Contract-based modeling and verification of timed safety requirements within SysML

Iulia Dragomir[1] · Iulian Ober[2] · Christian Percebois[2]

Abstract In order to cope with the growing complexity of critical real-time embedded systems, systems engineering has adopted a component-based design technique driven by requirements. Yet, such an approach raises several issues since it does not explicitly prescribe how system requirements can be decomposed on components nor how components contribute to the satisfaction of requirements. The envisioned solution is to design, with respect to each requirement and for each involved component, an abstract specification, tractable at each design step, that models how the component is concerned by the satisfaction of the requirement and that can be further refined toward a correct implementation. In this paper, we consider such specifications in the form of contracts. A contract for a component consists in a pair (assumption, guarantee) where the assumption models an abstract behavior of the component's environment and the guarantee models an abstract behavior of the component given that the environment behaves according to the assumption. Therefore, contracts are a valuable asset for the correct design of systems, but also for mapping and tracing requirements to components, for tracing the evolution of requirements during design and, most importantly, for compositional verification of requirements. The aim of this paper is to introduce contract-based reasoning for the design of critical real-time systems made of reactive components modeled with UML and/or SysML. We propose an extension of UML and SysML languages with a syntax and semantics for contracts and the refinement relations that they must satisfy. The semantics of components and contracts is formalized by a variant of timed input/output automata on top of which we build a formal contract-based theory. We prove that the contract-based theory is sound and can be applied for a relatively large class of SysML system models. Finally, we show on a case study extracted from the automated transfer vehicle (http://www.esa.int/ATV) that our contract-based theory allows to verify requirement satisfaction for previously intractable models.

Keywords Contract-based reasoning · Safety requirement · Component-based design · UML/SysML · Compositional verification · Timed input/output automata

Communicated by Dr. Gabor Karsai.

✉ Iulia Dragomir
iulia.dragomir@aalto.fi

Iulian Ober
iulian.ober@irit.fr

Christian Percebois
christian.percebois@irit.fr

[1] Department of Computer Science, Aalto University, Espoo, Finland

[2] IRIT - University of Toulouse, Toulouse, France

1 Introduction

The component-based design paradigm is one of the most used techniques for the development of critical real-time embedded systems. Integrated in a complete development process, it facilitates the system decomposition and, later, integration of components by delegating the responsibilities of developing correct components to different engineering teams. Yet, having multiple suppliers building integrated systems based on common system requirements entails a risk of errors during development due to the difficulty of decomposing global system requirements on components and the misinterpretation of the system requirements allocated to the software [25]. These misunderstandings have several sources but are often due to the use of ambiguous means to describe the system requirements and implementations thus leading to different interpretations. In industrial practice, early design models are often built in semi-formal languages such as UML [53], SysML [52] or AADL [63] which lack a proper mechanism for formalizing requirements and proving their satisfaction. The errors potentially introduced during the development are then discovered late and by very costly processes.

The last decades have seen an accelerating utilization of formal verification and validation techniques in the early phases of the development process in order to guarantee as soon as possible the correctness of the design, as well as for reducing production costs and increasing system quality. Design models are validated using an assortment of techniques, including design review [58], interactive simulation, and model checking [60]. While the first two explore a partial set of behaviors, model checking allows to verify the system requirements on all possible executions of the design, something which, for large systems with a high degree of parallelism, is often impossible to achieve due to the combinatorial explosion of the number of possible behaviors. For this reason, common verification techniques may find themselves powerless in front of the complexity of industrial-grade systems.

A way to tackle these problems is to use a compositional approach that allows for a requirement to be correctly decomposed into requirements on components and whose aim is to provide sound implementations. Therefore, we consider a development process based on partial and abstract specifications for components, traceable at each refinement step, and driven by requirements. Such specifications can then answer to two important open points in the design process: how are requirements mapped to each component involved in their satisfaction and how one component engages in the satisfaction of several requirements. We consider such specifications in the form of contracts: a *contract* for a component is defined by a pair (*assumption, guarantee*) where the assumption models an abstraction of the component's environment behavior and the guarantee models an abstraction of the component's behavior given that the environment behaves according to the assumption.

A contract-based reasoning technique enforces the three basic principles on which a component-based development process relies, which are component substitutivity and reuse, incremental development by successive refinements, and independent implementability. Moreover, it can provide a formal framework for proving the satisfaction of requirements. Besides being a solution for system design, contract-based reasoning offers diverse opportunities: mapping and tracing requirements to components, tracking the evolution of requirements during development, reviewing models, virtual integration of components [26], and, most importantly, compositional verification. Instead of reasoning with implementations during formal verification of requirements, one can use contracts and split the verification in two steps: (1) verify that each component satisfies its contract and (2) verify that the network of contracts correctly assembles and satisfies the requirement. Even thought the number of relations that need to be verified in order for contract-based reasoning to work is multiplied (linearly with the number of components), in general they involve more abstract specifications and thus they are less prone to combinatorial explosion, which makes them more tractable by automatic verification tools.

Despite these advantages, systems engineering has not yet widely adopted contract-based reasoning as a development technique for model design. The aim of this paper is to graft contract-based reasoning in the model-driven design and property verification process for critical real-time systems modeled in SysML. To the best of our knowledge, this study is the first to link industrially used standard modeling languages such as UML/SysML and formal behavioral contracts.

Paper structure Section 2 describes, in general terms, a method for reasoning with contracts which we use throughout this paper. The component-based model of UML/SysML and its extension with contract-related notions are presented in Sect. 3. The formal semantic model based on timed input/output automata on which our approach relies is presented in Sect. 4, as well as a mapping mechanism between the component model described in SysML and the formal semantic model. In Sect. 5, we develop the contract-based theory for our formal model and we discuss its implications on the SysML modeling. An automatic verification mechanism based on reachability analysis for the generated proof obligations is described in Sect. 6. The contract-based reasoning technique is applied in Sect. 7 on an industrial-scale system model extracted from the automated transfer vehicle system for the verification of a general safety requirement. The state of the art in contract-based approaches, discussing formal theories as well as works anchored in modeling languages, is presented in Sect. 8, before concluding.

2 A meta-theory for contract-based reasoning

In this section, we present the contract-based meta-theory proposed by Quinton et al. [61,62] on which we base our work. By meta-theory, we mean that the notion of component is kept abstract and the refinement relations used for contracts are not fully defined. In order to obtain a working contract framework, one has to formalize the notion of component and the refinement relations that are used, and prove the compositionality results required by the meta-theory. In exchange, the meta-theory provides a methodology for reasoning with contracts as illustrated in Fig. 1 and explained

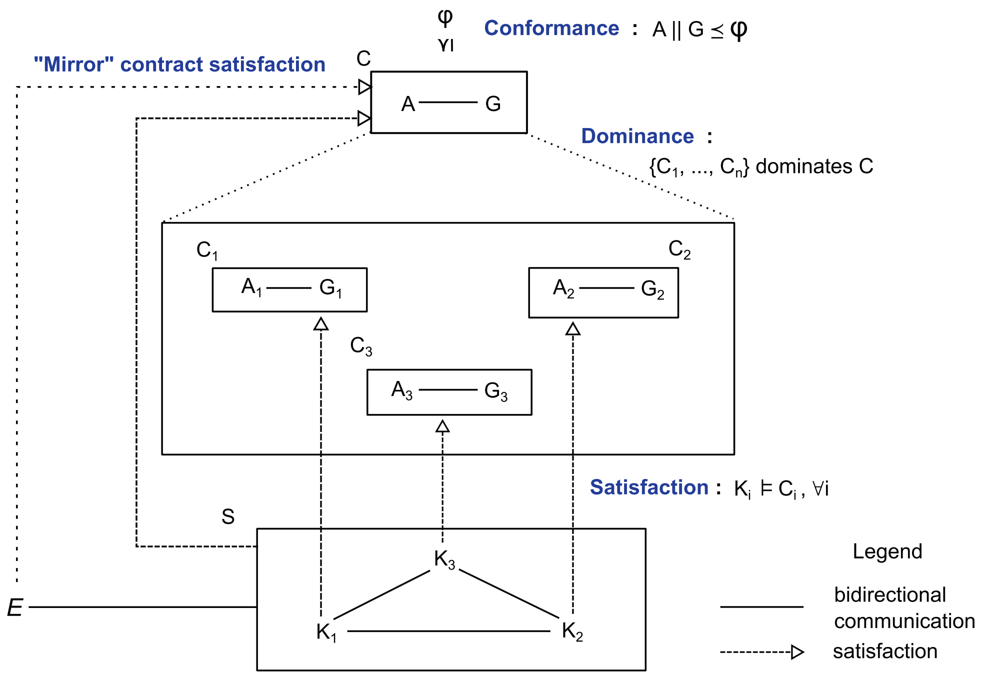

φ
≥
Conformance : A || G ≤ φ
"Mirror" contract satisfaction
C
A —— G
Dominance :
{C1, ..., Cn} dominates C
C1
C2
A1 —— G1
A2 —— G2
C3
A3 —— G3
Satisfaction : Ki ⊨ Ci , ∀i
S
K3
E
K1
K2
Legend
bidirectional communication
satisfaction

Fig. 1: Contract-based reasoning for a three-component subsystem [61]

below. The related work with respect to contract-based meta-theories and their implementations is discussed in Sect. 8.

Assume, at any level of the hierarchical decomposition of a system, a subsystem S obtained from the composition of several components $K_1, K_2, \ldots, K_n$ for which we want to prove that it satisfies a requirement φ. The meta-theory of [61,62] leaves open the choice of the composition operator in order to accommodate various definitions. In order to simplify the presentation, we will assume the existence of a notion of component compatibility, and the existence of a composition operator for every pair of compatible components, denoted $\|$, which is unique and associative. Then, S is obtained from the composition $K_1 \parallel K_2 \parallel \ldots \parallel K_n$. Figure 1 presents a subsystem S containing three components K_1, K_2 and K_3, and which communicates with an environment E. Again, on this example, we want to show that the requirement φ is satisfied by the subsystem S.

In order to use this contract-based methodology for proving requirement satisfaction by a subsystem S, we start by modeling a global *contract* $\mathcal{C}$ for S which *conforms* to the requirement φ.

Definition 1 (*Contract*) A *contract* $\mathcal{C}$ consists of a pair of compatible component specifications (A, G), where A is called the *assumption* and G is called the *guarantee*.

In this theory, A and G are specified using the same formalism as normal system components. Informally, the assumption of the contract $\mathcal{C}$ for the subsystem S is an abstract model of the behavior expected from the subsystem's environment E, while the guarantee is an abstract model of the behavior promised by S, given that the actual environment obeys to the assumption. Note that the same contract for a subsystem, and components in the general case, can be used to prove the satisfaction of several requirements.

The satisfaction of a requirement φ by a contract $\mathcal{C} = (A, G)$ is modeled by the *conformance* relation $A \parallel G \preceq \varphi$. In the meta-theory, $\preceq$ denotes a property conformance operator which is left open to be defined in its instances. In our instance of the meta-theory, we will use the same formalism for requirements as for components, and we will use the refinement relation between components as our conformance relation.

Since we are interested in working with contracts and discarding components compositions as much as possible from the reasoning, the methodology continues by designing a contract $\mathcal{C}_i = (A_i, G_i)$ for each component K_i of the subsystem S. The contract $\mathcal{C}_i$ is an abstract model of how the component K_i of S contributes toward the satisfaction of requirement φ. Indeed, the assumption A_i is expressed over the environment $E \parallel (\|_{j \neq i} K_j)$, while the guarantee G_i models the abstract behavior for K_i with respect to the running requirement φ. Note again that, when one is trying to prove the satisfaction of several requirements, the same contract can sometimes be used for one component, but also different contracts can be modeled for the same component with respect to different requirements. Figure 1 presents a set of contracts $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$, each $\mathcal{C}_i$ being modeled for the corresponding component K_i.

Next, the methodology requires for each component to *satisfy* its contract, denoted $K_i \models \mathcal{C}_i$. To define contract satisfaction, the meta-theory relies upon the existence of a *refinement under context* relation. This relation between two components K_i and K_j in an environment E, denoted $K_i \sqsubseteq_E K_j$, informally means that component K_i, when composed with the environment E, "behaves like" K_j when composed with the same environment. Although the choice of the refinement under context relation is left open, the meta-theory requires this operator to satisfy certain important properties, such as compositionality and correctness of circular reasoning; they are described below and are needed for proving Theorem 1.

Based on refinement under context, contract satisfaction is then defined as follows:

Definition 2 (*Contract satisfaction*) A component K *satisfies* a contract $\mathcal{C} = (A, G)$, denoted $K \models \mathcal{C}$, if and only if $K \sqsubseteq_A G$.

Note that the meta-theory does not impose any constraint with respect to the signature (i.e., the set of possible interactions with the environment) of the components K, A and G. As we will later see, in our instance of the meta-theory in Sect. 5.1, that we allow A and G to concentrate only on a subset of a component's signature and on a part of its behavior. This provides the ability to keep a contract abstract, with only the essential information for the running requirement.

The following step of the reasoning consists in proving the refinement of the global contract $\mathcal{C}$ by the set of contracts $\{\mathcal{C}_i\}_{i=\overline{1,n}}$. For this, the meta-theory introduces the notion of *dominance* relation.

Definition 3 (*Contract dominance*) A set of contracts $\{\mathcal{C}_i\}_{i=\overline{1,n}}$ *dominates* a contract $\mathcal{C}$ if and only if for any set of components $\{K_i\}_{i=\overline{1,n}}$ the following holds:

$$K_i \models \mathcal{C}_i, i = \overline{1,n} \implies K_1 \parallel K_2 \parallel \ldots \parallel K_n \models \mathcal{C}.$$

The dominance relation defined above involves component composition while avoiding defining a contract composition operator. However, in order to establish dominance, one would like to avoid component (implementation) composition, which is the main cause for combinatorial explosion in large systems. The crux of the meta-theory is the Theorem 1 below, which provides a set of sufficient conditions for dominance, allowing to boil dominance down to a set of contract satisfaction proof obligations. In order for it to work, the following compositionality conditions have to hold in the instance of the meta-theory:

1. Refinement under context is compositional: $K_1 \sqsubseteq_{E_1 \parallel E_2} K_2 \implies K_1 \parallel E_1 \sqsubseteq_{E_2} K_2 \parallel E_1$. This property allows for incremental design by successively incorporating parts of the environment in the components under study, while refinement under context holds.
2. Circular reasoning is sound: $K \sqsubseteq_A G \wedge E \sqsubseteq_G A \implies K \sqsubseteq_E G$. This property allows for independent implementability by breaking down the dependency between the components and their environment.

These two compositionality results ensure that the reasoning method of the meta-theory is sound.

Theorem 1 (Sufficient condition for dominance [61]) *If refinement under context is compositional and supports sound circular reasoning, then for establishing that* $\{C_i\}_{i=\overline{1,n}}$ *dominates* C *it is sufficient to prove that*

$$\begin{cases} G_1 \parallel \ldots \parallel G_n \models C, \textit{and} \\ A \parallel G_1 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \models \mathcal{C}_i^{-1}, \quad \forall i \in \overline{1,n} \end{cases}$$

where $\mathcal{C}_i^{-1} = (G_i, A_i)$ *denotes the* "mirror" contract *of* $\mathcal{C}_i$.

The first relation requires the refinement of a more abstract guarantee by a set of more specific guarantees, while the second expresses that individual assumptions need to be refined by the other components' guarantees together with the overall assumption.

Since real-life systems often exhibit a multi-layer architecture, the dominance step can be iterated at each architectural level and for each composed component until reaching the contracts for "atomic" components. To simplify the presentation, Fig. 1 only shows one dominance step.

Finally, in order to guarantee the satisfaction of the requirement φ, we have to make sure that the assumption A we made over the environment E and which is used in the global contract $\mathcal{C}$ is correct. The last step of the methodology consists in verifying the satisfaction of the "mirror" contract $\mathcal{C}^{-1}$. We note that this is necessary when the system under study S is an open subsystem, i.e., it interacts with an environment. If the requirement φ is expressed on a closed (i.e., not open) system, then there is no assumption to be defined and this step may be skipped.

As already mentioned, the framework presented above is a meta-theory, which has to be instantiated for a particular component model in order to yield a usable methodology. The instantiation means defining certain notions and proving certain results, something that we do for our concrete component model in the subsequent sections. The "to do" list is the following:

1. Formally define the component framework—the notions of component, parallel composition and refinement—the conformance relation $\preceq$ (if different from component refinement) and the refinement under context relation $\sqsubseteq_E$.
2. Prove that both conformance and refinement under context relations are preorders, i.e., they are reflexive and transitive. These conditions are needed for proving the compositionality results required by Therorem 1.

3. Prove that refinement under context is compositional. Compositionality is a prerequisite of Theorem 1.
4. Prove that circular reasoning is sound. This is also a prerequisite of Theorem 1.

For our component model of SysML, the formal model and the results mentioned above are provided in Sects. 4 and 5. Before that, in the next section we begin by examining the extensions of the UML/SysML meta-model needed in order to capture the syntactic aspects of contracts.

3 Modeling behavioral contracts in SysML

In order to be able to use contract-based reasoning in a model-driven development approach, we propose in the following an extension of modeling languages by introducing the notion of contract and the verification relations needed. First, since SysML is a rich standard supporting various modeling aspects, we identify a sufficient subset of modeling elements which allows to describe hierarchical component-based systems similar to the one depicted in Fig. 1. Next, we describe the contract-related notions presented in Sect. 2 by a domain meta-model illustrated in Fig. 5, as well as a set of well-formedness rules defined and formalized over these concepts in order to ensure that models comply to the meta-theory. Our domain meta-model is defined as an extension of the UML subset used in the definition of SysML (i.e., the *UML4SysML* package from [52]), thus the profile which is derived from it is applicable to both UML and SysML. We close this section by presenting this profile and how it can be used together with the meta-theory's methodology on a running example.

3.1 A UML/SysML subset for modeling timed asynchronous component-based systems

A component-based system, such as the one presented in Fig. 1, is structurally modeled in *UML4SysML* by two notions: the *class* which allows to define types that are to be used in the model and *composite structures* that allow to cope with the complexity of large systems by describing class instances and how they are composed and interconnected in a hierarchical structure.

In the following, we will assume the reader is familiar with the class, composite structure, and all the other relevant modeling elements from the UML and SysML standards. We impose certain constraints as to how these modeling elements are to be used, in order to avoid any modeling ambiguities, ensure rigorous static typing of composite structure models and make them comply with the formal model detailed later on in the paper. The constraints are only listed briefly in the following; they are described in full detail and justified in [49]. The components are deemed to communicate only by *asynchronous signals*, sent and received over *ports*. The type of a port consists of one *interface* that defines only the signal receptions that can travel through, while its direction specifies whether it is an entry (provided) or an exit (required) point. We denote by *component signature* the set of signal receptions defined within the component port types and their input/output direction. In a composite structure, the *ports* of the different *parts* (components) shall be connected with *connectors* whenever they communicate. Multicast ports, i.e., outbound ports that are connected to more than one inbound port having the same type, are forbidden since they induce an important overhead in the definition of the semantics. When needed, we require them to be modeled through multiple output event ports.

The behavior of an atomic component is modeled by a *state machine*. To describe actions (e.g., transition effects), we use a subset of the action language formalized by the fUML standard [56] consisting of signal output, assignment of structural features, and expression valuation. The behavior of a composed component should not be explicitly modeled as it is given by the parallel execution of all its subcomponents.

Since we aim to model real-time systems, we extend the modeling of component behavior with timed concepts as they are described in the MARTE profile [55] for physical/real-time. We define by *Timer* a *clockType* stereotyped class that has a dense time base and which contains the *getTime* and *setTime* operations. Thus, a designer can use Timer instances within a component behavior description in order to model time elapse and timeout events. In addition, we allow using *urgency* stereotypes on transitions for describing more flexibly how time can elapse in control states: ≪*eager*≫ models that the transition has to be executed as soon as it is enabled (i.e., time elapse is disabled) and ≪*lazy*≫ models that the transition can be executed at any moment in time (i.e., time elapse is enabled and unbounded). The notion of urgency was introduced for timed automata in [12] and was previously used in our work on UML [50].

Lastly, we are interested in verifying requirement satisfaction, and therefore in their formalization. Our interest is limited to safety requirements, i.e., properties that assert that nothing bad happens during system execution, and whose violation can always be described in terms of a finite trace. In our case, the properties are described in an automata-based language, in the form of *observers*. An observer is an object that monitors the system events and gives verdicts about the (non-)satisfaction of the requirement that it formalizes. It is modeled by a class stereotyped ≪*observer*≫ which has a local memory (attributes) and a state machine to describe the requirement's behavior. In order to model the divergence from a nominal scenario, observer states may be marked with the ≪*error*≫ stereotype when the executions leading to them

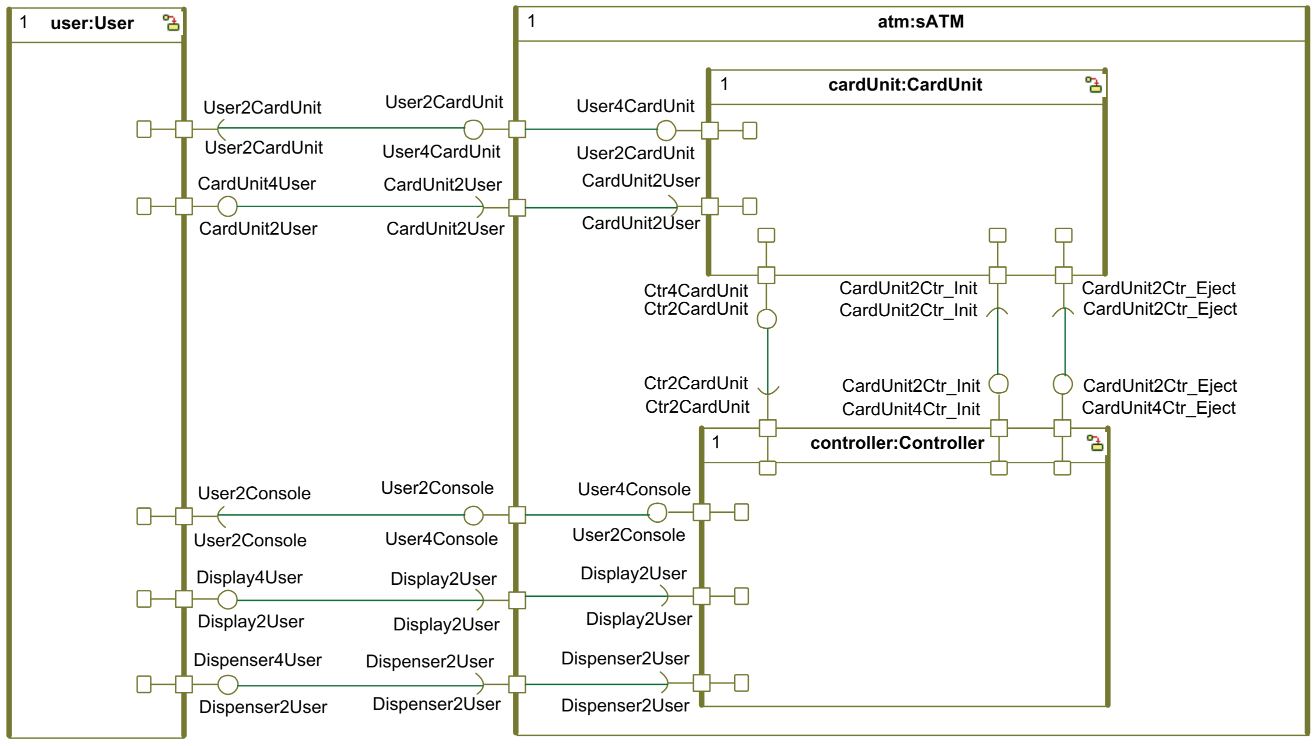


Fig. 2: Running example: the architecture of the simplified automated teller machine

have to be considered erroneous. The monitoring of actions is modeled by a set of special transition triggers called *match* clauses as follows: *send* for output actions and *acceptsignal* for triggers. The semantics of a match clause is to synchronize with the occurrence of the respective signal in the system at execution. Further details about the observer mechanism used in this paper can be found in our previous work [50].

3.2 The simplified automated teller machine running example

In order to illustrate the contract-based concepts for system designs, we describe in the following the running example of a simplified automated teller machine (sATM) for the withdrawal transaction only. The model has been edited using the IBM SysML Rhapsody tool.[1]

The architecture of the system is represented in Fig. 2 and consists of the following blocks: the *sATM* that contains an instance of the *CardUnit* responsible for the insertion and removal actions of a card and an instance of the *Controller* responsible for the withdrawal transaction, and the *User*. Note that the *User* models the environment of the sATM and, therefore, is not part of the system under study. We consider here only one of the possible behaviors a real customer can exhibit as described below.

The considered use case for the sATM is the following: a customer is required to insert a card into the card unit of a sATM. Then, the sATM will verify the amount available on the card and will propose several amounts (within the accepted range) to the user for withdrawal. The sATM interacts with the customer via a console and can handle only one user at a time. The customer chooses an amount and waits for the sATM to execute the transaction. The sATM will display a message and eject the card. If the card is removed within 5 time units after being ejected, the amount is distributed and the total amount available on the card updated. Otherwise, the card is retained and the sATM will become unavailable for further transactions. This behavior is modeled in Fig. 3 by a state machine for each component.

We are interested in showing with the contract-based approach that the current model satisfies the following requirement:

Requirement 1 *If the card is removed within 5 time units after being ejected, the amount released by the sATM is the amount requested by the customer.*

This requirement is formalized with an *observer* in Fig. 4 from the point of view of the sATM. This formalization captures several events that are briefly mentioned by the

[1] http://www.ibm.com/software/products/en/ratirhapfami.

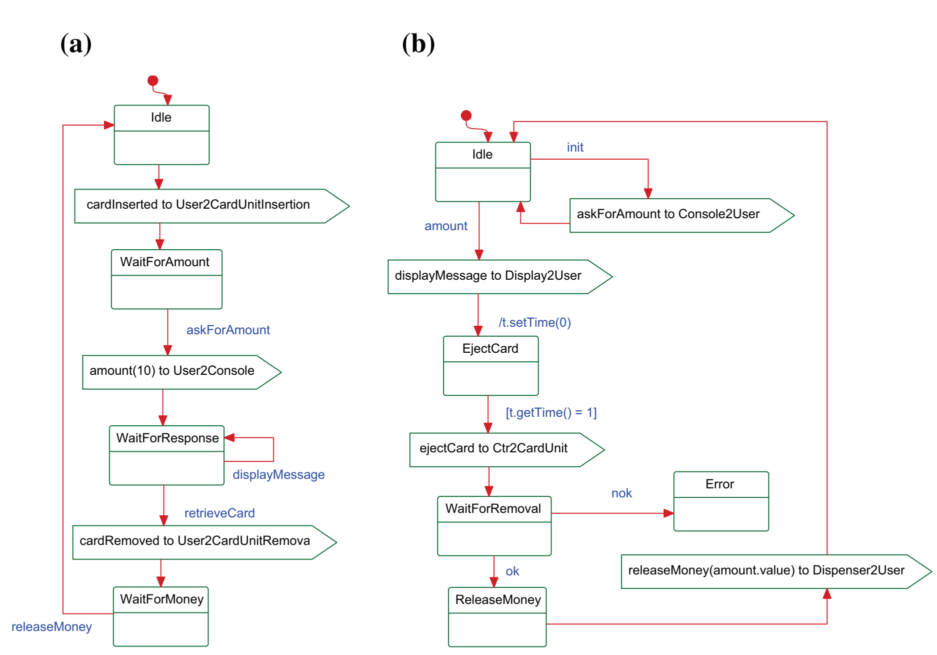


Fig. 3: State machines for the three main blocks of the sATM. **a** *User*. **b** *Controller*. **c** *CardUnit*

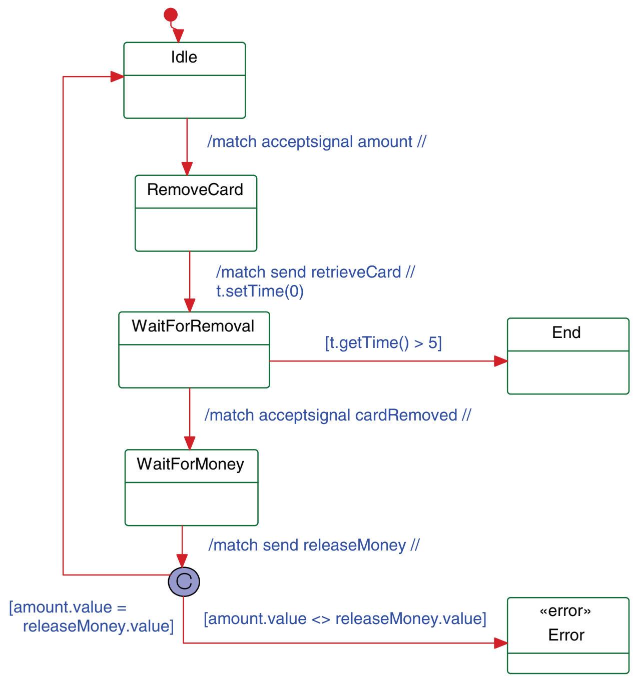

Idle
/match acceptsignal amount //
RemoveCard
/match send retrieveCard //
t.setTime(0)
WaitForRemoval
[t.getTime() > 5]
End
/match acceptsignal cardRemoved //
WaitForMoney
/match send releaseMoney //
C
[amount.value =
releaseMoney.value]
[amount.value <> releaseMoney.value]
«error»
Error

Fig. 4: SysML formalization with an *observer* of Requirement 1: *the amount released by the sATM is equal to the requested amount*

textual requirement as follows: the *amount* to be selected, *retrieveCard*, and *cardRemoved* to check the assumption over the environment and *releaseMoney* over which the requirement is expressed in conjunction with *amount*. Initially, the observer waits in the state *Idle* for the customer to insert a card and select an *amount*. Next, it expects for the customer to remove the card from its slot within 5 time units once the latter is ejected by the machine: the *retrieveCard* output signal followed by the *cardRemoved* input signal sequence. Then, the sATM executes the *releaseMoney* operation. The values of the requested and released amounts are modeled by a parameter of their corresponding signals. If the values coincide then the requirement is satisfied, otherwise the *Error* state is reached and the property is violated. In case the card is not removed within the allowed time interval, since this is in fact an assumption made over the sATM's environment, the requirement may be considered satisfied; the observer then goes into state *End* and stops.

3.3 A meta-model for behavioral contracts

The contract-based meta-theory presented in Sect. 2 is supported by a domain meta-model illustrated in Fig. 5. To explain this meta-model, we start by the meta-classes that are reused as such from *UML4SysML* [52]. The meta-class *Property* denotes the notion of part (component) in a composite structure, in the standard. The meta-class *Class* models component types.

An important concept that is not part of the UML meta-model, but which is defined within SysML is the *requirement*. However, the SysML definition and usage of requirements is informal, while contract-based reasoning needs to for-

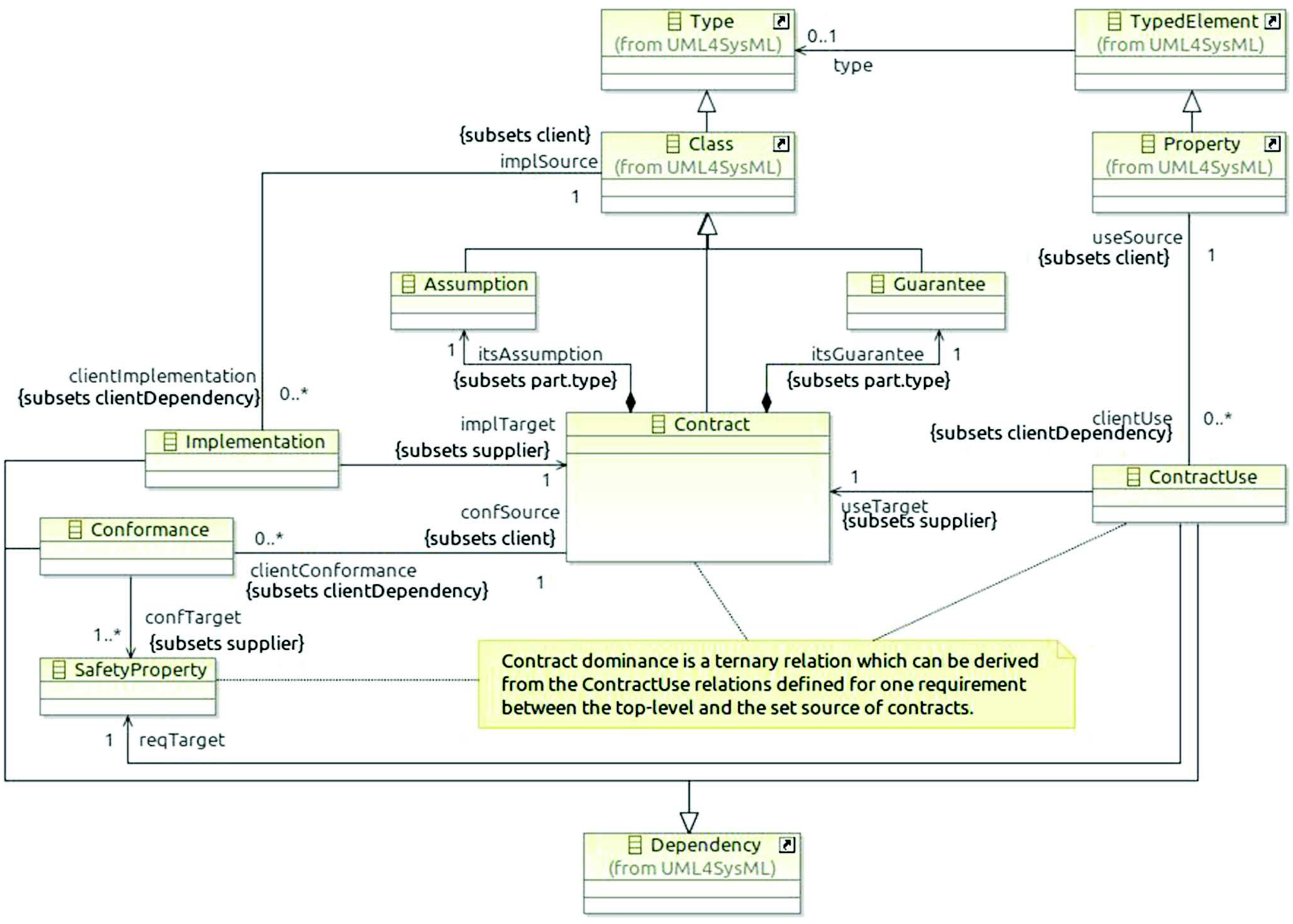


Fig. 5: An extension of the UML meta-model for contract-based reasoning

malize them within a framework. Since several formalisms are available to represent requirements (e.g., automata-based languages like the observer notion presented in Sect. 3.1, temporal logics, etc.) and implemented in different tools supporting UML/ SysML modeling, we prefer to add the concept to our meta-model and to keep it at an abstract level. Therefore, a requirement that a UML/SysML model has to satisfy is denoted by the meta-class *SafetyProperty* in Fig. 5.

There are two categories of notions that are defined within the meta-theory of Sect. 2: (1) those that define how to model a contract, represented in the upper part of the meta-model of Fig. 5 and (2) those that define the relations between components/ contracts, represented in the lower part of Fig. 5.

3.3.1 Modeling contracts

In order to introduce contracts, we firstly define the assumption and the guarantee which are, respectively, represented by the meta-classes *assumption* and *guarantee*. Both notions are subtypes of *Class*. The intuition behind this is straightforward: the meta-theory defines the assumption/guarantee as component, and therefore, to describe them, we can use the language element as for SysML components, i.e., a class with a behavior modeled by a state machine. One restriction that applies to this representation of assumptions/guarantees as classes is that they may not be involved in any relations (in the UML/SysML sense, such as associations or generalizations) except interface realizations needed to type ports. The reason is that assumptions/guarantees do not have the usual operational semantics of classes but are only used in refinement relations (in the sense of our formal framework), and therefore, their only relevant elements are their interface (ports) and behavior (state machine). This way of modeling assumptions/guarantees is convenient and supported by standard UML/SysML model editors.

A contract is represented by the meta-class *Contract*, a subtype of *Class*, whose composite structure must contain exactly one assumption and one guarantee, all other properties being forbidden. In order to comply to the definition of contract from the meta-theory, we need again to restrict the language a contract can be described with. Therefore, a contract does not exhibit any behavior and it is not involved in any other relations than the ones defined in the meta-theory. This modeling of a contract allows for reusability: a contract is defined only by instances, while types (assumption/guarantee) can be used within other contracts too.

The meta-theory requires for a contract to be a closed component. This means that all signals of a contract's assumption/guarantee have as source/target the other component. Since the communication for our system models is based on ports and connectors, we express this constraint on contracts with respect to port types that must be matched with reversed direction, in the following rule:

Rule 1 *Given a Contract, the assumption and the guarantee define a closed system: all ports of each type have a correspondent (by type and conjugated direction) within the ports of the other type.*

Moreover, the purpose of a contract is to model a partial behavior with respect to a requirement. We enable this by allowing a guarantee to define a set of ports that corresponds to a subset of the component's ports. The port correspondence is based on name, type, and direction, which must be the same as those of a port of the class typing the component. The aim of this rule is to contribute to requirement-driven design of systems: specifications are refined toward implementations based on requirements, and in order to support integrating multiple requirements in the same component, one needs to be able to specify contracts that concern only a subset of the component's ports.

Rule 2 *The set of ports of a contract's Guarantee is included in the set of ports of any Property (component) that satisfies the Contract.*

Rules 1 and 2 are part of a set of well-formedness rules defined for the meta-model of Fig. 5 such that the steps of the meta-theory from Sect. 2 can be applied on system models. This set of rules has been formalized with OCL [54], which allows to verify (using an OCL interpreter, Topcased[2] in our case) the static typing of a model extended with contracts, before applying verification and validation techniques for system behavior. We present here only a few important rules concerning the contract definition for the satisfaction and dominance relations of the meta-model and for which the OCL formalization is provided in "Appendix 1." Further details and the complete set of rules can be found in [35].

3.3.2 Modeling contract satisfaction

The satisfaction relation that relates a component to a contract is represented by the *Implementation* meta-class at the type level of the component. This relation, a subtype of *Dependency*, is defined between a *Class* and a *Contract* and expresses that the class satisfies the contract. This definition allows for the multi-view modeling of a system from the requirement perspective: a class can implement several contracts and a contract can be implemented by several classes.

Therefore, when verifying a particular requirement, the designer has to specify, for each component, which contract from the component's set of implemented contracts has to be used for that requirement. We define a second relation, also a subtype of *Dependency*, named *ContractUse*, that models contract satisfaction at the components' level and specifies the requirement concerned by the satisfaction relation. This prerequisite is modeled in Fig. 5 by the association from *ContractUse* to *SafetyProperty*. For a *ContractUse* relation to be correctly defined, an *Implementation* relation must exist between the component's type—a class—and the contract.

3.3.3 Modeling dominance

The dominance relation described in Sect. 2, between a more general contract and a set of more specific contracts, is not explicitly modeled in Fig. 5 since it can be deduced from the *ContractUse* relations. Indeed, each component of the system—atomic or composed—involved in the satisfaction of a requirement R must have a *ContractUse* relation to a contract supporting R, i.e., the *reqTarget* association of the *ContractUse* has to point to R. Therefore, when applying the methodology, in order to find the contracts that dominate a contract C of a component K needed to prove R, one simply has to look for the contracts of the subcomponents of K which have R as *reqTarget*.

Since we want to apply the methodology for the verification of system models, we require that each deduced dominance relation to be unique for a given context and a given requirement, as expressed in the following rule:

Rule 3 *There is one and only one ContractUse relation between a Property, a SafetyProperty and one Contract.*

A dominance relation can also be subject to signature refinement. This condition is illustrated with a rule and the corresponding formalization in [35].

3.3.4 Modeling conformance

Finally, the conformance relation is represented by the metaclass *Conformance* of type *Dependency* between a *Contract* and a *SafetyProperty*. We note that a contract can serve as source for checking several requirements.

We ensure the completeness of the reasoning by the following rule:

Rule 4 *Within a model, for any SafetyProperty, there is one and only one contract conforming to it, implemented by the*

[2] http://www.polarsys.org/.

component on which the requirement is expressed (which may be the whole system).

This set of rules fully presented and formalized in [35] allows us to generate an unambiguous set of proof obligations whose satisfaction ensure the satisfaction of system's requirements.

3.4 From domain meta-model to profile

In order to use contracts in a standard UML/SysML model, our choice is to define a profile and use the stereotype mechanism on the base UML4SysML meta-classes from which the new meta-classes from Fig. 5 are derived. For the meta-class *Class* the stereotypes that apply are: ≪*assumption*≫, ≪*guarantee*≫, ≪*contract*≫ and ≪*observer*≫, where *observer* is our instantiation of *SafetyProperty* presented in Sect. 3.1. For the meta-class *Dependency*, we define the ≪*contractConformance*≫, ≪*contractImplementation*≫ and ≪*contractUse*≫ stereotypes, where the latter stereotype defines the *reqTarget* property (or tagged value) that refers to the requirement for which the relation is defined.

3.5 Contracts in the sATM example

We describe in the following the instantiation of the contract meta-model and the application methodology on the sATM running example presented in Sect. 3.2, illustrated in Fig. 6. With respect to Requirement 1, the system under study S is given by the component *atm* of type sATM, while the environment E consists of the component *user*.

We start by defining a global contract *C_sATM* for the component under study *atm*. This contract is involved in several relations: first, it is used to verify the satisfaction of the expressed requirement which is modeled by the *contractConformance* dependency pointing to *Requirement1*. This contract is also used within the proof by the *atm* component modeled with the *contractUse* dependency tagged with *Requirement1* as running requirement. There is also a *contractImplementation* relation between *sATM*—the *atm* component's type—and *C_sATM* which is omitted from Fig. 6 in order to not overload the model.

Next, we model a contract for each of the *atm*'s subcomponents: *C_Controller* for the *controller* component and *C_CardUnit* for the *cardUnit* component. We link each component to its corresponding contract by *contractUse* relations tagged with the same *Requirement1*. Note that there is a derived dominance relation between *C_sATM* and {*C_Controller*, *C_CardUnit*}. Again, we omit from the representation the *contractImplementation* relations corresponding to the *contractUse* ones.

Finally, we detail each contract as a closed composite structure and their assumption/guarantee behavior with a state machine. Bear in mind that this behavior is described with respect to Requirement 1.

Figure 7a presents the contract *C_Controller* for the *controller* component. Since the requirement is expressed with respect to the release money functionality, we can abstract several interactions between the *controller* and its environment that do not impact its behavior: with the *cardUnit* component on the initialization process and with the *user* component on the display role. So, the signature of the

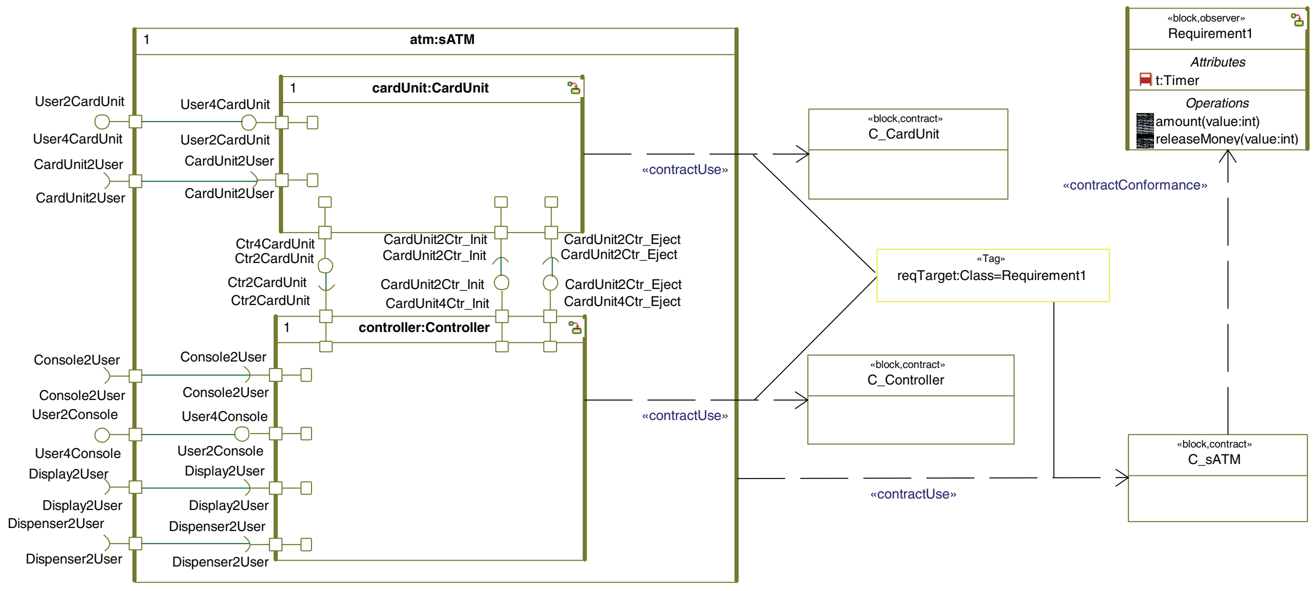


Fig. 6: Contracts and their relations in the sATM example

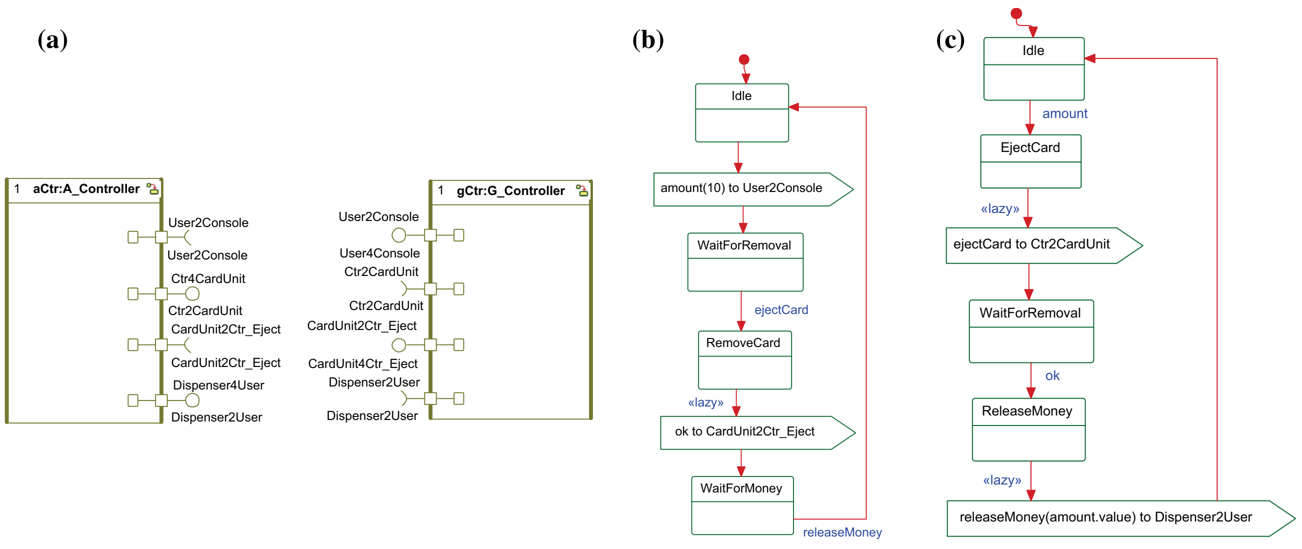


Fig. 7: Contract modeling for the *controller* component. **a** Contract architecture. **b** Behavior of the assumption. **c** Behavior of the guarantee

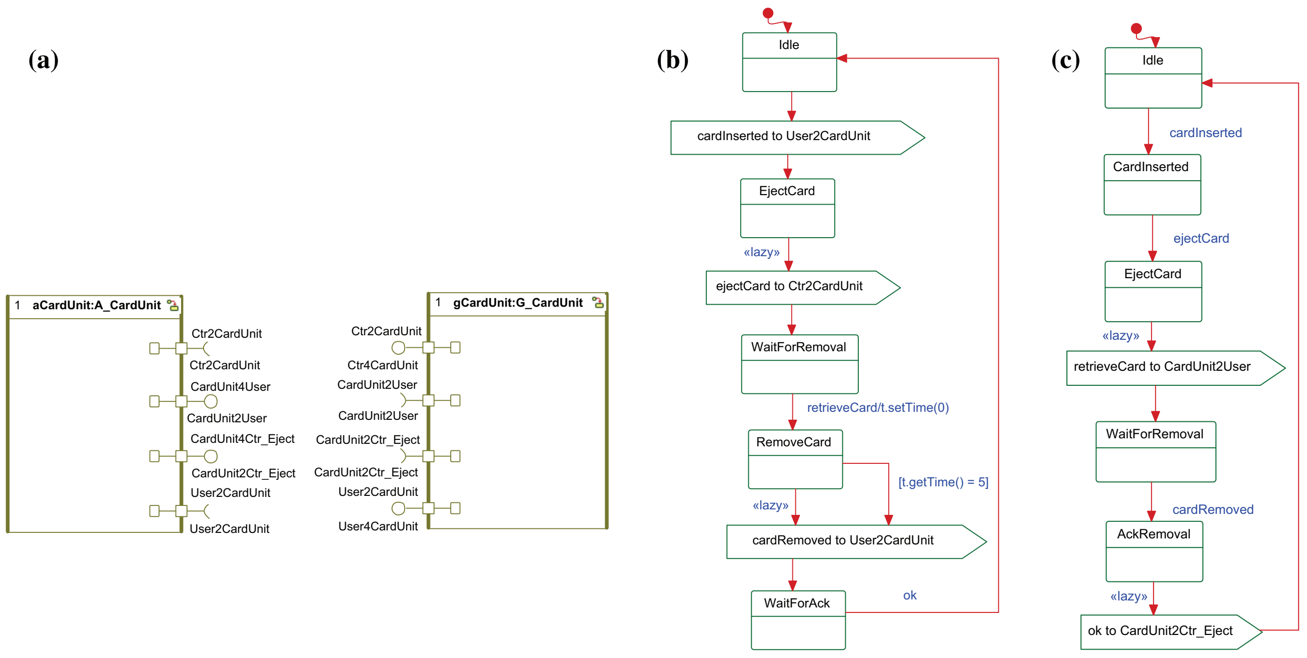


Fig. 8: Contract modeling for the *cardUnit* component. **a** Contract architecture. **b** Behavior of the assumption. **c** Behavior of the guarantee

contract is refined and consists in {*?amount*, *?ok*, *?nok*, *!ejectCard*, *!releaseMoney*}, where ? is used to denote an input while ! marks an output. The assumption represented in Fig. 7b models that after the *amount* is selected, the card is removed without the occurrence of an error, i.e., we assume that only *ok* may be sent. The component guarantees that if the amount is eventually released then it will have the same value as the one requested by the customer, modeled in Fig. 7c by the parameter *amount.value* of *releaseMoney*. The ≪*lazy*≫ urgency annotations in the contract state machines are justified by expressiveness constraints due to the semantics of contracts, which cannot be explained at this point but are detailed later in Sect. 5.2.

The contract for the *cardUnit* component, represented in Fig. 8a, is also modeled on a subset of the component's signature. Again, the initialization of the withdrawal process is not of concern for Requirement 1, yet the card insertion action has to be modeled in order not to encompass invalid

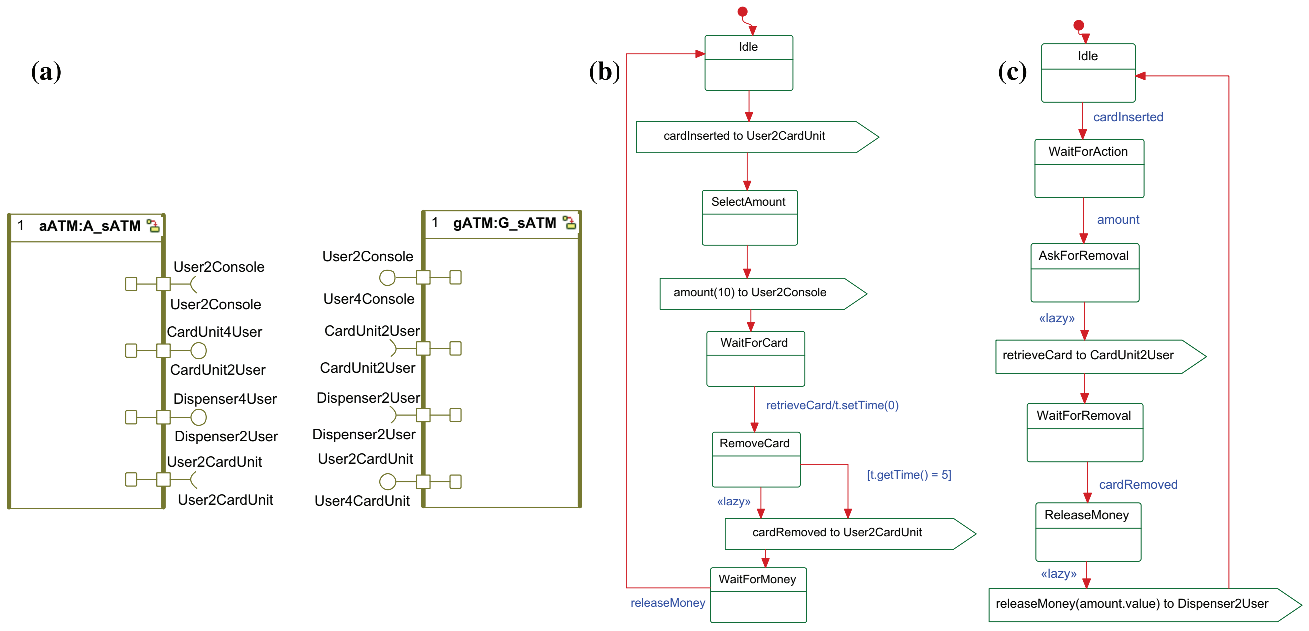


Fig. 9: Contract modeling for the *ATM* component. **a** Contract architecture. **b** Behavior of the assumption. **c** Behavior of the guarantee

executions of the component. Then, the contract signature consists of {*?cardInserted*, *?ejectCard*, *?cardRemoved*, *!retrieveCard*, *!ok*, *!nok*}. The assumption *A_CardUnit* modeled in Fig. 8b is informally described by the requirement: once the signal *retrieveCard* is handled, a *Timer t* is set to 0; then, the card is removed within at most 5 time units. This time delay is modeled by two outgoing transitions from *RemoveCard*: one with a lazy semantics which lets time elapse and one with an eager semantics enabled as the clock value is equal to 5 time units which ensures the execution of the action at the end of the deadline. The *cardUnit* will guarantee that only the *ok* signal is raised during the remove process thus eliminating the *nok* branch.

Finally, the top-level contract *C_sATM* is represented in Fig. 9a. Similarly as with the previous contracts we abstract the role of the display of the *atm*. Its signature is the set {*?cardInserted*, *?amount*, *?cardRe-moved*, *!retrieveCard*, *!releaseMoney*}. This signature is identical on inputs and outputs with the union signature of the *G_Controller* and *G_CardUnit* composition. The assumption, modeled in Fig. 9b, describes a behavior similar to the *A_CardUnit*. The guarantee, represented in Fig. 9c, expresses that if the amount is released then it will have the same value as the one requested by the customer.

4 A formal model for semantics: timed input/output automata

In order to apply verification and validation techniques on system models, we have, at first, to provide a formal model that describes their semantics. We chose to build our framework on a variant of *timed input/output automata* (TIOA) as defined in [42] since it is suitable to express the semantics of the timed reactive components of SysML. Moreover, it is thoroughly defined and it provides some ready-to-use compositionality results that are required by the meta-theory we are instantiating. A discussion with respect to other variants of TIOA is provided in Sect. 8. This formal framework is described in Sect. 4.1, while our view of the mapping from SysML notions to TIOA is presented in Sect. 4.2.

4.1 A flavor of timed input/output automata for SysML semantics

A SysML component is represented at the semantic level by a TIOA:

Definition 4 (*Timed input/output automaton*) A *timed input/output automaton* $\mathcal{A}$ is a tuple $(X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ where:

- X is a finite set of discrete variable symbols and Clk is a finite set of clock symbols. We denote by $Y = X \cup Clk$ the set of all internal variable symbols of $\mathcal{A}$.
- Q represents the set of states of $\mathcal{A}$. A state $q \in Q$ is a function $q : Y \to \mathcal{D}_Y$ which gives a value in a specific domain to each variable from Y. Note that for a discrete variable $x \in X$ the domain $\mathcal{D}_x$ is a finite set, while for clocks $c \in Clk$ the domain is the set of reals $\mathbb{R}^+$. Such a state q is also called a *valuation*. Let $val(Y)$ be the set of all valuations defined on Y; then $Q \subseteq val(Y)$.

– $\theta \in Q$ is the start state.
– I is a set of input actions, O a set of output actions and V a set of visible actions. We denote by $E = I \cup O \cup V$ the set of external actions which we call in the following the *signature of the automaton*.
– H is a set of internal actions. We denote by $A = E \cup H$ the set of all executable actions.
– I, O, V and H are pairwise disjoint sets.
– $D \subseteq Q \times A \times Q$ is a set of discrete transitions.
– $\mathcal{T}$ is the set of trajectories. Each trajectory is a function $\tau : J_\tau \to Q$, where J_τ is a real interval of type $[0, t]$ or $[0, \infty)$ with $t \in \mathbb{R}^+$, such that $\forall u \in J_\tau, \forall x \in X, \tau(u)(x)$ is constant, and $\forall c \in Clk, \tau(u)(c) = u + \tau(0)(c)$.

We note that there are two differences between the previous definition and the one presented in [42]. The first one relates to the extension of TIOA with *visible* actions, in addition to *inputs*, *outputs* and *internals*. Such actions find their rationale in the output–input matching of components. When computing a composition, in the asynchronous SysML semantics sending and receiving a signal (action) need only to leave a visible trace, whereas in [42] it becomes an output thus allowing for broadcast, which is inconsistent with the SysML semantics. Then, the need for visible actions is motivated by the system requirements which are often described with respect to closed systems and usually involve in their description the monitoring of input–output synchronization. Furthermore, visible actions support the definition of our refinement relation, as it will be shown in Sect. 5.

The second difference consists in the restriction of the state space: the domains of discrete variables are finite and the only allowed trajectories are constant functions for discrete variables and linear functions with the derivative equal to 1 for clock variables. While the definition from [42] allows for any functions to be used as trajectories and thus covers general hybrid systems, this restriction makes our timed model expressiveness equivalent to that of Alur–Dill timed automata [3]. This restriction opens the possibility of automatic reachability analysis or verification of simulation relations, which are undecidable for the more general TIOA of [42]. However, the compositionality results required by the meta-theory and provided in Sect. 5 are independent from this restriction, i.e., they can be proved also for hybrid systems as described by Kaynar et al. [42].

Notation We often denote elements of a TIOA $\mathcal{A}$ by $X_\mathcal{A}$, $Q_\mathcal{A}$, $\theta_\mathcal{A}$, $I_\mathcal{A}$, etc. We omit these subscripts where no confusion seems likely. We denote by $\mathrm{x} \xrightarrow{a}_\mathcal{A} \mathrm{x}'$ any $(\mathrm{x}, a, \mathrm{x}') \in D_\mathcal{A}$. Again, we drop the subscript when $\mathcal{A}$ is clear from the context. For a trajectory τ, we denote by $\tau.fval = \tau(0)$ and by $\tau.ltime$ the supremum of its domain. A trajectory τ is *closed* if its domain is a closed interval. Then $\tau.ltime$ is part of its domain, and we denote $\tau.lval = \tau(\tau.ltime)$. The notation $\mathrm{x} \xrightarrow{\tau}_\mathcal{A} \mathrm{x}'$ can be used if $\exists \tau \in \mathcal{T}$, $\mathrm{x} = \tau.fval$ and $\mathrm{x}' = \tau.lval$.

$\tau' = \tau\lceil[0, t]$ with $t \in J_\tau$ is called a *prefix*, where $\lceil$ denotes the operator which restricts a function to a subset of its domain. τ' is a *suffix* if $\exists t \in J_\tau$ such that $\tau' : [0, \tau.ltime - t] \to Q$ if τ is closed or $\tau' : [0, \infty) \to Q$ if τ is open, and $\tau'(u) = \tau(t + u)$, i.e., τ' is obtained by restricting τ to $J_\tau \cap [t, \infty)$ and left-shifting it such that $J_{\tau'}$ starts in 0.

Axioms A timed input/output automaton must satisfy the following axioms:

(A0) (*Existence of point trajectories*)
$\forall \mathrm{x} \in Q, \gamma(\mathrm{x}) \in \mathcal{T}$ where $\gamma(\mathrm{x}) : [0, 0] \to \mathrm{x}$ maps 0 to x.
(A1) (*Prefix closure*)
$\forall \tau \in \mathcal{T}, \forall \tau'$ a prefix of τ, $\tau' \in \mathcal{T}$.
(A2) (*Suffix closure*)
$\forall \tau \in \mathcal{T}, \forall \tau'$ a suffix of τ, $\tau' \in \mathcal{T}$.
(A3) (*Concatenation closure*)
Let $\tau_0\tau_1\tau_2\ldots$ be a finite or countably infinite sequence of trajectories in $\mathcal{T}$ such that, for each nonfinal index i, τ_i is closed and $\tau_i.lval = \tau_{i+1}.fval$. Then $\tau_0^\frown\tau_1^\frown\tau_2^\frown\ldots \in \mathcal{T}$, where $^\frown$ denotes the concatenation operator. Informally, the concatenation operator models the union between a first closed trajectory and a second one right-shifted such that its start time coincides to the limit of the first one.
(A4) (*Input actions enabling*)
$\forall \mathrm{x} \in Q, \forall a \in I, \exists \mathrm{x}' \in Q$ such that $\mathrm{x} \xrightarrow{a} \mathrm{x}'$.
(A5) (*Time-passage enabling*)
$\forall \mathrm{x} \in Q, \exists \tau \in \mathcal{T}$ such that $\tau(0) = \mathrm{x}$ and either

1. $\tau.ltime = \infty$, or
2. τ is closed and some $l \in H \cup V \cup O$ is enabled in $\tau.lval$.

Axioms A0–A3 are basic properties ensuring that the behavior of a TIOA is well defined (see definitions below). Axiom A4 corresponds to a semantic choice which is also made in [42]: a TIOA can never refuse an input; this choice is consistent with component models based on asynchronous messages such as that of SysML, in which a component can at any time receive a message and stores it for later use. Axiom A5 is a basic soundness property also: it states that a component can never block time progress, it either performs an action at a future moment or lets time pass up to infinity.

Behavior The behavior of timed input/output automaton is given by a set of *executions*. Informally, an execution records what happens during a particular run including discrete changes of states as well as changes that occur during time elapse (trajectories).

Definition 5 (*Execution*) An *execution* of a timed input/output automaton $\mathcal{A}$ is a (possibly infinite) sequence $\alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots$ where:

- each a_i is an action in $A_{\mathcal{A}}$,
- each τ_i is a trajectory in $\mathcal{T}_{\mathcal{A}}$,
- $\tau_0.fval = \theta$,
- all τ_i are closed except the last trajectory which can be either open or closed and
- if τ_i is not the last trajectory in α then $\tau_i.lval \xrightarrow{a_{i+1}} \tau_{i+1}.fval$,
- if α is a finite sequence then it ends with a trajectory.

The last item is added for convenience of notation, since an execution ending with a discrete transition can always be extended with a point trajectory.

An execution preserves all the information from the TIOA transitions. However, some elements do not present much interest when observing the behavior or checking for behavior refinement, such as the evolution of internal variables during time elapse or the execution of internal actions. We use the notion of *trace* introduced in [42] as the projection of an execution on external actions and time elapse intervals.

Definition 6 (*Trace*) Let α be an execution. Then $trace(\alpha)$ is the restriction of α to $(E_{\mathcal{A}}, \emptyset)$, denoted $trace(\alpha) = \alpha\lceil(E_{\mathcal{A}}, \emptyset)$, where:

- each a_i appearing in $trace(\alpha)$ is an action in $E_{\mathcal{A}}$, i.e., all actions from $H_{\mathcal{A}}$ are removed from α, and
- each $\tau_i : J_{\tau_i} \rightarrow \emptyset$, $J_{\tau_i} \subseteq \mathbb{R}^+$, records only the length of time-passage and ignores the evolution of variables.

If after action removal the trace contains adjacent trajectories, then the concatenation operator is applied in order to obtain only one trajectory. We denote by $traces_{\mathcal{A}}$ the set of traces of the automaton $\mathcal{A}$ and by $ftraces_{\mathcal{A}}$ the set of finite traces of $\mathcal{A}$ (traces with a finite number of discrete actions, but not necessarily time-bounded). Then $ftraces_{\mathcal{A}} \subseteq traces_{\mathcal{A}}$. The set of traces of an automaton can present two properties: *closure under limits* and *closure under time extension*. Closure under limits informally means that any infinite sequence whose prefixes are traces is also a trace. Closure under time extension means that any time-bounded trace can be extended with an open-interval trajectory which lets time progress to infinity without any other visible action occurring. The formal definitions of these two notions are those presented in [42].

Composition We define a *parallel composition* operator for allowing the automata to communicate and be executed in parallel. The following definition presents the conditions that have to be satisfied in order to compose two automata.

Definition 7 (*Compatible components*) Two timed input/output automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if $Y_{\mathcal{A}_1} \cap Y_{\mathcal{A}_2} = H_{\mathcal{A}_1} \cap A_{\mathcal{A}_2} = H_{\mathcal{A}_2} \cap A_{\mathcal{A}_1} = V_{\mathcal{A}_1} \cap A_{\mathcal{A}_2} = V_{\mathcal{A}_2} \cap A_{\mathcal{A}_1} = O_{\mathcal{A}_1} \cap O_{\mathcal{A}_2} = I_{\mathcal{A}_1} \cap I_{\mathcal{A}_2} = \emptyset$.

The parallel composition operator is based on the synchronization of inputs/outputs and on interleaving of all other unmatched actions:

Definition 8 (*Parallel composition*) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are two compatible timed input/output automata then their *composition* $\mathcal{A}_1 \parallel \mathcal{A}_2$ is defined to be the tuple $(X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ where:

- $X = X_{\mathcal{A}_1} \cup X_{\mathcal{A}_2}$ and $Clk = Clk_{\mathcal{A}_1} \cup Clk_{\mathcal{A}_2}$.
- $Q = \{\mathrm{x}_{\mathcal{A}_1} \cup \mathrm{x}_{\mathcal{A}_2} | \mathrm{x}_{\mathcal{A}_1} \in Q_{\mathcal{A}_1}, \mathrm{x}_{\mathcal{A}_2} \in Q_{\mathcal{A}_2}\}$. Note that $\mathrm{x}_{\mathcal{A}_1} \cup \mathrm{x}_{\mathcal{A}_2}$, which denotes the union of functions $x_{\mathcal{A}_1}$ and $x_{\mathcal{A}_2}$ is well defined since the domains of $\mathrm{x}_{\mathcal{A}_1}$ and $\mathrm{x}_{\mathcal{A}_2}$ are disjoint.
- $\theta = \theta_{\mathcal{A}_1} \cup \theta_{\mathcal{A}_2}$.
- $I = (I_{\mathcal{A}_1} \backslash O_{\mathcal{A}_2}) \cup (I_{\mathcal{A}_2} \backslash O_{\mathcal{A}_1})$, $O = (O_{\mathcal{A}_1} \backslash I_{\mathcal{A}_2}) \cup (O_{\mathcal{A}_2} \backslash I_{\mathcal{A}_1})$ and $V = V_{\mathcal{A}_1} \cup V_{\mathcal{A}_2} \cup (I_{\mathcal{A}_1} \cap O_{\mathcal{A}_2}) \cup (I_{\mathcal{A}_2} \cap O_{\mathcal{A}_1})$.
- $H = H_{\mathcal{A}_1} \cup H_{\mathcal{A}_2}$.
- D is the set of discrete transitions where for each $\mathrm{x} = \mathrm{x}_{\mathcal{A}_1} \cup \mathrm{x}_{\mathcal{A}_2}$, $\mathrm{x}' = \mathrm{x}'_{\mathcal{A}_1} \cup \mathrm{x}'_{\mathcal{A}_2} \in Q$ and each $a \in A$, $\mathrm{x} \xrightarrow{a} \mathrm{x}'$ if and only if for $i \in \{1, 2\}$, either
  1. $a \in A_{\mathcal{A}_i}$ and $\mathrm{x}_{\mathcal{A}_i} \xrightarrow{a}_{\mathcal{A}_i} \mathrm{x}'_{\mathcal{A}_i}$, or
  2. $a \notin A_{\mathcal{A}_i}$ and $\mathrm{x}_{\mathcal{A}_i} = \mathrm{x}'_{\mathcal{A}_i}$.
- $\tau \in \mathcal{T} \Leftrightarrow \tau\lceil X_{\mathcal{A}_i} \in \mathcal{T}_{\mathcal{A}_i}, i \in \{1, 2\}$.

The only difference between this definition and the one presented in [42] is related to the signature of the composite timed input/output automata: the input and output sets of actions consist in those that are not matched between components, while matched inputs/outputs become visible actions. By difference, in [42] matched inputs/outputs become outputs, which effectively means that outputs are treated as broadcasts and does not conform to the usual SysML semantics.

As we will show later in Sect. 4.2, although the semantics of inputs/outputs is synchronous, component models based in asynchronous communication such as that of SysML can be modeled with TIOA. This is done by using an internal *queue* variable and distinguishing between the *input* action which receives a message and stores it in the *queue*, and an internal *consumption* action which consumes the message from the queue at a later time.

The following theorem ensures that the parallel composition operator gives consistent results regardless of the order in which several components are composed:

Theorem 2 ($\mathcal{A}$, $\|$) *is a commutative monoid.*

The proofs of the theorems presented in this paper can be found in "Appendix 3".

Refinement As in [42], we use trace inclusion as the refinement relation between automata, but we limit our attention to *finite traces*, as we are interested in the satisfaction of safety properties for which we do not need infinite traces. However, as we will see, most results from the next section can easily be extended to fit a definition of refinement based on finite and infinite trace inclusion.

Definition 9 (*Comparable components*) Two timed input/output automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if they have the same signature, $E_{\mathcal{A}_1} = E_{\mathcal{A}_2}$.

Definition 10 (*Conformance*) Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two comparable timed input/output automata. $\mathcal{A}_1$ *refines* (*conforms to*) $\mathcal{A}_2$, denoted $\mathcal{A}_1 \preceq \mathcal{A}_2$, if $ftraces_{\mathcal{A}_1} \subseteq ftraces_{\mathcal{A}_2}$.

Note that we use the refinement relation between components also for checking *conformance* in the fourth step of the methodology for verifying that a top contract satisfies the requirement ($A \parallel G \preceq \varphi$). In the following we will use the term *conforms to* to denote the refinement of components relation. The following Theorems 3 and 4 presented in [42] and which are required by the meta-theory can be easily extended to our variant of TIOA.

Theorem 3 *The conformance relation is a preorder over a set of comparable components.*

Theorem 4 (Composability) *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two comparable timed input/output automata with $\mathcal{A}_1 \preceq \mathcal{A}_2$ and $\mathcal{B}$ a timed input/output automaton compatible with both $\mathcal{A}_1$ and $\mathcal{A}_2$. Then $\mathcal{A}_1 \parallel \mathcal{B} \preceq \mathcal{A}_2 \parallel \mathcal{B}$.*

4.2 From timed SysML models to TIOA

In this section, we discuss the transformation of a component-based system modeled with the SysML notions described in Sect. 3.1 into a timed input/output automata network. Our mapping rules are relatively straightforward and follow the same strategy that has been described in previous work like [43,46] or lately [48,51]; for this reason, we only give here an informal description of the mapping that we illustrate on the component from Fig. 8b—the assumption for the *cardUnit* of our sATM example. The fact that this component is part of a contract has no impact on the translation, the same mapping rules apply to system components and contract assumptions/guarantees. The complete formalization of this component is detailed in "Appendix 2." It is understood that, in order to have a fully formal approach, the mapping would have to be formalized, for example within a model transformation framework; however, this is a significant effort which goes beyond the scope of this paper.

The transformation proceeds by mapping each SysML component K to a timed input/output automaton $\mathcal{A}_K$. The set of clocks $Clk_{\mathcal{A}_K}$ consists in all the attributes defined by the component of type *Timer*, while all the other attributes form the set of discrete variables $X_{\mathcal{A}_K}$. In addition, each automaton $\mathcal{A}_K$ contains two implicit discrete variables: *location* models the current control state of the component as defined in its state machine and *queue* stores all incoming requests (inputs) prior to their handling by the automaton, which is modeled by an internal action consuming a message from *queue*. To ease definitions, the type of *queue* is not explicitly bounded, but the model checking method described in Sect. 6 works only if all queues of a model are bounded by construction, and our interactive simulation tool provides a method to easily detect situations where queues grow explosively. For the example from Fig. 8b, the set of discrete variables is made of the two predefined ones—location and queue previously described—while the set of clocks contains the defined Timer *t*.

In SysML, component types may model association and generalization relations. Associations are handled based on their end elements that represent attributes in the corresponding classes, while generalization between classes is flattened and all inherited attributes and association ends are duplicated in the automaton corresponding to the child class instance.

The set of states of the automaton $\mathcal{A}_K$ is given by the valuation of all variables, where the initial state θ_K contains the initial value for discrete variables (which can be defined in the model or predefined otherwise), the value 0 for clocks and $\emptyset$ for the message queue. For the *cardUnit*'s assumption, the initial state consists of *Idle* for the location, $\emptyset$ for the queue and 0 as clock *t* value.

The behavior of the component is modeled by its type's state machine that describes the transitions and trajectories of the automaton. A state machine transition is defined between a control source state s and a control target state s' on which we can evaluate a *guard* and execute several *effects*. A transition is usually enabled by a *trigger* or time delay deadline. Thus, for each state machine transition, a set of TIOA transitions is generated between two states q and q' where $q.location = s$ and $q'.location = s'$. The guard models the conditions for which the transition exists given that it is satisfied in the starting state q, otherwise no transition is generated. In each state of the automaton there is a predefined transition for each input action a. Its effect is to add the signal to the queue, i.e., $q'.queue = [q.queue; a]$. Then, a trigger m is transformed into a transition executing an internal action $\downarrow m$ that consumes the message m; thus, $s.queue = [m; a]$ and $s'.queue = a$. The set of effects defined on a transition can consist in several signal outputs and assignments.

For each effect an independent TIOA transition is generated. The signal sending action (or *sendAction*) becomes a transition with an output such that either the location in its target state is the target control state if there is only this effect modeled or an intermediate location is generated in case the effect is structured. This transition will synchronize with the input transition of the signal's target at composition and will modify the value of the queue. The *assignment* effect for discrete and clock variables is transformed into a TIOA transition that exists if and only if q' can be obtained from q by applying the assignment. As an example, from the state *WaitForRemoval* from Fig. 8b, the following set of transitions is generated for all the states q such that $q.location = WaitForRemoval$:

– corresponding to the *retrieveCard* message consumption, $\{q \xrightarrow{\downarrow retrieveCard} q'\}$ if and only if the message is at the top of the queue in state q, and the target state q' is obtained from q by changing location to $RemoveCard$ ($q.location = WaitForRemoval$ and $q'.location = RemoveCard$), removing the message from the top queue ($q.queue = [retrieveCard; q'.queue]$) and resetting the clock t ($q'.t = 0$), and
– corresponding to the input actions a, $\{q \xrightarrow{a} q'\}$ where q' is obtained from q by keeping the same location ($q.location = q'.location = WaitForRemoval$), the same value of the clock ($q.t = q'.t$) and adding the request a to the queue ($q'.queue = [q.queue; a]$).

From the state *RemoveCard*, the following set of transitions is generated for all the states q such that $q.location = RemoveCard$:

– $\{q \xrightarrow{a} q'\}$ where a is an input action, where the state stays the same and the request is added to the queue, as described before, and
– $\{q \xrightarrow{!cardRemoved} q'\}$ for the transition that sends *cardRemoved*, where q' changes the location ($q.location = RemoveCard$ and $q'.location = WaitForAck$), and the other two variables ($queue$ and t) are constant. As we will see below, this set of transitions is generated for all states where $q.t <= 5$, and 5 is the upper bound for the trajectories starting in q.

By default the time elapse in each state of the automaton is given by the set of all possible trajectories defined on $\{[0, t] | t \in \mathbb{R}^+\} \cup \{[0, \infty)\}$. This set of trajectories can be controlled by the urgency labels of the outgoing transitions from $s = q.location$ in the state machine as follows:

– *lazy* does not add any restrictions,
– *eager* with no clock guard restricts the set of trajectories to point trajectory only,
– *eager* with a clock guard restricts the set of trajectories so that they end in the smallest t where the guard is evaluated to true.

In our example from Fig. 8b, the set of trajectories enabled in a state q with $q.location = RemoveCard$ are all the trajectories defined for the domains $[0, x]$ with $x \in [0, 5 - q.t]$. Thus, the maximum length of the trajectories depends on the initial value of t in q, $q.t$, such that the maximum value for t reachable after a trajectory is 5. This is due to the *eager* transition with the clock guard $t == 5$ which only lets the time elapse up to 5 and enforces the output of the signal *cardRemoved*. For the states q with $q.location = WaitForRemoval$, two possibilities exist: either the request *retrieveCard* is present as the top of $q.queue$, in which case the set of trajectories contains only the point trajectory, or the request *retrieveCard* is not present as the top of the queue, in which case the trajectories let time elapse for any duration up to infinity.

In SysML, the set of signals that can be exchanged through a *port* are defined by its required/provided interface, and the set of signals exchanged by a component is the union of those of its ports. Thus, the sets of input actions $I_{\mathcal{A}_K}$ and output actions $O_{\mathcal{A}_K}$ of the automaton $\mathcal{A}_K$ are derived from the types of the ports of K. In the case of the running component, the set of inputs is made of $\{retrieveCard, ok, nok\}$ and the set of outputs is $\{cardInserted, ejectCard, cardRemoved\}$. The set of visible actions for an automaton mapped from an atomic component is the empty set, $V_{\mathcal{A}_K} = \emptyset$. The set of internal actions $H_{\mathcal{A}_K}$ consists in the set of message consumption actions $\downarrow m, \forall m \in I_{\mathcal{A}_K}$, plus a set of instantiations of anonymous silent actions ε corresponding to transitions fragments from the SysML state machine which do not have a trigger and do not perform an output.

A particular attention must be brought to the name of signals since a model usually contains several components of the same type and they react to the same stimuli, in contradiction with the compatibility condition. Our solution is to rename conflicting signals in the sender/receiver automata by appending their qualified name and the traveled chain of ports that can be statically computed via connectors. Then, if the receiving automaton handles a signal that has multiple senders, the transition that handles the signal is duplicated for each sender where the trigger contains the newly defined names of the signal.

Finally, a composite component is translated into an automaton obtained by applying the parallel composition operator on its parts. The set of input and output actions computed at composition must be identical to those defined by the ports of the component.

The requirement formalization given by an observer in our framework is also transformed into a TIOA by applying the same rules. The difference consists in the fact that an observer does not have any inputs or outputs, all its actions being defined as visible. A transition is typed with a visible action if it is preceded by a send clause and with an internal action $\downarrow a$ if it is preceded by an acceptsignal clause. In each state, the automaton defines visible transitions that add input signals to the queue. The added signal can be handled later on the consumption transitions. The timed semantics is defined as lazy for transitions resulting from send match clauses and as eager for the rest. This timed semantics formalizes a safety property.

As it can be seen, also from the detailed example formalization from "Appendix 2", the TIOA is an explicit representation of the (potentially infinite and time-dense) transition system, and although it is practical for reasoning and proving results such as the ones from the next section, it is ill suited for manipulating translations of real system components. For this reason, in our tools we represent TIOA symbolically in a timed automata-based language, IF [14].

5 Formal contract-based theory for timed systems

So far we have completely defined—by syntax and semantics—the component framework for which we want to use contract-based reasoning. In this section, we build the formal contract framework on top of the component theory and we show that this instantiation of the meta-theory from Sect. 2 can be applied for system models by proving the satisfaction of the required compositionality results. In this section, the term component is used to designate a TIOA.

5.1 Contract theory for TIOA

Contracts have been introduced in SysML in Sect. 3 where we have defined their syntax. In order to use them for the behavioral verification of a requirement, we have to define their semantics and several supporting concepts.

Definition 11 (*Environment*) An *environment* Env for a component K is a timed input/output automaton compatible with K and for which the following hold: $I_{Env} \subseteq O_K$ and $O_{Env} \subseteq I_K$.

Definition 12 (*Closed/open component*) A component K is *closed* if $I_K = O_K = \emptyset$. A component is *open* if it is not closed.

Closed components result from the composition of components having complementary interfaces, like it is often the case between a component and its environment. However, Definition 11 is relaxed and also includes *partial* environments. This allows to reason independently and compositionally also about a component's modeled environment based on the system architecture, e.g., by integrating in the component under study components that are part of its environment.

Definition 13 (*Contract*) A *contract* for a component K is a pair (A, G) of timed input/output automata such that:

- their composition gives a closed system, i.e., $I_A = O_G$ and $I_G = O_A$, and
- the signature of G is a subset of that of K, i.e., $I_G \subseteq I_K$, $O_G \subseteq O_K$ and $V_G \subseteq V_K$.

In the following, we use the term *signature of a contract* to designate the signature of the contract's guarantee.

Contract satisfaction has been introduced in Definition 2 based on a *refinement under context* relation. In our contract framework refinement under context at its turn relies on the conformance relation introduced at Definition 10. Since we are interested in allowing the concrete component K_i to have a larger signature than the abstract K_j and conformance can be defined only between comparable components, we have to compose each of the members of the obtained conformance relation with additional timed input/output automata such that the compatibility condition is satisfied.

Definition 14 (*Refinement under context*) Let K_1 and K_2 be two components such that $I_{K_2} \subseteq I_{K_1} \cup V_{K_1}$, $O_{K_2} \subseteq O_{K_1} \cup V_{K_1}$ and $V_{K_2} \subseteq V_{K_1}$. Let Env be an environment for K_1 compatible with both K_1 and K_2. We say that K_1 *refines* K_2 *in the context of* Env, denoted $K_1 \sqsubseteq_{Env} K_2$, if

$$K_1 \parallel Env \parallel Env' \preceq K_2 \parallel Env \parallel K' \parallel Env'$$

where K' and Env' are defined such that both members of the conformance relation are closed and comparable, as follows:

- $Env' = (\emptyset, \emptyset, \{\phi\}, \phi, (O_{K_1} \setminus I_{Env}), (I_{K_1} \setminus O_{Env}), \emptyset, \emptyset, \mathcal{D}_{Env'}, \mathcal{T}_{Env'})$ where ϕ is the valuation without variables $(\emptyset \to \emptyset)$, $\mathcal{D}_{Env'} = \{(\phi, a, \phi) | \forall a \in E_{Env'}\}$ and $\mathcal{T}_{Env'} = \{\tau : [0, t] \to \{\phi\} | t \in \mathbb{R}^+\} \cup \{\tau : [0, \infty) \to \{\phi\}\}$ contains all possible trajectories without variables.
- $K' = (\emptyset, \emptyset, \{\phi\}, \phi, ((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})), ((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_2})), (V_{K_1} \setminus E_{K_2}), \emptyset, \mathcal{D}_{K'}, \mathcal{T}_{K'})$ where ϕ is the valuation without variables $(\emptyset \to \emptyset)$, $\mathcal{D}_{K'} = \{(\phi, a, \phi) | \forall a \in E_{K'}\}$ and $\mathcal{T}_{K'} = \{\tau : [0, t] \to \{\phi\} | t \in \mathbb{R}^+\} \cup \{\tau : [0, \infty) \to \{\phi\}\}$ contains all possible trajectories without variables.

Informally, Env' is a partial environment having as signature the actions of the concrete component K_1 that are not present in the actions of Env with reversed directionality such that $K_1 \parallel Env \parallel Env'$ is a closed component. K' is a component that reacts to the actions defined as the set difference between the signatures of K_1 and the abstract

K_2 such that K_1 and $K_2 \parallel K'$ are comparable. Furthermore, these definitions verify that both $K_1 \parallel Env \parallel Env'$ and $K_2 \parallel K' \parallel Env \parallel Env'$ are closed comparable components. Their behavior is such that all actions are enabled at any moment and time can always elapse up to infinity.

The particular inclusion relation between the signatures of K_1 and K_2 in the definition is due to the fact that both K_1 and K_2 can be obtained from composition: $K_1 = K_1' \parallel K_3$ and $K_2 = K_2' \parallel K_3$, where $I_{K_2'} \subseteq I_{K_1'}$, $O_{K_2'} \subseteq O_{K_1'}$ and $V_{K_2'} \subseteq V_{K_1'}$. This happens in particular when K_2' is a contract guarantee for K_1'. Then, by composition, actions of K_3 may be matched by actions of K_1' but have no input/output correspondent in K_2'. This case also imposes the term $V_{K_1} \cap O_{K_2}$ for inputs of K', since the additional outputs of K_2 may belong to a different component, and the term $V_{K_1} \cap I_{K_2}$ for the outputs of K'.

Note Refinement under context can be easily extended to infinite traces by replacing, within the trace inclusion operator $\preceq$, $ftraces$ with $traces$. We denote the extended refinement under context operator with $\sqsubseteq^{\omega}$.

This definition of refinement under context satisfies the conditions described in Sect. 2 required for the meta-theory to hold, as shown in the following:

Theorem 5 *Given a component Env and a set $\mathcal{K}$ of components for which Env is an environment, the refinement under context $\sqsubseteq_{Env}$ is a preorder over $\mathcal{K}$.*

The following theorem that allows for *incremental design* holds in our framework:

Theorem 6 *(*Compositionality*) Let K_1 and K_2 be two components and E an environment compatible with both K_1 and K_2 such that $Env = Env_1 \parallel Env_2$. Then $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1$.*

The proofs of the previous two theorems can be found in "Appendix 3."

The soundness of *circular reasoning* is the main result that guarantees the correctness of the contract-based reasoning.

Theorem 7 (Circular reasoning) *Let K be a component, Env its environment and $\mathcal{C} = (A, G)$ a contract for K such that K and G are compatible with both Env and A. If*

1. *$ftraces_G$ is closed under time extension,*
2. *$K \sqsubseteq_A G$ and*
3. *$Env \sqsubseteq_G A$*

then $K \sqsubseteq_{Env} G$.

Proof sketch (the complete proof is found in [36]*)*:

The proof is built by induction for every closed trace of K.

1. If a closed trace α consists in a trajectory, from axiom A0 and closure under time extension, α is also a trace of G.
2. If a closed trace α is extended by an external action a, by induction α is a trace of the right-hand side member that can be extended by a executed either by G or one of the additionally generated automata. □

Note This theorem holds also for infinite traces by replacing $\sqsubseteq$ with $\sqsubseteq^{\omega}$, if in addition to the hypotheses we require $traces_G$ to be closed under limits and under time extension. The complete proof from [36] is provided for infinite traces and follows the same reasoning as *Theorem 8.8* from [42] to which it is similar. The difference with respect to [42] is that we prove that $K \parallel Env \preceq G \parallel Env$, while *Theorem 8.8* states that $K \parallel Env \preceq G \parallel A$, which is not sufficient for guaranteeing soundness of circular reasoning.

Based on the previous results, we can now establish the *sufficient condition for dominance* to be used in the "dominance" step of the reasoning methodology. The following theorem is a variant of Theorem 1, with the difference that, where the hypothesis of Theorem 1 was that circular reasoning is sound, this hypothesis is rewritten in Theorem 8 as "guarantees must be closed under time-extension", which, by Theorem 7, ensures that circular reasoning is sound. Note that, although Theorem 1 was already proved in [61] (*Theorem 2.3.5* therein), since that proof uses different notations and a quite different notion of composition operator, we feel compelled to restate the proof in "Appendix 3" with our own notions and notations. The proof technique is, however, fully inspired from [61].

Theorem 8 *$\{C_i\}_{i=1}^{n}$ dominates C if, $\forall i$, $traces_{G_i}$ and $traces_G$ are closed under time extension and*

$$\begin{cases} G_1 \parallel \ldots \parallel G_n \sqsubseteq_A G \\ A \parallel G_1 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i \end{cases}$$

Note This theorem holds also for infinite traces by replacing $\sqsubseteq$ with $\sqsubseteq^{\omega}$ and by requiring $traces_G$ and $traces_{G_i}$, $\forall i$, to also be closed under limits.

5.2 Contract expressiveness for timed SysML models

This theory can be applied on system models extended with contracts if the component playing the role of the guarantee satisfies the *closure under time extension* constraint. Therefore, we discuss here the restrictions that are imposed on the language for modeling contracts by this constraint.

Informally, closure under time extension lets time to elapse in any state of the automaton. An easy way to achieve time progress in all states of a state machine is to stereotype all outgoing transitions from the control states as *lazy*. However, this is not necessary for all types of transitions, and it is actually sufficient that transitions which perform an *output* be *lazy*. This allows for guarantees to specify more precise time progress constraints, e.g., by specifying transitions consisting only of internal actions (internal computation actions, internal signal consumption actions $\downarrow$) as *eager*.

Indeed, this setting of urgency is sufficient to ensure closure under time extension, provided that the state machine is non-Zeno, i.e., does not execute an infinity of internal actions in a finite time. This is because *eager* transitions are executed as soon as they are enabled thus eventually leading either to a state where an *output* may occur or to a final user-defined state. Transitions with an output are *lazy*, and therefore, time can progress to infinity. The case of final user-defined states is similar since they are termination states or states without outgoing transitions and time may progress to infinity.

We remark that this restriction does not allow our guarantees to specify that a certain output *!o will eventually occur*, but they can specify hard deadlines such as "if the output !*o* occurs, it will occur before a time instant T." Technically, the former type of property is a liveness property, while the latter is merely a safety property, although such properties are sometimes called *bounded liveness properties*. For this reason, we emphasize that our contract framework allows to specify and verify only *safety* properties, albeit these can be *timed safety* properties and thus quite expressive.

5.3 Application to the running example

For the example described in Sect. 3.5, the contract satisfaction step generates the following relations:

1. $controller \sqsubseteq_{aCtr} gCtr$ and
2. $cardUnit \sqsubseteq_{aCardUnit} gCardUnit$.

For the dominance step, since *G_ATM*, *G_Controller* and *G_CardUnit* satisfy the closure under time extension condition, we apply Theorem 8 and we obtain the following proof obligations:

3. $gCtr \parallel gCardUnit \sqsubseteq_{aATM} gATM$,
4. $aATM \parallel gCardUnit \sqsubseteq_{gCtr} aCtr$ and
5. $aATM \parallel gCtr \sqsubseteq_{gCardUnit} aCardUnit$.

We remark that output actions (e.g., *!ejectCard*, *!ok*) have a lazy semantics within the guarantees to ensure their closure under time extension, as explained above in Sect. 5.2. Since, within the dominance conditions, a guarantee composed with the global assumption $aATM$ must refine the assumption of the opposite component, it explains why output actions also have a *lazy* semantics in assumptions in Figs. 7b and 8b.

Note that outputs of the global environment (e.g., *!cardRemoved*) are not subject to the same restrictions when captured in the assumptions, as they do not appear in the guarantee of any component.

The satisfaction of the "mirror" *C_ATM* generates the next proof obligation:

6. $user \sqsubseteq_{gATM} aATM$

We remark that $aATM$ is a loose abstraction of the behavior of the $user$, as the *!cardRemoved* signal is assumed to arrive within a maximum of 5 time units. This is implied by the two transitions from state *RemoveCard*: the example shows how a couple of *lazy*/*eager* transitions are used to model a transition that may occur anywhere within a bounded interval, [0, 5] in this case.

The last proof obligation corresponds to the conformance step:

7. $aATM \parallel gATM \preceq Property$

6 Automatic verification of generated proof obligations

The contract theory we defined is based on the trace inclusion relation. However, this relation is undecidable in the general case and cannot be automatically verified by tools except for restricted categories of timed automata [57,66]. Two options are available to automatically verify refinement under context: either by making additional hypotheses on the form of the right-hand side (the abstract component), which allow one to use reachability analysis for guaranteeing trace inclusion or by using timed simulation [65] which also guarantees trace inclusion. In the following, we describe our technique for the automatic verification based on reachability analysis for property automata, which relies on the fact that the abstract component represents a deterministic safety property. This method is implemented in the IFx toolset [14] which allows to verify and simulate asynchronous communicating timed automata.

6.1 A method of automatic verification of refinement relations

Our method uses the notion of property automata. A *property automaton* is a complete definition of a safety requirement for a closed component $\mathcal{C}$: it defines an *error* state π to which incorrect behaviors will lead and synchronizes with $\mathcal{C}$ on common actions. The reasoning for proving contract satisfaction proceeds as follows: (1) transform the guarantee into a property automaton, (2) run in parallel the component $\mathcal{C}$ and the property automaton, and (3) explore the final state graph to check if the error state π has been reached. Reaching the error state π signifies the violation of the contract satisfaction.

We start by defining the transformation process from a deterministic guarantee automaton to a property automaton. The idea is similar to the one defined in [16] and later used for automated assume–guarantee reasoning in the LTSA tool [11,38], and is based on the classical method for constructing the complement of a deterministic timed automaton [3]. Note that the requirement of determinism is necessary because non-deterministic timed automata are not determinizable in general and are not closed under complement [3].

Definition 15 (*Property automaton*) Given a *deterministic* TIOA $\mathcal{A} = (X_{\mathcal{A}}, Clk_{\mathcal{A}}, Q_{\mathcal{A}}, \theta_{\mathcal{A}}, I_{\mathcal{A}}, O_{\mathcal{A}}, V_{\mathcal{A}}, H_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}, \mathcal{T}_{\mathcal{A}})$, the *property automaton for* $\mathcal{A}$ is defined as the TIOA $\mathcal{O}_{\mathcal{A}} = (X_{\mathcal{A}}, Clk_{\mathcal{A}}, Q, \theta_{\mathcal{A}}, \emptyset, \emptyset, V, H_{\mathcal{A}}, \mathcal{D}, \mathcal{T}_{\mathcal{A}})$ where:

- $Q = Q_{\mathcal{A}} \cup \{\pi\}$, where π is an additional error state,
- $V = I_{\mathcal{A}} \cup O_{\mathcal{A}} \cup V_{\mathcal{A}}$,
- $\mathcal{D} = \mathcal{D}_{\mathcal{A}} \cup \{(\mathrm{x}, a, \pi) | \mathrm{x} \in Q_{\mathcal{A}}, a \in V$ such that $(\nexists \mathrm{x}'. (\mathrm{x}, a, \mathrm{x}') \in \mathcal{D}_{\mathcal{A}}) \wedge (\nexists \varepsilon \in H_{\mathcal{A}} \wedge \mathrm{x}' \in Q_{\mathcal{A}}.(\mathrm{x}, \varepsilon, \mathrm{x}') \in \mathcal{D}_{\mathcal{A}})\}$.

The idea behind this transformation is that sequences of actions that are not explicitly modeled should be considered as erroneous behaviors. Since a property automaton is defined for a closed component, we consider the signature of $\mathcal{O}_{\mathcal{A}}$ to contain only visible actions, corresponding to the inputs, outputs and visible actions of $\mathcal{A}$. Then, in every state of the automaton from which there is no outgoing internal transition, we complement the set of transitions with those missing: for each visible action there must be a discrete transition either leading to a state defined in $\mathcal{A}$ or to π. So, the actions leading to π model the discrete actions that are not allowed to occur in a given timed sequence of $\mathcal{A}$.

However, for this method to work the component $\mathcal{A}$ must be a deterministic safety property both for visible actions and for internal actions. For internal actions, determinism means that there is at most one outgoing transition from a state. We remark that these conditions have to hold in the TIOA framework. It implies that in a SysML state machine, one is still able to model several outgoing internal transitions given that they are not to be enabled at the same time, e.g., two transitions having disjoint guards.

The synchronization at run-time between $\mathcal{C}$ and the property automaton $\mathcal{O}_{\mathcal{A}}$ is defined by the following composition operator, denoted $\bowtie$. It is similar with the previous parallel composition operator described in Definition 8 with synchronization on the common visible actions and interleaving of the others. The operator can be applied on two timed input/output automata if they do not share any internal actions (by label) and they do not exhibit any inputs/outputs. The latter condition is motivated by the fact that the property automaton always surveys a closed component.

Definition 16 (*Observer composition*) Let $\mathcal{A}_1$ be a closed component and $\mathcal{A}_2$ a property automaton such that $E_{\mathcal{A}_2} \subseteq A_{\mathcal{A}_1}$. Then $\mathcal{A}_1 \bowtie \mathcal{A}_2 = \mathcal{A}_1 \parallel \mathcal{A}_2$ where the compatibility condition (see Definition 7) is relaxed, the only remaining constraint being that $H_{\mathcal{A}_1} \cap H_{\mathcal{A}_2} = \emptyset$.

The following result links the unreachability of the error state of an observer with the trace inclusion relation. We denote the set of all reachable states of the automaton by $reach(\mathcal{A}) \subseteq Q$.

Theorem 9 *$K_1 \sqsubseteq_E K_2$ if K_2 is a deterministic safety property and $reach((K_1 \parallel E \parallel E') \bowtie \mathcal{O}_{K_2}) \cap \{\pi\} = \emptyset$.*

The proof of this theorem can be found in "Appendix 3."

6.2 Verifying compliance to assumptions

The fact that the abstract TIOA has to be deterministic is a limitation of this verification method that has to be taken into account in the methodology. The limitation is usually not problematic for verifying contract satisfaction as safety guarantees have to be expressed as time- and limit-closed TIOA and they can often be determinized. However, in order to establish dominance, one has to verify also "mirror" contract satisfaction, which is more problematic since we do not require assumptions to be safety properties. In consequence, modeling assumptions as deterministic safety properties becomes necessary for using model checking in combination with timed property automata on all proof obligations.

When the assumptions cannot be described using deterministic safety properties, there are two solutions which may open up the possibility of automatically verifying proof obligations. The first option is to use as assumption the component's actual environment, if its model is available (i.e., if the modeled system is closed). In this case, the proof obligations concerning "mirror" contract satisfaction (second hypothesis of Theorem 8) become trivial, as the left and right-hand side members of a satisfaction relation are identical, and the dominance step sums up to verifying only the refinement of the global guarantee by individual ones in this concrete environment (first hypothesis of Theorem 8). This situation is exemplified on the case study from Sect. 7. Note that this case also allows using more expressive assumptions which could include non-deterministic behavior. The second option could be to verify timed simulation, which implies trace inclusion and is decidable for certain classes of automata, but for which we lack tool support for the time being. Note that replacing timed trace inclusion with a simulation relation could, in addition, relax the constraint of closure under time extension for guarantees. This idea constitutes future work.

6.3 Error diagnostic for contract-based reasoning

Within the proof obligations set, one or even several checks may not be satisfied. In this case, we have to perform a diagnosis in order to establish if either the requirement is not satisfied or the set of defined contracts needs to be refined in order to prove the satisfaction of the requirement. We base this diagnosis on the generation of a counterexample which for the previous method will lead to the error state π and use the same approach as for counterexample guided abstraction refinement [22].

We can distinguish two cases for which the solution depends on whether the reasoning has been applied for design or for verification: (1) a contract satisfaction or dominance verification fails or (2) the conformance verification fails. For the first case, if we are in a design approach and all previous steps have been proved correct, we have to refine the source component/contracts such that the counterexample is eliminated. This guarantees that the developed components are correct by construction with respect to the requirement. If we are in a verification approach with a completely modeled system, one should refine the target contract(s) since it is more frequent that the abstraction is erroneous.

For the latter case, we have at first to verify on the concrete system if the generated counterexample is a spurious one due to the abstractions defined or is a relevant one, which means that the system does not satisfy the requirement. For a spurious counterexample, one should refine the top contract and re-verify at least the upper dominance step and the “mirror” contract satisfaction. Possibly, iterations of refinement of contracts must be performed until all checks pass. For a correct counterexample, the modeler should redesign the implementations i.e., leaf components, on which the requirement is expressed. For this, the contract-based reasoning can be applied in a design approach in order to refine the correct contracts that satisfy the relations they are involved in, toward correct implementations.

We remark that, in the case of our tools, the generated counterexample is expressed on the TIOA level, while the refinement of contracts/components is specified in a high-level modeling language. The bridge between the two frameworks is unidirectional, since the transformation presented above is given from SysML to TIOA. In order to exploit the error scenario, the developer has to apprehend in detail the system and to make use of his experience for performing refinement. This point is an open question for which current research provides some options [2,24] but is out of the scope of this paper.

7 The solar generation wing management case study

This section presents the Solar Generation Wing Management System (SGS) of the automated transfer vehicle (ATV) and its verification and validation with the contract-based reasoning technique. The ATV, developed by Airbus Defence and Space[3] (ADS) is a space cargo ship launched into orbit by the European Ariane-5 launcher with the aim of resupplying the International Space Station. The SGS system described here is responsible for the management of the solar arrays that provide the vehicle with the energy needed to fulfill its mission. It contains the functional chains that perform the solar arrays deployment and rotation.

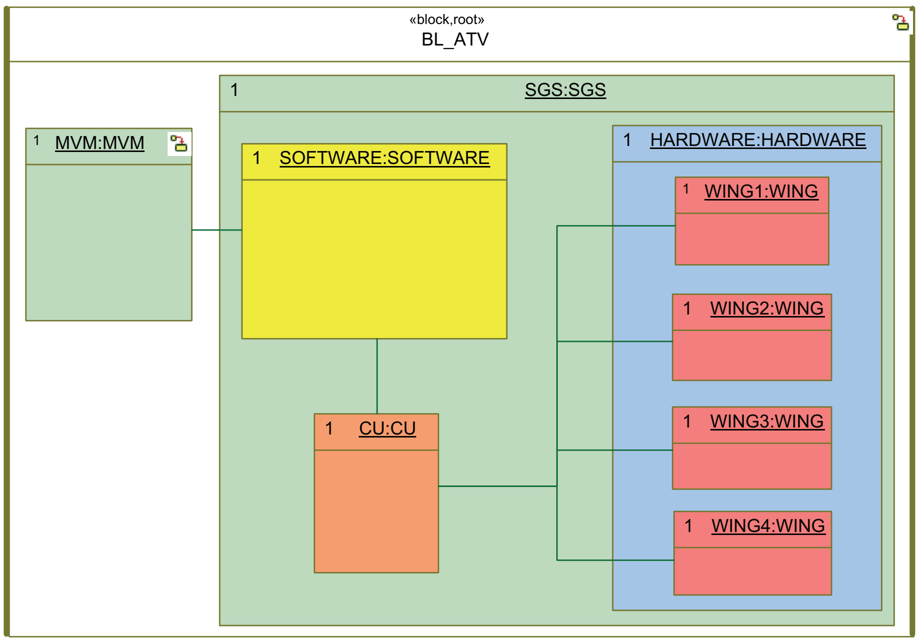
7.1 System description and architecture

The model represented in Fig. 10 has been reverse engineered from the actual system by the ADS engineers, for the purpose of this case study. It is described in SysML within the IBM Rhapsody tool. The model has a 4-layer architecture structured in a set of hardware and software entities that capture its timed behavior. Figure 10 adopts a high-level view of the main components, without the details of their substructures:

- the mission and vehicle management component (*MVM*) that “simulates” a mission scenario going through the two operating modes of the SGS that are described below.
- the *SOFTWARE* component that consists of three subcomponents, each with a specific function. They react to requests coming from the *MVM* and control the hardware by executing automated procedures in response to *MVM* demands.
- the *HARDWARE* component that contains the four solar arrays of the ATV. This component has more than 70 pieces of equipment with multiple levels of redundancy for achieving reliability and availability in case of failures. Every wing is held in its initial position by four hold-down and release systems (HDRS). In order for the deployment to occur, each HDRS has to be set loose. This is realized by eight thermal knives (TK) for each wing, two for each HDRS, one for the nominal case, and a redundant one in case of anomaly. Each time a HDRS is cut, a wing locking mechanism evolves to a deployed state for that array.
- the command units (*CU*) component which may also be subject to failure. It has numerous interconnections both nominal and redundant with the wings, connections that are abstracted to 4 in Fig. 10. It contains 4 power units (PCDU) and 4 thermal control units (TCU) that are responsible for the enabling/deactivation of the TKs,

[3] http://www.astrium.eads.net/.

Fig. 10: An overview of the SGS model in Rhapsody SysML



each of them being connected to two different wings. Two command and monitoring units (CMU) supervise the entire system, i.e., all requests from the software transit the CMUs.

The SGS describes two operating modes: (1) the deployment of the solar arrays and (2) their rotation. We are interested here only in the first mode. Initially, the four solar arrays of the ATV are stowed. Their deployment starts by removing the safety barriers from the thermal control units. Safety barriers prevent an unwanted unfolding of the wings by blocking the enabling of the thermal knives. Next the HDRSs are cut by at least 4 of the 8 thermal knives of each wing. In order for a HDRS to be cut, the knife has to be active for 50 consecutive seconds. The deployment of the wing starts immediately after the last HDRS is cut. After the deployment is completed, the safety barriers are restored.

The system's redundancy, if an anomaly occurs at execution, is explicitly modeled for the TKs and HDRSs of each wing, TCUs and PCDUs. There are 56 possible failures and each may occur at an arbitrary moment during execution. The hypothesis is that the system may be subject to at most one failure, i.e., 1-fault tolerance. In order to ease the generation of verification configurations, a special *SIMULATION* component is added to the model to command non-deterministically the failure of an equipment based on a parameter that can be provided prior to the verification session.

In terms of metrics, the model defines a total of 21 block types (7 of which are refined by means of 24 Internal Block Diagrams) with 348 port types and 372 connector types for communication. At run-time, the system contains 96 block instances running in parallel with a total of 651 ports and 504 connectors.

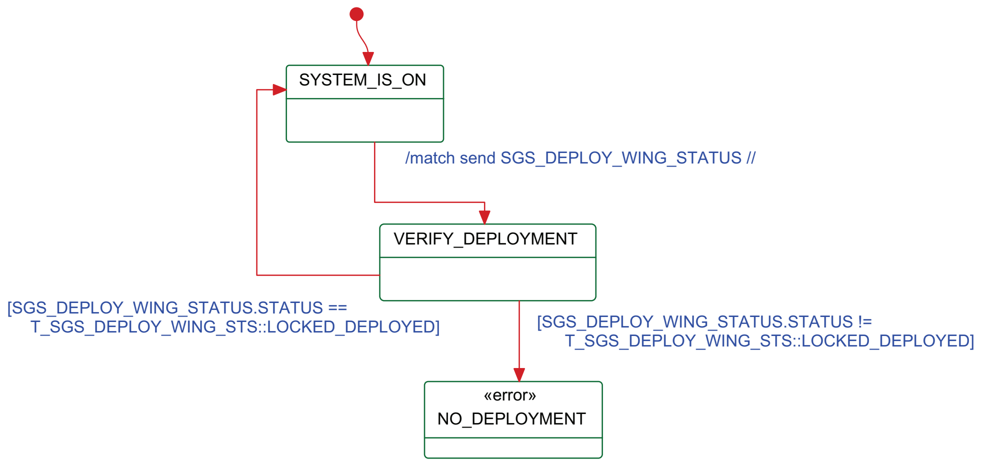
We are interested in proving that the system is indeed 1-fault tolerant. Informally, it means that no matter what error occurs to equipment devices and at what moment, the software will attain its purpose—the correct deployment of the wings. It is expressed by the following requirement modeled in Fig. 11.

Requirement 2 *At the end of the deployment sequence, all four wings are deployed.*

We formalize this requirement also with an observer. We add to the system model a block *phi* whose state machine describes the safety property to be verified: initially we wait in the state *SYSTEM_IS_ON* for the wing status interrogation to be executed. After the entire sequence deployment is executed, a software piece verifies the locked status of the wings. When asked, the target wings answer with a *SGS_DEPLOY_WING_STATUS* message that has as parameter its current status. When the action is matched, the automaton passes into the *VERIFY_DEPLOYMENT* state where it checks the value of the parameter. If it is *LOCKED_DEPLOYED*, then it will wait for another occurrence of the interrogation for another or the same wing. Otherwise, something wrong has occurred at deployment and it advances to the error state *NO_DEPLOYMENT*. The reaching of the error state during verification means that the requirement is violated.

7.2 Preliminary verification results without contracts

We start by reviewing the system model and performing some preliminary verification and validation in order to detect



SYSTEM_IS_ON
/match send SGS_DEPLOY_WING_STATUS //
VERIFY_DEPLOYMENT
[SGS_DEPLOY_WING_STATUS.STATUS == T_SGS_DEPLOY_WING_STS::LOCKED_DEPLOYED]
[SGS_DEPLOY_WING_STATUS.STATUS != T_SGS_DEPLOY_WING_STS::LOCKED_DEPLOYED]
«error»
NO_DEPLOYMENT

Fig. 11: SysML formalization of the Requirement 2 modeling that all four wings are deployed

modeling errors that may lead to the violation of the requirement. In this, we do not yet make use of contracts. The results presented here pre-date the work on contracts and are thoroughly described in [34]. Yet we chose to review them here briefly, since they participated in eliminating some errors in the model and in showing that directly model checking Requirement 2 is not feasible.

Interactive simulation of nominal scenarios and execution of random scenarios allowed us to discover several modeling errors. Most of them concerned unexpected message receptions that blocked the execution of components thus leading to general deadlocks, as it was the case for TKs. Message receptions had to be modeled for correct behavior. Other modeling errors were due to the intricate behaviors and the incomplete specification of the system. An examined system scenario has around 2400 transitions fired and needs around a minute to be executed on a regular desktop machine.

We also inspected formally the system model for the absence of deadlocks. A deadlock may also occur due to incorrect interpretations of the model that lead to unrealistic behaviors as it was the case for the *MVM*.

Yet, performing model checking on the current configuration is not possible due to the combinatorial explosion of the state space. This is caused by the large number of component instances at run-time, which can be seen in the communication graph represented in Fig. 12. As a first way around the explosion problem, we used in the beginning a non-exhaustive exploration by limiting concurrency in the system to two threads, one for the *SIMULATION* component and one for all the other components. This allowed us to discover several missing transition for TKs and, most importantly, incorrect connections between the PCDU and the wings. Each PCDU was erroneously connected to the same wing by both connections, while it had to be nominally connected to one wing and redundantly connected to another wing.

Once corrections were made to the model, the exploration of the state space in the 2-thread configuration produced no further errors. However, this is not sufficient to establish the satisfaction of the requirement in the general case. For this reason, we set out to use contract-based reasoning, which is described in the following section.

7.3 Applying the contract-based verification technique

We start by identifying the components that represent the system under study S and the environment E. Since Requirement 2 is expressed with respect to the behavior of the four wings that are contained in the *HARDWARE* block, with regard to the methodology of Fig. 1, we consider the subsystem S to be the *HARDWARE* and the K_i the *WINGi*, $i = \overline{1,4}$. The environment of the subsystem is given by the parts with which it communicates: bidirectional communication is established between *CU* and *HARDWARE*, while *CU* depends on the behavior of *SOFTWARE* and *MVM*. So, the environment E of Fig. 1 is represented here by the composition of *MVM*, *SOFTWARE* and *CU*. The application methodology on the SGS system model is illustrated in Fig. 13.

Next, we define a contract $C\_Wi = (A\_Wi, G\_Wi)$ for each *WINGi* and prove that *WINGi* satisfies $C\_Wi, i = \overline{1,4}$. We chose for *WINGi* to use as assumption the concrete environment of the subsystem *HARDWARE* composed with an abstraction *WAj* for each *WINGj* with $j \neq i$. We propose the following abstraction *WAj*: the wing consumes all requests coming from the environment, and answers to any status request with *deployed*. Then, the assumption A_i is given by the parallel composition of *MVM*, *SOFTWARE*, *CU* and *WAj* with $j \neq i$. This abstraction of the environment is sufficient to drastically reduce the state space of the verification model, since the exponential explosion in the original model is mainly due to the parallelism of the hardware pieces which are abstracted to the three leaf parts *WAj*. We want to guarantee that even if *WINGi* exhibits a failure it ends up being deployed. The contract C_Wi is defined as follows:

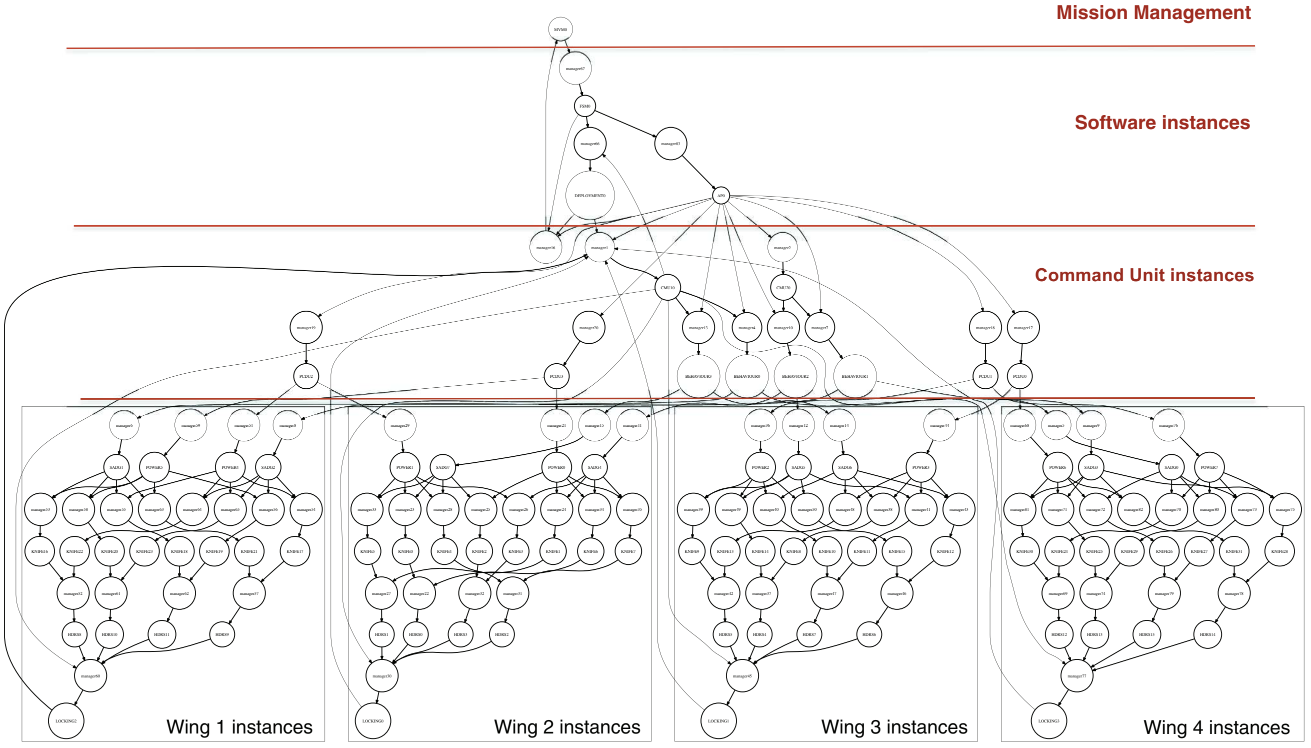


Fig. 12: System's communication graph displaying the components—represented as nodes—and their unidirectional communication—represented as *arrows*

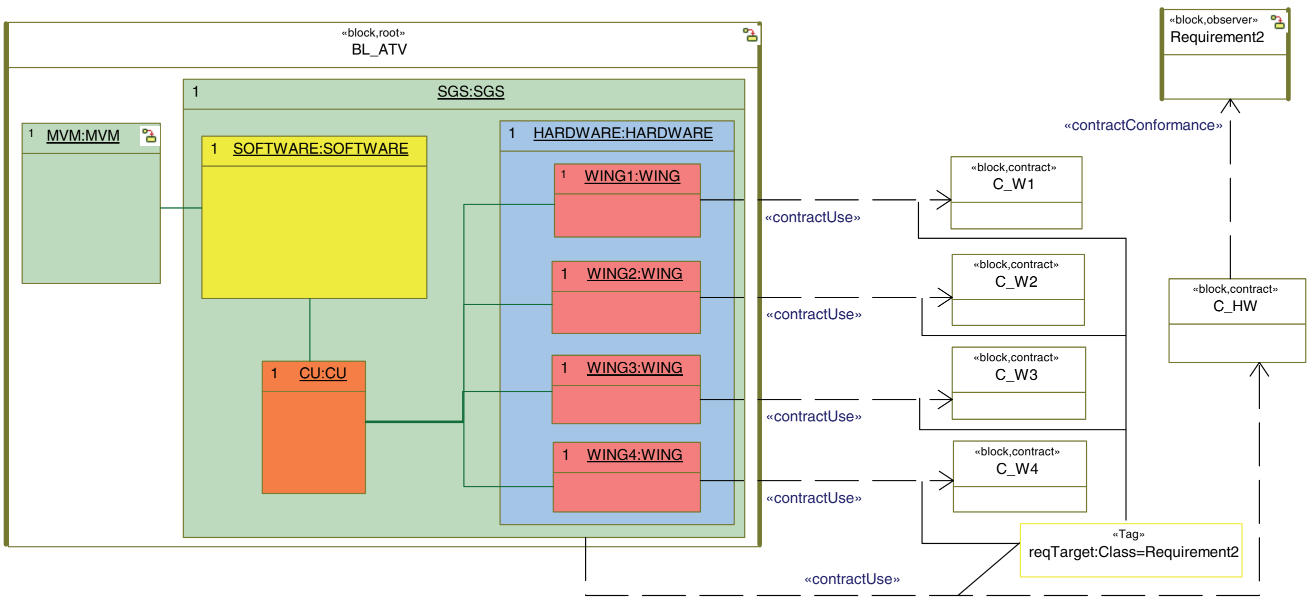


Fig. 13: The SGS model extended with contracts for verifying Requirement 2

Contract $C\_Wi = (A\_Wi, G\_Wi)$ where:

– $A\_Wi = MVM \parallel SOFTWARE \parallel CU \parallel (\parallel_{j \neq i} WAj)$.
– $G\_Wi = WAi$: the wing answers to requests about its status with *deployed* and ignores all other requests.

The contract is modeled in Fig. 14, while Fig. 15 presents the behavior of the guarantee. We note that since we use as assumption the concrete environment, the signature of the guarantee remains the same as that of the component. For this reason, we have to add consuming transitions in every state

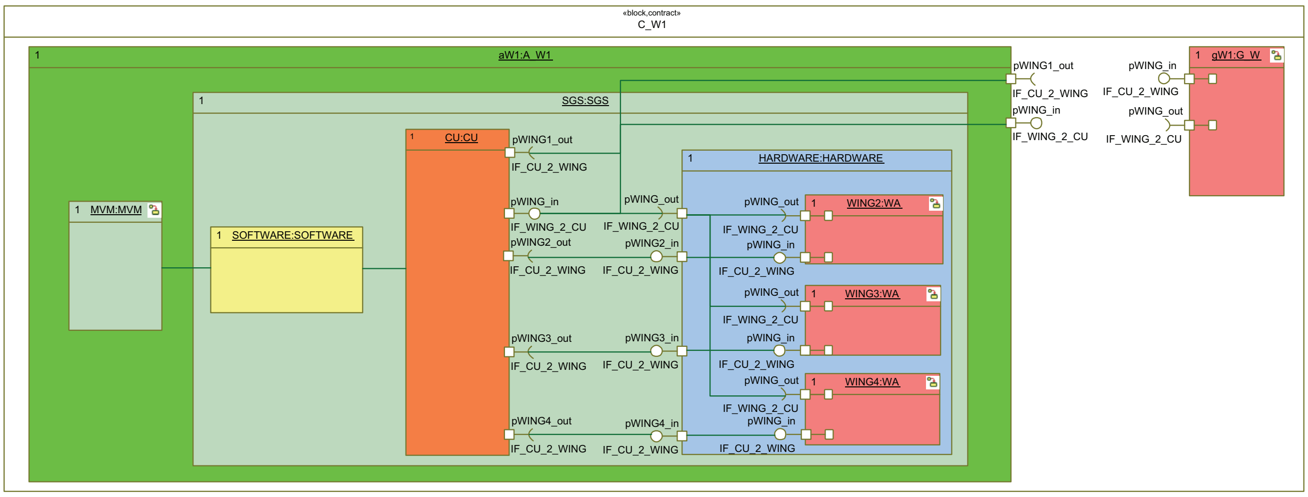


Fig. 14: The contract C_W1 for *WING*1 in SysML

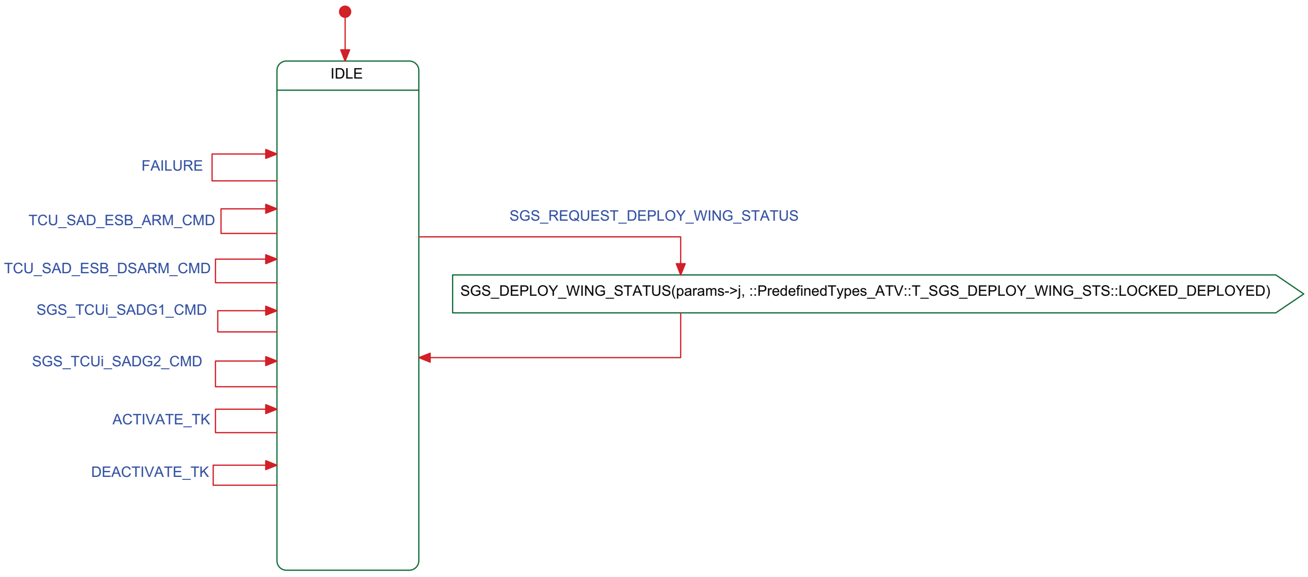

IDLE
FAILURE
TCU_SAD_ESB_ARM_CMD
TCU_SAD_ESB_DSARM_CMD
SGS_TCUi_SADG1_CMD
SGS_TCUi_SADG2_CMD
ACTIVATE_TK
DEACTIVATE_TK
SGS_REQUEST_DEPLOY_WING_STATUS
SGS_DEPLOY_WING_STATUS(params->j, ::PredefinedTypes_ATV::T_SGS_DEPLOY_WING_STS::LOCKED_DEPLOYED)

Fig. 15: The modeled behavior for all G_Wi and G_HW - parameter j ranges through 1–4

for all inputs corresponding to the wing deployment process. Also, the guarantee G_Wi is identical to the abstraction *WAi*. Furthermore, one can remark that the guarantee is stronger than the projection of the Requirement 2 on *WINGi*. The abstraction *WAj* can also be subject to one failure since this case was not excluded from its behavior. Then, the fault tolerance property that we verify via contracts is stronger than the one intended: we guarantee that the system is four-fault tolerant if faults occur in separate wings.

Secondly, we define a global contract $C\_HW = (A\_HW, G\_HW)$ for *HARDWARE* and prove that the contract is dominated by $\{C\_W1, C\_W2, C\_W3, C\_W4\}$. Again, we use as assumption A_HW the concrete environment of *HARDWARE*. The guarantee G_HW is the composition of the four *WAi*. All *WAi*, $i = \overline{1,4}$, and G_HW as defined satisfy the closure conditions for applying Theorem 8.

Contract $C\_HW = (A\_HW, G\_HW)$ where:

– A_HW = *MVM* ∥ *SOFTWARE* ∥ *CU*
– G_HW : for each wing status interrogation answers with *deployed*, while all other requests are ignored.

The next step of the reasoning consists in proving the satisfaction of the “mirror” contract C_HW^{-1}. This verification is trivial since the concrete environment is used as assumption and the proof obligation is written: *MVM* ∥ *SOFTWARE* ∥ *CU* $\sqsubseteq_{G\_HW}$ *MVM* ∥ *SOFTWARE* ∥ *CU*.

Table 1: Average verification time for each contract C_i per induced failure group

	Average verification time (s)			
Type of induced failure	Wing 1	Wing 2	Wing 3	Wing 4
Thermal knife	13,993	6869	18,842	11,412
Hold-down and release system	12,672	6516	16,578	9980
Solar array driving group	11,527	5432	13,548	6807

The last step consists in verifying that the composition $A\_HW \parallel G\_HW$ conforms to Requirement 2.

The proofs of all three steps have been automatically verified within the OMEGA-IFx Toolset with the method described in Sect. 6. For each step of the verification methodology, we have manually modeled the contracts: assumptions as blocks that we had to connect via ports to the other components and guarantees as independent components. The first step gave 4 possible configurations with one concrete wing and 3 abstract ones that were each verified with respect to all 14 possible failures. The average time in seconds needed for the verification of the satisfaction relation for each contract with respect to each class of failures is presented in Table 1. Even though the system model looks symmetrical, the command units do not have a symmetrical behavior and due to their interconnections with the wings the state space of system's abstraction for *WING1* and *WING3* is larger than the one of *WING2* and *WING4*: the CMU1 is responsible for *WING1* and *WING3* during wing deployment but transfers requests to the four wings during preparation, whereas CMU2 handles only the wing deployment for *WING2* and *WING4*. For the second step, the following obligations proofs have to be verified:

1. $WA1 \parallel WA2 \parallel WA3 \parallel WA4 \sqsubseteq_{MVM \parallel SOFTWARE \parallel CU} G\_HW$
2. $MVM \parallel SOFTWARE \parallel CU \parallel WA2 \parallel WA3 \parallel WA4 \sqsubseteq_{WA1} MVM \parallel SOFTWARE \parallel CU \parallel WA2 \parallel WA3 \parallel WA4$
3. $MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA3 \parallel WA4 \sqsubseteq_{WA2} MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA3 \parallel WA4$
4. $MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA4 \sqsubseteq_{WA3} MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA4$
5. $MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA3 \sqsubseteq_{WA4} MVM \parallel SOFTWARE \parallel CU \parallel WA1 \parallel WA2 \parallel WA3$

We remark that the last 4 items are trivial since we have the same member on both sides of the relations. Then, only the verification of the first item was necessary, which took 1 second. Finally, the verification that contract C conforms to Requirement 2 also took 1 second.

This industrial-grade system model shows how our approach can be applied, but also provides positive feedback with respect to the verification results. We can ascertain that previously intractable models can be tamed by the verification methodology and technique described in this paper.

8 Related work

Recent work has explored the notion of contract in the form of meta-theories, but also to directly provide theories and tools for specific component formalisms. However, the application of these theories in high-level modeling languages has been left aside. Until now, the concept of contract as it is used in systems engineering refers to pre-/post-conditions for operations or for model transformations. To the best of our knowledge, this work is the first to propose a reasoning theory based on behavioral contracts for timed asynchronously communicating components within UML/SysML together with a framework that facilitates its use.

8.1 Contract-based meta-theories and their implementations

The meta-theory that we use as basis in this paper [61] has already been instantiated for other formalisms: Labeled Transition Systems with priorities [61] and data [41], Modal Transition Systems [62], BIP components [6,39] and Heterogeneous Rich Components [9]. Yet, this is the first documented application for timed systems.

For this meta-theory to hold, *circular reasoning* has to be sound: for K and E two components and $\mathcal{C}=(A,G)$ a contract for K such that $K \sqsubseteq_A G$ and $E \sqsubseteq_G A$ then $K \sqsubseteq_E G$. A slightly different version of reasoning on which most of the related work relies consists in strengthening the hypothesis and requiring for the assumption of a contract to be refined by the environment regardless of how the component behaves. Formally, this rule called in general *assume/guarantee reasoning* is expressed as follows: if $K \sqsubseteq_A G$ and $E \sqsubseteq A$ then $K \sqsubseteq_E G$. Assume/guarantee reasoning is harder to achieve for complex systems such as ours that have mutual dependencies between components: it demands to find a strategy for breaking the symmetry of the dependency between the component and its environment by finding a component which can guarantee its property independently of its environment. Therefore, we consider that sound circular reasoning is more interesting in a contract-based approach and, at the same time, sufficient to allow for independent implementability of components.

Contract meta-theories have been developed also for *specification theories*. A specification theory is a complete algebra that, besides the parallel composition and refinement operators, defines a logical conjunction operator which allows to derive a component satisfying two different specifications and a quotient operator, which allows to compute from a system specification that is partially implemented, the coarsest specification of the remaining (not implemented) part. Therefore, the aim for a specification theory is to provide substitutability results allowing for compositional design, while its usage in compositional verification did not receive as much attention. In contrast, the meta-theory we use provides the minimal set of operators needed for both design and verification, even thought it may incur an important overhead especially during design. Logical conjunction and quotient can be formalized and added to our framework, whereas their use should be independent from the reasoning described above that is sound and sufficient for verification.

The meta-theory of [7] is built on a specification theory and is similar in many points with [61]. The main differences concern (1) the specification of contracts and (2) the method for reasoning with contracts. Regarding the first item, the framework of [7] does not support signature refinement, i.e., the ability of a contract to concentrate only on some of the inputs/outputs of the component while abstracting away the others, which is explicitly handled in our framework. A partial solution is presented in [8] where contracts are defined on a subset of the component's signature via ports. However, [8] does not allow to reason individually on contracts: at each design step, a specification consists in a component and the set of contracts on ports such that the union of their signatures is equal to the signature of the component. Moreover, signature refinement between specifications is not allowed since the definitions of refinement for both component and contract require for all elements from specifications to have the same signature.

Regarding the method for reasoning with contracts, the meta-theories from [7,8] subscribe to the assume/ guarantee reasoning. For proving dominance, the meta-theory of [7] makes use of a (derived) contract composition operator, which can be used to compute the "strongest" contract $C_1 \boxtimes C_2$ satisfied by the composition of two components that satisfy C_1 and, respectively, C_2. The dominance step is reduced to the following proof obligation: $C_1 \boxtimes C_2 \boxtimes \cdots \boxtimes C_n \preceq C$, where C is the dominated contract. However, contract composition is partial, i.e., it can be undefined for certain pair of contracts since it is based on the quotient operator which is itself partial. The meta-theory of [8] also uses a specification composition operator which in this case requires for port contracts and the contained components to be pairwise composed. So, the use of such operators may turn out difficult for large and complex systems. Moreover, in both meta-theories, the failure of proving dominance is not explicitly discussed and it is not clear how the user could derive a counterexample, identify the cause, and mitigate it. In contrast, the method from [61] allows to infer, based on the proof obligations that constitute the sufficient conditions for dominance, which contract is faulty. In the case of our TIOA framework, it is possible to derive a counterexample trace. On a more general note, neither the meta-theory of [7] nor [8] explicitly describe how a system requirement has to be formalized and how its satisfaction can be achieved via contracts. We consider that the reasoning methodology introduced by Quinton et al. [61,62] and depicted in Sect. 2 is an important asset in the application of the meta-theory to concrete domains.

The meta-theory of [7] has been applied in [27] for a Timed Input/Output Automata specification theory [29,30] and implemented in the ECDAR toolset [28]. However, some aspects of this theory make it impractical for representing the semantics of timed components described in SysML or UML. The synchronization between an input of one component and an output of another component becomes an output of the composite, which equates to considering outputs as broadcasts and which is not consistent with the UML/SysML semantics. Moreover, the formalism forbids non-determinism due to the timed game semantics [13] and does not handle silent transitions, which is problematic for representing the semantics of complex components performing internal computation steps. Finally, alternating timed simulation is used as refinement relation which is rather tailored for verifying composition compatibility, not the satisfaction of general safety properties. The same meta-theory has been applied for communicating components modeled as a set of traces in [20,21] and is implemented in the OCRA tool [19]. In this instantiation, a contract is given by a pair of hybrid LTL assertions on the set of traces and dominance is verified with a sufficient condition similar to the one defined in [61], even though circular reasoning is not explicitly stated and proved.

Assume–guarantee reasoning is a long-standing line of research, although classical approaches deal with logical specifications [1,23,40]. The more recent approach of [17] deals with specifications in the form of sets of I/O traces and handles the signature refinement in contract satisfaction although only for untimed specifications. Working directly on traces without an intermediate operational specification is delicate since system engineering teams are not necessarily experts in behavior formalization. Moreover, the reasoning approach is similar with the one proposed in [7], in particular with regard to contract composition. As such, the remarks made above with respect to the reasoning methodology remain valid for [17].

Interface theories [15,31–33,44] are also related to contracts since they can be used to express assumptions and guarantees for a component, albeit they do it in the same specification. An interface encompasses the assumption represented by its inputs and the guarantee represented by its outputs and describes how a component and its environment are expected to interact. Some of the specification theories developed in this context are based on variants of TIOA close to the one that we are using, like in the case of [18]. The theory of [18] allows reasoning for safety and bounded liveness properties on finite-timed traces. Generally, the notion of a contract that merges the assumption and the guarantee is well suited to derive compatible environments in which components can work together since it models how a component should behave. In fact, an interface plays the same role as the composition between an assumption and a guarantee as they are presented in our framework and which has to be manually computed by the designer. Keeping assumptions and guarantees separate has several advantages: (1) allows to model component properties as standalone automata and (2) allows for the refinement of the assumption or guarantee as well as the component to be performed independently.

8.2 Contracts in high-level modeling languages

Contracts in software engineering are classified in [10] in four categories: *syntactical* ones that describe the types a component can handle, *behavioral* ones that add constraints about the use of a component, *synchronization* ones that specify the global behavior and interaction of components, and *quality-of-service* ones that can quantify the expected behavior of components. Our definition of contracts, even though we call them behavioral due to the fact that they model a behavior, falls under the category of synchronization contracts since they explicitly describe the call order of requests and their synchronization. However, we consider that our contracts can also be used as syntactical and behavioral ones: the contract signature specifies also the signature of the component and assumption/guarantee can be considered as a generic view of the pre-/post-condition for component use.

Plentiful work has focused on syntactic and behavioral contracts in order to provide a mechanism for the composability problem. In [67], the authors make the distinction between an output contract which is offered by the component and an input contract on which the component relies, where a contract is a generic view of an interface. Then contracts serve as type specifications for components during the development phases of a system.

The Kmelia component model [4,45] provides means to verify the functional correctness of behavioral contracts for services, as well as compatibility issues between components. The meta-model defines contracts for operations or interfaces and models explicitly the verification results. With respect to behavioral contracts, one is modeled for an operation as a pre-/post-condition pair, while the behavior of the operation is modeled as an extended Labeled Transition System. Formal verification of contract satisfaction is performed by transformation of the component language to formalisms such as CADP[4] or AtelierB.[5]

[4] http://www.inrialpes.fr/vasy/cadp/.

[5] http://www.atelierb.eu/.

Synchronization contracts have been considered in [59] where on each component boundary an interface is defined with attributes, operations and a protocol state machine that describes the response of the component to sequences of events constraining their order. A contract is defined on connectors between two component boundaries, where protocol state machines are used to verify the compatibility of components. Despite the instantiation of these notions in SysML via the stereotype mechanism, the theory is not supported by a formal framework which can provide answers about the satisfaction of a contract.

In [5], the notion of contract is used as means to formalize requirements which are verified on system-of-systems expressed as Stochastic State Transition Systems. A contract is modeled by an OCL expression which may contain Contract Specification Language patterns [64] that correspond to Bounded Linear Temporal Logic operators. Then model checking is directly applied in order to prove the satisfaction of the requirement. Yet, this approach does not offer a compositional reasoning for the design of systems or for their verification. Their notion of contract equates to our formalization of requirements by observers which in our theory can be further decomposed.

Contracts have also been considered for synchronous SysML and AADL architectures in [37,68] which propose a reasoning similar to ours. A contract is defined as a pre-/post-condition pair on components with respect to their inputs/outputs, while each condition is modeled by a past-time linear temporal logic formula. Informally, contract satisfaction models that if the assumption holds at any previous moments, then the guarantee will also hold at the current moment. The theory provides a mechanism to verify dominance directly on contracts similar to Theorem 8: iterative verification of the satisfaction of individual assumptions by the other guarantees on which it depends and the global assumption and verification of the global guarantee by the global assumption and individual guarantees. However, circular reasoning is required to hold only locally at one moment in time: components are allowed to refer to guarantees of the others in earlier instants in time such that at one particular moment there is no circularity in the model. This is supported by the synchronous communication of components with one-step communication delay.

On the modeling side, requirements—both globally and locally in the form of guarantees—are modeled with the Property Specification Language for each component, therefore having a textual representation [47]. Several requirements can be proved by using the same set of contracts, yet the case when different contracts are needed is not explicitly handled. By difference, we use the same graphical language to represent components and contracts, thus removing the need for field engineers to master another specification language.

To conclude, our contract theory is complementary with respect to the previous presented approaches for SysML and it deals with the timed behavior of system models and requirement satisfaction via contracts.

9 Conclusion

In this paper, we described a complete method for reasoning with behavioral contracts both for the design and the verification of system models in SysML. Firstly, we proposed an extension of UML/SysML allowing to model contracts and to use them for the compositional verification of requirements. The extension is defined as a meta-model enriched with well-formedness rules such that a system model extended with contracts is unambiguous and sound. An instantiation of the meta-model based on the stereotype mechanism was provided in order to make the extension usable with standard model editors.

Next, we formalized the semantics of the SysML component framework by a variant of timed input/output automata and we provided a mapping between the SysML notions and their TIOA counterparts. We built upon the TIOA framework a formal contract-based theory and we established compositionality results that allow reasoning with contracts for the satisfaction of timed safety requirements. Automatic verification can be achieved based on a reachability analysis method given that the contract satisfies certain constraints, while the entire technique is partially implemented in the IFx toolset. Yet, some of the steps for generating the intermediate contract-based verification models remain manual. Future work consists in automating all intermediate model generation steps and adding functionality to manage the proof obligations and results.

Finally, we illustrated our method on a case study extracted from an industrial-scale system model, and we showed how contract-based reasoning can alleviate the problem of combinatorial explosion for the verification of large systems.

However, the method described does not explicitly prescribe how to derive contracts for the whole system and for its components. In our case study, the task was rather straightforward: the guarantee G is almost identical to the requirement to satisfy, while component guarantees are the projection of G on each component. There may be cases where the model for contracts is less obvious and the overhead may be significant. This is one of the reasons why previous attempts to introduce contracts in programming languages have not enjoyed an extensive popularity. Nevertheless, we believe that the case for contracts in early phases of development is different and the contract concept is necessary in order to cope with the growing complexity of system models that need to be verified before implementation. We consider that applying verification techniques is less expensive than finding errors in the hardware or software after implementation. Future work is guided by the urgency to make contract-based reasoning widespread in systems engineering, which can be achieved by providing methods or methodological guidelines for deriving intermediate contracts from the properties one is trying to prove.

Appendix 1: OCL formalization of the well-formedness set of rules for contracts in UML/SysML

In this appendix, we discuss the OCL formalization of the well-formedness rules presented in Sect. 3.3.

Listing 1 presents the OCL code corresponding to Rules 1 and 2. For their formalization, we have defined two helper functions *isConjugated* and *isIdenticalTo* for the verification of conjugated, respectively, corresponding, ports. The formalization of Rule 1 consists in verifying that for each port of one type there is at least one port of the other matching the criteria. In order to avoid possible broadcast, we verify that the assumption and guarantee have the same number of ports. The formalization of Rule 2 summarizes to iterating the set of ports of the guarantee and verifying that for each port there is one and only one correspondent in the definition of the component.

Rule 3 ensures the uniqueness of a dominance relation in a given context. Indeed, if a component uses a contract for which a correct refinement is provided based on its subcomponents, there is no need to define a second refinement for the same contract and the same components. Listing 2 provides the OCL formalization of this rule: for each component we compute the set of *contractUse* relations and we verify that the set does not include two or more relations pointing to the same requirement.

Finally, Listing 3 describes the completeness Rule 4: within the set of conformance relations defined in a model, there is one and only one relation having as target the current *SafetyProperty*.

Listing 1 OCL code for well-formedness of contracts.

```
context Port

def: isConjugatedOf(p:Port) : Boolean = self.direction <> p.direction and self.interface = p.interface
def: isIdenticalTo(p:Port) : Boolean = self.name = p.name and self.direction = p.direction and self.
     interface = p.interface

context Contract

-- Rule: The assumption and guarantee of a contract define a closed system with respect to ports
def: haveIdenticalNoOfPorts : Boolean =
  self.itsAssumption.ownedPort->size() = self.itsGuarantee.ownedPort->size()
def: assumptionPortsSubsetGuaranteePorts : Boolean =
  self.itsAssumption.ownedPort->forAll(p1 | self.itsGuarantee.ownedPort->select(p2| p1.isConjugatedOf(
      p2))->size() >= 1)
def: guaranteePortsSubsetAssumptionPorts : Boolean =
  self.itsGuarantee.ownedPort->forAll(p1 | self.itsAssumption.ownedPort->select(p2 | p1.isConjugatedOf
      (p2))->size() >= 1)

def: contractAGPortsWellFormed : Boolean =
  self.haveIdenticalNoOfPorts and self.assumptionPortsSubsetGuaranteePorts and self.
      guaranteePortsSubsetAssumptionPorts

inv contractClosedSystem : self.contractAGPortsWellFormed

context Implementation

-- Rule: The set of ports of the guarantee is a subset or equal to the set of ports of the part
     implementing it
def: guaranteePortsSubsetPartPorts : Boolean =
  self.implTarget.itsGuarantee.ownedPort->forAll(p1 | self.implSource.ownedPort->select(p2 | p2.
      isIdenticalTo(p1))->size() = 1)

def: guaranteePortsWellFormed : Boolean =
  self.guaranteePortsSubsetPartPorts

inv implementationGuaranteePortsWellFormed : self.guaranteePortsWellFormed
```

Listing 2 OCL code for uniqueness of contract-based proof obligations.

```
context Property

def: getContractUseRelations : Set(Dependency) = self.clientDependency->select(d | d.isUsage)
def: isUsingContracts : Boolean = self.getContractUseRelations->size() > 0

-- Rule: A component can use at most one contract for the satisfaction of one requirement and within
     one dominance
def: isContractUniqueForRequirementAndRefinement : Boolean =
     let r:Set(Dependency) = self.getContractUseRelations in
     if self.type.oclIsTypeOf(uml::Class) and self.isUsingContracts
```

```
        then r->forAll(d1 | r->excluding(d1)->select(d2 | d1.reqTarget = d2.reqTarget)->size() = 0)
    else
        true
    endif

inv contractUseUniqueRR : self.isContractUniqueForRequirementAndRefinement
```

Listing 3 OCL code for completeness of contract-based proof obligations.

```
context SafetyProperty

-- Rule: All safety properties have a contract conforming to them
def: isVerified : Boolean =
    self.oclAsType(uml::Classifier).getModel().getDependencies->select(d | d.isConformance and d.
        confTarget->includes(self))->size() = 1

inv safetyPropertyIsVerified : self.isVerified
```

Appendix 2: SysML-TIOA mapping example

The TIOA corresponding to the component from Fig. 8b is the tuple $(X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ defined as follows:

- $X = \{location, queue\}$ only contains the predefined variables for the state machine location and for the input queue. The domains of these are as follows:
  - $Dom_{location} = \{Idle, EjectCard, WaitForRemoval, RemoveCard, WaitForAck\}$
  - $Dom_{queue} = I^*$, i.e., the set of finite sequences of elements of I (the input actions, defined below). The concatenation of two sequences a and b of I^* is denoted by $[a; b]$.
- $Clk = \{t\}$ contains the Timer t.
- $Q = Dom_{location} \times Dom_{queue} \times \mathbb{R}$ is the set of possible valuations of the variables (location, queue, t) listed above. For a triple $q = (\lambda, \rho, \delta) \in Q$, for simplicity, we denote λ as $q.location$, ρ as $q.queue$ and δ as $q.t$.
- $\theta = (Idle, \emptyset, 0)$ is the initial state, where $\emptyset$ denotes the empty sequence of signals from I.
- $I = \{retrieveCard, ok, nok\}$
- $O = \{cardInserted, ejectCard, cardRemoved\}$
- $V = \emptyset$
- $H = \{\downarrow retrieveCard, \downarrow ok\}$ is the set of internal actions corresponding to the consumption of the input signals from the *queue*. In other contexts, there may be additional internal actions corresponding to transitions without any visible activity, however this is not the case for the state machine in Fig. 8b.
- $D = Inp \cup (\cup_{loc \in Dom_{location}} D_{loc})$ where
  - Inp represents the input transitions, defined as follows (remember that the TIOA is input-complete): $Inp = \{q \xrightarrow{i} q' | i \in I \wedge q'.location = q.location \wedge q'.t = q.t \wedge q'.queue = [q.queue; i]\}$
  - D_{loc} represents the discrete TIO transitions corresponding to the state machine transitions leaving state loc in Fig. 8(b):
    - $D_{Idle} = \{q \xrightarrow{!cardInserted} q')|q.location = Idle \wedge q'.location = EjectCard \wedge q.queue = q'.queue \wedge q.t = q'.t\}$
    - $D_{EjectCard} = \{q \xrightarrow{!ejectCard} q'|q.location = EjectCard \wedge q'.location = WaitForRemoval \wedge q.queue = q'.queue \wedge q.t = q'.t\}$
    - $D_{WaitForRemoval} = \{q \xrightarrow{\downarrow retrieveCard} q'|q.location = WaitForRemoval \wedge q'.location = RemoveCard \wedge q.queue = [retrieveCard; q'.queue] \wedge q'.t = 0\}$

- $D_{RemoveCard} = \{q \xrightarrow{!cardRemoved} q' | q.location = RemoveCard \wedge q'.location = WaitForAck \wedge q.queue = q'.queue \wedge q.t = q'.t\}$
- $D_{WaitForAck} = \{q \xrightarrow{\downarrow ok} q' | q.location = WaitForAck \wedge q'.location = Idle \wedge q.queue = [ok; q'.queue] \wedge q.t = q'.t\}$

– $\mathcal{T} = \cup_{loc \in Dom_{location}} \mathcal{T}_{loc}$ where $\mathcal{T}_{loc}$ represents the trajectories starting in states q with $loc = q.location$. They are defined as follows:

– $\mathcal{T}_{Idle} = \{\tau : [0, 0] \to Q | \tau(0) = q, \forall q \in Q$ with $q.location = Idle\}$
(i.e., only point trajectories are allowed since the outgoing transition is eager and has no input)
– $\mathcal{T}_{EjectCard} = \{\tau : I \to Q \mid I$ is any interval of form $[0, x]$ or $[0, \infty) \wedge \forall y \in I.\ (\tau(y)(location) = EjectCard \wedge \tau(y)(queue) = \tau(0)(queue) \wedge \tau(y)(t) = y + \tau(0)(t))\}$
(i.e., trajectories may go to infinity, and only change t with derivative 1)
– $\mathcal{T}_{WaitForRemoval} = \mathcal{T}^{Pres}_{WaitForRemoval} \cup \mathcal{T}^{Abs}_{WaitForRemoval}$ where:
- $\mathcal{T}^{Pres}_{WaitForRemoval}$ are (point) trajectories from states in which the signal $retrieveCard$ is present in front of the queue:
$\mathcal{T}^{Pres}_{WaitForRemoval} = \{\tau : [0, 0] \to Q | \tau(0) = q, \forall q \in Q$ with $q.location = WaitForRemoval \wedge \exists w$ such that $q.queue = [retrieveCard; w]\}$
- $\mathcal{T}^{Abs}_{WaitForRemoval}$ are trajectories (to infinity) from states in which the signal $retrieveCard$ is not in front of the queue:
$\mathcal{T}^{Abs}_{WaitForRemoval} = \{\tau : I \to Q \mid I$ is any interval of form $[0, x]$ or $[0, \infty) \wedge \forall y \in I.\ (\tau(y)(location) = WaitForRemoval \wedge \tau(y)(queue) = \tau(0)(queue) \wedge \tau(y)(t) = y + \tau(0)(t) \wedge \nexists w$ such that $q.queue = [retrieveCard; w])\}$

– $\mathcal{T}_{RemoveCard} = \{\tau : [0, x] \to Q | x \in [0, 5 - \tau(0)(t)] \wedge \forall y \in [0, x].\ (\tau(y)(location) = RemoveCard \wedge \tau(y)(queue) = \tau(0)(queue) \wedge \tau(y)(t) = y + \tau(0)(t))\}$
(i.e., trajectories may go up to $t = 5$ and only change t with derivative 1)
– $\mathcal{T}_{WaitForAck} = \mathcal{T}^{Pres}_{WaitForAck} \cup \mathcal{T}^{Abs}_{WaitForAck}$ where:
- $\mathcal{T}^{Pres}_{WaitForAck}$ are (point) trajectories from states in which the signal ok is present in front of the queue:
$\mathcal{T}^{Pres}_{WaitForAck} = \{\tau : [0, 0] \to Q | \tau(0) = q, \forall q \in Q$ with $q.location = WaitForAck \wedge \exists w$ such that $q.queue = [ok; w]\}$
- $\mathcal{T}^{Abs}_{WaitForAck}$ are trajectories (to infinity) from states in which the signal ok is not in front of the queue:
$\mathcal{T}^{Abs}_{WaitForAck} = \{\tau : I \to Q \mid I$ is any interval of form $[0, x]$ or $[0, \infty) \wedge \forall y \in I.(\tau(y)(location) = WaitForAck \wedge \tau(y)(queue) = \tau(0)(queue) \wedge \tau(y)(t) = y + \tau(0)(t) \wedge \nexists w$ such that $q.queue = [ok; w])\}$

Appendix 3: Proofs for the formal contract-based framework

Theorem 2 $(\mathcal{A}, \|)$ is a commutative monoid.

Proof Let $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ be three timed input/output automata.

1. *Commutativity*: $\mathcal{A}_1 \parallel \mathcal{A}_2 = \mathcal{A}_2 \parallel \mathcal{A}_1$ is true since only set operations are used by the composition operator.
2. *Associativity*: By applying the composition operator we obtain $(\mathcal{A}_1 \parallel \mathcal{A}_2) \parallel \mathcal{A}_3 = \mathcal{A}_1 \parallel (\mathcal{A}_2 \parallel \mathcal{A}_3) = (X, Clk, Q, \theta, I, O, V, H, D, \mathcal{T})$ where:

– $X = X_{\mathcal{A}_1} \cup X_{\mathcal{A}_2} \cup X_{\mathcal{A}_3}$.
– $Clk = Clk_{\mathcal{A}_1} \cup Clk_{\mathcal{A}_2} \cup Clk_{\mathcal{A}_3}$.
– $Q = \{\mathrm{x}_{\mathcal{A}_1} \cup \mathrm{x}_{\mathcal{A}_2} \cup \mathrm{x}_{\mathcal{A}_3} | \mathrm{x}_{\mathcal{A}_1} \in Q_{\mathcal{A}_1}, \mathrm{x}_{\mathcal{A}_2} \in Q_{\mathcal{A}_2}$ and $\mathrm{x}_{\mathcal{A}_3} \in Q_{\mathcal{A}_3}\}$.
– $\theta = \theta_{\mathcal{A}_1} \cup \theta_{\mathcal{A}_2} \cup \theta_{\mathcal{A}_3}$.
– $I = (I_{\mathcal{A}_1} \setminus (O_{\mathcal{A}_2} \cup O_{\mathcal{A}_3})) \cup (I_{\mathcal{A}_2} \setminus (O_{\mathcal{A}_1} \cup O_{\mathcal{A}_3})) \cup (I_{\mathcal{A}_3} \setminus (O_{\mathcal{A}_1} \cup O_{\mathcal{A}_2}))$.
– $O = (O_{\mathcal{A}_1} \setminus (I_{\mathcal{A}_2} \cup I_{\mathcal{A}_3})) \cup (O_{\mathcal{A}_2} \setminus (I_{\mathcal{A}_1} \cup I_{\mathcal{A}_3})) \cup (O_{\mathcal{A}_3} \setminus (I_{\mathcal{A}_1} \cup I_{\mathcal{A}_3}))$.
– $V = V_{\mathcal{A}_1} \cup V_{\mathcal{A}_2} \cup V_{\mathcal{A}_3} \cup (O_{\mathcal{A}_1} \cap (I_{\mathcal{A}_2} \cup I_{\mathcal{A}_3})) \cup (O_{\mathcal{A}_2} \cap (I_{\mathcal{A}_1} \cup I_{\mathcal{A}_3})) \cup (O_{\mathcal{A}_3} \cap (I_{\mathcal{A}_1} \cup I_{\mathcal{A}_2}))$.
– $H = H_{\mathcal{A}_1} \cup H_{\mathcal{A}_2} \cup H_{\mathcal{A}_3}$.

- D is the set of discrete transitions where for each $x = x_{\mathcal{A}_1} \cup x_{\mathcal{A}_2} \cup x_{\mathcal{A}_3}$ and $x' = x'_{\mathcal{A}_1} \cup x'_{\mathcal{A}_2} \cup x'_{\mathcal{A}_3} \in Q$ and each $a \in A$, $x \xrightarrow{a} x'$ if and only if for $i \in \{1, 2, 3\}$, either
  (a) $a \in A_i$ and $x_i \xrightarrow{a} x'_i$, or
  (b) $a \notin A_i$ and $x_i = x'_i$.
- $\mathcal{T} \subseteq trajs(Q)$ is given by $\tau \in \mathcal{T} \Leftrightarrow \tau \lceil X_i \in \mathcal{T}_i, i \in \{1, 2, 3\}$.

3. The identity element is the *empty* timed input/output automaton: it has no internal variables, it does not perform any actions and can let time elapse to infinity. □

Theorem 5 Given a component Env and a set $\mathcal{K}$ of components for which Env is an environment, the refinement under context $\sqsubseteq_{Env}$ is a preorder over $\mathcal{K}$.

Proof 1. *Reflexivity*: $K \sqsubseteq_{Env} K \overset{\Delta}{\Leftrightarrow} K \parallel Env \parallel Env' \preceq K \parallel Env \parallel Env'$ which is true from the definition of the conformance relation. K' is not represented since it is the identity element of the composition operator.
2. *Transitivity*: $K_1 \sqsubseteq_{Env} K_2 \wedge K_2 \sqsubseteq_{Env} K_3 \implies K_1 \sqsubseteq_{Env} K_3$.

$K_1 \sqsubseteq_{Env} K_2 \overset{\Delta}{\Leftrightarrow} K_1 \parallel Env \parallel Env' \preceq K_2 \parallel Env \parallel K'_2 \parallel Env'$ (1)

We write the automaton $Env' = Env'_1 \parallel Env'_2$ where :

- $Env'_1 = (\emptyset, \emptyset, \{\phi\}, \phi, ((O_{K_1} \cap O_{K_2}) \setminus I_E), ((I_{K_1} \cap I_{K_2}) \setminus O_E), \emptyset, \emptyset, D_{Env'_1}, \mathcal{T}_{Env'_1})$,
- $Env'_2 = (\emptyset, \emptyset, \{\phi\}, \phi, ((O_{K_1} \setminus O_{K_2}) \setminus I_E), ((I_{K_1} \setminus I_{K_2}) \setminus O_E), \emptyset, \emptyset, D_{Env'_2}, \mathcal{T}_{Env'_2})$.

Remark that the sets of input and output actions are pairwise disjoint for Env'_1 and Env'_2.
We write the automaton $K'_2 = K''_2 \parallel Env'_3$ where:

- $K''_2 = (\emptyset, \emptyset, \{\phi\}, \phi, (I_{K_1} \setminus I_{K_2}), (O_{K_1} \setminus O_{K_2}), (V_{K_1} \setminus E_{K_2}), \emptyset, D_{K''_2}, \mathcal{T}_{K''_2}$,
- $Env'_3 = (\emptyset, \emptyset, \{\phi\}, \phi, (V_{K_1} \cap O_{K_2}), (V_{K_1} \cap I_{K_2}), \emptyset, \emptyset, D_{Env'_3}, \mathcal{T}_{Env'_3})$.

Similarly, the sets of inputs, outputs and visible actions are pairwise disjoint for K''_2 and Env'_3.
With this notation:

(1) $\Leftrightarrow K_1 \parallel Env \parallel Env'_1 \parallel Env'_2 \preceq K_2 \parallel Env \parallel K''_2 \parallel Env'_3 \parallel Env'_1 \parallel Env'_2$ (2)
$K_2 \sqsubseteq_{Env} K_3 \overset{\Delta}{\Leftrightarrow} K_2 \parallel Env \parallel Env'' \preceq K_3 \parallel Env \parallel K'_3 \parallel Env''$ (3)

With the same notation we obtain that $Env'' = Env'_1 \parallel Env'_3$, and

(3) $\Leftrightarrow K_2 \parallel E \parallel Env'_1 \parallel Env'_3 \preceq K_3 \parallel E \parallel K'_3 \parallel Env'_1 \parallel Env'_3$ (4)

Composing (4) with $K''_2 \parallel Env'_2$ and from Theorem 4 we get:

$$K_2 \parallel Env \parallel Env'_1 \parallel Env'_3 \parallel K''_2 \parallel Env'_2 \preceq K_3 \parallel Env \parallel K'_3 \parallel Env'_1 \parallel Env'_3 \parallel K''_2 \parallel Env'_2 \Leftrightarrow$$
$$\left.\begin{array}{r}\Leftrightarrow K_2 \parallel Env \parallel K''_2 \parallel Env'_3 \parallel Env'_1 \parallel Env'_2 \preceq K_3 \parallel Env \parallel K'_3 \parallel K'_2 \parallel Env'_1 \parallel Env'_2 \\ (2)\ K_1 \parallel Env \parallel Env'_1 \parallel Env'_2 \preceq K_2 \parallel Env \parallel K''_2 \parallel Env'_3 \parallel Env'_1 \parallel Env'_2\end{array}\right\} \overset{Transitivity\ of\ \preceq}{\Longrightarrow}$$
$$\implies K_1 \parallel Env \parallel Env'_1 \parallel Env'_2 \preceq K_3 \parallel Env \parallel K'_3 \parallel K'_2 \parallel Env'_1 \parallel Env'_2 \Leftrightarrow\Leftrightarrow K_1 \parallel Env \parallel Env'$$
$$\preceq K_3 \parallel Env \parallel K'_2 \parallel K'_3 \parallel Env'$$

By denoting $K' = K'_2 \parallel K'_3$ we have:

$K_1 \parallel Env \parallel Env' \preceq K_3 \parallel Env \parallel K' \parallel Env' \overset{\Delta}{\Leftrightarrow} K_1 \sqsubseteq_{Env} K_3$

The last step consists in proving that K' is indeed the automaton generated by the refinement under context relation. Since K_2' and K_3' are built from the hypothesis by the refinement under context relation, by composition they define the correct structure for K'. Moreover:

- $I_{K'} = (I_{K_1} \setminus I_{K_3}) \cup (V_{K_1} \cap O_{K_3})$,
- $O_{K'} = (O_{K_1} \setminus O_{K_3}) \cup (V_{K_1} \cap I_{K_3})$ and
- $V_{K'} = V_{K_1} \setminus E_{K_3}$.

The proofs on the sets of actions for Env'' and K' are detailed in [36]. □

Theorem 6 Let K_1 and K_2 be two components and E an environment compatible with both K_1 and K_2 such that $Env = Env_1 \parallel Env_2$. Then $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1$.

Proof First, let us rewrite the two refinement relations to be proved equivalent as conformance relations, based on the definition of refinement under context:

- $K_1 \sqsubseteq_{Env_1 \parallel Env_2} K_2 \Leftrightarrow K_1 \parallel (Env_1 \parallel Env_2) \parallel Env' \preceq K_2 \parallel (Env_1 \parallel Env_2) \parallel K' \parallel Env'$
- $K_1 \parallel Env_1 \sqsubseteq_{Env_2} K_2 \parallel Env_1 \Leftrightarrow (K_1 \parallel Env_1) \parallel Env_2 \parallel Env'' \preceq (K_2 \parallel Env_1) \parallel Env_2 \parallel K'' \parallel Env''$

Based of the associativity of $\parallel$ we have that the two relations are identical, where: $Env' = Env'' = (\emptyset, \emptyset, \{\phi\}, \phi, (O_{K_1} \setminus (I_{Env_1} \cup I_{Env_2})), (I_{K_1} \setminus (O_{Env_1} \cup O_{Env_2})), \emptyset, \emptyset, D_{Env'}, \mathcal{T}_{Env'})$ and $K' = K'' = (\emptyset, \emptyset, \{\phi\}, \phi, ((I_{K_1} \setminus I_{K_2}) \cup (V_{K_1} \cap O_{K_2})), ((O_{K_1} \setminus O_{K_2}) \cup (V_{K_1} \cap I_{K_1})), (V_{K_1} \setminus E_{K_2}), \emptyset, D_{K'}, \mathcal{T}_{K'})$. □

Theorem 8 $\{C_i\}_{i=1}^n$ dominates C if, $\forall i$, $traces_{G_i}$ and $traces_G$ are closed under time extension and

$$\begin{cases} G_1 \parallel ... \parallel G_n \sqsubseteq_A G \\ A \parallel G_1 \parallel ... \parallel G_{i-1} \parallel G_{i+1} \parallel ... \parallel G_n \sqsubseteq_{G_i} A_i, \ \forall i \end{cases}$$

Proof Let $K_i, i = \overline{1, n}$, a set of components such that:

(1) $K_i \sqsubseteq_{A_i} G_i$,
(2) $G_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G$,
(3) $A \parallel G_1 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i$.

We have to prove that $K_1 \parallel K_2 \parallel \ldots \parallel K_n \sqsubseteq_A G$.
The proof is built by induction on j where $j = \overline{0, n}$ is the number of guarantees replaced by their corresponding component. More precisely, we will prove by induction that $K_1 \parallel ... \parallel K_{j-1} \parallel G_j \parallel ... \parallel G_n \sqsubseteq_A G$. In parallel, we will also need to prove that $A \parallel K_1 \parallel K_2 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i > j$.
Step $j = 0$. Then the conclusion becomes $G_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G$ which is true by hypothesis (2)
Step $j = 1$.

$$\left.\begin{array}{r} (1)\ K_1 \sqsubseteq_{A_1} G_1 \\ \text{From (2) for i=1} \Rightarrow A \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_{G_1} A_1 \end{array}\right\} \overset{Theorem\ 7}{\Rightarrow}$$

$\Rightarrow K_1 \sqsubseteq_{A \parallel G_2 \parallel \ldots \parallel G_n} G_1$ (4)

$$\left.\begin{array}{r} (4) \overset{Theorem\ 6}{\Rightarrow} K_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G_1 \parallel G_2 \parallel \ldots \parallel G_n \\ (2)\ G_1 \parallel \ldots \parallel G_n \sqsubseteq_A G \end{array}\right\} \overset{Transitivity}{\Rightarrow}$$

$\Rightarrow K_1 \parallel G_2 \parallel \ldots \parallel G_n \sqsubseteq_A G$ (5)

$$\left.\begin{array}{r} (4) \overset{Theorem\ 6}{\Rightarrow} A \parallel K_1 \parallel G_2 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A \parallel G_1 \parallel G_2 \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel \\ \parallel G_n, \forall i > 1 \\ (3)\ A \parallel G_1 \parallel \ldots G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i \end{array}\right\} \overset{Transitivity}{\Rightarrow}$$

$\Rightarrow A \parallel K_1 \parallel G_2 \parallel \ldots G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i > 1$ (6)

Relations (5) and (6) constitute the hypotheses for the induction step at $j = 2$.

Induction step Let j be fixed. The induction hypotheses for this step are:

$K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_n \sqsubseteq_A G$ (7)
$A \parallel K_1 \parallel K_2 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \quad \forall i > j$ (8)

Then we want to prove that:

$K_1 \parallel \ldots \parallel K_j \parallel K_{j+1} \parallel G_{j+2} \parallel G_n \sqsubseteq_A G$ (9) and
$A \parallel K_1 \parallel \ldots \parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \quad \forall i > j+1$ (10)

We proceed as follows:

$$\left.\begin{array}{r}(1)\ K_{j+1} \sqsubseteq_{A_{j+1}} G_{j+1} \\ \text{From (8) for i = j + 1} \Rightarrow A \parallel K_1 \parallel K_2 \parallel \ldots \parallel K_j \parallel G_{j+2} \parallel \ldots \parallel G_n \sqsubseteq_{G_{j+1}} A_{j+1}\end{array}\right\} \overset{Theorem\ 7}{\Rightarrow}$$

$\Rightarrow K_{j+1} \sqsubseteq_{A \parallel K_1 \parallel \ldots \parallel K_j \parallel G_{j+2} \parallel \ldots \parallel G_n} G_{j+1}$ (11)

$$\left.\begin{array}{r}(11) \overset{Theorem\ 6}{\Rightarrow} K_1 \parallel \ldots \parallel K_j \parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_n \sqsubseteq_A K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_n \\ (7)\ K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_n \sqsubseteq_A G\end{array}\right\} \overset{Transitivity}{\Rightarrow}$$

$\Rightarrow K_1 \parallel \ldots \parallel K_j \parallel K_{j+1} \parallel G_{j+2} \parallel G_n \sqsubseteq_A G$ (9)

$$\left.\begin{array}{r}(11) \overset{Theorem\ 6}{\Rightarrow} A \parallel K_1 \parallel \ldots \parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A \parallel K_1 \parallel \ldots \parallel \\ \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n, \forall i > j+1 \\ (8)\ A \parallel K_1 \parallel \ldots \parallel K_j \parallel G_{j+1} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i > j+1\end{array}\right\} \overset{Transitivity}{\Rightarrow}$$

$\Rightarrow A \parallel K_1 \parallel \ldots \parallel K_{j+1} \parallel G_{j+2} \parallel \ldots \parallel G_{i-1} \parallel G_{i+1} \parallel \ldots \parallel G_n \sqsubseteq_{G_i} A_i, \forall i > j+1$ (10)

Step $j = n$. From (9), for $j = n$, we obtain $K_1 \parallel K_2 \parallel \ldots \parallel K_n \sqsubseteq_A G$ which implies dominance. □

Theorem 9 $K_1 \sqsubseteq_E K_2$ if K_2 is a deterministic safety property and $reach((K_1 \parallel E \parallel E') \bowtie \mathcal{O}_{K_2}) \cap \{\pi\} = \emptyset$.

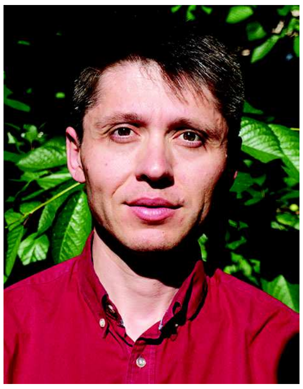
Proof Notation. We note by $reach(\mathcal{A})(\sigma)$ the set of reached states after the execution σ
This proof is built by contradiction. We suppose that $K_1 \not\sqsubseteq_E K_2$.
It implies that $\exists \sigma \in tr(K_1 \parallel E \parallel E') \wedge \sigma \notin tr(K_2 \parallel E \parallel E' \parallel K')$.
Let $\sigma' a$ be a prefix of σ such that $\sigma' \in tr(K_1 \parallel E \parallel E') \cap tr(K_2 \parallel E \parallel E' \parallel K')$ and $\sigma' a \notin tr(K_2 \parallel E \parallel E' \parallel K')$, where a is a visible action. Such a prefix exists because K_2 is a safety property.
Then $reach((K_1 \parallel E \parallel E') \bowtie \mathcal{O}_{K_2})(\sigma') = \{(q_1, q_2)\}$.
Concatenating a we obtain: $(q_1, q_2) \xrightarrow{a}_{(K_1 \parallel E \parallel E') \bowtie \mathcal{O}_{K_2}} \pi \implies reach((K_1 \parallel E \parallel E') \bowtie \mathcal{O}_{K_2}) \cap \{\pi\} \neq \emptyset$ in contradiction with the hypothesis. □

References

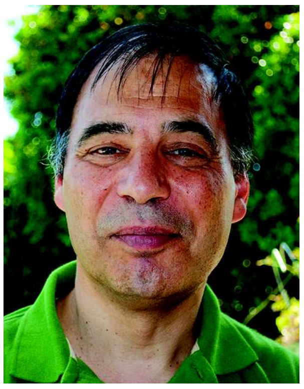
1. Abadi, M., Plotkin, G.D.: A logical view of composition. Theor. Comput. Sci. **114**(1), 3–30 (1993)
2. Aboussoror, E., Ober, I., Ober, I.: Seeing errors: model driven simulation trace visualization. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 7590, pp. 480–496. Springer, Berlin (2012)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
4. André, P., Gilles, A., Messabihi, M.: Vérification de contrats logiciels à l'aide de transformations de modèles. In: 7èmes journées sur l'Ingénierie Dirigée par les Modèles (IDM) (2011)
5. Arnold, A., Boyer, B., Legay, A.: Contracts and behavioral patterns for SoS: the EU IP DANSE approach. In: Larsen, K.G., Legay, A., Nyman, U. (eds.) AiSoS, EPTCS, vol. 133, pp. 47–66 (2013)
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM'06, pp. 3–12 (2006)
7. Bauer, S.S., David, A., Hennicker, R., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Moving from specifications to contracts in component-based design. In: de Lara, J., Zisman, A. (eds.) FASE, Lecture Notes in Computer Science, vol. 7212, pp. 43–58. Springer (2012)
8. Bauer, S.S., Hennicker, R., Legay, A.: Component interfaces with contracts on ports. In: Pasareanu, C.S., Salaün, G. (eds.) Formal Aspects of Component Software, Lecture Notes in Computer Science, vol. 7683, pp. 19–35. Springer, Berlin (2013)
9. Benvenuti, L., Ferrari, A., Mangeruca, L., Mazzi, E., Passerone, R., Sofronis, C.: A contract-based formalism for the specification of heterogeneous systems. In: FDL'08. Forum on, pp. 142–147. IEEE (2008)
10. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. Computer **32**(7), 38–45 (1999)
11. Bobaru, M.G., Pasareanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) CAV, Lecture Notes in Computer Science, vol. 5123, pp. 135–148. Springer (2008)
12. Bornot, S., Sifakis, J.: An algebraic framework for urgency. Inf. Comput. **163**(1), 172–202 (2000)
13. Bourke, T., David, A., Larsen, K.G., Legay, A., Lime, D., Nyman, U., Wasowski, A.: New results on timed specifications. In: Mossakowski, T., Kreowski, H.J. (eds.) WADT, Lecture Notes in Computer Science, vol. 7137, pp. 175–192. Springer (2010)
14. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. Lecture Notes in Computer Science, vol. 3185, pp. 237–267. Springer, Berlin (2004)
15. Chen, T., Chilton, C., Jonsson, B., Kwiatkowska, M.Z.: A Compositional specification theory for component behaviours. In: Seidl, H. (ed.) ESOP, Lecture Notes in Computer Science, vol. 7211, pp. 148–168. Springer (2012)
16. Cheung, S.C., Kramer, J.: Checking safety properties using compositional reachability analysis. ACM Trans. Softw. Eng. Methodol. **8**(1), 49–78 (1999)
17. Chilton, C., Jonsson, B., Kwiatkowska, M.Z.: Assume-guarantee reasoning for safe component behaviours. In: Pasareanu, C.S., Salaün, G. (eds.) Formal Aspects of Component Software, Lecture Notes in Computer Science, vol. 7683, pp. 92–109. Springer, Berlin (2013)
18. Chilton, C., Kwiatkowska, M.Z., Wang, X.: Revisiting Timed specification theories: a linear-time perspective. In: Jurdzinski, M., Nickovic, D. (eds.) FORMATS, Lecture Notes in Computer Science, vol. 7595, pp. 75–90. Springer (2012)
19. Cimatti, A., Dorigatti, M., Tonetta, S.: OCRA: a tool for checking the refinement of temporal contracts. In: Denney, E., Bultan, T., Zeller, A. (eds.) 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11–15, 2013, pp. 702–705. IEEE (2013)
20. Cimatti, A., Tonetta, S.: A property-based proof system for contract-based design. In: Cortellessa, V., Muccini, H., Demirörs, O. (eds.) 38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2012, Cesme, Izmir, Turkey, September 5–8, 2012, pp. 21–28. IEEE Computer Society (2012)
21. Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Sci. Comput. Progr. **97**, 333–348 (2015)
22. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E., Sistla, A. (eds.) Computer Aided Verification. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer, Berlin (2000)
23. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: LICS, pp. 353–362. IEEE Computer Society (1989)
24. Combemale, B., Gonnord, L., Rusu, V.: A generic tool for tracing executions back to a DSML's operational semantics. In: France, R.B., Küster, J.M., Bordbar, B., Paige R.F. (eds.) ECMFA, Lecture Notes in Computer Science, vol. 6698, pp. 35–51. Springer (2011)
25. Conquet, E., Dormoy, F.X., Dragomir, I., Graf, S., Lesens, D., Nienaltowski, P., Ober, I.: Formal model driven engineering for space onboard software. In: Proceedings of Embedded Real Time Software and Systems (ERTS2), Toulouse. SAE (2012)
26. Damm, W., Hungar, H., Josko, B., Peikenkamp, T., Stierand, I.: Using contract-based component specifications for virtual integration testing and architecture design. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2011, pp. 1–6 (2011). doi:10.1109/DATE.2011.5763167
27. David, A., Larsen, K.G., Legay, A., Møller, M.H., Nyman, U., Ravn, A.P., Skou, A., Wasowski, A.: Compositional verification of real-time systems using ECDAR. STTT **14**(6), 703–720 (2012)
28. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Methodologies for specification of real-time systems using timed I/O automata. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO, Lecture Notes in Computer Science, vol. 6286, pp. 290–310. Springer (2009)
29. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: ECDAR: an environment for compositional Design and analysis of real time systems. In: Proceedings of the 8th International Conference on Automated Technology for Verification and Analysis. ATVA'10, pp. 365–370. Springer, Berlin (2010)
30. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 91–100. ACM (2010)
31. de Alfaro, L., Henzinger, T.: Interface automata. In: Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM, pp. 109–120. Press (2001)
32. de Alfaro, L., Henzinger, T.: Interface theories for component-based design. In: Henzinger, T., Kirsch, C. (eds.) Embedded Software. Lecture Notes in Computer Science, vol. 2211, pp. 148–165. Springer, Berlin (2001)
33. de Alfaro, L., Henzinger, T., Stoelinga, M.: Timed interfaces. In: Sangiovanni-Vincentelli, A., Sifakis, J. (eds.) Embedded Software. Lecture Notes in Computer Science, vol. 2491, pp. 108–122. Springer, Berlin (2002)
34. Dragomir, I., Ober, I., Lesens, D.: A case study in formal system engineering with SysML. In: Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on, pp. 189–198 (2012)

35. Dragomir, I., Ober, I., Percebois, C.: Integrating Verifiable Assume/Guarantee Contracts in UML/SysML. Tech. Rep., IRIT (2013). http://www.irit.fr/Iulian.Ober/docs/TR-Syntax.pdf
36. Dragomir, I., Ober, I., Percebois, C.: Safety Contracts for Timed Reactive Components in SysML. Tech. Rep., IRIT (2013). http://www.irit.fr/Iulian.Ober/docs/TR-Contracts.pdf
37. Gacek, A., Katis, A., Whalen, M.W., Cofer, D.: Hierarchical Circular Compositional Reasoning. Tech. Rep. 2014-1, University of Minnesota Software Engineering Center, 200 Union St., Minneapolis, MN 55455 (2014)
38. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on, pp. 3–12 (2002)
39. Graf, S., Quinton, S.: Contracts for BIP: hierarchical interaction models for compositional verification. In: Derrick, J., Vain, J. (eds.) FORTE, Lecture Notes in Computer Science, vol. 4574, pp. 1–18. Springer (2007)
40. Grumberg, O., Long, D.E.: Model checking and modular verification. In: CONCUR, LNCS, vol. 527, pp. 250–265. Springer (1991)
41. Hafaiedh, I.B., Graf, S., Quinton, S.: Reasoning about safety and progress using contracts. In: Dong, J.S., Zhu H. (eds.) ICFEM, Lecture Notes in Computer Science, vol. 6447, pp. 436–451. Springer (2010)
42. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata, 2nd edn. Morgan and Claypool Publishers, San Rafael (2010)
43. Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In: Damm, W., Olderog, E.R. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 2469, pp. 395–414. Springer, Berlin (2002)
44. Larsen, K., Nyman, U., Wasowski, A.: Interface input/output automata. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods. Lecture Notes in Computer Science, vol. 4085, pp. 82–97. Springer, Berlin (2006)
45. Messabihi, M., André, P., Attiogbé, C.: Multilevel contracts for trusted components. In: International Workshop on Component and Service Interoperability, EPTCS, vol. 37, pp. 71–85 (2010)
46. Mikk, E., Lakhnechi, Y., Siegel, M.: Hierarchical automata as model for statecharts. In: Shyamasundar, R., Ueda, K. (eds.) Advances in Computing Science—ASIAN'97. Lecture Notes in Computer Science, vol. 1345, pp. 181–196. Springer, Berlin (1997)
47. Murugesan, A., Whalen, M.W., Rayadurgam, S., Heimdahl, M.P.: Compositional verification of a medical device system. Ada Lett. **33**(3), 51–64 (2013)
48. Ober, I., Dragomir, I.: OMEGA2: a new version of the profile and the tools. In: Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on, pp. 373–378. IEEE (2010)
49. Ober, I., Dragomir, I.: Unambiguous UML composite structures: the OMEGA2 experience. In: Cerná, I., Gyimóthy, T., Hromkovic, J., Jeffery, K.G., Královic, R., Vukolic, M., Wolf, S. (eds.) SOFSEM, Lecture Notes in Computer Science, vol. 6543, pp. 418–430. Springer (2011)
50. Ober, I., Graf, S., Ober, I.: Validating timed UML models by simulation and verification. STTT **8**(2), 128–145 (2006)
51. Ober, I., Ober, I., Dragomir, I., Aboussoror, E.: UML/SysML semantic tunings. Innov. Syst. Softw. Eng. **7**(4), 257–264 (2011)
52. Object Management Group: Systems Modelling Language (SysML) v1.1 (2008). http://www.omg.org/spec/SysML/1.1/
53. Object Management Group: Unified Modelling Language (UML) v2.2 (2009). http://www.omg.org/UML/2.2/
54. Object Management Group: Object Constraint Language (OCL) v2.2 (2010). http://www.omg.org/spec/OCL/2.2/
55. Object Management Group: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems v1.1 (2011). http://www.omg.org/spec/MARTE/
56. Object Management Group: Semantics of a Foundational Subset For Executable UML Models (fUML) v1.1 (2013). http://www.omg.org/spec/FUML/1.1/
57. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: closing a decidability gap. In: Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on, pp. 54–63 (2004). doi:10.1109/LICS.2004.1319600
58. Parnas, D., Weiss, D.: Active design reviews: principles and practices. In: ICSE'85. IEEE Computer Society (1985)
59. Payne, R., Fitzgerald, J.: Contract-Based Interface Specification Language for Functional and Non-Functional Properties. Tech. Rep., Newcastle University (2011). http://www.ncl.ac.uk/computing/research/publication/176971
60. Peled, D.: Software Reliability Methods. Texts in Computer Science. Springer, Berlin (2001)
61. Quinton, S.: Design, vérification et implémentation de systèmes à composants. Ph.D. thesis, Université de Grenoble (2011)
62. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: SEFM'08, pp. 377–381 (2008)
63. SAE: Architecture Analysis and Design Language (AADL). Document No. AS5506/1 (2004). http://www.sae.org/technical/standards/AS5506/1
64. SPEEDS: D 2.5.4: Contract Specification Language (2008). http://speeds.eu.com/downloads/D_2_5_4_RE_Contract_Specification_Language.pdf
65. Wang, F.: Symbolic simulation-checking of dense-time automata. In: Raskin, J.F., Thiagarajan, P. (eds.) Formal Modeling and Analysis of Timed Systems. Lecture Notes in Computer Science, vol. 4763, pp. 352–368. Springer, Berlin (2007)
66. Wang, T., Sun, J., Liu, Y., Wang, X., Li, S.: Are timed automata bad for a specification language? Language inclusion checking for timed automata. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 8413, pp. 310–325. Springer, Berlin (2014)
67. Weis, T., Becker, C., Geihs, K., Plouzeau, N.: A UML meta-model for contract aware components. In: 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML) 2001, pp. 442–456. Springer (2001)
68. Whalen, M.W., Gacek, A., Cofer, D., Murugesan, A., Heimdahl, M.P., Rayadurgam, S.: Your "what" is my "how": iteration and hierarchy in system design. IEEE Softw. **30**(2), 54–60 (2013)

Iulia Dragomir is a postdoctoral researcher in the Department of Computer Science of Aalto University School of Science, Finland. She received a PhD in computer science from the University of Toulouse in 2014. Her research focuses on the correct design of real-time, embedded and cyber-physical systems. She is particularly interested in model-driven design, formal methods, computer-aided verification and compositionality, and their transfer in industrial applications. She has actively participated to the development of the Omega-IFx toolbox, which bridges UML/SysML models and model checking and which has been employed in the design of several aeronautics systems.



Iulian Ober is an Associate Professor at the University of Toulouse, France. He obtained a PhD from the Institut National Polytechnique of Toulouse, in 2001. Until 2005, he was a post-doctoral researcher at the Verimag Laboratory in Grenoble, France. He has joined the faculty at the University of Toulouse in 2005. His research area covers model-driven design and validation of real-time embedded systems, using industry standards such UML or SysML and SDL. Since 2002, Iulian has been leading the development of Omega-IFx (http://www.irit.fr/ifx), a model simulation and verification platform for UML and SysML models, which started as part of the IST Omega project and was further developed in several EU and ESA projects. In this context, he has developed model semantics and analysis techniques that have been successfully used in several industrial-grade projects in the aerospace domain.



Christian Percebois is a professor of Computer Science at the University of Toulouse since 1992. He was always interested in software engineering. He worked on Lisp and Prolog interpreters, garbage collecting for symbolic computations, asynchronous backtrackable communications in parallel logic languages, abstract machine construction through operational semantics refinements, typing in object-oriented programming and multiset rewriting techniques in order to coordinate concurrent objects. Today, his main research tries to combine formal methods and software engineering, in particular for graph rewriting systems.