AsmetaComp: a Tool for Runtime Contract Checking with I/O Abstract State Machines*

Silvia Bonfanti[1 [0000−0001−9679−4551]], Angelo Gargantini[1 [0000−0002−4035−0131]], Elvinia Riccobene[2 [0000−0002−1400−1026]], and Patrizia Scandurra[1 [0000−0002−9209−3624]]

[1] University of Bergamo, Bergamo, Italy
{silvia.bonfanti,angelo.gargantini,patrizia.scandurra}@unibg.it
[2] Università degli Studi di Milano, Milano, Italy elvinia.riccobene@unimi.it

Abstract. Contract-based Design specifies components via *assume–guarantee* contracts, enabling modular development and verification. In state-based system modeling, it helps manage complexity through decomposition into analyzable sub-models coordinated by I/O events. However, its adoption is hindered by the lack of integrated languages and tools for the specification and verification of component contracts.
This paper presents AsmetaComp, a tool that supports Contract-based Design for systems specified using I/O Abstract State Machines (ASMs), a state-based formalism for modeling and analyzing discrete-event systems. Originally developed for compositional simulation of interacting I/O ASM models, AsmetaComp has been extended to support Contract-based Design by integrating assume–guarantee contracts with executable I/O ASM models and enabling automated runtime contract checking during compositional execution. A healthcare case study is used to illustrate the tool and evaluate its effectiveness in supporting modular reasoning and dynamic contract checking in safety-critical, state-based systems.

Keywords: Contract-based Design · Models composition · Runtime Contract Checking · Abstract State Machines

1 Introduction

Contract-based Design (CbD) [6,7,27,20] is a well-established paradigm for building reliable systems through the explicit specification of behavioral *contracts*. Contracts define assumptions on inputs, guarantees on outputs, and invariants preserved during execution, supporting assume/guarantee reasoning and modular verification. While Design-by-Contract (DbC) [32] introduced contracts at the programming level, CbD generalizes these principles to components and systems, enabling scalable and compositional design.

Despite its conceptual maturity, practical tool support for CbD in formal system modeling remains fragmented. Existing solutions either embed contracts

* Artefacts available in https://doi.org/10.5281/zenodo.18678488.

within programming languages (e.g., Eiffel, JML, ACSL) or provide partial support within verification frameworks. However, seamlessly integrating contract specification, compositional modeling, and automated runtime checking in a unified environment is still a challenge [25,8,21].

In previous work [13,14], we introduced I/O Abstract State Machines (I/O ASMs) as a state-based formalism for modeling interacting components in discrete-event systems. We also developed `AsmetaComp`, a compositional simulation engine that orchestrates the execution of I/O ASM components through dedicated composition operators.

This paper presents a tool-supported contract-based framework built on top of I/O ASMs and their simulation engine [13,14]. Our contribution is threefold:

- We introduce a dedicated contract language for I/O ASMs, explicitly reflecting the assume/guarantee paradigm.
- We define inference rules for compositional operators, enabling systematic derivation of contracts for composed components.
- We extend the `AsmetaComp` engine to support automated runtime contract checking during compositional simulation.

This framework provides a unified environment in which component modeling, contract specification, and contract checking are seamlessly combined. Contracts are checked dynamically during orchestrated execution of models. Unlike traditional contract-based approaches that primarily rely on static compositional verification through model checking, our framework supports runtime validation under concrete execution scenarios. Rather than aiming at exhaustive state-space exploration, it complements static verification by enabling scalable, scenario-driven detection of contract violations in dynamically composed systems.

To show the tool applicability, we apply the framework to a healthcare system case study, illustrating how runtime contract monitoring supports correctness assurance and complexity management in safety-critical, state-based systems.

The paper is structured as follows. Section 2 relates our work within existing CbD frameworks. Section 3 introduces the case study. Section 4 provides background on I/O ASMs and their composition operators. Section 5 recalls component contracts in CbD and presents CbD for I/O ASMs as the underlying theory of the tool. Section 6 describes the `AsmetaComp` tool for CbD. Section 7 reports the validation results of the case study. Section **??** provides information about the tool artifact, and Section 8 concludes the paper.

2 Related work

We focus on tool-supported formal approaches to CbD, distinguishing between those aimed at contract verification and refinement, and those oriented toward simulation and runtime contract checking.

On CbD for formal compositional verification and refinement. Traditionally, CbD has been used to manage complexity and scalability in embedded system verification. Together with assume/guarantee reasoning, it mainly supports

formal compositional verification and stepwise refinement from design to implementation. We highlight a few, inspiring contributions in this direction.

Cimatti et al. [20] propose a trace-based contract framework built upon HRELTL, an expressive specification language extending Linear Temporal Logic (LTL) to hybrid traces, together with the OCRA verification engine for assessing contract refinement. Their approach analyzes the verification conditions necessary to establish the correctness of contract refinement in a *top-down decomposition*, emphasizing the critical role of architectural connections in component composition. The notion of contract decomposition and refinement is also explored in [35], where contract patterns are translated into automata and contract refinement is defined in terms of trace-set inclusion. Related A/G (assume/guarantee) contract frameworks are based on variants of I/O automata, e.g., Interface Automata [29] and choreography automata [24]. AGREE provides A/G compositional reasoning for architecture models [22], while an information-flow–based tool for enforcing system-wide security properties is presented in [3]. Bocchi et al. [9] propose a Design-by-Contract framework for distributed multiparty interactions, based on global assertions, their projection onto endpoint assertions, and a compositional proof system for process verification. Their approach provides a rigorous formal basis for specifying and verifying multiparty protocols by enriching session types with logical predicates, thus mainly supporting reasoning at design time. All these frameworks operate at the design stage, abstracting away from component implementations.

Other approaches more closely tied to component implementations are the following. In [19], a compositional specification theory is proposed for components interacting via I/O synchronization. Specifications constrain the temporal ordering of interactions and support composability, assumption modeling, and runtime reasoning, using either operational models (I/O-labeled transition systems with inconsistency predicates) or trace-based declarative models. Inconsistency is captured over states or traces, refinement ensures safe substitutability, and operators such as parallel composition, conjunction, and quotient support incremental design. In [30], a methodology applies contract theory to embedded software design by instantiating contracts with component-level syntaxes and enabling formal reasoning via semantically parameterized abstractions. The approach is demonstrated on a realistic example using TLA (Temporal Logic of Actions) at the system level and ACSL pre/post-conditions for C components. More recently, [31] proposes a runtime verification framework that uses LTL to express contracts and complements static model checking by monitoring individual execution traces.

On runtime contract checking. While formal verification remains the dominant application, newer research is broadening its scope toward compositional system design and analysis in different formalisms, and practical composition workflows with contract checking [37]. The work in [36], for example, introduces an approach for automatically checking semantic constraints by transforming Palladio architecture models into MontiArc, where constraint validation is performed using the A/G paradigm. By combining evaluation and soundness within a single

framework, the method enhances architecture reliability and avoids redundant modeling across different formalisms. Our approach pursues a similar intent, i.e. to ensure the semantic soundness of compositions during composition. Our focus is not on proving global correctness properties exhaustively, but on enabling compositional runtime assurance through executable contract-aware models.

Formal approaches to CbD that rely on simulation/testing of models and runtime contract checking include CARE [4] with Uppaal model checker, runtime verification in Simulink [23], and a stochastic contract framework for real-time systems [33] (to name a few). These methods extend traditional A/G reasoning by validating contracts dynamically during execution, and share aspects with the synthesis/verification of runtime monitors ([4], [33]) and quantitative verification via statistical inference of properties [33]. Our compositional simulation approach shares the same intent of runtime checking, and like in [4] is a *bottom-up analysis*.

In line to support continuous assurance, we aim to use ASMs as runtime formal models [18], exploiting their feature of being executable besides being verifiable. Like runtime monitoring, runtime simulation identifies violations of A/G assertions only after they occur, recording the corresponding execution runs. Since these techniques analyze one run at a time, they scale efficiently and do not suffer from state-space explosion. However, unlike model checking, they cannot ensure the complete absence of constraint violations.

Channel-based coordination models for component composition have been proposed in the literature, including Reo [2] and BIP [5]. Reo is an exogenous coordination model in which complex *connectors* are built by composing simpler ones, with the emphasis placed on connectors rather than on the interacting entities themselves. BIP (*Behaviour-Interaction-Priority*) is a component-based framework for heterogeneous concurrent systems that separates component behavior from coordination. In BIP, systems are constructed by coordinating atomic components through *connectors*, which define interaction patterns, and *priorities*, which resolve conflicts among enabled interactions. It also supports validation of key safety properties and has been applied to real-time embedded systems, including industrial case studies such as robot controllers and satellite on-board software. Reo and BIP primarily focus on coordination and component interaction rather than providing an explicit CbD in the classical assume/guarantee sense. This is especially true for Reo, while contract-based reasoning has been investigated in connection with the BIP framework in subsequent works. For example, Quinton et al. address contract-based verification of hierarchical component systems using BIP glue operators [34], while later work studies contract-based reasoning for component systems with rich interactions, explicitly considering BIP as a representative framework [26].

3 Running case study

We illustrate the usage of AsmetaComp through a system example in the medical domain, modeled as a composition of I/O ASMs. Inspired by [15,14], the case study (shown in Fig.1) is about a *Medicine Reminder and Monitoring (MRM)*

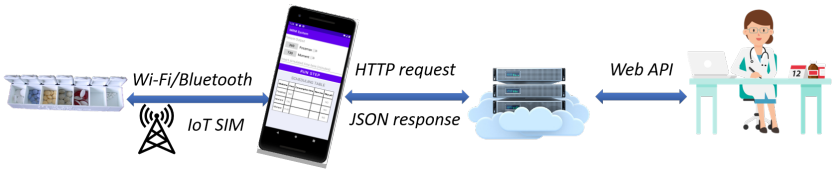

Fig. 1. MRM system: high-level interaction flow (taken from [15]).

System designed to control a smart medicine dispenser (a remotely controllable pillbox) for medicine administration. The (client side) system consists of two main sub-systems:

- *Electronic pill reminder and dispenser* (pillbox) with several compartments and slots per compartment;
- *Remote control mobile app* running on a smartphone, which monitors and configures the pillbox via local wireless (Bluetooth or Wi-Fi Direct).

A cloud-based information system provides persistent services for health records, prescriptions, and treatments entered by doctors through a web application. In this architecture, the smartphone acts as a *gateway* between the pillbox and the cloud, to synchronize patient data with the cloud using the smartphone's Internet connection (Wi-Fi or cellular). Our running example focuses only on the patient side system. The mobile app is responsible for checking the pillbox configuration after it is filled by the caregiver and connected to power, and for dynamically rescheduling pills to ensure safe drug intake. Unlike typical systems that only notify caregivers of missed doses, this solution automatically reschedules missed doses while ensuring no harmful drug interferences occur.

4 Background concepts

This section introduces the formal method underlying the tool, along with the specification of components and their compositional execution semantics.

4.1 Abstract State Machines and the ASMETA toolset

ASMs [17,16] extend Finite State Machines (FSMs) where unstructured FSM control states are replaced by first-order logic structures comprising domains of objects with functions defined on them, and state *transitions* are expressed by transition rules describing how function values saved into *locations* [3] change from one state to the next. Functions are *static* or *dynamic*. Dynamic functions can be *monitored* (input values), *controlled* (internal state), *out* (output values), or *derived* (computed in terms of other functions).

The basic unit of rule construction is the *update* rule of the form: ***if*** *condition* ***then*** *Updates* where *Updates* is a set of simultaneous assignments $f(t_1, \ldots, t_n) :=$

[3] An ASM location is a memory unit represented by a function together with specific argument values, identifying where a value is stored in a given state.

v, being f an n-ary dynamic function, t_i terms, and v the value of $f(t_1, \ldots, t_n)$ in the next state. Other transition rules *constructors* are used depending on the update structure they express: guarded updates (*if-then-else*), parallel updates (*par*), non-determinism (*choose*), unrestricted synchronous parallelism (*forall*).

An ASM model *run* is a finite or infinite sequence $S_0, S_1, \ldots, S_i, \ldots$ of states: starting from an initial state S_0, a *run step* from S_i to S_{i+1} consists of the execution of all the transition rules invoked from a *main rule*[4] and enabled to fire, and leads to simultaneous updates of a number of locations. The model execution fails if an inconsistent update occurs, when a location is assigned conflicting values, or if an invariant, expressed as a first-order logic formula that must hold in every state, is violated.

ASMETA (ASM mETAmodeling) [1,10] is a set of tools based on ASMs to support the specification, validation, and verification of systems' behavior. In this work, we rely on ASMETA, using it as a comprehensive toolset to specify ASM models and to analyze their correctness in a systematic manner.

An ASMETA model is specified in a `.asm` file using the textual language `AsmetaL`. Listing 1 shows a model excerpt, which is structured into five sections:
- `import`: Imports definitions from external modules, including the StandardLibrary which provides names for basic domains and functions for the operations over these domains (line 2).
- `signature`: Declares domains and functions modeling knowledge about the system (lines 3-21).
- `definitions`: Contains static functions, transition rules declaration, and invariants – first-order formulas that must hold in all states – (lines 22-38).
- `main rule`: Defines the entry point for computation at each run step. It may invoke other transition rules by name[5].
- `default init`: Initializes dynamic domains and controlled/out functions.

4.2 I/O Abstract State Machines

In [13,14], we defined an *I/O ASM* as an ASM with input/output interfaces, and orchestration operators enabling compositional simulation of I/O ASMs that interact via I/O events.

Definition 1 (I/O ASM). *An* I/O ASM *is an ASM m with a non-empty set I_m of input (or monitored) functions and a non-empty set O_m of output (or out) functions in its signature. We denote by (I_m, m, O_m) an I/O ASM, and by curr_state(m) the set of its locations values.*

Case study. Fig. 2 depicts several examples of I/O ASM for the MRM system in a black-box manner. The I/O interface, defined in terms of monitored and out ASM functions, represents the set of interaction ports (or points) with the

[4] According to the working definition of ASM model implemented in ASMETA.

[5] A named rule in the `definitions` section is defined as `r_rule(`*`params`*`)`, while it is invoked from another rule as `r_rule[`*`params`*`]`.

```
1  asm configurationMgr
2  import StandardLibrary
3  signature:
4   enum domain States = {INIT | NORMAL}
5   ...
6   //INPUT from Pillbox
7   monitored time_consumption: Compartment −> Seq(Natural)
8   monitored name: Compartment −> String
9   monitored drugIndex: Compartment −> Natural
10  monitored redLed: Compartment −> LedLights
11  //INPUT from prescription
12  monitored medicine_list: Seq(String)
13  monitored amount: String −> Natural
14  ...
15  controlled state: States
16  controlled prevRedLed: Compartment −> LedLights
17  ...
18  derived ledStatusUpdateOk: Compartment −> Boolean
19  static compartment1: Compartment
20  static compartment2: Compartment
21  ...
22 definitions:
23  function ledStatusUpdateOk($compartment in Compartment) =
24          ((prevRedLed($compartment) = OFF and redLed($compartment)= BLINKING) or
25          (prevRedLed($compartment) = BLINKING and redLed($compartment)= ON))

27  rule r_INIT =
28          par
29              forall $s in asSet(medicine_list) do extend Medicine with $m do id($m) := $s
30              state := NORMAL
31          endpar
32  //The only valid traces of a led light are the sequences: OFF, ON, BLINKING and BLINKING, OFF
33  invariant inv_A_redLed over redLed: ( forall $c in Compartment with not ledStatusUpdateOk($c) )
34  //Pills type consistency between the prescription and the compartments of the pillbox
35  invariant inv_A_medicineList over name(Compartment): ( forall $c in Compartment with (contains(
        medicine_list,name($c)))) // compartments contain only prescribed pills
36  ...
37  main rule r_Main = if state = INIT then r_INIT[] else r_keepPrevLiths[] endif
38  default init s0:
39     function state = INIT
40     function medicine_list = ["fosamax","moment"] // Medicine prescription (e.g., from a JSON file)
41     ...
```

Listing 1. Excerpt of an ASM model in `AsmetaL` from the MRM running example.

environment or other ASMs, while the set of controlled locations represents the internal state of the I/O ASM. An excerpt of the ASM *ConfigurationMgr* has been already reported in Listing 1.

4.3 Compositional execution of I/O ASMs

I/O ASMs can be connected and interact *by binding* [13,14] input and out functions with the same name and type. The resulting *assembly* of I/O ASM models can be then executed according to a *composition formula* denoting an orchestration schema for the co-simulation of the composed I/O ASMs. A composition formula uses the composition operators defined in [13,14]. Formally, a (recursive) composition formula of I/O ASMs is defined as follows:

Table 1. I/O ASM composition operators (from [14]).

Operator name	Symbol	Semantics description
(Simplex) pipe or sequence	$m_1 \mid m_2$	First execute m_1, then use the output of m_1 as input to execute m_2.
Half-duplex bidirectional pipe	$m_1 <\mid> m_2$	First execute m_1, subsequently use the output of m_1 as input to execute m_2; then, return the output of m_2 as input to m_1 for its next future execution (and not the current one).
Full-duplex bidirectional pipe	$m_1 <\parallel> m_2$	First execute both m_1 and m_2 simultaneously, then return the outputs of m_1 to m_2 and the outputs of m_2 to m_1 for their next execution.
Synchronous parallel split (or fork-join)	$m_1 \parallel m_2$	Execute both m_1 and m_2 separately on their inputs, then return the union of their outputs.

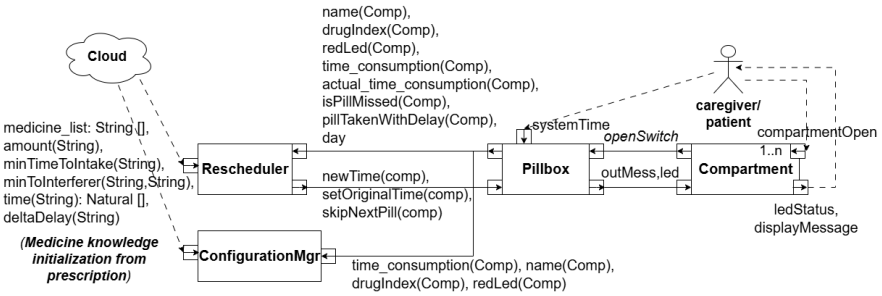

Fig. 2. MRM I/O ASM assembly.

Definition 2 (I/O ASM composition formula). *A* composition *formula c over an assembly A of I/O ASMs is: a single I/O ASM m of A or $c_1 \mid c_2$ or $c_1 <\mid> c_2$ or $c_1 <\parallel> c_2$ or $c_1 \parallel c_2$ where c_1 and c_2 are composition formulas and $\{\mid, <\mid>, <\parallel>, \parallel\}$ are composition operators.*

Table 1 reports the basic binary composition operators over two I/O ASMs m_1 and m_2 (the basic cases of composition formula in Def 2 when operands are I/O ASMs), and informally recalls their execution semantics. More details on the simulation mechanism of an I/O ASM assembly can be found in [13,14].

Case study. Fig. 2 shows the I/O ASM assembly (the architecture) of the MRM system. To ensure patient safety, the mobile app includes two key components:

- *ConfigurationMgr*, which checks the actual pillbox configuration against the medicine prescription;
- *Rescheduler*, which manages critical situations by applying corrective actions on the time scheduling for medicine intake when possible.

The user manually fills the pillbox compartments' slots (one medicine type per compartment) according to the prescription. A patient's prescription contains medicine details like: name, amount, minimum time separation between drugs, time schedule, and retry intervals for missed doses. This data are used to initialize the configuration manager and the rescheduler. When it is time to

take a medicine, the pillbox sends a notification message to the caregiver/patient through the corresponding compartment, which lights up and unlocks. The patient confirms intake by opening and closing the compartment. If the medicine is not taken within 10 minutes, the compartment's LED flashes for another 10 minutes before the dose is marked as missed. Runtime knowledge, including the current pillbox configuration, the state of each compartment, the actual intake time of a pill, etc., is constantly monitored by the Pillbox and sent in a targeted way to the configuration manager and the rescheduler according to their supervisioning goal.

Once all of the components have been initialized, the MRM assembly can be executed using the composition formula:

$$((PB <|> (CM \parallel RES)) <|> (C_1 \parallel \ldots \parallel C_n))$$

where PB is the *Pillbox* component, CM is the *ConfigurationMgr*, RES is the *Rescheduler*, and $C_1 \ldots C_n$ are different compartments of the pill dispenser. Intuitively, the pillbox operates first, and its intended actions are then supervised by the *ConfigurationMgr*—which verifies the pillbox configuration against the prescription—and by the *Rescheduler*, which may adjust the medication schedule if needed, before the final confirmation and actuation of the compartments.

5 Underlying theory: CbD for compositional I/O ASMs

In CbD [6,7,27,20], a system architecture is defined by means of components, which encompass both implementations and contracts. An implementation is an instantiation of a component. A contract is a description of the expected interaction of the component with the environment (i.e., other components). A contract specifies the expected behavior of a component by defining the *assumptions* A that must be satisfied by the environment and the *guarantees* G satisfied by the component in response. Thus, we can denote the contract of a component c as the pair (A, G), and, following the Hoare-style notation [28], we express contract satisfaction as the triple $\{A\}c\{G\}$ meaning that, if the component implementation c is executed from a state where A holds (the pre-state) and its execution terminates, then G will hold in the resulting state (the post-state).

Following this assume-guarantee paradigm, we define the contract of an I/O ASM, which, in our framework, models a component behavior (i.e., the implementation). In our underlying theory, we take into consideration that the externally observable part of a component model is the set of I/O location values, which change along an ASM run (see Section 4.1)[6].

Definition 3 (Contract of an I/O ASM). *The contract* ctr_m *for an* I/O ASM (I_m, m, O_m) *is the pair* $ctr_m = (A_m, G_m)$ *where assumptions* A_m *are assertions (i.e., first-order logic formulas) over* I_m *and guarantees* $\{G_m\}$ *are assertions (i.e., first-order logic formulas) over* O_m.

[6] This notion is similar to the notion of *trace* introduced in the contract framework presented in [20].

Case study. Listing 1 reports two of the assumptions in A_{MRM} of the MRM component: the first one, `inv_A_redLed` (see line 33), checks the valid sequence of led lights in the compartment of the MRM system; the second one, `inv_A_medicineList` (see line 35), verifies that the medicine loaded in each compartment corresponds to those prescribed by the doctor (see the ASMETA model of the *ConfigurationMgr* in the replication package [11] for the complete list of the MRM component's assumptions).

We now define the condition for an I/O ASM to satisfy its contract.

Definition 4 (Contract satisfaction of an I/O ASM). *We say that an I/O ASM m satisfies a contract $ctr_m = (A_m, G_m)$ iff every time m is executed from a current state where the assumptions A_m hold and its execution step terminates, then the guarantees G_m will hold in the resulting next state. In this case, we can write: $\{A_m\}\ m\ \{G_m\}$.*

To deal with the composition of contracts for I/O ASMs and ensure the correctness of an I/O ASM composition formula (so called *compositional correctness*) w.r.t. a contract, we define how contract composition is expressed and computed for the basic composition operators of I/O ASMs. For this purpose, we introduce inference rules defining how to derive the composition of the contracts from the sub-component's contracts and suitable logical constraints.

To express these inference rules, we use the following notation:

$$\frac{\{A_{c_1}\}c_1\{G_{c_1}\},\ \{A_{c_2}\}c_2\{G_{c_2}\}\dots\{A_{c_n}\}c_n\{G_{c_n}\}\ , l_1 \dots l_n}{\{A_c\}c = \bigotimes(c_1, \cdots, c_n)\{G_c\}}$$

where $\bigotimes(c_1, \cdots, c_n)$ is the composite model c of the n component models $c_1, \cdots, c_n$ according to a composition formula c (see Definition 2 in Section 4). The rule states that if the components $c_1, \dots, c_n$ satisfy their contracts and the logical assertions $l_1 \dots l_n$ hold, then we can derive the contract for the composed model c with assumption A_c and guarantee G_c.

Note that every composition rule holds modulo invariants, i.e., assuming that the state invariants of an I/O ASM are always preserved.

The definition of the contract for each compositional operator is given below.

Definition 5 (Contract of composition $c_1 \mid c_2$). *The contract of composition $c_1 \mid c_2$ is defined as:*

$$\frac{\{A_{c_1}\}\ c_1\ \{G_{c_1}\}, \{A_{c_2}\}\ c_2\ \{G_{c_2}\}, G_{c_1} \Rightarrow A_{c_2}}{\{A_{c_1}\}\ c_1 \mid c_2\ \{G_{c_2}\}}$$

thus: $A_{c_1|c_2} \equiv A_{c_1}$ and $G_{c_1|c_2} \equiv G_{c_2}$.

Definition 6 (Contract of composition $c_1||c_2$). *The contract of composition $c_1 \parallel c_2$ is defined as:*

$$\frac{\{A_{c_1}\}\ c_1\ \{G_{c_1}\}, \{A_{c_2}\}\ c_2\ \{G_{c_2}\}}{\{A_{c_1} \wedge A_{c_2}\}\ c_1 \parallel c_2\ \{G_{c_1} \wedge G_{c_2}\}}$$

thus: $A_{c_1\|c_2} \equiv A_{c_1} \wedge A_{c_2}$ and $G_{c_1\|c_2} \equiv G_{c_1} \wedge G_{c_2}$.

Definition 7 (Contract composition $c_1 <|> c_2$). *The contract of composition $c_1 <|> c_2$ is defined as:*

$$\frac{\{A_{c_1}\}\ c_1\ \{G_{c_1}\}, \{A_{c_2}\}\ c_2\ \{G_{c_2}\}, G_{c_1} \Rightarrow A_{c_2}, G_{c_2} \Rightarrow A'_{c_1}}{\{A_{c_1}\}\ c_1 <|> c_2\ \{A'_{c_1}\}}$$

where A'_{c_1} is the evaluation of the assumptions of component c_1 after executing c_2; thus: $A_{c_1<|>c_2} \equiv A_{c_1}$ and $G_{c_1<|>c_2} \equiv A'_{c_1}$.

Definition 8 (Contract of composition $c_1 <||> c_2$). *The contract of composition $c_1 <||> c_2$ is defined as:*

$$\frac{\{A_{c_1}\}\ c_1\ \{G_{c_1}\}, \{A_{c_2}\}\ c_2\ \{G_{c_2}\}, G_{c_1} \Rightarrow A'_{c_2}, G_{c_2} \Rightarrow A'_{c_1}}{\{A_{c_1} \wedge A_{c_2}\}\ c_1 <||> c_2\ \{A'_{c_1} \wedge A'_{c_2}\}}$$

where A'_{c_1} is the evaluation of the assumptions of component c_1 after executing c_2, and A'_{c_2} is the evaluation of the assumptions of component c_2 after executing c_1; thus: $A_{c_1<||>c_2} \equiv A_{c_1} \wedge A_{c_2}$ and $G_{c_1<||>c_2} \equiv A'_{c_1} \wedge A'_{c_2}$.

These inference rules are sound; formal proofs are omitted, as the paper focuses on the tool[7]. Besides the rules presented above, classical rules that strengthen the assumption A and/or weaken the guarantee G [28] can be used.

Case study. As an example of contract composition, let us consider the parallel composition $CM \mid\mid RES$ in the MRM system. The contract of this composition, denoted as $ctr_{CM||RES}$, is obtained by combining the assumptions A and guarantees G of the two specifications CM and RES according to Def 6. Given the following contracts for CM and RES[8]:

- $ctr_{CM} = (A_{CM}, G_{CM})$, where A_{CM}={`inv_A_redLed`, `inv_A_medicineList`, `inv_A_compName`, `inv_A_CompSize`, `inv_A_CompMed`, `inv_A_timeConsumption`}, G_{CM}=true
- $ctr_{RES} = (A_{RES}, G_{RES})$, where A_{RES}={`inv_A_actual_time`}, G_{RES}={`inv_G_newTime`, `inv_G_skipNextPill`}

the contract of the composition $CM \mid\mid RES$ is:
$ctr_{CM||RES} = (A_{CM||RES}, G_{CM||RES})$, where $A_{CM||RES} = A_{CM} \wedge A_{RES}$ ={`inv_A_redLed`, `inv_A_medicineList`, `inv_A_comp_Name`,`inv_A_CompSize`, `inv_A_CompMed`, `inv_A_time Consumption`,`inv_A_actual_time`} and $G_{CM||RES} = G_{CM} \wedge G_{RES}$ ={`inv_G_newTime`, `inv_G_skipNextPill`}.

6 The AsmetaComp tool for Runtime Contract Checking

The inference approach for deriving and checking composed contracts presented in the previous section has been implemented by extending the ASMETA tool `AsmetaComp` [14]. This section presents the AsmetaComp architecture, the notations for contract

[7] The proof of the definitions' soundness is available as additional material associated with this paper at [12].

[8] The complete ASMETA models of CM and RES are in the replication package [11].

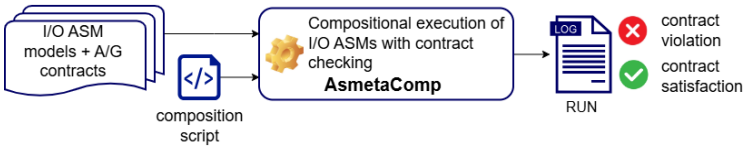

Fig. 3. `AsmetaComp` and artifacts.

definition and composition, the mechanism for runtime contract checking, and a summary of the tool features and its limitations.

A replication package containing the `AsmetaComp` tool (a Java archive file), all MRM ASMETA models, and composition scripts for validation scenarios reproducibility are available online at [11]. A demo video of the tool is also available online at: `https://foselab.unibg.it/asmeta/videos/AsmetaComp@FORTE26.mp4`.

6.1 Overview and component architecture

For contract checking, `AsmetaComp` can be invoked via a Java CLI program (see Fig. 3) that reads from a script a sequence of commands for setting (*setup*) and running (*run*) the composition formula(s) for I/O ASMs. I/O ASMs are ASMETA models that are written in the `AsmetaL` textual notation, and enriched with A/G contracts. They are executed in a compositional manner, with the contracts being validated inline on the models themselves. The output of the compositional simulation is an ASM run that is recorded on the standard console.

The core component of `AsmetaComp` is `CompositionManager`, which implements the interface `IModelComposition` to simulate a composition—represented by the abstract class `Composition`—of ASMETA models (see Fig. 4). It may throw exceptions during the compositional simulation in case a model execution fails (the exception class `CompositionException`) or an A/G contract is violated (the class `CBDException`). The methods `eval()` of `Composition` and `run()` of `IModelComposition` have been appropriately overloaded to enable (through a boolean flag) runtime contract checking. Details on the class hierarchy representing the entity types involved in the composition of I/O ASMs, where the abstract class `Composition` serves as the root node, and on the compositional simulation algorithm over these entities are in [14].

6.2 Contract definition and composition

As *assertion language* to define contracts we adopt the `AsmetaL` textual notation. Contracts and components are therefore specified in a unified way using the same modeling notation. Specifically, contracts are defined in terms of *state invariants* of an I/O ASM model. The variables that are used in the contracts are locations local to the owning I/O ASM. Concretely, *assumptions* A are state invariants over *input locations* of the owning I/O ASM, while *guarantees* G are state invariants over *output locations* of the I/O ASM. At each model execution step, `AsmetaS`, the model simulator in ASMETA, checks for the state invariants over the input locations, and over the output locations before updating them according to the ASM transition rules. A failure of the A invariants means the environment misuses the model, a failure of the invariant G means that the model operates incorrectly.

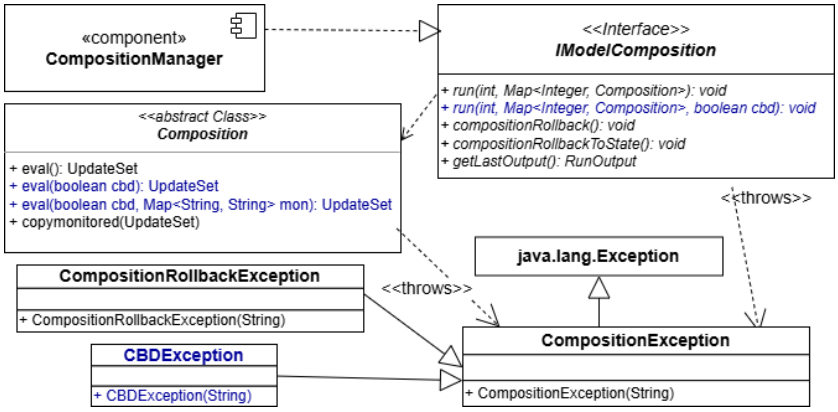

Fig. 4. `AsmetaComp` class diagram: blue elements are extensions for contract checking.

To compose and run I/O ASMs, `AsmetaComp` provides a lightweight *scripting language*, executed via a Java-based CLI (see Fig. 3). A composition script contains *setup* and *run* commands to, respectively, define and execute I/O ASM composition formulae. An example of composition script for the MRM system is reported in Listing 2. Note that ASM models can be closed (all input locations are bound to output locations of other models) or open (some input locations are unbounded or, in other words, are bound to an abstract environment). For open models, the key–value pairs within curly braces in a *run* command provide an explicit initialization for unbound input locations, thus all models before running a composition are closed. For example, in Listing 2 (see line 11), before executing the composed system, initialization runs are performed. The scheduling subsystem is initialized (see line 8), while each compartment is set to closed, its LED is switched off, and a unique identifier (`myID`) is assigned (see lines 9 - 10). The next commands advance the fully composed system (`pillbox_comp`). In each run (from line 11), both compartments remain closed, and the global clock `systemTime` (simulated time, in minutes) is assigned a value. The chosen time stamps correspond to moments of interest for observing scheduling and dispensing behaviors. Running the system at these discrete times allows systematic evaluation of the timed interactions among all of the components.

6.3 Runtime Contract Checking Mechanism

During the execution of an I/O ASM composition, `AsmetaComp` leverages the *invariant checking* mechanism of the `AsmetaS` simulator to check the A/G assertions of the whole composition. Indeed, during the execution of a composition operator in the I/O ASM composition, `AsmetaComp` checks for the validation of the components' contract and the further assertions required in the premises of the inference rule of the contract composition operator (see definitions of the inference rules in Section 5), so guaranteeing the contract of the resulting composition before proceeding. On the atomic components, the contract satisfaction is performed as previously explained on the single model. In case of contract violation, `AsmetaComp` throws an exception, which keeps track of the violated A/G assertion. This violation is recorded in the output console; the composite model execution fails, but models are rolled-back to their previous state.

```
1  setup CM as Pillbox_CBD/configurationMgr.asm
2  setup RES as Pillbox_CBD/rescheduler.asm
3  setup C1 as Pillbox_CBD/compartment.asm
4  setup C2 as Pillbox_CBD/compartment.asm
5  setup PB as Pillbox_CBD/pillbox.asm
6  setup setup_comp as ( CM || RES )
7  setup pillbox_comp as ( ( PB <|> ( CM || RES ) ) <|> ( C1 || C2 ) )
8  run(setup_comp, {pillboxSystemTime=0;day=0;isPillMissed(compartment1)=false;pillTakenWithDelay(
       compartment1)=false;isPillMissed(compartment2)=false;pillTakenWithDelay(compartment2)=false;
       redLed(compartment1)=OFF;redLed(compartment2)=OFF;name(compartment1)="fosamax";name(
       compartment2)="moment";time_consumption(compartment1)=[360];time_consumption(compartment2)
       =[730,1140];actual_time_consumption(compartment1)=[0];actual_time_consumption(compartment2)
       =[0,0];drugIndex(compartment1)=0;drugIndex(compartment2)=0})
9  run(C1, {compartmentOpen(1)=false;outMess(1)="";led(1)=OFF;myID=1})
10 run(C2, {myID=2;compartmentOpen(2)=false;outMess(2)="";led(2)=OFF})
11 run(pillbox_comp, {compartmentOpen(1)=false;compartmentOpen(2)=false;systemTime=10n})
12 run(pillbox_comp, {compartmentOpen(1)=false;compartmentOpen(2)=false;systemTime=361n})
13 run(pillbox_comp, {compartmentOpen(1)=false;compartmentOpen(2)=false;systemTime=365n}) ...
```

Listing 2. Composition script

6.4 Tool features and limitations

Overall, `AsmetaComp` offers an executable environment that integrates modeling, composition, and runtime contract verification, providing native support for Contract-based Design with I/O ASMs and the following ***key functionalities***:

- **Unified Modeling and Contracts.** The tool supports specifying both component behavior and A/G contracts within a unified modeling environment, using a contract language aligned with I/O ASM semantics.
- **Compositional Execution.** Independent components modeled as I/O ASMs can be composed using dedicated operators. The execution engine orchestrates their interaction according to the defined composition semantics.
- **Inference Support for Composed Contracts.** The framework provides inference rules that allow the systematic derivation of contracts for composed components from the contracts of their constituents.
- **Automated Runtime Contract Checking.** During compositional simulation, contracts are automatically checked, enabling early detection of assumption violations and guarantee breaches as components interact.

Despite its capabilities, the current implementation exhibits ***limitations*** primarily related to *scalability* and the *user-friendly output interface*. Broader empirical validation is required to assess scalability and usability in large-scale industrial case studies. In addition, improving the output interface would facilitate clearer reporting of contract violations, execution traces, and verification results in a more accessible and interpretable format. These aspects will be addressed as future improvements of our tool.

7 Validation

This section presents the scenario-based experimental validation of AsmetaComp on the MRM case study, framed around the following research question:

RQ — Can `AsmetaComp` *check assume/guarantee (A/G) contracts during the compositional simulation of I/O ASMs?*

Table 2. MRM model size metrics (#m: monitored, #c: controlled, #d: derived, #o: out, #s: static, #r: rule constructors, #rd: rule declarations, #ct: contracts)

MRM model	#m	#c	#d	#o	#s	#r	#rd	#ct
pillbox.asm	7	3	3	12	4	135	13	5
configurationMgr.asm	4	9	1	0	2	13	4	6
compartment.asm	4	2	0	3	0	9	1	6
knowledge.asm	0	1	8	0	0	0	0	0
rescheduler.asm	9	9	1	5	2	48	5	3

Table 3. Validation Scenarios

Scenario	Error	Expected violated assertion	Description
CS1	none	–	A pill is missed, and not rescheduled since it interferes with the next pill.
CS2	none	–	The first pill is taken on time, and the second is rescheduled and taken at the new time.
WP	Wrong pill	`inv_A_medicineList`	The patient loads an incorrect pill in the second compartment.
PO	Pill overlap	`inv_A_timeConsumption`	The rescheduler sets a new time consumption that violates the minimum interval before the next pill.

Models and validation scenarios. To provide an indication of the complexity of the MRM system model, Table 2 lists the sub-models along with their size metrics, including the number of functions (by type), rule constructors, rule declarations, and contracts. It is a medium-size modular system model with a contract mainly consisting of functional assertions (adherence to functional requirements), interaction assertions (for correct sequencing of outputs, e.g. valid led light color transitions), safety assertions (to avoid unsafe states, e.g., no invalid medicine intake), and temporal assertions (e.g., scheduling of pill intakes). We introduce here four scenarios (see Table 3): two that result in contract violations (*WP* and *PO*) and two that do not (*CS1* and *CS2*). In *CS1*, a pill (e.g., pill "fosamax" in the first compartment) is missed and it is expected that it cannot be rescheduled because the minimum required interval before the next pill (e.g., "moment" in the second compartment) is not satisfied. The second non-violation scenario *CS2* models the situation in which the "fosamax" dose is taken on time, while the "moment" dose is rescheduled and subsequently taken at the new time. Other scenarios are in the online replication package.

In scenario *WP*, we set the second pillbox's compartment to "pantoprazole" instead of the prescribed "moment". This setting should result in a violation of the safety assertion `inv_A_medicineList` for the *ConfigurationMgr* (see Listing 1, line 35), which checks consistency between the prescribed pills and those contained in the pillbox. In scenario *PO*, we set the assumption time for the "fosamax" pill in the first compartment in a way that causes a dose overlap with the next pill ("moment" in the second compartment) in case of rescheduling. We simulate the patient not taking the "fosamax" pill by keeping the `compartmentOpen(1)` to false. When the *Rescheduler* intervenes to reschedule the missed "fosamax" pill, the temporal assertion `inv_A_timeConsumption` (see line 67 of the ASMETA model *ConfigurationMgr* in the replication package [11]) is violated, which ensures that any two potentially interacting doses, either across compartments or within the same compartment, are scheduled with a time gap of at least the minimum safe interval specified by `minToInterferer`.

```
1  Composition setup for alias: PB —> ASM pillbox: {}
2  {RES=ASM reschedulerNonOptimal: {}, PB=ASM pillbox: {}, setup_comp= ( ASM configurationMgr: {} ||
       ASM reschedulerNonOptimal: {} ) , CM=ASM configurationMgr: {}, C1=ASM compartment: {}, C2=
       ASM compartment: {}} ...
3  RUN STEP 1
4  Running composition: setup_comp with input: {name(compartment2)="moment", name(compartment1)="
       fosamax", ...} ...
5  RUN STEP 4
6  Running composition: pillbox_comp with input: {compartmentOpen(2)=false, compartmentOpen(1)=false, ...}
7  Monitored locations pillbox: {compartmentOpen(2)=false, compartmentOpen(1)=false, ...}
8  Out locations: pillbox: {name(compartment2)=pantoprazole , name(compartment1)=fosamax, ...}
9  Monitored locations configurationMgr: {name(compartment1)="fosamax", name(compartment2)=
       "pantoprazole", ...}
10 Error in composition pillbox_comp: Assumption violation over inv_A_medicineList
11 Composition script interrupted due to contract violation.
```

Listing 3. Composition contract violation: wrong pill

```
1  Composition setup for alias: PB —> ASM pillbox: {}
2  {RES=ASM reschedulerNonOptimal: {}, PB=ASM pillbox: {}, setup_comp= ( ASM configurationMgr: {} ||
       ASM reschedulerNonOptimal: {} ) , CM=ASM configurationMgr: {}, C1=ASM compartment: {}, C2=
       ASM compartment: {}} ...
3  RUN STEP 10
4  Running composition: pillbox_comp with input: {compartmentOpen(2)=false, compartmentOpen(1)=false,
       systemTime=753n}
5  Monitored locations pillbox: {systemTime=753n, openSwitch(1)=false, openSwitch(2)=false,
       setNewTime(compartment1)=true, displayMessage(1)="fosamax missed", displayMessage(2)="Take
       moment in 10 minutes"}
6  Out locations pillbox: {led(1)=OFF, led(2)=OFF, outMess(1)=fosamax rescheduled, outMess(2)=moment
       missed, redLed(compartment1)=OFF, redLed(compartment2)=OFF,
       time_consumption(compartment1)=[420], time_consumption(compartment2)=[730,1140], name(
       compartment1)=fosamax, name(compartment2)=moment, pillboxSystemTime=753}
7  Monitored locations configurationMgr: {minToInterferer(fosamax,moment)=360,
       time_consumption(compartment1)=[420], time_consumption(compartment2)=[730,1140],... }
8  Error in composition pillbox_comp: Assumption violation over inv_A_timeConsumption
9  Composition script interrupted due to contract violation.
```

Listing 4. Composition contract violation: time overlap with the next medicine

Validation results. We report here the execution of scenarios WP and PO, which lead to contract violations; the others are available in the online repository.

Listing 3 shows the execution of scenario *WP* in which, the patient placed "pantoprazole" in the second compartment (see the prompted pill names at line 8), which does not match the prescribed medicines (see the prompted pill names at line 4).

Listing 4 shows the execution of scenario *PO*: the rescheduler updates the administration time of "fosamax" pill to 420 (see line 6) because the patient missed the dose. However, this adjustment fails to respect the minimum required interval before the subsequent dose —"moment", scheduled at 730 — since the time difference is less than 360 (see line 7).

8 Conclusion

This paper presented AsmetaComp, a tool for Contract-based Design and runtime contract checking of I/O ASM models. Future work includes improving the tool output features and applying it to further industrial case studies.

References

1. ASMETA (ASM mETAmodeling) toolset, https://asmeta.github.io/
2. ARBAB, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science **14**(3), 329–366 (2004). https://doi.org/10.1017/S0960129504004153
3. Bartocci, E., Ferrère, T., Henzinger, T.A., Nickovic, D., da Costa, A.O.: Information-flow interfaces. In: Johnsen, E.B., Wimmer, M. (eds.) Fundamental Approaches to Software Engineering. pp. 3–22. Springer International Publishing, Cham (2022)
4. Basile, D., ter Beek, M.H.: A runtime environment for contract automata. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) Formal Methods. pp. 550–567. Springer International Publishing, Cham (2023)
5. Basu, A., Combaz, J., Nguyen, T.H., Jaber, M., Sifakis, J., Bensalem, S., Bozga, M.: Rigorous Component-Based System Design Using the BIP Framework . IEEE Software **28**(03), 41–48 (May 2011). https://doi.org/10.1109/MS.2011.27, https://doi.ieeecomputersociety.org/10.1109/MS.2011.27
6. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. pp. 200–225. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
7. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T.A., Larsen, K.G.: Contracts for system design. Foundations and Trends® in Electronic Design Automation **12**(2-3), 124–400 (2018). https://doi.org/10.1561/1000000053, http://dx.doi.org/10.1561/1000000053
8. Bicarregui, J., Fitzgerald, J., Larsen, P.G., Woodcock, J.: Industrial practice in formal methods: A review. Lecture Notes in Computer Science **4156**, 182–195 (2006). https://doi.org/10.1007/11813040$_1$4
9. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 162–176. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
10. Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: ASMETA tool set for rigorous system design. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14934, pp. 492–517. Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_28, https://doi.org/10.1007/978-3-031-71177-0_28
11. Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Accompanying tool artifact for the paper "AsmetaComp: a tool for Runtime Contract Checking with I/O Abstract State Machines" (2026), https://doi.org/10.5281/zenodo.18678488
12. Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: AsmetaComp: a Tool for Runtime Contract Checking with I/O Abstract State Machines (2026), https://github.com/asmeta/asmeta_based_applications/tree/main/MRM_CBD
13. Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Compositional simulation of Abstract State Machines for safety critical systems. In: Tarifa, S.L.T., Proença, J. (eds.) Formal Aspects of Component Software - 18th International

Conference, FACS 2022, Proceedings. LNCS, vol. 13712, pp. 3–19. Springer (2022). https://doi.org/10.1007/978-3-031-20872-0_1
14. Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: A compositional simulation framework for Abstract State Machine models of Discrete Event Systems. Formal Aspects of Computing (2024)
15. Bonfanti, S., Riccobene, E., Scandurra, P.: A runtime safety enforcement approach by monitoring and adaptation. In: Biffl, S., Navarro, E., Löwe, W., Sirjani, M., Mirandola, R., Weyns, D. (eds.) Software Architecture. pp. 20–36. Springer International Publishing, Cham (2021)
16. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer, Berlin, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56641-1
17. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer Verlag (2003)
18. Calinescu, R., Kikuchi, S.: Formal methods @ runtime. In: Calinescu, R., Jackson, E. (eds.) Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems. pp. 122–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
19. Chen, T., Chilton, C., Jonsson, B., Kwiatkowska, M.: A compositional specification theory for component behaviours. In: Seidl, H. (ed.) Programming Languages and Systems. pp. 148–168. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
20. Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Science of Computer Programming **97**, 333–348 (2015). https://doi.org/https://doi.org/10.1016/j.scico.2014.06.011, https://www.sciencedirect.com/science/article/pii/S0167642314002901, object-Oriented Programming and Systems (OOPS 2010) Modeling and Analysis of Compositional Software (papers from EUROMICRO SEAA'12)
21. Cimatti, Alessandro, E.G.H.P., Rozier, K.Y.: Software contracts meet system contracts (2026), https://www.dagstuhl.de/26031, dagstuhl Seminar 26031, January 11–16, 2026
22. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Goodloe, A.E., Person, S. (eds.) NASA Formal Methods. pp. 126–140. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
23. Ferrante, O., Passerone, R., Ferrari, A., Mangeruca, L., Sofronis, C.: Bcl: A compositional contract language for embedded systems. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA). pp. 1–6 (2014). https://doi.org/10.1109/ETFA.2014.7005353
24. Gheri, L., Lanese, I., Sayers, N., Tuosto, E., Yoshida, N.: Design-by-contract for flexible multiparty session protocols. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, Berlin, Germany, June 6-10, 2022. LIPIcs, vol. 222, pp. 8:1–8:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPICS.ECOOP.2022.8, https://doi.org/10.4230/LIPIcs.ECOOP.2022.8
25. Gleirscher, M., Marmsoler, D.: Formal methods: Oversold? underused? a survey. arXiv preprint arXiv:1812.08815 (2018), https://arxiv.org/abs/1812.08815
26. Graf, S., Passerone, R., Quinton, S.: Contract-based reasoning for component systems with rich interactions. In: Embedded Systems Development: From Functional Models to Implementations, pp. 139–154. Springer (2014). https://doi.org/10.1007/978-1-4614-3879-3_8

27. Gurov, D., Hähnle, R., Huisman, M., Reger, G., Lidström, C.: Principles of Contract Languages (Dagstuhl Seminar 22451). Dagstuhl Reports **12**(11), 1–27 (2023). https://doi.org/10.4230/DagRep.12.11.1, https://drops.dagstuhl.de/entities/document/10.4230/DagRep.12.11.1
28. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259, https://doi.org/10.1145/363235.363259
29. Incer, I., Benveniste, A., Sangiovanni-Vincentelli, A., Seshia, S.A.: From Interface Automata to Hypercontracts, pp. 477–493. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-22337-2$_2$3, https://doi.org/10.1007/978-3-031-22337-2_23
30. Lidström, C., Gurov, D.: Contract based embedded software design. In: David, C., Sun, M. (eds.) Theoretical Aspects of Software Engineering. pp. 77–94. Springer Nature Switzerland, Cham (2023)
31. Lu, X., Tian, C., Gu, B., Yu, B., Chen, C., Duan, Z.: A contract-based framework for formal verification of embedded software. In: Bourke, T., Chen, L., Goharshady, A. (eds.) Dependable Software Engineering. Theories, Tools, and Applications. pp. 180–196. Springer Nature Singapore, Singapore (2025)
32. Meyer, B.: Applying 'design by contract'. Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
33. Nandi, C., Monot, A., Oriol, M.: Stochastic contracts for runtime checking of component-based real-time systems. In: Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering. p. 111–116. CBSE '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737166.2737173, https://doi.org/10.1145/2737166.2737173
34. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008). pp. 377–381. IEEE Computer Society (2008). https://doi.org/10.1109/SEFM.2008.28
35. Stierand, I., et al.: Using contract-based component specifications for virtual integration testing and architecture design. In: Proceedings of the DATE Conference (2010)
36. Taghavi, B., Weber, S., Marin, A., Rumpe, B., Stüber, S., Henß, J., Weber, T., Heinrich, R.: Modeling the composition of analysis components and automatic constraint checking for semantic soundness. Journal of Systems and Software **231**, 112637 (2026). https://doi.org/https://doi.org/10.1016/j.jss.2025.112637, https://www.sciencedirect.com/science/article/pii/S0164121225003061
37. Talcott, C.L., Ananieva, S., Bae, K., Combemale, B., Heinrich, R., Hills, M., Khakpour, N., Reussner, R.H., Rumpe, B., Scandurra, P., Vangheluwe, H.: Composition of languages, models, and analyses. In: Heinrich, R., Durán, F., Talcott, C.L., Zschaler, S. (eds.) Composing Model-Based Analysis Tools, pp. 45–70. Springer (2021). https://doi.org/10.1007/978-3-030-81915-6_4, https://doi.org/10.1007/978-3-030-81915-6_4