Repairing Conformance Violations with a Reference Model in the Context of Automata Decomposition

Bahareh Taghavi
Karlsruhe Institute of Technology
Karlsruhe, Germany
bahareh.taghavi@kit.edu

Max Stachon
RWTH Aachen University
Aachen, Germany
stachon@se-rwth.de

Adrian Marin
RWTH Aachen University
Aachen, Germany
marin@se-rwth.de

Lan Bao Quang Le
Ulm University
Ulm, Germany
lan.le@uni-ulm.de

Bernhard Rumpe
RWTH Aachen University
Aachen, Germany
rumpe@se-rwth.de

Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
ralf.reussner@kit.edu

Robert Heinrich✉
Ulm University
Ulm, Germany
robert.heinrich@uni-ulm.de

Abstract

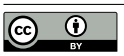
Existing conformance-checking approaches typically compare a reference model with a monolithic concrete model and therefore do not support decompositions into interacting subcomponents. We present a conformance-checking and repair approach for time-synchronous component-and-connector subarchitectures with statechart-based component behavior. Given a reference component, a concrete cascade composition with delayed feedback, and an incarnation mapping, the approach checks whether the composition's observable behavior conforms to the reference. On failure, it returns a counterexample, localizes the violation to the responsible subcomponent and repair site, and synthesizes a repair via MaxSMT, minimizing syntactic change, bounded behavioral loss, and distance from the blamed component. Re-checking validates each repair. Experiments with seeded faults and generated composition chains show accurate localization and conformance-restoring repairs, with repair synthesis dominating runtime.

CCS Concepts

• **Software and its engineering** → **Formal software verification**; *Architecture description languages*; Semantics; Model-driven software engineering; • **Theory of computation** → Formal languages and automata theory; *Automated reasoning*.

Keywords

Model Repair, Conformance, Refinement, Semantics, Reference Model, Component Architecture, Automata, Decomposition



This work is licensed under a Creative Commons Attribution 4.0 International License.
MODELS Companion 2026, Málaga, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2903-4/26/10
https://doi.org/10.1145/3837062.3839047

ACM Reference Format:
Bahareh Taghavi, Max Stachon, Adrian Marin, Lan Bao Quang Le, Bernhard Rumpe, Ralf Reussner, and Robert Heinrich. 2026. Repairing Conformance Violations with a Reference Model in the Context of Automata Decomposition. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3837062.3839047

1 Introduction

Model-driven engineering aims to improve software and system development by treating models as primary source artifacts [16] and relying on automation for tasks such as model-to-code translation and model-based analysis. In model-based analysis, a monolithic reference analysis may be decomposed into interacting analysis components to improve modularity, reuse, or scalability [21], goals also pursued at the metamodel level [22]. The resulting decomposition must preserve the externally observable analysis behavior of the reference specification, i.e., it must conform to the reference model.

The conformance framework introduced in [32] establishes a rigorous foundation for checking whether a concrete model adheres to the specification of a reference model, using a formally defined mapping, known as an *incarnation mapping* [32], to specify which elements of the concrete model correspond to which elements in the reference model. This approach enables precise reasoning about correctness by relying on a formal notion of model refinement based on a denotational semantics definition for a given modeling language [20].

Existing conformance checks based on the aforementioned conformance framework [32, 41] are currently limited to comparing a monolithic concrete model with a reference model of the same language. Modern systems, however, are rarely monolithic. They are increasingly constructed by decomposing a system into subcomponents, each responsible for a specific task. This design principle introduces new challenges for conformance checking, and the existing approaches provide no means to (i) check whether a concrete

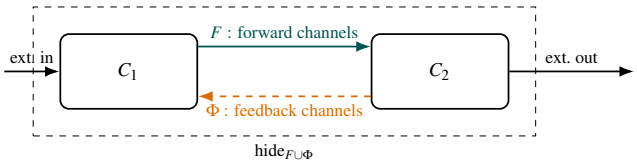


Figure 1: General two-component composition with a forward channel set F and a feedback channel set Φ.

subsystem correctly decomposes a reference component specification, (ii) trace a conformance violation back to the responsible syntactic elements in the specific subcomponent, or (iii) repair it. We close this gap by building on compositional refinement ideas from formal specification frameworks such as FOCUS [7] that address the design and refinement of compositional architectures to extend the existing conformance framework.

In particular, in this work, we extend the conformance check for MontiArcAutomata [53] introduced in [32] to support the compositional analysis setting of component-and-connector architectures in MontiArc [19]. We consider a concrete subarchitecture consisting of two or more subcomponents, obtained by a task-driven decomposition of a reference component specification using a MontiArcAutomaton. The subcomponents interact through cascade composition, in which forward channels carry the output of one component to the input of the next, and feedback channels return values in the opposite direction, with the internal channels hidden from the outside. Throughout, we assume a time-synchronous setting with at most one message per channel per step, unit-delay feedback channels that rule out zero-time cycles, and finite message alphabets and control state spaces, so that the composed configuration space is finite. Figure 1 shows this topology for the general case of two components. This pattern is common in data-flow and pipeline architectures, but it complicates conformance checking, since correctness must be established across the composition rather than for a single component in isolation.

Our approach checks a whole subarchitecture of the system against its corresponding reference model under the given incarnation mapping (Figure 2). When non-conformance is detected, it generates counterexamples that expose the deviation between expected and observed behavior and localize it to the responsible subcomponent. We assume that faults originate in the concrete model, consistent with viewing the concrete model as a refinement and decomposition of a trusted reference. The objective is therefore not only to detect violations but also to localize and repair them within the concrete subcomponents, while changing the concrete model as little as possible, both syntactically and behaviorally.

This paper makes the following contributions:

- **C1** A compositional conformance checking approach for systems assembled from MontiArcAutomata subcomponents through cascade composition. It checks such a system against the reference model it was decomposed from, under a given incarnation mapping.
- **C2** A counterexample-guided repair for composed architectures. It localizes a composition-level conformance violation to the responsible subcomponent. It then synthesizes a minimal-deviation fix as a MaxSMT problem, minimizing a cost function over three dimensions: syntactic distance, bounded behavioral loss, and distance from the localized subcomponent.

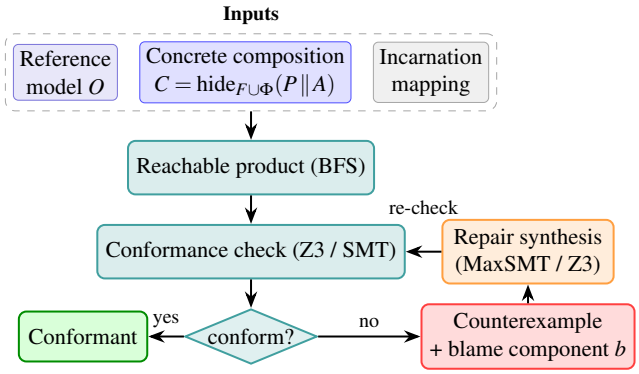


Figure 2: Overview of the conformance checking and repair pipeline.

2 Preliminaries

We briefly introduce the modeling and analysis concepts used in this paper: component-and-connector architectures, conformance under incarnation mappings, and SMT-based optimization.

2.1 Component-and-Connector Architectures

Component-and-connector (C&C) architectures model systems as networks of components that communicate through typed, directed ports connected by channels [4, 34, 55, 58]. Components may be hierarchically decomposed, so that an abstract component can be realized by a subarchitecture of interacting components. This makes C&C models well suited for systems whose functionality is distributed over multiple collaborating parts, but it also raises the question whether the composed subarchitecture still preserves the behavior required by the original component.

We use MontiArc [19], a custom architecture description language, to model C&C systems. MontiArc defines the architectural structure in terms of components, ports, and connectors, while the input-output behavior of atomic components is specified by MontiArcAutomata [53], a statechart variant. Transitions react to input messages, may evaluate guards and update variables, and produce output messages. Thus, a MontiArc component can be understood both structurally, through its interface and internal wiring, and behaviorally, through the automata assigned to its atomic parts.

The semantic foundation of MontiArcAutomata is inspired by FOCUS [7], which models component behavior using streams over directed channels and provides a formal notion of refinement. In such a setting, a concrete model refines an abstract one if it admits no externally observable behavior that the abstract model forbids. This refinement view is central to our approach, because a decomposition should not merely be structurally well-formed; it should also behave like a correct realization of the reference component.

In this paper, we restrict communication to a finite, time-synchronous setting in which a component consumes at most one message per input channel and produces at most one message per output channel per step; feedback channels are interpreted with a unit delay.

2.2 Conformance and Refinement

Conformance checking verifies whether a concrete model preserves the relevant properties of a reference model. The conformance framework of [32] makes the correspondence between both models explicit through an *incarnation mapping*, which relates selected concrete model elements, called incarnations, to their reference counterparts. Such mappings are necessary because a concrete model may use different names, introduce additional structure, or realize a reference element by several concrete elements.

The precise notion of conformance depends on the modeling language and the intended preservation criterion. At the semantic level, conformance is commonly based on refinement: a concrete model is considered correct with respect to an abstract reference if all relevant behaviors of the concrete model are admitted by the reference. Under a denotational view of semantics [20], refinement is a subset relation between the semantics of the refined and the abstract model.

For behavioral models such as automata or statecharts, the semantic instances are observable behaviors, typically input-output traces, so a failed check is witnessed by a concrete behavior that the reference does not admit. This view is closely related to semantic differencing [38], where model differences are represented by semantic witnesses rather than by syntactic changes. In our setting, such witnesses take the form of counterexamples that guide localization and repair.

2.3 SMT and MaxSMT

Satisfiability Modulo Theories (SMT) [2] decides logical constraints over background theories such as arithmetic, bit-vectors, and uninterpreted functions. SMT solvers thus combine propositional reasoning with theory-specific decision procedures. We use SMT to encode the conformance obligations generated from the reachable composed behavior and the incarnation mapping.

MaxSMT extends SMT with optimization over hard and soft constraints [48]. Hard constraints must be satisfied, while soft constraints may be violated at a cost. In the weighted variant, the objective is to minimize the cost of violated soft constraints. We use weighted MaxSMT for repair synthesis: hard constraints enforce well-formedness and elimination of the counterexample, while soft constraints encode the repair cost, including syntactic distance, bounded behavioral loss, and distance from the blamed component.

3 Running Example

As a running example, we use a model of *resource simulation* taken from *Slingshot* [30], a discrete-event simulator for the Palladio Component Model [3] that follows an event-driven architecture. In Slingshot, serving a resource request intertwines two concerns, the contention and queueing of a *passive* resource (a bounded capacity that must be acquired and released) and the actual processing of an *active* resource (a server that executes an admitted request and thereby determines its service time and, hence, the resulting utilization). The model used here is a simplified abstraction of this behavior that keeps the request lifecycle (arrival, admission or rejection, execution, and completion) while abstracting away timing values and the concrete scheduling policy, so that the example stays small enough to inspect.

Figure 3 presents the example in two forms. Before decomposition (top), the reference model ResourceSimulation captures the entire request lifecycle in a single monolithic component that handles both queueing and execution. To improve modularity and enable compositional reasoning, we decompose it into two interacting subsystems (bottom) that mirror this passive/active distinction. A passive admission stage P that owns all queueing and contention and reports the waiting time as `waitTime`, and an active execution stage A that runs only admitted requests and reports the service time as `serviceTime`, from which resource utilization is derived.

The two stages communicate over a forward channel `admit` and a unit-delay feedback channel `release`, on which A sends completion messages such as `ResourceReleased` back to P. The feedback is essential because P can only release the capacity held by a finished request, and thereby admit a queued one, after A signals completion. The two stages are therefore mutually dependent and do not form a one-way pipeline. The decomposition is intended to be behaviorally equivalent to the monolith. The figure deliberately includes a non-conformant grant edge (highlighted in red) whose composed behavior diverges from the reference. This motivates the conformance check of the following sections.

Note that in the figure, port names and message values are shown together. The forward port `admit` carries values such as `ActiveStartRequested` and `ActiveAbortRequested` from P to A. The delayed feedback port `release` carries completion information from A back to P. In the composition, both ports are internal and therefore hidden from the external interface.

4 Automata Composition Model

Motivated by model-based analyses structured as interacting analysis components, we model the concrete realization of a reference analysis specification as a composition of automata that exchange information over hidden internal channels. This section presents contribution **C1**; it fixes the cascade composition operator and the conformance relation, makes precise what a state of the composed system is, states the conditions under which an output of one component may legitimately become an input of another, and realizes the conformance check as a breadth-first construction of the reachable product configurations.

Throughout, we write P for the upstream and A for the downstream component, corresponding to the concrete components `PassiveResourceSimulation` and `ActiveResourceSimulation` of our running example. The reference model is `ResourceSimulation`, denoted O.

4.1 Composition Operator and Conformance Relation

Composition operator. We take as our composition operator $C = \mathrm{hide}_{F \cup \Phi}(P \parallel A)$, where the shared alphabet between P and A is partitioned into a forward channel set F (outputs of P consumed by A) and a feedback channel set Φ (outputs of A consumed by P),

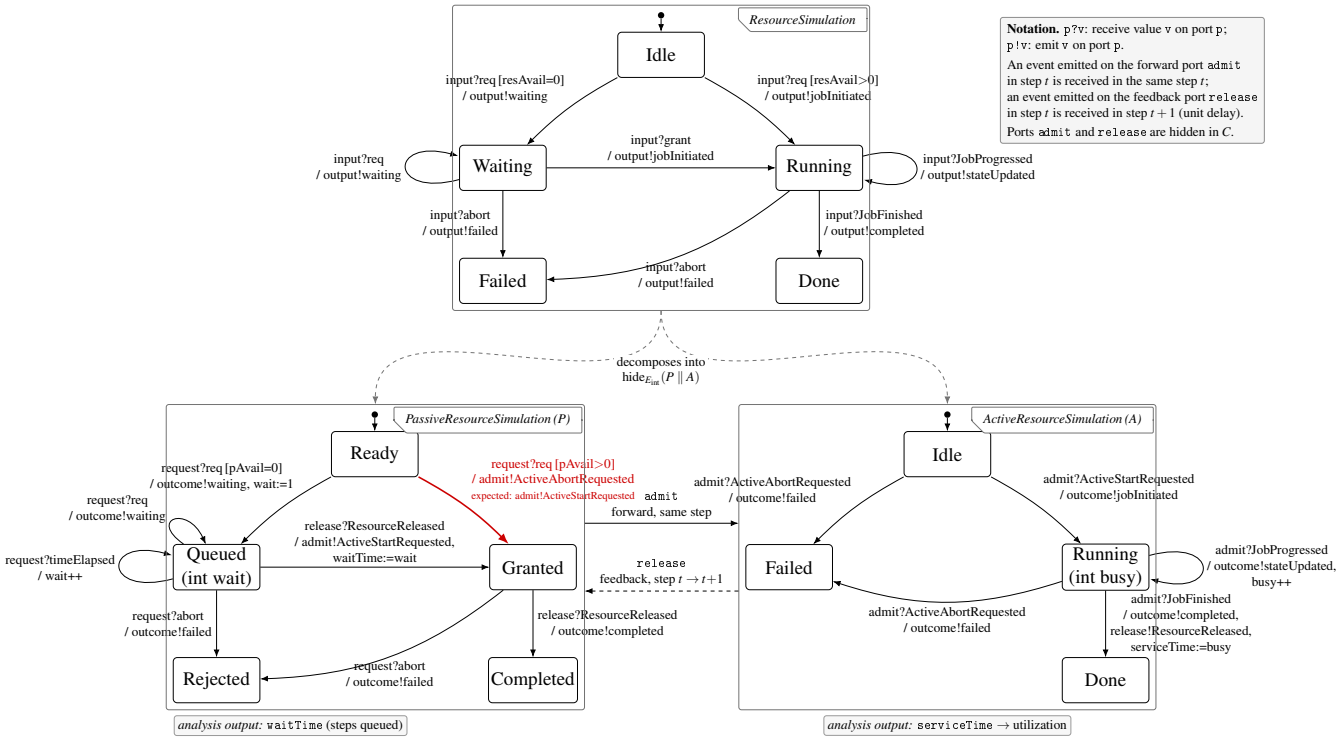


Figure 3: Running example: the reference automaton ResourceSimulation (top) and its decomposition into P and A (bottom). The red edge is the seeded fault.

with both sets hidden from C's external interface. The pure cascade $C = P \circ A$ is the special case $\Phi = \emptyset$.

Two components cannot in general interact correctly through a single one-directional channel whenever the downstream component holds state on which the upstream component's future decisions depend. The downstream component must be able to signal back when an in-flight operation completes, and the channel carrying this information necessarily flows against the direction of control. In our running example Φ carries the completion signal from the active resource simulation back to the passive resource simulation, so that the freed capacity can be granted to a queued request.

Requiring at least one channel in each direction for an interaction to be well-formed is the standard convention in C&C architecture description languages such as MontiArc [19]; a unit delay on each feedback rules out zero-time cycles. In MontiArc this delay is expressed directly by declaring the feedback port as «delayed».

Conformance relation. According to [32], conformance requires semantic refinement of the original system. However, we need not consider elements in the concrete model that do not correspond to elements in the reference model. Hiding F and Φ removes the forward and feedback events from C's external interface. So, from the outside, only the external input and output of C remain visible. The labels of the conformance obligation are therefore only external, so the observable alphabet is that of a black-box refinement, although the buffered feedback value remains part of the configuration.

Let $[\![M]\!]_\mu$ denote the set of observable input-output traces of a model M after applying the incarnation mapping μ. We say that the concrete composition C refines the reference model O iff

$[\![C]\!]_\mu \subseteq [\![O]\!]$.

Thus, every externally observable behavior of the concrete composition must be admitted by the reference model. If both models are deterministic and input-enabled, this reduces to equality of the produced output sequence for every input sequence. The checker does not discharge this inclusion directly. Since μ assigns a reference state $\mu_{\mathrm{st}}(c)$ to every reachable configuration $c = (s_P, s_A, \phi)$ (Listing 2), we adopt as our conformance relation the step-wise obligation that for every enabled step $c \xrightarrow{x/y} c'$, O admits $\mu_{\mathrm{st}}(c) \xrightarrow{\mu_{\mathrm{in}}(x)/\mu_{\mathrm{out}}(y)} \mu_{\mathrm{st}}(c')$. This simulation implies $[\![C]\!]_\mu \subseteq [\![O]\!]$ but is strictly stronger, and it is what attributes a violation to a single step and hence to a single transition.

Composition and mapping. The composition C is specified by an enclosing MontiArc component specification that contains instances of the subcomponents P and A, and specifies how they are connected in the subarchitecture. The external inputs and outputs of C are specified by connecting the input and output ports of the enclosing component to the corresponding input and output ports of the subcomponents. The enclosing component also connects output channels of subcomponent instances to input channels of subsequent subcomponent instances to form the channels for the intermediate messages. It is parsed and checked alongside the other

```
component ActivePassiveComposition {
  port
    sync in  Input  request,     // external input of C
    sync out Output outcome;     // external output of C
    sync out int waitTime;
    sync out int serviceTime;

  PassiveResourceSimulation passive;   // stage 1
  ActiveResourceSimulation  active;    // stage 2

  request -> passive.request;

  passive.admit   -> active.admit;     // forward
  active.release  -> passive.release;  // feedback

  active.outcome -> outcome;
}
```

Listing 1: The composition expressed as a composed MontiArc component.

```
mapping AutomataMapping {
  stateMap {
    PassiveResourceSimulation.state == Ready
      && ActiveResourceSimulation.state == Idle
      ==> ResourceSimulation.state == Idle;
    PassiveResourceSimulation.state == Granted
      && ActiveResourceSimulation.state == Running
      ==> ResourceSimulation.state == Running;
    // ...
  }
  inputMap {
    request == Input.ACQUIRE ==> input == Input.ACQUIRE;
    // ...
  }
  outputMap {
    outcome == [Output.STARTED]
        ==> output == [Output.STARTED];
    outcome == []
        ==> output == [];
    // ...
  }
}
```

Listing 2: Excerpt of the mapping between the reference model and the decomposed concrete components.

component specifications. A simplified version from our running example is shown in Listing 1.

Connecting the components yields the composed object C but says nothing about what it is meant to compute; that correspondence is recorded in a separate artifact, the *mapping* (Listing 2). The composition relates concrete ports to concrete ports and never refers to the reference, whereas the mapping relates C back to the original monolithic model O. It reuses the same form as the single-automaton mapping of the base setting, with one generalization. In the base setting each state rule abstracts a single concrete state to a reference state; under composition there is no single concrete state, so a reference state is realized *jointly* by a combination of component states. The input and output maps keep their original single-interface form, because C still exposes one external input and one external output. In our example case the relevant concrete input and output messages have the same form as the corresponding input and output messages in the reference model. This does not need to be the case in general, and renaming, as well as multiple variants for inputs and output incarnations are permitted.

Separating the specification of C and the mapping from the specifications of the sub-components also reflects the responsibilities of the two engineering roles. In the context of model-based analysis, the *analysis component developer* specifies an individual analysis component together with its transition system, input and output alphabets, execution semantics, and observable events. This role has a white-box view and may modify the component's internal behavior, state structure, and semantic annotations [31]; outside model-based analysis, it generalizes to that of a *component developer*. The *composition expert*, in turn, defines the sequential interaction between components, that is, the interaction protocol, the event-propagation rules, and the causality constraints, and authors the mapping between the concrete and the reference automata. To define these correctly, the composition expert needs access to the semantic structure and observable behavior of the components, but does not necessarily modify their internal implementation.

Reachable product states. Following the standard treatment of systems of communicating automata, in which a global configuration is the tuple of local control states together with the contents of the channels [5], we define a *configuration* of C as

$$c = (s_P, s_A, \phi) \in S_P \times S_A \times D_\Phi,$$

where s_P, s_A are the component control states and ϕ is the value buffered on the feedback channel (one coordinate per channel in Φ). Unlike the unbounded FIFO channels of the classical model, each feedback channel here is a *one-place, unit-delay* buffer over a finite value domain D_Φ, so the configuration space remains finite.

The control component of the configuration space is the Cartesian product $S_P \times S_A$, multiplicative in the component sizes; adding the buffer coordinate multiplies the worst-case bound by a further factor of $|D_\Phi|$. Only the configurations reachable from the initial one are relevant for conformance checking, following the standard treatment of finite-state systems in model checking [9].

The checker first constructs the reachable fragment of the composed system by breadth-first search (BFS). For each reachable configuration and each step enabled in it, the checker then verifies the conformance obligations induced by the incarnation mapping. These obligations are encoded as SMT constraints. If an obligation is unsatisfiable, the corresponding execution step in the reachable fragment is returned as a counterexample. The reachable fragment is far smaller than the bound. Each shared step records the same change on both sides at once, so the two components can never disagree about the state they share, and inconsistent configurations are never generated. In particular, the buffered value ϕ is fixed by the producer's last transition, so every reachable control pair occurs with exactly one buffer value. Consequently, the buffer coordinate enlarges the bound but not the reachable set.

In our running example, P has the five control states and A has four, so the control product has $|S_P| \cdot |S_A| = 5 \cdot 4 = 20$ pairs. The feedback channel ranges over the result alphabet, and the worst-case bound is $20 \cdot |D_\Phi|$ configurations. The BFS reaches only a small fraction of these, all of which the mapping relates to reference states; the remainder correspond to inconsistent situations the synchronization discipline never produces. Crucially, each of

these reachable pairs occurs with a single buffer value, so tracking the buffer leaves the reachable count unchanged even though it multiplies the bound.

4.2 Integration Conditions

Integration conditions [49] define the exact circumstances under which an output of one component is allowed to become an input of another component in a cascade composition.

Let $C_1 = (Q_1, \Sigma_1, \delta_1, q_{01})$ be the producer component, $C_2 = (Q_2, \Sigma_2, \delta_2, q_{02})$ be the consumer component, with states Q_i, alphabet Σ_i, transition relation δ_i, and initial state $q_{0i} \in Q_i$. The *bridge relation* $B \subseteq \Sigma_1^{\text{out}} \times \Sigma_2^{\text{in}}$ pairs an output of C_1 with the input of C_2 it feeds. For a producer transition $q_1 \xrightarrow{o} q_1'$ and a candidate consumer transition $q_2 \xrightarrow{i} q_2'$, the integration condition is the predicate

$$IC_{12}\big(q_1 \xrightarrow{o} q_1',\ q_2 \xrightarrow{i} q_2',\ t,\ \Gamma\big) \in \{\mathit{true}, \mathit{false}\},$$

where t is the current logical step and Γ is the interaction context (buffered feedback, guards, and annotations). It holds exactly when all three conditions below are satisfied.

- **Interface compatibility.** The produced output must be connected to a valid input of the second component, this is checked through the bridge relation: $(o, i) \in B$. In the running example (*Passive.admit*, *Active.admit*) $\in B$, so the admit signal of the passive stage may be propagated to the active stage. This condition ensures that propagated events are structurally connectable across automata boundaries.
- **Behavioral admissibility.** The consumer's current state q_2 must have a transition $q_2 \xrightarrow{i} q_2'$ that *matches* the producer transition $q_1 \xrightarrow{o} q_1'$, i.e. with $(o, i) \in B$ and a guard that holds in Γ; determinism makes this transition unique. This condition guarantees that interaction does not introduce undefined or semantically inconsistent behavior in the composed system.
  In the running example the active stage accepts an event on *admit* only in a state with a matching transition (e.g. *Idle*), and not in one without (e.g. `ActiveStartRequested` in *Running*).
- **Causality preservation.** A connection that closes a cycle must not take effect in the step that produced it; the consumer may react only at a later step. This is what rules out zero-time feedback loops. In the running example the feedback signal *release* produced by the active stage at step t is observed by the passive stage at step $t + 1$, realizing the unit delay. A forward connection such as *admit*, which closes no cycle, may be consumed in the same step.

The integration conditions make each composed step well defined and unique, so C is deterministic whenever its components are; with finite state spaces and a finite D_Φ, the configuration space is finite and the reachable construction terminates. Composition is monotone with respect to trace refinement, so refining a subcomponent cannot invalidate an established trace-conformance verdict.

5 Repair computation

This section delivers **C2**. We formulate automated repair as a MaxSMT optimization problem, where the goal is to restore conformance while minimizing the deviation from the original model. The formulation separates correctness requirements from optimization criteria and allows repairs to be both sound and minimal. Given a counterexample witnessing non-conformance, we derive a finite set of candidate edits $E = \{e_1, \ldots, e_m\}$. Each edit modifies one candidate repair site, for example by changing an emitted output value, strengthening or weakening a guard, or modifying an assignment. The MaxSMT encoding contains hard constraints ensuring that the edited model is syntactically well-formed, type-correct, and eliminates the current counterexample. Since the repair is synthesized from a finite counterexample, soundness with respect to the full conformance relation is obtained by re-running the conformance checker on the repaired model.

For each candidate edit e, we assign a cost

$$C(e) = w_1\, d_{\text{syn}}(e) + w_2\, \ell_{\text{beh}}(e) + w_3\, d_{\text{blame}}(e),$$

where all weights default to 1. The MaxSMT objective is to find a hard-constraint-satisfying edit of minimal cost.

Here, *Syntactic distance* d_{syn} captures the principle of minimal structural change. It increases with the size of the edited syntax fragment and the number of distinct semantic entities involved in the change.

$$d_{\text{syn}} = \max\big(1, \text{affectedAstNodes} + \text{touchedSemanticElements} - 1\big)$$

affectedAstNodes shows the number of AST nodes inside the edited fragment and *touchedSemanticElements* counts how many distinct variables, ports, or guards are involved.

Comparing the k-step observable traces produced by the component before and after the edit, denoted T_{old} and T_{new}, respectively, where k is the automaton diameter, we define

$$\ell_{\text{beh}}(e) = \min\left(\lambda,\ \left\lceil \left(1 - \frac{|T_{\text{old}} \cap T_{\text{new}}|}{|T_{\text{old}}|}\right)\lambda \right\rceil\right), \quad \lambda = 4.$$

Setting $\lambda = 4$ discretizes the loss into quartiles and keeps this dimension on the same small-integer scale as the others; λ is a tunable parameter. If $T_{\text{old}} = \emptyset$, we define $\ell_{\text{beh}}(e) = 0$, because the edit does not remove any previously observable behavior. The weights and λ are pragmatic defaults rather than tuned values; they are parameters of the MaxSMT instance, so other weightings and discretizations are supported.

Component blame distance d_{blame} is the dimension that accounts for the compositional structure of the system. It measures how directly a repair targets the component responsible for the violation. Because only the last component's output is observed, a violation could in principle be patched at several components. We therefore first localize the fault by walking the chain and taking as blame component b the earliest component whose own output already makes the correct reference output unreachable for all components downstream of it. A repair at component r then costs $d_{\text{blame}}(e) = |r - b|$, so fixing the blamed component scores 0, and the penalty grows with each component between the repair and the true source of the fault.

Table 1: Effectiveness validation cases: detection, localization, and repair outcome per case. †conformance restored under re-checking, not fault reverted.

Case	Fault Kind	Position	Det. / Loc. / Rep.†
1	output	downstream	✓/ ✓/ ✓
2	guard	upstream	✓/ ✓/ ✓
3	variable	upstream	✓/ ✓/ ✓
4	output	upstream	✓/ ✓/ ✓
5	guard	downstream	✓/ ✓/ ✓
6	variable	downstream	✓/ ✓/ ✓
7	—	(conformant)	no violation (correct)
8	output	middle	✓/ ✓/ ✓
9	guard	middle	✓/ ✓/ ✓
10	variable	middle	✓/ ✓/ ✓

6 Evaluation

We evaluate our compositional conformance repair approach with respect to two research questions. The first asks whether the approach produces correct and well-targeted repairs (effectiveness), and the second asks how it behaves as the composed system grows (scalability). We study both on our running example, which we instantiate into controlled fault scenarios for effectiveness and into progressively larger systems for scalability.

All experiments were executed on a machine with a 12th Gen Intel Core i7-1260P CPU, 32 GB RAM, running Windows 11 Pro OS. The implementation uses Java 17, Z3 version 4.13.0, and a timeout of 600 seconds per analysis case. Unless stated otherwise, each measurement reports the median of 3 repetitions.

Data Availability. We provide a dataset (https://doi.org/10.5281/zenodo.21216345) containing all code artifacts, models, and evaluation scripts required to reproduce the results in this paper.

6.1 Effectiveness

RQ1: Does the approach detect a conformance violation, localize it to the responsible component, and synthesize a repair that restores it?

We construct ten validation cases over the running example. Each injects exactly one fault, leaving the rest of the model correct, so that every case isolates a single repair scenario and controls for confounding faults. The cases cover all three repair kinds (output, guard, and variable-assignment) at all three positions in the chain (upstream, middle, and downstream) together with one conformant case as a negative control (Table 1). The middle-position cases use a three-stage chain in which a scheduler component is inserted between the admission and execution stages.

Crucially, every repair is validated independently. The tool materializes the synthesized edit back into the concrete model and re-runs the full conformance check end-to-end. Every injected fault was detected, and the conformant case was correctly reported as conforming, so the approach produced no false positives.

In all nine faulty cases, the approach pinpointed the exact component and transition containing the fault. This is non-trivial in a sequential chain, since the violation is only ever observed at the final output regardless of where the fault originates; correct localization across upstream, middle, and downstream positions is precisely what the chain walk is designed to achieve; d_{blame} then steers the repair toward the localized component. In each case the repair takes the form of an output correction on the localized transition, which is the minimal edit that re-establishes the observable conformance relation.

6.2 Scalability

RQ2: How does the cost of conformance checking and repair synthesis scale as the composed system grows?

We scale the running example along two independent axes: the *chain length n*, the number of composed components, and the *state count* per component. For each configuration we generate a complete instance with the same structure as the running example (an admission stage, forwarding stages, and a final execution stage with delayed feedback). Into each instance we inject one seeded fault, of a type matching one of the repair kinds, an OUTPUT, a GUARD, or a VARIABLE fault, placed in the final stage so its effect is directly observable. The final-stage placement is a control for the scalability study only, so that runtime is not confounded by fault position; faults in intermediate components are evaluated in RQ1 (Table 1). We vary the chain length $n \in \{2, 3, 4\}$ and eight state counts (3 to 22 states) and combine each with all three fault kinds, giving $3 \cdot 8 \cdot 3 = 72$ configurations. Repair synthesis is encoded as a MaxSMT problem and solved via Z3's [12] Java API. We record conformance-checking, repair-search, and total time separately.

Across all 72 configurations the approach detected the violation and synthesized a conformance-restoring repair, with no timeouts or failures. Conformance checking was negligible throughout (17–312 ms, 0.70% of the total on average and never above 2.3%), so the cost lies almost entirely in repair synthesis. Figure 4 shows that repair time grows super-linearly with the per-component state count, from a few seconds at 3 states to 48–388 s at 22 states, while chain length has a milder and less consistent effect (between 0.7× and 2.2× from two to four components beyond the smallest instances).

Repair type also has a consistent effect, with guard repairs the most expensive and output repairs the cheapest. A guard repair must search over many candidate guard conditions, whereas an output repair only substitutes one value from a small finite set.

As the per-component state count grows, the curves for different chain lengths spread apart, so the penalty for adding components is largest when they are large. This cost, however, comes almost entirely from repair synthesis; conformance checking stays negligible. Since realistic decompositions split a component into a few parts, not many tiny ones, our chain lengths cover typical cases.

7 Threats to Validity

We discuss threats to validity along the four standard dimensions [56], considering our theory, our tool, and our experiments.

Construct validity. The verdict is relative to a hand-authored mapping, so soundness is conditional on a correct `mapping`. We reduce this risk by deriving the mapping directly from the reference model's states and checking it against the reachable product configurations, so that an inconsistent `stateMap` is exposed during checking rather than being silently accepted. The three-dimensional repair cost approximates minimality, but it may not coincide with a

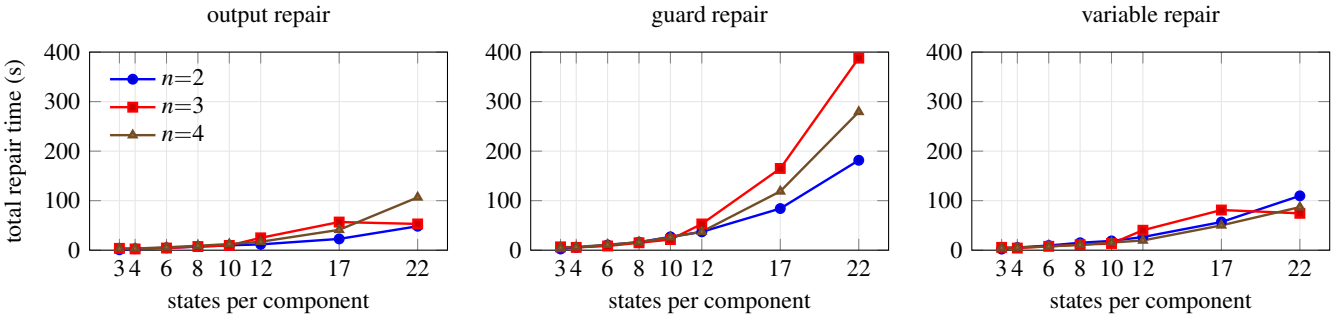


Figure 4: Total repair time as a function of per-component state count, for chain lengths $n \in \{2, 3, 4\}$ and the three fault types. Conformance checking accounts for under 3% of each total.

domain expert's notion of the best repair. Different weight choices can lead to different optimal repairs.

Internal validity. The reachability claim is mitigated by exhaustive BFS over the finite configuration space, which makes the reachable set a computed result rather than an assumption. Blame localization is derived from a single counterexample, so with several coexisting faults the attributed component can depend on traversal order; we therefore report blame per counterexample rather than as a global diagnosis. When several repairs have identical minimal cost, the selected repair may depend on solver tie-breaking. We therefore interpret the reported repair as one optimal repair, not necessarily the unique best repair.

External validity. Our effectiveness evidence comes from a single running decomposition (ResourceSimulation into its passive and active variants) in the Palladio/PCM domain. The faults are synthetic and each case injects a single fault, so interacting or multi-component faults are out of scope. We therefore do not claim that the measured effectiveness transfers unchanged to other systems; broadening this evidence across domains, defect types, and fault combinations is future work. We also do not compare empirically against alternative localization or repair techniques. Our current implementation targets cascade compositions with time-synchronous communication and delayed feedback. The results may not transfer to arbitrary graph-shaped architectures, asynchronous/event-based communication, multiple feedback cycles, or dynamically reconfigurable systems.

Reliability. A reliability concern is that several different repairs can share the same minimal cost, and in that case the tool reports just one of them. For the seeded-fault study, faults were generated from a fixed mutation scheme applied systematically to the model elements rather than chosen by hand, which limits but does not eliminate selection bias. To support reproducibility, we plan to release the generated benchmark instances, the seeded-fault scripts, and the solver configuration.

8 Related Work

Our work combines conformance checking for decomposed architectures with counterexample-guided repair.

Conformance and semantic differencing. We build on the conformance framework of [32], which defines language-specific conformance relations under incarnation mappings and provides checks for class diagrams, feature models, and MontiArcAutomata. A corresponding conformance-checking approach for process models is presented in [41]. These approaches compare a reference model with a monolithic concrete model, whereas we are now concerned with a concrete subarchitecture obtained by decomposition. Compared to the conformance checking approach introduced in [32], which checks if each transition of the concrete model is permitted by the reference model, we verify transition sequences in a composed concrete model. This required an extension of the incarnation mapping: whereas the original mapped individual states of the concrete automaton to states of the reference automaton, the concrete model may now consist of multiple automata and we therefore map tuples of states to states of the original.

Conformance checking is closely related to semantic differencing [38], where differences are represented by semantic witnesses rather than syntactic edits. Such operators have been studied for, among others, activity diagrams [27, 39], class diagrams [39, 52], feature models [14], use-case diagrams [28], sequence diagrams [26], statecharts [13], and bounded Data Petri Nets [45]. Semantic differencing has also been considered for model compositions, including combinations of class diagrams, OCL constraints, and object diagrams [54], as well as C&C architectures with statechart behavior [8, 18]. Unlike these existing semantic differencing approaches for C&C architectures, we rely on the mapping-witnessed simulation of Section 4.1, which is strictly stronger than black-box trace refinement, requires no trace-inclusion check, and yields counterexamples localized to a single step, and hence to a single transition.

Compositional refinement of architectures. Complementary approaches to semantic differencing are formal verification methods based on FOCUS [7]. Recent work translates SysML v2 specifications into Isabelle/HOL theories to prove refinement across architectural abstraction levels [24, 25]. This theorem-prover-based line is complementary to our counterexample-oriented approach, in which we check a given decomposition and use violations to guide repair.

Compositional verification is also addressed by assumption-guarantee reasoning, where components are verified under assumptions about their environments [10]. These techniques aim to decompose proof obligations for a fixed property, whereas our

approach checks a given architectural decomposition against a reference model and uses conformance counterexamples to localize and repair faults.

Automata-based component theories provide another foundation for compositional reasoning. I/O automata [37] model reactive and distributed components by input, output, and internal actions; refinement is typically expressed through trace inclusion or simulation. Interface automata [11] build on a similar input/output view but emphasize interface compatibility and substitutability under environmental assumptions, using alternating simulation as a refinement relation. Both lines provide general theories for composing automata and reasoning about component behavior at their boundaries. Our setting is more specific: we consider MontiArcAutomata embedded in C&C architectures, check whether a concrete cascade decomposition conforms to a monolithic reference under an incarnation mapping, and exploit the resulting counterexamples for localization and MaxSMT-based repair.

The preservation of semantics during system decomposition has also been addressed previously in [29] which focuses on automatic parallel decomposition. In contrast, we verify and repair user-provided cascade decompositions with delayed feedback.

FOCUS is one of several formal foundations for reasoning about communicating and component-based systems. Related formalisms include process algebras such as CSP [23, 46], CCS [42], and the π-calculus [43], as well as Ptolemy [36], TLA [1], and Petri nets [50]. These formalisms support, in different ways, nondeterminism, communication, refinement, timing, and hierarchical composition. Petri nets are also used in the Palladio context [51] as a semantic basis for simulation-based performance analyses [33]. Closer to our concern with semantic soundness, recent work models the composition of analysis components and automatically checks soundness constraints over composed analyses [57]. In contrast, our work does not introduce a new semantic foundation or performance analysis; it uses a statechart-based C&C setting to check whether a concrete cascade decomposition conforms to a reference component and to repair violations automatically.

Automated repair. Automated repair has been studied extensively for programs and models [17, 44]. Our repair formulation is closest to counterexample-guided, MaxSMT-based model repair. Kölbl et al. [35] repair networks of timed automata from diagnostic traces by encoding candidate changes as MaxSMT constraints. We share the counterexample-guided optimization view, but repair decomposed component architectures against a reference model and include blame distance to steer repairs toward the responsible subcomponent. Kautz [26] repairs single models by searching syntactic edit sequences that restore a violated property; our setting instead repairs a composed system under a multi-dimensional cost.

Program-repair approaches such as SemFix [47] and DirectFix [40] synthesize patches from logical constraints. Quantitative repair approaches, including QLOSE [15] and AuRUS [6], optimize the syntactic and semantic similarity to the original artifact. Our distinguishing feature is that the repair is driven by a compositional conformance counterexample and explicitly accounts for the location of the blamed component.

9 Conclusion and Future Work

We presented a compositional conformance-checking and repair approach for component-based systems modeled in MontiArcAutomata under cascade composition. Given a reference model, a concrete composition, and an incarnation mapping, the approach checks whether the decomposed system conforms to the reference, localizes violations to responsible subcomponents and transitions, and synthesizes MaxSMT repairs. Repairs minimize syntactic change, bounded behavioral loss, and distance from the blamed component, and are validated by re-checking the repaired model. The evaluation detected and localized all nine injected faults, synthesized conformance-restoring repairs for each, and correctly reported the conformant case. All 72 configurations completed successfully, with conformance checking negligible and repair synthesis dominating and growing super-linearly in the state count.

Future work includes adopting a more open-world interpretation of statechart semantics, extending the approach beyond time-synchronous cascade composition to richer and event-based interaction topologies, evaluating it on larger systems with naturally occurring faults, and reducing the reliance on hand-authored mappings through partial automatic derivation and validation. Comparing against alternative localization and repair techniques would additionally require a shared benchmark, since those approaches take diagnostic traces or violated properties rather than a reference model and mapping as input.

Acknowledgments

This work was funded by the DFG (German Research Foundation) – project number 499241390 (FeCoMASS), and supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263.

References

[1] Martín Abadi and Leslie Lamport. 1994. Open Systems in TLA. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing - PODC '94*, James Anderson, David Peleg, and Elizabeth Borowsky (Eds.). ACM Press, New York, New York, USA, 81–90. doi:10.1145/197917.197960

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 2009. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software* 82, 1 (Jan. 2009), 3–22. doi:10.1016/j.jss.2008.03.066

[4] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. 2017. Component and Connector Views in Practice: An Experience Report. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)* (Austin). IEEE, 167–177.

[5] Daniel Brand and Pitro Zafiropulo. 1983. On Communicating Finite-State Machines. *J. ACM* 30, 2 (April 1983), 323–342. doi:10.1145/322374.322380

[6] Matías Brizzio, Maxime Cordy, Mike Papadakis, César Sánchez, Nazareno Aguirre, and Renzo Degiovanni. 2023. Automated Repair of Unrealisable LTL Specifications Guided by Model Counting. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '23)*. ACM, 1499–1507. doi:10.1145/3583131.3590454

[7] Manfred Broy and Ketil Stølen. 2001. *Specification and Development of Interactive Systems.* Springer New York. doi:10.1007/978-1-4613-0091-5

[8] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2019. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software (JSS)* 149 (March 2019), 437–461. doi:10.1016/j.jss.2018.12.016

[9] Edmund M Clarke. 1997. Model checking. In *International conference on foundations of software technology and theoretical computer science*. Springer, 54–56.

[10] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. 2003. Learning Assumptions for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS, Vol. 2619)*. Springer, 331–346.

[11] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface Automata. In *Proceedings of the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 109–120.

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[13] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. 2019. Semantic Differencing of Statecharts for Object-oriented Systems. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)* (Prague), Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić (Eds.). SciTePress, 274–282.

[14] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. 2019. Semantic Evolution Analysis of Feature Models. In *International Systems and Software Product Line Conference (SPLC'19)* (Paris), Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 245–255.

[15] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. *Qlose: Program Repair with Quantitative Objectives*. Springer International Publishing, 383–401. doi:10.1007/978-3-319-41540-6_21

[16] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)* (May 2007), 37–54.

[17] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[18] Johanna Grahl, Bernhard Rumpe, Max Stachon, and Sebastian Stüber. 2026. Dynamic Symbolic Execution of Component-and-Connector Models for Semantic Differencing. In *Modellierung 2026 (LNI, Vol. P378)*. GI, Bayreuth, 89–105. doi:10.18420/modellierung2026-06

[19] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2012. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Technical Report AIB-2012-03. RWTH Aachen University.

[20] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal* 37, 10 (October 2004), 64–72.

[21] Robert Heinrich, Francisco Durán, Carolyn L Talcott, and Steffen Zschaler. 2021. *Composing Model-Based Analysis Tools*. Springer.

[22] Robert Heinrich, Misha Strittmatter, and Ralf Reussner. 2021. A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis. *IEEE Transactions on Software Engineering* 47, 4 (April 2021), 775–800. doi:10.1109/tse.2019.2903797

[23] C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice Hall International, Englewood Cliffs, N.J.

[24] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. 2025. Enhancing System Model Quality: Evaluation of the Systems Modeling Language (SysML)-Driven Approach in Avionics. *Journal of Aerospace Information Systems (JAIS)* 22, 5 (May 2025), 367–378. doi:10.2514/1.I011476

[25] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. 2025. Model-driven Development for Functional Correctness of Avionics Systems: A Verification Framework for SysML Specifications. *Council of European Aerospace Societies Aeronautical Journal (CEAS* 16, 1 (January 2025), 33–48. doi:10.1007/s13272-024-00762-6

[26] Oliver Kautz. 2021. *Model Analyses Based on Semantic Differencing and Automatic Model Repair*. Shaker Verlag.

[27] Oliver Kautz and Bernhard Rumpe. 2018. Semantic Differencing of Activity Diagrams by a Translation into Finite Automata. In *Proceedings of MODELS 2018. Workshop ME* (Copenhagen).

[28] Oliver Kautz, Bernhard Rumpe, and Louis Wachtmeister. 2022. Semantic Differencing of Use Case Diagrams. *Journal of Object Technology (JOT)* 21 (July 2022), 3:1–14.

[29] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2020. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering Journal* 27 (April 2020), 119–151.

[30] Floriment Klinaku, Sarah Sophie Stieß, Alireza Hakamian, and Steffen Becker. 2025. An architectural view type for elasticity modeling and simulation—The Slingshot approach. *Journal of Systems and Software* 228 (Oct. 2025), 112432. doi:10.1016/j.jss.2025.112432

[31] Sandro Koch. 2024. *A reference structure for modular model-based analyses*. KIT Scientific Publishing.

[32] Marco Konersmann, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo. 2024. Towards a Semantically Useful Definition of Conformance with a Reference Model. *Journal of Object Technology (JOT)* 23, 3 (July 2024), 1–14. doi:10.5381/jot.2024.23.3.a5

[33] Heiko Koziolek and Ralf Reussner. 2008. A model transformation from the palladio component model to layered queueing networks. In *SPEC International Performance Evaluation Workshop*. Springer, 58–78.

[34] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. 2017. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)* (Marburg) *(LNCS 10376)*. Springer, 34–50.

[35] Martin Kölbl, Stefan Leue, and Thomas Wies. 2021. Automated repair for timed systems. *Formal Methods in System Design* 59, 1-3 (Dec. 2021), 136–169. doi:10.1007/s10703-022-00397-5

[36] Edward Lee. 2016. Fundamental Limits of Cyber-Physical Systems Modeling. *ACM Transactions on Cyber-Physical Systems* 1 (11 2016), 1–26. doi:10.1145/2912149

[37] Nancy A. Lynch and Mark R. Tuttle. 1989. An Introduction to Input/Output Automata. *CWI Quarterly* 2, 3 (1989), 219–246.

[38] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2010. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10) (LNCS 6627)*. Springer, 194–203.

[39] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP 2011 - Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, 230–254.

[40] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 448–458.

[41] Judith Michael, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo. 2026. Automated Conformance Checking Between Process Models and Their Generalized Reference Process Models. In *Mining a Scientist's Process: Essays Dedicated to Wil van der Aalst on the Occasion of His 60th Birthday*, Mendling, Jan and Leemans, Sander and van Dongen, Boudewijn F. and Reijers, Hajo (Ed.). Springer, 525–550. doi:10.1007/978-3-032-17618-9_35

[42] R. Milner. 1982. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg.

[43] Robin Milner. 1999. *Communicating and mobile systems: the pi calculus*. Cambridge university press.

[44] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24.

[45] Marco Montali and Sarah Winkler. 2024. Relating behaviour of data-aware process models. *Data & Knowledge Engineering* 154 (2024), 102363.

[46] Toby Murray and Gavin Lowe. 2009. On Refinement-Closed Security Properties and Nondeterministic Compositions. *Electr. Notes Theor. Comput. Sci.* 250 (09 2009), 49–68. doi:10.1016/j.entcs.2009.08.017

[47] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781.

[48] Robert Nieuwenhuis and Albert Oliveras. 2006. *On SAT Modulo Theories and Optimization Problems*. Springer Berlin Heidelberg, 156–169. doi:10.1007/11814948_18

[49] Frederik Reiche, Ralf Reussner, and Robert Heinrich. 2025. Detecting Information Flow Security Vulnerabilities by Analysis Coupling. *IEEE Transactions on Software Engineering* 51, 10 (Oct. 2025), 2710–2743. doi:10.1109/tse.2025.3589647

[50] Wolfgang Reisig. 1985. *Petri Nets: An Introduction*. Springer, Berlin, Heidelberg.

[51] Ralf H Reussner, Steffen Becker, Jens Happe, Robert Heinrich, and Anne Koziolek. 2016. *Modeling and simulating software architectures: The Palladio approach*. MIT Press.

[52] Jan Oliver Ringert, Bernhard Rumpe, and Max Stachon. 2023. On Implementing Open World Semantic Differencing for Class Diagrams. *Journal of Object Technology (JOT)* 22, 2 (July 2023), 2:1–14. doi:10.5381/jot.2023.22.2.a11

[53] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2014. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Shaker Verlag.

[54] Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Valdes Voufo. 2026. Semantic difference analysis for model compositions of class diagrams, OCL constraints, and object diagrams. *Innovations in Systems and Software Engineering* 22, 1 (May 2026), 14. doi:10.1007/s11334-026-00630-y

[55] Bernhard Rumpe and Andreas Wortmann. 2018. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, Lohstroh, Marten and Derler, Patricia Sirjani, Marjan (Ed.). Springer, 383–406.

[56] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (Dec. 2008), 131–164. doi:10.1007/s10664-008-9102-8

[57] Bahareh Taghavi, Sebastian Weber, Adrian Marin, Bernhard Rumpe, Sebastian Stüber, Jörg Henß, Thomas Weber, and Robert Heinrich. 2026. Modeling the composition of analysis components and automatic constraint checking for semantic soundness. *Journal of Systems and Software* 231 (Jan. 2026), 112637. doi:10.1016/j.jss.2025.112637

[58] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. 2009. *Software Architecture: Foundations, Theory, and Practice*. Wiley.