cup \Sigma_A^{\mathrm{H}}) \setminus \Sigma_A^{\mathrm{m}}$: a component invoking a may require or not the assignment of its return value.

Well-formedness

Object-oriented implementation rules should be covered by the runs of interface automata. A provided public non-void method should be specified at least by a sequence of events starting and ending resp. by an input method action and an output return one interposed, by calls of local private or environment public methods and the assignment of their return values. They may be interleaved optionally by catching or throwing exceptions events. A call of a non-void method, made by a component requiring the assignment of its return value, is followed necessarily by a return input action, and optionally by an exception catch one. All the actions of a component are *autonomous* (controllable), except method or exception input actions. It's up to the environment to enable or not these actions. In [13, 5], only output and hidden actions are required to be autonomous. From our perspective, input return actions of non-void method calls, made by a component, are also autonomous because the environment is expected to provide their return values and the component has the option to assign them or not.

The set Σ_A^{aut} of autonomous actions is $\Sigma_A \setminus (\Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Ie}})$. We define by $\Sigma_A^*(s)$ where $* \in \{\mathrm{I}, \mathrm{O}, \mathrm{H}, \mathrm{Im}, \mathrm{Om}, \mathrm{Hm}, \mathrm{Ir}, \mathrm{Or}, \mathrm{Hr}, \mathrm{Ie}, \mathrm{Oe}, \mathrm{He}, \mathrm{m}, \mathrm{r}, \mathrm{e}, \mathrm{aut}\}$ the set of actions in Σ_A^* enabled from $s \in \Upsilon_A$. $\Sigma_A(s)$ is the set of all enabled actions from s. The run $\sigma = s_0[a_0]...[a_{n-1}]s_n$ is called autonomous in A if $\Sigma_A\langle\sigma\rangle \subseteq \Sigma_A^{\mathrm{aut}}$ for all $k \in \mathbb{N}_{<n}$. It is called *exception-free* if $\Sigma_A\langle\sigma\rangle \subseteq \Sigma_A \setminus \Sigma_A^{\mathrm{e}}$ for all $k \in \mathbb{N}_{<n}$. A state $s' \in \Upsilon_A$ is reachable autonomously (resp. without exceptions) from $s \in \Upsilon_A$ in A if there is an autonomous (resp. exception free) run between s and s'.

Definition 2. *An interface automaton A is* well-formed *iff for all state $s \in \Upsilon_A$, and action $a \in \Sigma_A^{\mathrm{m}}(s)$ where $\mathsf{R}_A(a) \in \Sigma_A^{\mathrm{r}}$, there is at least a state $t \in \Upsilon_A$, where $\mathsf{R}_A(a) \in \Sigma_A^{\mathrm{r}}(t)$, reachable autonomously without exceptions from $Succ_A(s, a)$.*

3.2 Method semantics

The semantics of a provided method consists of: (i) a precondition representing the environment assumptions on input parameters, (ii) an abstract specification of the return parameter computation using input parameters and attributes, (iii) a termination postcondition on the return parameter depending on input parameters and attributes, and (iv) an extra postcondition describing exception conditions on parameters and attributes. A method call semantics is defined only by a precondition on input parameters and a postcondition on input and return parameters.

Given a set of variables V, a *condition* on v is a subtype of $\mathbb{T}[v]$. A condition Q on V is a subtype of $\mathbb{T}[\![V]\!]$. We denote by $Q[w_1, ..., w_n]$ (or $Q[\![W]\!]$), the *projection* of Q on variables in $W = \{w_1, ..., w_n\} \subseteq V$. These conditions can be concretely defined as predicates in a theory adapted to the variable types. Consider the set $Z \subseteq W$, and two conditions P and Q subtypes of $\mathbb{T}[\![V]\!]$, we set the following equivalences to define semantic formulas in the rest of the paper:

- $\bot[\![W]\!] \equiv P[\![W]\!] = \emptyset$; $\top[\![W]\!] \equiv P[\![W]\!] = \mathbb{T}[\![W]\!]$;
- $\neg P[\![W]\!] \equiv \mathbb{T}[\![W]\!] \setminus P[\![W]\!]$;
- $P[\![Z]\!] \wedge Q[\![W]\!] \equiv (P[\![Z]\!] \times Q[\![W \setminus Z]\!]) \cap Q[\![W]\!]$;
- $P[\![Z]\!] \vee Q[\![W]\!] \equiv (P[\![Z]\!] \times Q[\![W \setminus Z]\!]) \cup Q[\![W]\!]$;
- $P[\![W]\!] \Rightarrow Q[\![W]\!] \equiv P[\![W]\!] \subseteq Q[\![W]\!]$.

Definition 3. *Given an interface automaton A, an* input semantics $I_a = (P_a, B_a, Q_a, E_a)$ *of an action $a \in \Sigma_A^{\mathrm{Im}}$ is defined by: a precondition $P_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a)]\!]$, a specification $S_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\mathrm{o}}(a)]\!]$, a termination postcondition $Q_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\mathrm{o}}(a)]\!]$, and an exception postcondition $E_a \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(a) \cup \Lambda_A(a) \cup \Psi_A^{\mathrm{o}}(a)]\!]$.*

An output semantics $O_b = (P_b, Q_b)$ *of an action $b \in \Sigma_A^{\mathrm{Om}}$ is defined by: a precondition $P_b \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(b)]\!]$ and a postcondition $Q_b \subseteq \mathbb{T}[\![\Psi_A^{\mathrm{i}}(b) \cup \Psi_A^{\mathrm{o}}(b)]\!]$. These conditions are denoted resp. by $I_a.P$, $I_a.S$, $I_a.Q$, $I_a.E$, $O_b.P$, and $O_b.Q$.*

We consider only the semantics of observable method actions ($a \in \Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Om}}$). We omit the semantics of private method actions ($a \in \Sigma_A^{\mathrm{Hm}}$) because they are not relevant for interoperability. We define behavioral contracts as follows.

Definition 4. *A* behavioral contract *B of a component is a tuple $(\mathcal{A}, \mathcal{I}, \mathcal{O})$ such that $\mathcal{A}$ is an interface automaton, $\mathcal{I}$ is a map associating each $a \in \Sigma_A^{\mathrm{Im}}$ to an input semantics I_a, and $\mathcal{O}$ is a map associating each $a \in \Sigma_A^{\mathrm{Om}}$ to an output semantics O_a. We denote by, $B.\mathcal{A}$, the interface automaton of B, $B.\mathcal{I}$, the map $\mathcal{I}$ of B, and $B.\mathcal{O}$, the map $\mathcal{O}$ of B.*

The following definition establishes the different relations between the specification and the pre/postconditions of an input method action $a \in \Sigma_A^{\mathrm{Im}}$.

DEFINITION 5. *Given a behavioral contract B and an action $a \in \Sigma_A^{\mathrm{Im}}$ where $B.\mathcal{A} = A$ and $B.\mathcal{I}(a) = (P_a, B_a, Q_a, E_a)$, for all $(i, f, o) \in \Psi_A^{\mathrm{i}}(a) \times \Lambda_A(a) \times \Psi_A^{\mathrm{o}}(a)$,*

- *a is* correct *with respect to $B.\mathcal{I}(a)$ iff $P_a[i] \wedge S_a[i, f, o] \Rightarrow Q_a[i, f, o]$;*
- *a* terminates *with respect to $B.\mathcal{I}(a)$ iff $P_a[i] \wedge S_a[i, f, o] \Rightarrow Q_a[i, f, o] \wedge \neg E_a[i, f, o]$;*
- *a* throws exceptions *with respect to $B.\mathcal{I}(a)$ iff $P_a[i] \wedge S_a[i, f, o] \Rightarrow E_a[i, f, o]$;*

The stated conditions are based on the Hoare triplet [15]: a provided method is correct if its behavior under the precondition ensures the postcondition; it terminates if it is correct and the exception postcondition is not satisfied, and throws exceptions if the exception postcondition is satisfied.

3.3 Design of the railway case study

The UML-like component architecture in Figure 2 presents the different ATP equipments mentioned in Section 2. We count four component classes: OnBoardDevice, DataExchangeUnit, MovementControlUnit, and SubRouteBuilder instantiated resp. by the components OBD, DEU, MCU, and SRB. The last three ones implement resp. the interfaces DataExchange, MovementControl, and RouteBuilder.

The component DEU implements the public (+) method *covReq* (coverage request), whose arguments are: *tel* and *ts*, resp. the coordinate of TEL, sent by OBD, and the identifier of the segment containing TEL, *hel* and *hs*, resp. the coordinate of HEL and the identifier of the segment containing HEL, and t, the train identifier. According to the interface automaton A_d of DEU (cf. Figure 3(b)), the method *covReq* transfers the coverage request to MCU by invoking the method *isCovered*. MCU responds OBD, via DEU, by returning 2 (resp. 1) if it covers completely (resp. partially) the train (signal *covered*), or by throwing *uncovered* if not.

Subsequently, if the train is covered by MCU, OBD requests its VMAZ (*vmazReq*). DEU transfers the request by calling *computeVmaz* implemented by MCU. In turn, MCU calls the method *chain* of SRB to perform chaining on segments in order to compute the VMAZ bounds within the sequence of segments from *start* to *end*, the arguments of *chain*. If MCU covers only *hel*, the argument *start* is set to the first segment in the trains path fully covered by MCU. Otherwise, it is set to *ts*. The argument *end* is always set to the last segment fully covered by MCU. According to A_m in Figure 3(c), if chaining is interrupted by an uncontrolled switch, MCU handles the exception *uncontSW* expected to be thrown by *chain* and in turn, throws *default*.

Based on the path database *bdd*, SRB returns VMAZ segments in the table *segs* of size *max* the maximum number of segments covered by MCU. The field *useful_nb* $\leq$ *max* indicates the number of segments included in VMAZ. MCU computes accordingly the VMAZ bounds coordinates on the path frame based on informations of useful segments (identifiers, beginning coordinates, and lengths saved in data structures of type *Seg*). In the case where MCU covers only a part of the train VMAZ, it returns a pair *vmaz* of coordinates where one of them is null and the other is a positive real. Otherwise, the two coordinates are positive reals. The map attribute *cst* (covered segments and trains) is finally updated such that segments covered both by MCU and VMAZ of the train are associated to its identifier.

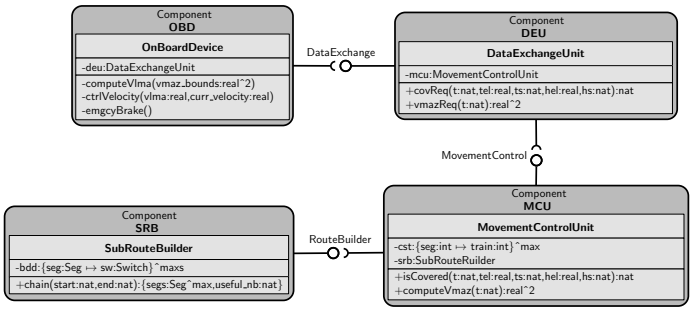
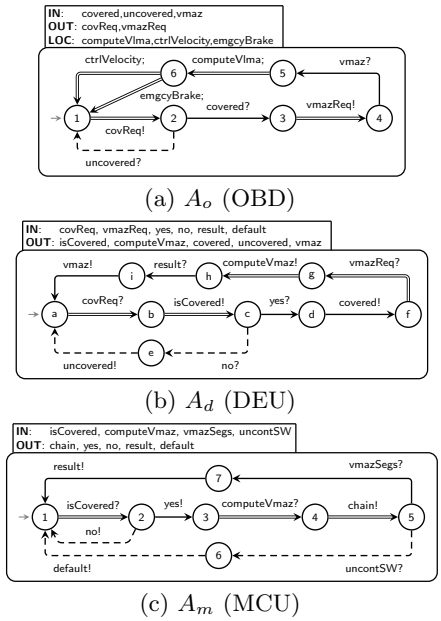


Figure 2: UML-like component architecture.



(a) A_o (OBD)

(b) A_d (DEU)

(c) A_m (MCU)

Figure 3: Interface automata of OBD, DEU, and MCU: method actions (double transitions); return actions (simple transitions); exception actions (dashed transitions).

Finally, according to A_o in Figure 3(a), OBD fixes VLMA by calling its private (−) method *computeVlma* before the final bound of VMAZ. Consequently, it controls the train speed if HEL is sufficiently far from VLMA (*ctrlVelocity*), or performs an emergency brake (*emgcyBrake*) otherwise.

Let us consider three behavioral contracts B_o, B_d, and B_m resp. for components OBD, DEU, and MCU where $B_o.\mathcal{A}$ is A_o, $B_d.\mathcal{A}$ is A_d, and $B_m.\mathcal{A}$ is A_m. Table 1 shows an example of the method *covReq* semantics in B_o and B_d whose signature is *covReq* $(t, tel, ts, hel, hs) \rightarrow$ *covered* $\sharp$ *uncovered* (parameter types are given in Figure 2). The semantics of *covReq* in B_o and B_d states that the minimal and maximal identifiers t of trains, are resp. 0 and 30, and those of segment identifiers (*ts* and *hs*), are resp. 0 and 500. The precondition of *covReq* in B_o states that the conditions $tel, hel \in [0, 5000]$ and $tel < hel$ have to be satisfied by calling the method, where 5000um (unit of measurement) is the size of the longest trains path. In B_d, the precondition states simply that $tel, hel \in [0, 5000]$. In B_o, the postcondition of *covReq* states that the return parameter *covered* is a sig-

Table 1: Semantics of the method action *covReq*.

Output semantics $B_o.\mathcal{O}$	Input semantics $B_d.\mathcal{I}$
$B_o.\mathcal{O}(covReq).P \equiv t \in \{0,...,30\} \wedge ts, hs \in \{0,...,500\} \wedge tel, hel \in [0,5000] \wedge tel < hel$	$B_d.\mathcal{I}(covReq).P \equiv t \in \{0,...,30\} \wedge ts, hs \in \{0,...,500\} \wedge tel, hel \in [0,5000]$
	$B_d.\mathcal{I}(covReq).S \equiv \perp[t, tel, hel, ts, hs, covered]$
$B_o.\mathcal{O}(covReq).Q \equiv covered \in \{0,1,2\}$	$B_d.\mathcal{I}(covReq).Q \equiv covered \in \{1,2\}$
	$B_d.\mathcal{I}(covReq).E \equiv covered = 0$

nal in $\{0,1,2\}$. However, in B_d, it states only that *covered* is a signal in $\{1,2\}$ because if it is equal to 0, the exception *uncovered* is thrown. The specification $B_d.\mathcal{I}(covReq).S$ is not defined ($\perp[t, tel, hel, ts, hs, covered]$): at the level of B_d, there is no parameter or attribute ($\Lambda_{A_d}(covReq) = \emptyset$) describing how the return parameter *covered* is computed. MCU, after receiving the coverage request from OBD, is expected to ask SRB to check in *bdd* whether *tel* and *hel* are really placed resp. on *ts* and *hs*, as claimed by OBD. This functionality of SRB does not appear intentionally at this stage. We expect using this detail to justify our refinement approach presented in Section 5.

Finally, by considering that $\mathsf{R}_{A_o}(covReq)$ is $covered \in \Sigma^{\mathrm{r}}_{A_o}$, $\mathsf{R}_{A_o}(vmazReq)$ is $vmaz \in \Sigma^{\mathrm{r}}_{A_o}$, and $\mathsf{E}_{A_o}(covReq)$ is $uncovered \in \Sigma^{\mathrm{e}}_{A_o}$, we can deduce that A_o is well-formed. The reader can easily deduce the well-formedness of A_d and A_m by finding their method, return and exception actions.

4. COMPONENTS COMPOSITION

The composition of two behavioral contracts may induce deadlock situations caused by potential semantic or protocol incompatibilities. At the protocol level, the composition of two interface automata may contain *deadlock* states. From that states, one of the two interface automata requests an input not accepted by the other. For example, a component calls a method throwing exceptions without handling them.

At the semantic level, the synchronization of shared input/output method actions with incompatible semantics, leads to deadlock states. A component outputting a method call have more informations about its arguments. Thus, the call precondition is stronger than that of the method implementation: the environment is expected to provide input arguments included in the implementation precondition. In return, the component providing the method communicates to the environment a postcondition on its return parameter: it vouches to provide only return values that satisfy the postcondition. The calling component cannot have more detailed informations about the return parameter than the implementing one. That's why the postcondition of a method invocation is weaker than that provided by its implementation. Note that preconditions, like postconditions, of provided observable methods are required to be satisfiable. Not all calling environments satisfy the precondition, or expect return guarantees larger than the postcondition [5]. In this case, synchronization disparities are detected.

4.1 Synchronization of interface automata and semantic compatibility

The synchronization of two interface automata A_1 and A_2 is possible only if they are mutually composable *i.e.* $\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{I}}_{A_2} = \Sigma^{\mathrm{O}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2} = \Sigma^{\mathrm{H}}_{A_1} \cap \Sigma_{A_2} = \Sigma^{\mathrm{H}}_{A_2} \cap \Sigma_{A_1} = \emptyset$. The set of shared input/output actions in A_1 and A_2 is $Shared(A_1, A_2) = (\Sigma^{\mathrm{I}}_{A_1} \cap \Sigma^{\mathrm{O}}_{A_2}) \cup (\Sigma^{\mathrm{I}}_{A_2} \cap \Sigma^{\mathrm{O}}_{A_1})$. For simplicity, we denote the couple of states (s_1, s_2) by $s_1 s_2$. By synchronizing A_1 and A_2, transitions labeled by shared actions synchronize and the others are interleaved. The *synchronized product* $A_1 \otimes A_2$ of A_1 and A_2 is an interface automaton where $\Upsilon_{A_1 \otimes A_2} = \Upsilon_{A_1} \times \Upsilon_{A_2}$, $\imath_{A_1 \otimes A_2} = \imath_{A_1} \imath_{A_2}$, $\Sigma^{\mathrm{I}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{I}}_{A_1} \cup \Sigma^{\mathrm{I}}_{A_2}) \setminus Shared(A_1, A_2)$, $\Sigma^{\mathrm{O}}_{A_1 \otimes A_2} = (\Sigma^{\mathrm{O}}_{A_1} \cup \Sigma^{\mathrm{O}}_{A_2}) \setminus Shared(A_1, A_2)$, $\Sigma^{\mathrm{H}}_{A_1 \otimes A_2} = \Sigma^{\mathrm{H}}_{A_1} \cup \Sigma^{\mathrm{H}}_{A_2} \cup Shared(A_1, A_2)$, and $(s_1 s_2, a, s'_1 s'_2) \in \delta_{A_1 \otimes A_2}$ iff :

- $a \in Shared(A_1, A_2) \wedge (s_1, a, s'_1) \in \delta_{A_1} \wedge (s_2, a, s'_2) \in \delta_{A_2}$;
- $a \notin Shared(A_1, A_2) \wedge$ $((s_1, a, s'_1) \in \delta_{A_1} \wedge s_2 = s'_2 \vee (s_2, a, s'_2) \in \delta_{A_2} \wedge s_1 = s'_1)$.

Given tow behavioral contracts B_1 and B_2 where $B_1.\mathcal{A} = A_1$ and $B_2.\mathcal{A} = A_2$, B_1 and B_2 are *composable* if A_1 and A_2 are composable, and each $a \in Shared(A_1, A_2) \cap \Sigma^{\mathrm{m}}_{A_1}$ has the same signature in A_1 and A_2. We deduce, from the composability of B_1 and B_2, that for each $a \in \Sigma^{\mathrm{m}}_{A_i} \cap Shared(A_1, A_2)$ for $i \in \{1,2\}$, if $\mathsf{R}_{A_i}(a), \mathsf{E}_{A_i}(a) \in \Sigma_{A_i}$ for all $i \in \{1,2\}$, then $\mathsf{R}_{A_1}(a) = \mathsf{R}_{A_2}(a) = r_a$, $\mathsf{E}_{A_1}(a) = \mathsf{E}_{A_2}(a) = e_a$ and $r_a, e_a \in Shared(A_1, A_2)$. In the following definition, we provide the semantic compatibility conditions of input/output method actions shared between A_1 and A_2.

Definition 6. *Given an action $a \in Shared(A_1, A_2) \cap \Sigma^{\mathrm{m}}_{A_1}$, for all $(i, o) \in \Psi^{\mathrm{i}}_{A_1}(a) \times \Psi^{\mathrm{o}}_{A_1}(a)$, if one of the following conditions holds, then the action a in B_1 is* semantically compatible *with a in B_2 i.e. $SemComp_a(B_1, B_2)$:*

- $B_1.\mathcal{O}(a).P[i] \Rightarrow B_2.\mathcal{I}(a).P[i] \wedge$ $B_1.\mathcal{O}(a).Q[i,o] \Leftarrow B_2.\mathcal{I}(a).Q[i,o]$, *if* $a \in \Sigma^{\mathrm{Om}}_{A_1}$;
- $B_1.\mathcal{I}(a).P[i] \Leftarrow B_2.\mathcal{O}(a).P[i] \wedge$ $B_1.\mathcal{I}(a).Q[i,o] \Rightarrow B_2.\mathcal{O}(a).Q[i,o]$, *if* $a \in \Sigma^{\mathrm{Im}}_{A_1}$.

Example 1. According to our case study (cf. Section 3.3), $B_o.\mathcal{A}$ and $B_d.\mathcal{A}$ are composable. The set $Shared(A_o, A_d)$ is defined by $\{covReq, vmazReq, covered, uncovered, vmaz\}$. Based on Table 1, $SemComp_a(B_o, B_d)$ is true for $a = covReq$.

Assume that B_1 and B_2 are composable, we define by $B_1|B_2$, the *synchronized* behavioral contract of B_1 and B_2 where $(B_1|B_2).\mathcal{A}$ is $A_1 \otimes A_2$ restricted to the set of reachable states, $(B_1|B_2).\mathcal{I} = B_i.\mathcal{I}(a)$ for all $a \in \Sigma^{\mathrm{Im}}_{A_i} \setminus Shared(A_1, A_2)$, and $(B_1|B_2).\mathcal{O} = B_i.\mathcal{O}(a)$ for all $a \in \Sigma^{\mathrm{Om}}_{A_i} \setminus Shared(A_1, A_2)$ with $i \in \{1,2\}$. We set $(B_1|B_2).\mathcal{A} = A_{12}$ for simplicity.

Deadlock states in A_{12} represent possible deadlocks during the communication between the components specified by B_1 and B_2 at the protocol and semantic levels. They are states $s_1 s_2$ such that (i) there exists at least $a \in Shared(A_1, A_2)$ enabled from s_1 and not from s_2 or inversely, or (ii) a is a method action enabled from s_1 and s_2 but, the condition $SemComp_a(B_1, B_2)$ is falsified.

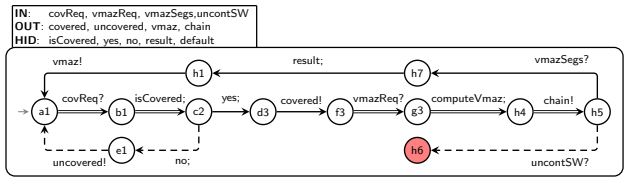


Figure 4: Interface automaton $(B_d|B_m).\mathcal{A}$.

DEFINITION 7. *The set of* deadlock states $Dead(A_1, A_2)$ *in* A_{12} *is* $\{s_1s_2 \in \Upsilon_{A_{12}} \mid (\exists\ a \in Shared(A_1, A_2).\ D_1(s_1s_2) \vee D_2(s_1s_2))\}$ *where*

- $D_1(s_1s_2) \equiv (a \in \Sigma^{\mathrm{O}}_{A_1}(s_1) \wedge a \notin \Sigma^{\mathrm{I}}_{A_2}(s_2)) \vee (a \in \Sigma^{\mathrm{Om}}_{A_1}(s_1) \wedge a \in \Sigma^{\mathrm{Im}}_{A_2}(s_2) \wedge \neg SemComp_a(B_1, B_2))$;
- $D_2(s_1s_2) \equiv (a \in \Sigma^{\mathrm{O}}_{A_2}(s_2) \wedge a \notin \Sigma^{\mathrm{I}}_{A_1}(s_1)) \vee (a \in \Sigma^{\mathrm{Om}}_{A_2}(s_2) \wedge a \in \Sigma^{\mathrm{Im}}_{A_1}(s_1) \wedge \neg SemComp_a(B_1, B_2))$.

Example 2. According to Figure 3, the interface automata A_d and A_m are composable. Let us consider two composable behavioral contracts B_d and B_m where $B_d.\mathcal{A} = A_d$ and $B_m.\mathcal{A} = A_m$. By supposing that actions *isCovered* and *computeVmaz* are semantically compatible between B_d and B_m, the state $h6$ is the only deadlock state in $(B_d|B_m).\mathcal{A}$: the exception action $default \in \Sigma^{\mathrm{e}}_{A_m}(6) \cap Shared(A_d, A_m)$ is not enabled from the state h in A_d (cf. Figure 4).

4.2 Optimistic approach of composition

The incremental bottom-up design means that the compatibility checking between components can be performed for partial descriptions of the system. The optimistic approach of interface automata composition is closely consistent with the incremental design oncoming.

In this approach, the presence of deadlock states in A_{12} doesn't imply necessarily the incompatibility of B_1 and B_2: the existence of a suitable environment E where $E.\mathcal{A}$ provides good input steps and semantics for A_{12} and prevents reaching deadlock states, implies that they are *compatible*. E must satisfy the following conditions: (1) E and $B_1|B_2$ are composable, (2) $E.\mathcal{A}$ is non-empty interface automaton, (3) $Dead(A_{12}, E.\mathcal{A}) = \emptyset$, and (4) no state in the set $Dead(A_1, A_2) \times \Upsilon_{E.\mathcal{A}}$ is reachable in $((B_1|B_2)|E).\mathcal{A}$ [13].

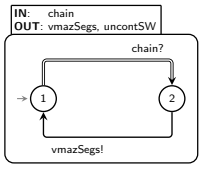


Figure 5: Interface automaton A_s of SRB.

Example 3. We assume that SRB does not throw the exception *uncontSW* if an uncontrolled switch is detected during chaining. The train VAMZ is limited by the switch position: for example, in Fig 1, if p_1 is uncontrolled, VMAZ of T1 is bounded by the end of segment s_2. Let us consider a behavioral contract B_s for SRB composable with $B_d|B_m$ where $B_s.\mathcal{A} = A_s$ (cf. Figure 5) and $SemComp_a(B_d|B_m, B_s)$ is valid for $a = chain$. B_s is a suitable environment for $B_d|B_m$. In $((B_d|B_m)|B_s).\mathcal{A}$, the states $h61$ and $h62$ are not reachable because from the state 2 in A_s the action *uncontSW* is not enabled. Consequently, B_d and B_m are compatible behavioral contracts.

In the product A_{12}, all states s_1s_2 from which deadlock states are autonomously reachable, are considered as incompatible and must be removed from A_{12}. No environment can prevent reaching deadlocks from those states as explained in Section 3.1. A state $s_1s_2 \in \Upsilon_{A_{12}}$ is *compatible* in A_{12} if there is no state $s'_1s'_2 \in Dead(A_1, A_2)$ autonomously reachable from s_1s_2. We denote, by $Cmp(A_1, A_2)$, the set of compatible states in A_{12}. B_1 and B_2 are *compatible* iff they are composable and $\imath_{A_{12}} \in Cmp(A_1, A_2)$. The interface automaton of the composition of two behavioral contracts is restricted to the set of compatible states of their synchronized product.

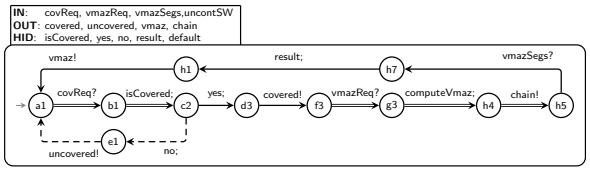


Figure 6: Interface automaton $(B_d\|B_m).\mathcal{A}$.

DEFINITION 8. *The* composition $B_1\|B_2$ *of* B_1 *and* B_2 *is a behavioral contract such that* $(B_1\|B_2).\mathcal{I} = (B_1|B_2).\mathcal{I}$, $(B_1\|B_2).\mathcal{O} = (B_1|B_2).\mathcal{O}$, *and* $(B_1\|B_2).\mathcal{A}$ *is an interface automaton where* $\Upsilon_{(B_1\|B_2).\mathcal{A}}$ *is restricted to* $Cmp(A_1, A_2)$, $\imath_{(B_1\|B_2).\mathcal{A}} = \imath_{A_{12}}$, $\Sigma^*_{(B_1\|B_2).\mathcal{A}} = \Sigma^*_{A_{12}}$ *for* $* \in \{\mathrm{I}, \mathrm{O}, \mathrm{H}\}$, *and* $\delta_{(B_1\|B_2).\mathcal{A}} = \{(s, a, s') \in \delta_{A_{12}} \mid s, s' \in Cmp(A_1, A_2)\}$.

Example 4. The interface automaton $(B_d\|B_m).\mathcal{A}$ (cf. Figure 6) is the restriction of $(B_d|B_m).\mathcal{A}$ to the set of compatible states $\Upsilon_{(B_d|B_m).\mathcal{A}} \setminus \{h6\}$. Assume that A_d is not well-formed and do not expect to assign the return value of *computeVmaz*, $h7$ is a deadlock state in $(B_d|B_m).\mathcal{A}$. In this case, states $h5$, $h4$, and $g3$ are incompatible (the path between $g3$ and $h7$ is autonomous). The call of *vmazReq* leads inevitably to a deadlock for all possible environments.

The proofs of the claimed theorems in the rest of the paper are detailed in [22]. The following theorem states the preservation of interface automata well-formedness by composition of behavioral contracts.

THEOREM 1. *If* A_1 *and* A_2 *are well-formed and* B_1 *is compatible with* B_2, *then* $(B_1\|B_2).\mathcal{A}$ *is well-formed.*

The following theorem is in the heart of incremental design of component-based systems. It is a straightforward generalization of interface automata associativity [13] to behavioral contracts.

THEOREM 2. *The composition operation* $\|$ *between compatible behavioral contracts is commutative and associative.*

The compatibility check procedure of two behavioral contracts is similar to that described in [13] for interface automata, by considering the semantic layer of actions and the new definition of autonomous runs. The linear complexity of the proposed algorithm is extended by the satisfiability decision problems of the semantic compatibility conditions of shared method actions [5].

5. REFINEMENT

Refinement embodies with more details an abstract specification of a component in a more concrete one. It guarantees a safe substitutability of an abstract version of a component by a refined one. We propose a refinement approach for behavioral contracts at the protocol and semantic levels suitable to the object-oriented context. We start by introducing refinement at the level of interface automata.

5.1 Expanding simulation

The original refinement approach of interface automata is *contravariant* [13]: a refined version of a component must accept the same or more inputs and provide the same or fewer outputs, than the abstraction. It is based on an alternating simulation relation [7]: an interface automaton A' refines an interface automaton A if each input event of A can be simulated by A', and each output event of A' can be simulated by A. At the protocol level in OOCBD, refinement ensures that a refined specification of component (i) may contain more details about the common provided methods with the abstract one, which are output and hidden method calls encapsulated in their implementations, and (ii) may provide more methods than the abstract one. In order to satisfy the previous requirements, we define refinement as a *covariant expanding simulation* relation between interface automata: A' refines A if A' accepts (resp. issues) more inputs (resp. outputs) than A, and each input, output, or local event of A is simulated in A' by the same one followed or preceded by other events. To formalize this relation, we define the *closure* set $Clos_A(s, \Sigma)$ of $s \in \Upsilon_A$ under actions in $\Sigma \subseteq \Sigma_A$ by the largest set $\Upsilon \subseteq \Upsilon_A$ such that $s \in \Upsilon$ and if $t \in \Upsilon$, $t' = Succ_A(t, a)$, and $a \in \Sigma$, then $t' \in \Upsilon$ *i.e.* $Clos_A(s, \Sigma)$ contains states reachable from the state s by enabling actions of Σ.

Definition 9. *Given two interface automata A and A', a binary relation $\gtrsim \subseteq \Upsilon_A \times \Upsilon_{A'}$ is an* expanding simulation *from A to A' iff for all states $s \in \Upsilon_A$ and $s' \in \Upsilon_{A'}$ such that $s \gtrsim s'$, for all $a \in \Sigma_A(s)$ and $t = Succ_A(s, a)$, the following conditions hold:*

(1) if $a \in \Sigma_A^{\mathrm{Om}}(s) \cup \Sigma_A^{\mathrm{Ir}}(s) \cup \Sigma_A^{\mathrm{Ie}}(s)$, then $a \in \Sigma_{A'}(s')$ and $t \gtrsim t'$ for $t' = Succ_{A'}(s', a)$;

(2) if $a \in \Sigma_A^{\mathrm{Im}}(s) \cup \Sigma_A^{\mathrm{Hm}}(s)$, then $a \in \Sigma_{A'}(s')$, and there is a subset $\Sigma \subseteq ((\Sigma_{A'}^{\mathrm{aut}} \setminus \Sigma_{A'}^{\mathrm{Or}}) \setminus \Sigma_{A'}^{\mathrm{e}}) \setminus \Sigma_A$ and a state $t' \in Clos_{A'}(Succ_A(s', a), \Sigma)$ such that $t \gtrsim t'$;

(3) if $a \in \Sigma_A^{\mathrm{Or}}(s) \cup \Sigma_A^{\mathrm{Hr}}(s)$, then there is $v' \in Clos_{A'}(s', \Sigma)$ such that $\Sigma = ((\Sigma_{A'}^{\mathrm{aut}} \setminus \Sigma_{A'}^{\mathrm{Or}}) \setminus \Sigma_{A'}^{\mathrm{e}}) \setminus \Sigma_A$, $a \in \Sigma_{A'}(v')$, and $t \gtrsim t'$ for $t' = Succ_{A'}(v', a)$;

(4) if $a \in \Sigma_A^{\mathrm{Oe}}(s) \cup \Sigma_A^{\mathrm{He}}(s)$, then there is $v' \in Clos_{A'}(s', \Sigma)$ where $\Sigma = ((\Sigma_{A'}^{\mathrm{aut}} \setminus (\Sigma_{A'}^{\mathrm{Oe}} \cup \Sigma_{A'}^{\mathrm{Or}})) \cup \Sigma_{A'}^{\mathrm{Ie}}) \setminus \Sigma_A$, $a \in \Sigma_{A'}(v')$, and $t \gtrsim t'$ for $t' = Succ_{A'}(v', a)$.

Our expanding simulation relation pinpoints where refinement details are added in the abstract version of an interface automaton. Condition (1) of Definition 9 states that every transition labeled by an output method action, or an input return or exception action must be matched by a transition labeled by the same action in A'. Method calls sent to the environment, the reception of their return values, and catching their thrown exceptions, cannot be refined.

Condition (2) states that every transition labeled by an input or hidden method action in A is matched in A' by a transition labeled by the same action followed by zero or more transitions labeled by a "subset" of new autonomous non-exception actions in $((\Sigma_{A'}^{\mathrm{aut}} \setminus \Sigma_{A'}^{\mathrm{Or}}) \setminus \Sigma_{A'}^{\mathrm{e}}) \setminus \Sigma_A$. A provided public method in the abstraction of a component can be refined by adding to its body new private or public method calls. In addition, since providing private methods is not specified by actions in interface automata (cf. Section 3), our simulation relation allows adding refinement details about private methods after their calls.

Condition (3) states that every transition labeled by an output or hidden return action a in A is matched in A' by zero or more transitions labeled by new autonomous non-exception actions in $((\Sigma_{A'}^{\mathrm{aut}} \setminus \Sigma_{A'}^{\mathrm{Or}}) \setminus \Sigma_{A'}^{\mathrm{e}}) \setminus \Sigma_A$ followed by a transition labeled by a. The return event of a private or public provided method in the abstraction is computed based on the return values of new calls of private or public methods added as refinement details.

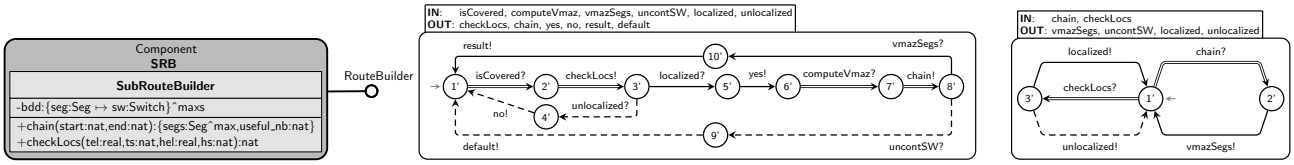
Condition (4) states that every transition labeled by an output or hidden exception action a in A is matched in A' by zero or more transitions labeled either by new autonomous and hidden exception actions in $(\Sigma_{A'}^{\mathrm{aut}} \setminus (\Sigma_{A'}^{\mathrm{Oe}} \cup \Sigma_{A'}^{\mathrm{Or}})) \setminus \Sigma_A$, or by new input exception actions in $\Sigma_{A'}^{\mathrm{Ie}} \setminus \Sigma_A$, followed by a transition labeled by a. The exception events of a provided private or public method in the abstraction is the propagation of catching exception events of new calls of private or public methods added as refinement details. From the previous definition, we establish refinement as follows.

Definition 10. *A' refines A ($A \succeq A'$) iff (1) $\Sigma_A^{\mathrm{I}} \subseteq \Sigma_{A'}^{\mathrm{I}}$, $\Sigma_A^{\mathrm{O}} \subseteq \Sigma_{A'}^{\mathrm{O}}$, $\Sigma_A^{\mathrm{H}} \subseteq \Sigma_{A'}^{\mathrm{H}}$, and (2) there is an expanding simulation $\gtrsim$ from A to A' such that $\imath_A \gtrsim \imath_{A'}$.*

A trivial consequence of condition (1) of Definition 10 is covariance from A to A' on method, return, and exception actions: $\Sigma_A^{\mathrm{m}} \subseteq \Sigma_{A'}^{\mathrm{m}}$, $\Sigma_A^{\mathrm{r}} \subseteq \Sigma_{A'}^{\mathrm{r}}$, and $\Sigma_A^{\mathrm{e}} \subseteq \Sigma_{A'}^{\mathrm{e}}$. Condition (2) requires the existence of an expanding simulation from A to A' relating their initial states $\imath_A$ and $\imath_{A'}$ and recursively propagated to their successor states.

We infer from conditions of Definition 9 that extra new input method actions are not considered as refinement details by the expanding simulation relation, which obviously makes sense. By cons, it allows the extension of interface automata by adding protocols related to additional methods provided by a component extended interface. They can be enabled for example separately from the initial state.

Example 5. After receiving a train coverage request, MCU asks SRB to check if *tel* and *hel* are really on segments *ts* and *hs* respectively by calling the method *checkLocs*, presented in Figure 7(left), as a new service of the class SubRouteBuilder and the interface RouteBuilder. If true, SRB responds by sending the status (*localized*), and MCU in turn, responds OBD, via DEU, by returning *yes* if the train is completely (or partially) included in its coverage area. Otherwise, SRB throws the exception *unlocalized* to MCU, which in turn, propagates it to DEU by throwing the exception *no*. In A'_m shown in Figure 7(a), the method call *checkLocs*! is encapsulated in the runs describing the body of the method *isCovered* provided by MCU. Providing the public method *checkLocs* is equally depicted in the interface automaton A'_s shown in Figure 7(b) by a new input method action enabled separately from $\imath_{A'_s} = 1'$. A'_m and



(a) A'_m (MCU) (b) A'_s (SRB)

Figure 7: Extended class SubRouteBuilder (left); refined interface automata of MCU and SRB (right).

A'_s resp. refine A_m and A_s (shown resp. in Figure 3(c) and Figure 5): condition (1) of Definition 10 is met by A'_m and A_m, as well by A'_s and A_s, and there are two expanding simulations $\gtrsim_m = \{11', 23', 36', 47', 58', 69', 7(10')\}$ from A_m to A'_m with $\imath_{A_m} \gtrsim_m \imath_{A'_m}$ and $\gtrsim_s = \{11', 22'\}$ from A_s to A'_s with $\imath_{A_s} \gtrsim_s \imath_{A'_s}$.

5.2 Semantic substitutability

The semantic substitutability of method actions between an abstract and a concrete versions of a component behavioral contract is based on *behavioral sub-typing principles* introduced in [8, 20]: in the refined specification, a common provided method must have a weaker precondition, a stronger termination postcondition, and does not introduce exceptions by supplying a stronger exception condition, than the abstraction. Inversely, a common method call must have a stronger precondition and a weaker postcondition than the abstraction. Given tow behavioral contracts B and B', we denote $B.\mathcal{A}$ by A and $B'.\mathcal{A}$ by A'.

Definition 11. *Given an action* $a \in \Sigma_A^{\mathrm{Im}}$, $B'.\mathcal{I}(a) = (P'_a, B'_a, Q'_a, E'_a)$ substitutes $B.\mathcal{I}(a) = (P_a, B_a, Q_a, E_a)$ *i.e.* $SemSub_a(B, B')$, *iff for all* $(i, f, o) \in \Psi_A^{\mathrm{i}}(a) \times \Lambda_A(a) \times \Psi_A^{\mathrm{o}}(a)$, *the following conditions hold:*

(1) $P_a[i] \Rightarrow P'_a[i]$, $R[i, f, o] \Leftarrow R'[i, f, o]$ *for* $R \in \{Q_a, E_a\}$;

(2) $P_a[i] \wedge S'_a[i, f, o] \Rightarrow S_a[i, f, o]$.

Given an action $b \in \Sigma_A^{\mathrm{Om}}$, $B'.\mathcal{O}(b) = (P'_b, Q'_b)$ substitutes $B.\mathcal{O}(b) = (P_b, Q_b)$ *i.e.* $SemSub_b(B, B')$, *iff for all* $(i, o) \in \Psi_A^{\mathrm{i}}(b) \times \Psi_A^{\mathrm{o}}(b)$, *the following condition holds:*

(3) $P_b[i] \Leftarrow P'_b[i]$ *and* $Q_b[i, o] \Rightarrow Q'_b[i, o]$.

Property 1. *Given* $a \in \Sigma_A^{\mathrm{Im}}$ *where* $SemSub_a(B, B')$, *for all* $(i, f, o) \in \Psi_A^{\mathrm{i}}(a) \times \Lambda_A(a) \times \Psi_A^{\mathrm{o}}(a)$,

- $P_a[i] \wedge S'_a[i, f, o] \Rightarrow Q_a[i, f, o]$,
- $P_a[i] \wedge S'_a[i, f, o] \Rightarrow Q_a[i, f, o] \wedge \neg E_a[i, f, o]$, *or*
- $P_a[i] \wedge S'_a[i, f, o] \Rightarrow E_a[i, f, o]$

resp. iff the action a is correct, terminates, or throws exceptions with respect to $B.\mathcal{I}(a)$.

The previous property is evident based on definitions 5 and 11. The correctness, termination and exception preservation is what we expect for a correct refinement at the level of provided methods semantics: if a refined semantics of a provided method satisfies the condition (2) of Definition 11, then any property holding for a specification S under the precondition in the abstract method semantics, holds also for the refined specification S' under the same precondition, and thus S' may be used instead of S [21, 3].

Property 2 says that the semantic compatibility validity of shared observable method actions, between a component behavioral contracts and its environment, is preserved by the semantic substitutability. The property is obvious based on Definition 6 and conditions (1) and (3) of Definition 11. Given a behavioral contract E, we set $E.\mathcal{A} = A_E$.

Property 2. *Given an action* $a \in Shared(A, A_E) \cap \Sigma_A^{\mathrm{m}}$, *for all* $(i, o) \in \Psi_A^{\mathrm{i}}(a) \times \Psi_A^{\mathrm{o}}(a)$, *if* $SemSub_a(B, B')$, *then* $SemComp_a(B, E) = SemComp_a(B', E)$.

Finally, we can define refinement of behavioral contracts based on refinement of interface automata and the semantic substitutability of observable method actions.

Definition 12. B' refines B ($B \sqsupseteq B'$) *iff* $A \succeq A'$ *and for all* $a \in \Sigma_A^{\mathrm{Im}} \cup \Sigma_A^{\mathrm{Om}}$, $SemSub_a(B, B')$.

5.3 Refinement properties

In this subsection, we present the properties and requirements under which our refinement approach allows independent implementability of components using their behavioral contracts. We recall that these results are proved in [22].

Reflexivity and transitivity

Theorem 3. *Refinement* $\sqsupseteq$ *between behavioral contracts is a preorder i.e. reflexive and transitive.*

The previous theorem states that a behavioral contract can be gradually refined in several steps while remaining consistent with its abstract specification. It is provable based on the transitivity of the expanding simulation relation *i.e.* given three interface automata A, A', and A'', and two expanding simulations $\gtrsim' \subseteq \Upsilon_A \times \Upsilon_{A'}$ and $\gtrsim'' \subseteq \Upsilon_{A'} \times \Upsilon_{A''}$, then the composite relation $\gtrsim'' \circ \gtrsim' \subseteq \Upsilon_A \times \Upsilon_{A''}$ is an expanding simulation.

Independent implementability

Refinement is expected to allow independent implementability of components: compatible behavioral contracts can be refined separately, while still maintaining compatibility. It lets industrials unrestricted to outsource the implementation of the different components by different suppliers, after the refinement process, even if they do not communicate [5].

Our refinement approach guarantees the consistency between two behavioral contracts B and B' where $B \sqsupseteq B'$ if they are considered "isolated" from their use context. However, it does not prevent the introduction of poorly designed behaviors in their interface automata. Since refinement may issues new outputs, the designer should "safely" define it to preserve compatibility with the environment within the abstraction is incorporated without altering their communication scenarios. For example, according to Definition 2 and

conditions (2) and (3) of Definition 9, the proposed expanding simulation relation preserves well-formedness in refinement only for method actions events common with the abstraction. By cons, it does not guarantee that new method actions events are followed necessarily by their return events. In general, the higher the refinement design respects the environment requirements and well-formedness, the safer the refinement is considered to be.

In the rest of the section, we consider three behavioral contracts B_1, B'_1 and B_2 such that B_1 and B_2 are composable and compatible, B'_1 and B_2 are composable, and $B_1 \sqsupseteq B'_1$. Let $B_1.\mathcal{A} = A_1$, $B'_1.\mathcal{A} = A'_1$, $B_2.\mathcal{A} = A_2$, $(B_1|B_2).\mathcal{A} = A_{12}$, and $(B'_1|B_2).\mathcal{A} = A'_{12}$, we set $EnabRiseDead(A_1, A_2) = \{a \in (\Sigma^{\mathrm{Im}}_{A_{12}} \cup \Sigma^{\mathrm{Ie}}_{A_{12}}) \cap \Sigma_{A_{12}}\langle\sigma\rangle \mid \sigma \in \Theta_{A_{12}}(d_1d_2), d_1d_2 \in Dead(A_1, A_2)\}$: the set of non-autonomous actions enabled by runs $\sigma \in \Theta_{A_{12}}(d_1d_2)$ for all $d_1d_2 \in Dead(A_1, A_2)$. Since B_1 and B_2 are compatible, $EnabRiseDead(A_1, A_2) \neq \emptyset$ if $Dead(A_1, A_2) \neq \emptyset$.

Given $\gtrsim$ an expanding simulation from A_1 to A'_1 such that $\imath_{A_1} \gtrsim \imath_{A'_1}$, B'_1 is a *safe refinement* of B_1 compared to B_2, denoted $B_1 \sqsupseteq^s_{B_2} B'_1$ if the following conditions hold for the interface automata A_1, A'_1, and A_2:

(1) for all deadlock state $d'_1d_2 \in Dead(A'_1, A_2)$, there is a deadlock state $d_1d_2 \in Dead(A_1, A_2)$ such that $d_1 \gtrsim d'_1$ or $d'_1 \in Clos_{A'_1}(c'_1, \Sigma_{A'_1} \setminus \Sigma_{A_1})$ and $d_1 \gtrsim c'_1$, and

(2) $Shared(A'_1, A_2) \cap EnabRiseDead(A_1, A_2) = \emptyset$.

The previous conditions establish requirements whereby B'_1 is considered to be a safe refinement of B_1 compared to B_2. Condition (1) says that A'_{12} does not introduce new deadlocks compared to A_{12} by guaranteeing that all states in $Dead(A'_1, A_2)$ are simulated by states in $Dead(A_1, A_2)$. Condition (2) says that A'_1 does not share non-autonomous actions in $EnabledRiseDead(A_1, A_2)$ with A_2 if they are enabled by the environment in A_{12} may lead inevitably to deadlock states. We claim the following theorem.

THEOREM 4. *If $B_1 \sqsupseteq^s_{B_2} B'_1$, then B'_1 is compatible with B_2 and $B_1\|B_2 \sqsupseteq B'_1\|B_2$.*

Given a fourth behavioral contract B'_2 such that B'_2 is composable with B'_1 and $B_2 \sqsupseteq B'_2$, the independent implementability property of behavioral contracts is established by the following corollary, which is obviously deductible from theorems 3 and 4.

COROLLARY 5. *If $B_1 \sqsupseteq^s_{B_2} B'_1$ and $B_2 \sqsupseteq^s_{B'_1} B'_2$, then B'_1 is compatible with B'_2 and $B_1\|B_2 \sqsupseteq B'_1\|B'_2$.*

Refinement $\succeq$ of two interface automata A and A' is checkable in time $\mathbf{O}((|\delta_A| + |\delta_{A'}|).(|\Upsilon_A| + |\Upsilon_{A'}|))$ [7, 5], where $|S|$ is the cardinality of a set S. The algorithm of checking refinement between interface automata, in our approach, can be deduced naturally form that proposed in [13]. Safe refinement can be checked in linear time by forward or backward traversals, that is $B_1 \sqsupseteq^s_{B_2} B'_1$ can be checked in time $\mathbf{O}(|\delta_{(B_1|B_2).\mathcal{A}}|.|\delta_{(B'_1|B_2).\mathcal{A}}|)$. The previous complexity is extended by the satisfiability decision problems related to the semantic substitutability conditions of common observable method actions between the refinement of a behavioral contract and its abstraction.

6. DISCUSSIONS AND RELATED WORKS

This work is the result of a critical feedback after a considerable experience in the railway industry. Its main motivation is to provide innovative solutions for the development of correct-by-design critical component-based systems by checking rigorously the functional interoperability between software components. Many industrial actors still checking integrity and safety after the development phase by using the V-Model or formal methods. The V-Model and classic testing techniques are out of phase compared to the complexity of contemporary critical systems. Besides, the use of formal methods only for verification, after the design phase, is heavy in general, and model-checkers or proof assistants are not scalable to support large industrial applications in this case. The last European Norm EN 50128:2011 [1] of railway applications, recommends using formal methods, from design to code generation, to distribute the verification complexity throughout the whole development cycle.

Our work is considered as part of this changing context. It is illustrated by a design case study of ATP functions in railway CBTC systems. We introduced behavioral contracts combining protocol and semantic levels of component interface specifications. Our choice of interface automata, to model component protocols, is motivated by their simplicity, and their optimistic approach of composition, which is less restrictive for components manufacturers. The proposed refinement approach is defined from the perspective of OOCBD. It is based on a simulation relation allowing the addition of details about behaviors of common provided services between an abstract and a concrete versions of a component. Our refinement approach gives solutions for substitutability problems which remain almost without accomplished concrete processes in the industry.

We see three main directions for future work. First, scalability of behavioral contracts is possible since their translation to object-oriented languages such as ADA or Java and specification languages such as SPARK [4] or JML [18] is feasible. Second, the formalism can be extended to support non-functional properties such as real-time aspects by using formalisms like *timed interfaces* [6]. Third, the proposed approach can be strengthened by studying the preservation and deduction of safety and liveness properties by component composition and refinement.

In the rest of the paper, we quote some related industrial and academic works. We start by the B Method [3] and its industrial impact. This formal method is among the few ones used to build correct-by-design critical systems by many industrial major actors like RATP, Alstom, Siemens, etc [19]. Its design approach, based on gradual refinements of abstract specifications of system parts until reaching their implementations, responds pertinently to many issues addressed in this paper. Besides, it is worthy to mention also Lurette [17] and its innovative design method closely aligned with the paper targets. It is a design testing tool for synchronous controllable systems. Its principle is to refine an abstract specification of a system by modeling its environment. The tool performs gradually guided random tests of the system reaction under the environment constraints. By observing the relation between the system inputs and outputs, one can decide whether a test succeeds or fails depending on these constraints. This design procedure was among the encouraging reasons for launching ARGOSIM around the tool Stimulus [9] based in part on the Lurette idea.

In [29], the authors produce a formal description of safety communication protocols in train control systems TCS using interface automata and UML sequence diagrams. Deadlocks, live-locks, and some mandatory consistency properties of the proposed case study were checked by SPIN [16]. The presented experimental results show both potential efficiency and practical usefulness of the approach. In [26], the authors propose a controller for the cooperation protocol of the European Train Control System (ETCS) [14]. The informal system specification of ETCS is generic, requiring parameter settings depending on the deployment platform. They identify constraints on these free parameters to ensure collision freedom. They model these constraints in terms of reachability properties of the system hybrid dynamics. Controllability, safety, liveness, and reactivity properties were checked by the deductive theorem prover KeYmaera [25]. The work presented in [11] is an incremental methodology to specify and verify component-based systems using SysML [24] requirement diagrams, and verify their architectural consistency and safety requirements. At each step of the incremental design, they translate an atomic requirement to an LTL property, and check it on a component (specified by a SysML sequence diagram) using SPIN. Next, they check the component compatibility with its environment in the system architecture using interface automata. The work was illustrated on airbag and seat-belts protecting devices in automotive systems. In [10], the authors provide a V&V alignment approach of SysML using a subset of the B Method semantically compatible with block definitions and state machine diagrams. This transformation was applied to verify safety properties of a railway industrial case study.

7. REFERENCES

[1] Railway applications – communications, signalling and processing systems – software for railway control and protection systems. *CENELEC, EN 50128*, 2001 (revised at 2011).

[2] IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements. *IEEE Std 1474.1-2004 (Revision of IEEE Std 1474.1-1999)*, pages 1–45, 2004 (reaffirmed at 2009).

[3] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[4] AdaCore. SPARK 2014 reference manual, 2011-2014. `docs.adacore.com/spark2014-docs/html/lrm`.

[5] L. d. Alfaro and T. A. Henzinger. Interface-based design. In *Engineering Theories of Software-intensive Systems*, NATO Science Series: Mathematics, Physics, and Chemistry 195, pages 83–104. Springer, 2005.

[6] L. d. Alfaro, T. A. Henzinger, and M. Stoelinga. Timed interfaces. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, pages 108–122, London, UK, 2002. Springer-Verlag.

[7] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *Proceedings of the 9th Int. Conf. on Concurrency Theory*, pages 163–178, London, UK, 1998. Springer-Verlag.

[8] P. America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK, 1991. Springer-Verlag.

[9] ARGOSIM. Stimulus. `www.argosim.com/products`.

[10] E. Bousse, D. Mentré, B. Combemale, B. Baudry, and T. Katsuragi. Aligning SysML with the B Method to provide V&V for systems engineering. In *Proceedings of the Workshop MoDeVVa'12*, pages 11–16, New York, NY, USA, 2012. ACM.

[11] O. Carrillo, S. Chouali, and H. Mountassir. Incremental modeling of system architecture satisfying sysml functional requirements. In *Proceedings of FACS 2013, (Revised Selected Paper)*, LNCS, pages 79–99. Springer International Publishing, 2014.

[12] S. Chouali, H. Mountassir, and S. Mouelhi. An I/O automata-based approach to verify component compatibility: Application to the CyCab car. *Electron. Notes Theor. Comput. Sci.*, 238:3–13, June 2010.

[13] L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[14] European Railway Agency. ERTMS/ETCS System Requirements Specification. Technical report, 2010.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[16] G. J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23:279–295, May 1997.

[17] E. Jahier. The Lurette V2 User Guide. Research report, VERIMAG, UMR 5104 CNRS, 2014.

[18] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. JML reference manual, 2002-2013. `eecs.ucf.edu/~leavens/JML/jmlrefman/`.

[19] T. Lecomte. Applying a formal method in industry: A 15-year trajectory. In *Proceedings of 14th International Workshop, FMICS 2009*, LNCS, pages 26–34. Springer Berlin Heidelberg, 2009.

[20] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16:1811–1841, November 1994.

[21] J. Mikác and P. Caspi. Temporal refinement for lustre. In *Proceedings of Languages Applications and Programming, SLAP'05*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[22] S. Mouelhi, K. Agrou, S. Chouali, and H. Mountassir. Object-oriented component-based design using behavioral contracts (version with proofs). Research report, DISC department, FEMTO-ST Institute, UMR CNRS 6174, 2015. `hal.archives-ouvertes.fr`.

[23] S. Mouelhi, S. Chouali, and H. Mountassir. Refinement of interface automata strengthened by action semantics. *Electron. Notes Theor. Comput. Sci.*, 253:111–126, October 2009.

[24] Object Management Group (OMG). OMG Systems Modeling Language SysML. Technical report, 2012.

[25] A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In *Proceedings of IJCAR 2008*, volume 5195 of *LNCS*, pages 171–178. Springer Berlin Heidelberg, 2008.

[26] A. Platzer and J.-D. Quesel. European train control system: A case study in formal verification. In *Formal Methods and Software Engineering*, volume 5885 of *LNCS*, pages 246–265. Springer, 2009.

[27] W. Schön, G. Larraufie, G. Moens, and J. Pore. *Railway Signalling and Automation Volume 1*, volume 3. La Vie du Rail, 2013.

[28] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[29] Y. Zhang, T. Tang, K. Li, J. Mera, L. Zhu, L. Zhao, and T. Xu. Formal verification of safety protocol in train control system. *Science China Technological Sciences*, 54(11):3078–3090, 2011.