The Journal of Systems and Software 231 (2026) 112637

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Modeling the composition of analysis components and automatic constraint checking for semantic soundness☆,☆☆

Bahareh Taghavi [a,*], Sebastian Weber [b], Adrian Marin [c], Bernhard Rumpe [c], Sebastian Stüber [c], Jörg Henß [b], Thomas Weber [a], Robert Heinrich [d]

[a] Karlsruhe Institute of Technology, Karlsruhe, 76131, Germany
[b] FZI Research Center for Information Technology, Karlsruhe, 76131, Germany
[c] Software Engineering, RWTH Aachen University, Aachen, 52062, Germany
[d] Ulm University, Ulm, 89081, Germany

ARTICLE INFO

Dataset link: https://github.com/FeCoMASS/Model-Transformation-for-Automated-Constraint-Validation

Keywords:
Semantic constraint checking
Software architecture
Model transformation
Palladio
MontiArc

ABSTRACT

Component-based software architecture enables software architects to design complex systems by composing components that interact through well-defined, syntactically specified interfaces. A special kind of component we investigated in our previous work is the analysis components. Analysis components support the evaluation and prediction of system's functional and non-functional properties. Evaluating these properties early in the development process helps optimize system performance and ensure compliance with requirements. While approaches for modeling and analyzing such systems, such as the Palladio approach, support syntactic validation of the composition, they often lack mechanisms to ensure the semantic soundness of compositions. In this paper, we present a model transformation approach to help architects ensure that system models are semantically sound and behave as expected. This approach enables the transformation of Palladio models into MontiArc models, allowing architects to enrich their system representations with semantic constraints and validate these constraints with the MontiArc workbench. This ensures that component interactions are consistent with both structural composition and intended semantics. We evaluate our approach through two different case studies. From these case studies, we derived several scenarios with varying constraints and states to assess the accuracy and performance of our approach. To evaluate accuracy, we examined our approach's ability to check semantic constraints and detect violations. We observed high accuracy across the case studies. For performance, we analyze time complexity in different constraint types. The approach performed well when applied to arithmetic constraints, with its effectiveness decreasing when applied to more complex string-centered constraints.

1. Introduction

Software plays a significant role in various domains, including engineering, society, and the economy. Software is the backbone of modern technology because it enables the functionality of a wide range of systems, from desktop applications to enterprise-level technologies. Hence, software must be deemed trustworthy; in other words, the software must be ensured to fulfill the desired functionality and quality properties (e.g., performance and security). Effective assurances of trustworthiness can be given by analysis techniques.

Analysis techniques are designed to investigate specific questions because complex systems exhibit diverse behaviors that require specialized and targeted analysis approaches. Examples of questions that may need investigating are "What happens if the number of users of my system doubles?" or "What happens if my system is attacked?" To answer such specific questions, each analysis technique must employ a modeling formalism tailored to the particular aspect it is intended to analyze. For instance, analyses are tailored to specific disciplines, such as modeling user interactions through the usage analysis technique (Hamlet et al., 2004) or examining internal behavior and interactions within system components using system-level analysis techniques (Bertolino and Mirandola, 2004). Additionally, software analysis must be capable of evaluating and predicting how well the software aligns with functional requirements and quality properties. To this end, analysis components are designed as modular building blocks (Talcott et al.,

☆ This article is part of a Special issue entitled: 'SoftArch' published in The Journal of Systems & Software.
☆☆ Editor: Raffaela Mirandola.
* Corresponding author.
E-mail address: bahareh.taghavi@kit.edu (B. Taghavi).

https://doi.org/10.1016/j.jss.2025.112637
Received 3 March 2025; Received in revised form 11 July 2025; Accepted 12 September 2025
Available online 26 September 2025

2021), and further, the components are also composed to scale analyses to specific investigation goals. This composition poses a challenge: Now, for each composition, it must be verified whether the composed analysis components interact in such a way that the intended behavior is preserved and the expected results are produced. In other words, every composition must be verified for semantic soundness, meaning that the composed components conform to their individual assumptions and guarantees and produce valid analysis results.

One of the leading frameworks for modeling and analyzing component-based software architectures is the Palladio simulation approach. With its metamodel, Palladio Component Model (PCM) (Reussner et al., 2016), and its simulation tool Palladio Bench (Heinrich et al., 2018), the framework allows developers to create abstract representations of software architectures, simulate their behavior, and investigate quality properties such as performance, scalability, and resource utilization. Its structured modeling approach and extensibility make it a suitable foundation for reasoning about system component-level interactions.

In spite of Palladio's strength in architectural-level analysis, PCM supports only syntactic validation, such as checking for interface compatibility, and lacks support for reasoning about whether compositions are semantically sound. This limitation, as discussed in Heinrich et al. (2021), restricts the applicability of Palladio in scenarios where the semantic soundness of component interactions and their emergent behaviors is critical. In this work, we focus on modeling analysis component compositions in Palladio to better support the extension and replacement of analysis components by enabling the verification of their composition. As highlighted by Koch (2024), finding reusable analysis components remains a challenge due to factors such as a lack of standardization, insufficient maintenance, and inadequate documentation. These challenges hinder the reusability of analysis components. To address this, and to ensure the reusability of an analysis component within a composition at design time, Palladio will benefit from the capability to verify the semantic soundness of analysis component compositions through semantic constraint checking and to reveal violated constraints to analysis architects.

Hence, we propose a complementary approach to address the lack of semantic soundness in the composition of analysis components, instead of introducing direct support for semantic constraints within Palladio itself. This approach bridges Palladio's existing strengths in quality property analysis with the semantic soundness capabilities of a framework for semantic constraint validation. In particular, we enable the transformation of PCM models into MontiArc models (Haber, 2016). MontiArc supports the textual modeling of Component & Connector architectures and allows the static validation of semantic constraints specified at the ports of the components, which are connected through the connectors. Our approach includes a systematic model-to-model transformation process, where the architectural elements of PCM, such as components, interfaces, and connectors, are mapped to their corresponding representations in MontiArc. Once the PCM model is transformed into a MontiArc model, semantic constraints can be applied to verify whether the modeled system's behavior aligns with expected logical and functional requirements. These constraints are specified in an additional model using a technique similar to tagging (Greifenberg et al., 2015), where metadata or annotations are attached to elements of the model without altering its core structure. This avoids adding more complexity to the PCM and enriches the MontiArc model generated from the PCM. This approach contributes to trustworthy software architecture by ensuring that the models of analysis components adhere to defined constraints, which leads to more predictable and reliable system behavior. By ensuring the reliable behavior of analysis components in the static context of architectural models, we support the semantic soundness of their composition by enabling well-defined and meaningful interactions among them.

Contributions: In this article, we extend (Weber et al., 2024) work to contribute to the trustworthiness of PCM as a foundation for early architectural validation, by enabling constraint checking in scenarios where ensuring the semantic soundness of analysis component compositions is critical. In Weber et al. (2024), a robust transformation of PCM models to MontiArc models is provided that takes into account the architectural details in both models and also considers how architectural modeling in PCM can be abstracted and mapped to MontiArc. The method in Weber et al. (2024) specifies constraints in Palladio and also presents a solution for integrating these constraints into MontiArc models. Additionally, Weber et al. (2024) include a tool for constraint checking based on an assumption-guarantee formalism within the MontiArc framework. However, the approach of Weber et al. (2024) is only a technical demonstration and, consequently, is limited in capability, lacking support for more complex constraints and a comprehensive evaluation. Therefore, we build on that foundation and extend the range of supported constraint types, including more complex regular expression constraints. Furthermore, we analyze the accuracy and performance of constraint checking in the MontiArc framework. Lastly, we provide a more detailed comparison with related work and apply our extended approach to a broad set of case studies.

Our contributions in this paper are as follows:

- C1 We incorporate support for constraint checking using regular expressions into the semantic constraint analysis, thereby providing a more expressive and powerful specification formalism. This enhancement enables our approach to support a broader range of constraint types within MontiArc and Palladio, in particular complex naming schemes present in e.g., file endings. Unlike existing OCL extensions (Lano, 2021; Damus and Sánchez-Barbudo, 2002), our lightweight approach avoids complex library dependencies by using a simple formalism focused only on language membership.
- C2 We conducted a literature review to provide an overview of existing related approaches. This review revealed that while several approaches (Meyer, 1988, 1992; Swamy et al., 2013, 2016) address formal verification, they typically lack support for architectural modeling. In contrast, our approach explicitly integrates such mechanisms into the semantic constraint analysis, enabling validation of component interactions.
- C3 We conducted an evaluation and structured it systematically based on relevant development scenarios in order to examine how changes in component composition affect semantic constraint checking. To this end, we employed a new case study to evaluate regular expression support in addition to systematically deriving relevant scenarios for the original case study. Furthermore, we assessed both the accuracy of semantic constraint checking and the performance of our approach.

The paper is structured as follows: Section 2 provides an introduction to the Palladio approach, its simulators, and the modeling formalism PCM and MontiArc. This is followed by a discussion of related approaches to address C2 in Section 3. Section 4 introduces a running example that we reference throughout the paper. Our approach is presented in Section 5, where we outline the transformation process from Palladio to MontiArc, followed by the constraint-checking process. Section 6 presents the evaluation of the accuracy and performance of our approach. Finally, Section 7 focuses on conclusions and future research directions.

2. Background

This section provides an overview of the two frameworks used in this paper: Palladio and MontiArc, both of which adhere to a Component & Connector (C&C) approach. In these frameworks, the logic is encapsulated within components that communicate through explicitly

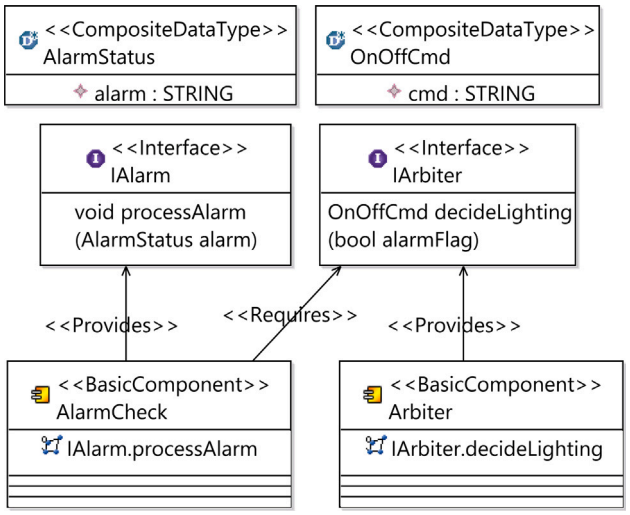


Fig. 1. LightCtrl repository diagram.

defined communication channels. The shared C&C architecture of Palladio and MontiArc enables a translation between the two models, which preserves both structure and meaning. During this translation process, essential information – such as components, communication channels, and message types – is preserved, thereby facilitating comprehensive analysis in the context of software validation and verification.

Palladio (Reussner et al., 2016) is an approach to simulating software architecture, aiming to analyze and predict performance, among other quality properties. The tooling that implements the Palladio approach is known as the Palladio-Bench (Heinrich et al., 2018). The Palladio Component Model (PCM) is a domain-specific modeling language and is composed of multiple sub-models, each targeting a particular developer role. Component developers contribute by specifying behavioral aspects of their components and interfaces in the repository model. Subsequently, system architects leverage these repositories to assemble concrete component-based software systems in the assembly model. Meanwhile, system deployers focus on modeling the resource environment and allocating components across different resources. Business domain experts are also responsible for providing usage models that describe critical usage scenarios and outline user behavior. Palladio facilitates model evaluation through simulation, enabling the prediction of performance metrics like response times and hardware utilization under specified workloads.

To provide a clearer understanding of Palladio's modeling capabilities, Fig. 1 presents the Repository model of a system called LightCtrl. The Repository model defines the reusable component types, their interfaces, and associated data types or service operations. This serves as the foundation for modeling our components and their interactions in Palladio. Fig. 1 illustrates a simplified version of the LightCtrl system, which is a component-based architecture designed to process alarm inputs and generate corresponding control commands. This system consists of two components: `AlarmCheck` and `Arbiter`. `AlarmCheck` component receives an input of type `AlarmStatus` via the interface `IAlarm`. It processes this input according to its internal logic and emits a Boolean flag. `Arbiter` component receives the Boolean flag through its interface `IArbiter`. Based on the value of this flag, it determines whether a control command should be issued. This decision is exposed through its interface, `IArbiter`, which returns an `OnOffCmd` (e.g., to activate or deactivate a device).

In addition to the components and interfaces, the Repository model also includes composite data types that facilitate structured data exchange between components. The composite data types `AlarmStatus` and `OnOffCmd` are depicted in Fig. 1. `AlarmStatus` encapsulates the

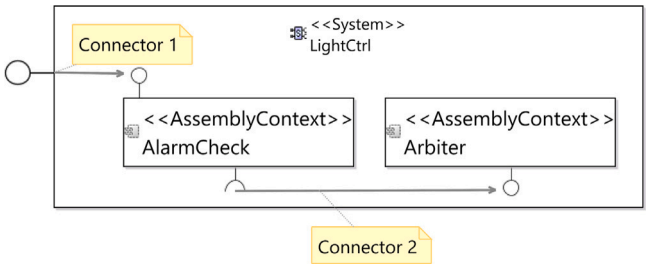


Fig. 2. LightCtrl assembly diagram.

alarm information received by the `AlarmCheck` component through the `IAlarm` interface. `OnOffCmd` represents the output generated by the `Arbiter` component through the `IArbiter` interface. It contains a `cmd` attribute of type `String` that specifies the resulting control command (e.g., "ON", "OFF").

Fig. 2 shows the Assembly model of the LightCtrl system in Palladio. Each box labeled with a component name (`AlarmCheck` and `Arbiter`) represents an `AssemblyContext`, which is a concrete instantiation of a repository component in the system. The connections between these `AssemblyContexts` are modeled using `AssemblyConnectors`, which bind required roles of one component instance to the provided roles of another. For example, the required interface `IArbiter` of `AlarmCheck` is connected to the provided interface of `Arbiter` that establishes the communication path between them (Connector2). Additionally, the Assembly model includes `DelegationConnector` (Connector1) that route the system-level inputs/outputs to the appropriate internal components. Specifically, the alarm input received by the overall `LightCtrl` system is delegated to `AlarmCheck`.

Montiarc[1] (Haber, 2016) is a textual modeling language to describe C&C systems. The components receive input messages and send output messages via typed and directed ports. Communication is only possible through these explicitly defined ports, which helps reduce hidden links. MontiArc has a precisely defined semantic foundation in FOCUS (Broy and Stølen, 2001). The textual MontiArc models can be mapped into the mathematical FOCUS space to define the semantics of the models (Harel and Rumpe, 2004), which enables formal interpretation and reasoning. This enables formal verification of MontiArc models at design time. In MontiArc, semantics are used to support formal reasoning about system behavior, particularly with respect to component composition and communication. For instance, properties such as the absence of message loss, preservation of message order, and correctness of component composition can be formally proven. The formal proofs are performed using MontiBelle (Kausch et al., 2020b,a) and the interactive theorem prover Isabelle (Nipkow et al., 2002). By leveraging this formal semantics, MontiArc supports early detection of design issues and ensures that system models satisfy critical correctness properties before implementation.

Listing 1 shows the example of LightCtrl modeled in MontiArc. As shown, the main component `LightCtrl` represents the system-level assembly, which is depicted in the Assembly Diagram in Fig. 2. In MontiArc, the keyword component signifies a local component type definition. This can then be used inside a hierarchical composition by declaring an instance of the component type, much like Palladio's use of assembly contexts within a system model. The ports of the main component, such as `alarm` and `cmd`, serve as the external interfaces of the system (lines 3 and 4, Listing 1). These correspond to Palladio's

[1] "the montiarc architecture description language", https://github.com/monticore/montiarc (accessed: january 23, 2025).

system-level required and provided roles, which are linked through delegation connectors. For instance, the `alarm` port in MontiArc delegates its input to the internal component `alarmCheck` via a connection specified using an arrow notation (line 21, Listing 1). Additionally, outputs like `cmd` can also be modeled in Palladio as return values of operations defined in a provided interface (depends on the design decision). The connection between the `AlarmCheck` and `Arbiter` components is represented in line 22 of Listing 1. This connection is equivalent to an assembly connector in Palladio, where the required role of `AlarmCheck` is connected to the provided role of `Arbiter`. Finally, the output from the `Arbiter` is routed to the system's output port, as shown in line 23 of Listing 1.

```
1  component LightCtrl {
2    port
3      in AlarmStatus alarm,
4      out OnOffCmd cmd;
5
6    component AlarmCheck {
7      port
8        in AlarmStatus alarm,
9        out Boolean alarmFlag;
10   }
11
12   component Arbiter {
13     port
14       in Boolean alarmFlagIn,
15       out OnOffCmd cmd;
16   }
17
18   AlarmCheck alarmCheck;
19   Arbiter arbiter;
20
21   connect alarm -> alarmCheck.alarm;
22   connect alarmCheck.alarmFlag -> arbiter.
         alarmFlag;
23   connect arbiter.cmd -> cmd;
24 }
```

Listing 1: MontiArc model of the LightCtrl component.

3 Related work

Model-Driven Software Engineering (MDSE) is a method to bridge the gap between a problem and the design and implementation of a solution for the problem (France and Rumpe, 2007) and offers a comprehensive view to develop systems (Brambilla et al., 2017). The MDSE approach for software development has been proven to increase efficiency and effectiveness (Acerbis et al., 2007). By applying MDSE, verification through certification of source code can be reduced to certifying the code generator and the model the code was generated from, reducing the expense. In this way, formal modeling notations and development tools that operate on architectural specifications are needed to support architecture-based development (Medvidovic and Taylor, 1997). That is why Architecture Description Languages (ADLs) offer an avenue for modeling different aspects of the system under development, allowing verification to be performed on the structure of the system by employing composition (Allen et al., 2002; Haddad et al., 2011). Designers can use an ADL aimed at a specific domain to verify their system against specific properties (Allen et al., 2002). Although there are quite a few ADLs available, they are mostly used in research environments and are not widely adopted by industry and some of them are only compatible with certain architectural styles (Gacek and de Lemos, 2006). Furthermore, refinement is explicitly supported only in a few ADLs, helping to ensure that lower-level constraints do not violate higher-level constraints (Moriconi et al., 1995).

Model checking is an established verification technique that involves evaluating a system's model against its specification, considering all possible execution traces, to systematically determine whether the model satisfies the desired property (Baier and Katoen, 2008). Model checking relies on formal constraints to define the properties that a system must satisfy, ensuring compliance with its intended behavior. Czepa et al. (2017) build upon this principle by introducing Card, an Eclipse plug-in that automates conformance checking between UML 2.0 models and their corresponding Java implementations to mitigate architecture erosion. The evaluation of Card demonstrates its effectiveness in detecting violations, usability within the Eclipse IDE, and scalability across different project sizes. Czepa and Zdun (2019) also conducted a controlled experiment with 116 participants to compare novice software designers' understanding of graphical and textual behavioral constraint representations. Ly et al. (2012) introduced a framework that provides a formal foundation for developing constraint-aware process management systems (PrMS). It provides comprehensive support for semantic constraints in PrMS, enabling the validation and enforcement of business rules and policies throughout the entire process lifecycle. Cardei et al. (2008) presented a methodology for requirements specification and validation that uses an ontology-based language for the semantic description of functional product requirements. The aim of the paper is to detect omissions and consistency errors in the requirements specification early on, before the design stage.

The FOCUS framework (Broy et al., 1992; Ringert and Rumpe, 2011) enables modular system development by providing precise specifications for component behavior and interactions, along with a mathematical semantics for defining the structure and behavior of software systems. It supports various specification styles, including assumption-guarantee reasoning, which allows system components to be defined based on assumptions about their environment and the guarantees they must fulfill under those conditions. Broy and Stølen (2012) build upon the FOCUS framework, offering models, formal methods, and verification techniques for the stepwise specification and development of distributed interactive systems. They present a rigorous mathematical and logical foundation for software and systems engineering, enabling the verification and refinement of complex system interactions through formal reasoning. Kausch et al. (2024) built on the former and applied a model-driven approach, by verifying liveness properties specified in the SysML v2 ADL over an uplink feed of an avionic system in cooperation with Airbus. This approach employs theorem proving to reach its goal of formal verification, by transforming components and their behavioral specification into the interactive theorem prover Isabelle and thus validating compositions by construction. This theorem proving approach is more potent in showing semantic soundness of a composition by handling history-based input–output specifications. But generating fully automated proofs in Isabelle is not yet generalizable.

Formal verification and software correctness are critical in ensuring software reliability. Eiffel (Meyer, 1988) is an object-oriented programming language and environment that extends the concept of assumption-guarantee reasoning through its native support for Design by Contract (DbC) (Meyer, 1992), explicitly enforcing these interactions within its programming model. DbC introduces a software development methodology that enhances reliability by incorporating formal contracts into code. It is based on the principle that if a client satisfies a supplier's precondition, the supplier is obliged to fulfill its postcondition. These contracts serve as formal agreements between different parts of a system, ensuring software correctness, robustness, and maintainability. Eiffel demonstrates how contracts improve software integrity by explicitly defining expected behaviors and preventing unexpected failures. F* (Swamy et al., 2013, 2016) is a dependently-typed, functional programming language designed for formal verification, incorporating effectful programming, satisfiability modulo theories SMT-based verification, and manual proofs. It enables users to specify preconditions and post-conditions with mathematical precision, allowing for rigorous formal reasoning about program correctness. These two integrated approaches act at the source code level. While

they do perform, and in the case of F*, can also leverage underspecification, the composition of analyses modeled through these can only be performed at the source code level. Architecture analysis validation through semantic soundness assessments of compositions, however, is better suited for ADLs enriched with an assumption-guarantee formalism.

Moreover, several studies have addressed the verification of UML/OCL models. The UML can be considered a generic ADL, thus qualifying it for this use case. But as stated by Pandey (2010), its inconsistencies and ambiguities regarding formal semantics outside of state charts and class diagrams hinder its use for analysis and code generation. Tools such as the USE Validator (Richters and Gogolla, 2000) support checking OCL constraints over UML models, particularly focusing on class and object diagrams. Similarly, Cabot et al. (2008) present an approach for verifying the correctness of UML class diagrams annotated with OCL constraints by transforming them into constraint satisfaction problems. However, they are not designed for component-based architectures. Moreover, OCL lacks the extensibility required to capture system-level architectural constraints that span interfaces, ports, and component interactions, as supported in MontiArc.

To summarize this section, while there are works that employ DbC on the source code level and others that perform requirement-to-realization tracing and validation, there are no works that explore applying automatic composition validations on the architectural modeling level, apart from theorem-proving-based approaches which cannot guarantee full automation.

4 Running example

To illustrate the concepts described in our approach and to briefly demonstrate how it is beneficial, we use the Slingshot simulator (Katić et al., 2021; Klinaku et al., 2025) as a running example. We selected Slingshot because our focus is on composition case studies involving analysis components. Slingshot is a simulation environment as part of the Palladio approach and is based on an event-driven architecture. This simulator is employed to analyze the dynamic behavior and performance of component-based software architectures. For the purposes of this paper, we employ a simplified model of Slingshot (building on our earlier work, which particularly modeled its behavioral aspects (Taghavi et al., 2025)) to make the concepts easier to understand and to focus on the core aspects relevant to our proposed approach.

Slingshot currently comprises three simulation components: the `UsageSimulation`, the `SystemSimulation`, and the `ResourceSimulation` components. As shown in Fig. 3, the Repository Model Diagram in Palladio illustrates the components, their interfaces, and their connections. We represent the three Slingshot components using `BasicComponents` and establish their connections through `Interface` model elements. To initiate a simulation in Slingshot, Slingshot begins with the `UsageSimulation` component, which reads and interprets predefined usage scenarios. These usage scenarios describe user behavior and interaction with the system over time. One key parameter in the usage scenario, `numOfUsers`, defines the number of users concurrently present in the system during simulation. As the first constraint in this composition, we define that this parameter must be greater than zero to ensure meaningful results. Based on this and other usage scenario parameters, the `UsageSimulation` component generates `systemCall`, which may include the name of the invoked method. These calls are then processed by the `SystemSimulation`. The `SystemSimulation` component identifies the demand for computational resources. Then it passes these resource demands, such as CPU or memory usage, to the `ResourceSimulation` component, which simulates hardware-level resource consumption. This interaction allows Slingshot to simulate how the behavior of components and different usage conditions affect overall performance. The simulation results include performance predictions such as response time and resource utilization. For example, in Fig. 3, a composite data type named `ResourceSimulationReturn` is defined, which includes an attribute `utilization`.

The components from the Repository are instantiated and their interconnections within the system are defined in the Assembly model of Palladio, as shown in Fig. 4. These `AssemblyContexts`, which contain component types, are connected through directed connectors, including delegation connectors that link the system's required or provided roles to the corresponding roles of internal `AssemblyContexts`. We show the constraints for each port in Slingshot in Fig. 4 to provide a clearer understanding of their placement.

5 An automated approach for semantic constraint checking

In this section, we first introduce the underlying structural definitions of both source and target languages, and we also provide an overview of our proposed approach.

Fig. 5 depicts the transformation and enrichment process applied by the approach to check semantic constraints in the composition of systems modeled in the PCM. PCM is based on an Eclipse Modeling Framework (EMF)-compatible Ecore metamodel (Steinberg et al., 2008), which explicitly defines its core modeling constructs, including components, interfaces, data types, and system architectures. However, MontiArc defines its modeling language using MontiCore grammars,[2] from which the abstract syntax trees are generated.

This section discusses the contributions provided in Weber et al. (2024), as well as those related to C1. The primary goal of the approach is to enrich PCM models in order to check semantic constraints for software architectures, a capability that was previously unavailable. We achieve this goal in two main steps: Transform PCM into MontiArc models (Section 5.1) and Semantic Checking of Transformed Constraints (Section 5.2).

At a high level, our approach (depicted in Fig. 5) receives a PCM model and transforms it into a MontiArc model. It is possible to annotate the PCM model with constraints, which are also transformed and enrich the MontiArc model. These constraints can then be validated on the MontiArc model using our constraint checker.

The key PCM metamodel elements (detailed in Section 2) in the Repository model are `BasicComponent`, `Interface` (with operation signatures), `ProvidedRole`, `RequiredRole`, and `CompositeDataType` (illustrated in Fig. 3); and in the Assembly model, the main elements are `System`, `AssemblyContext`, and `Connector`, which define how components are instantiated and interconnected (illustrated in Fig. 4). In MontiArc, the key language elements (detailed in Section 2) are `Component`, `Port`, and `Connector`, with connections specified using the `connect` keyword.

The proposed approach begins with the specification of components using Palladio, which facilitates early performance predictions and supports architectural decision-making during the design phase. By modeling the architecture in Palladio, the components' behavior and resource demands can be defined, enabling a simulated performance analysis of the system under varying conditions. This process is closely interconnected with MontiArc, another approach that supports the design of components and their interactions while enabling the specification of communication patterns. Although both Palladio and MontiArc approaches share a common focus on defining and managing components, their goals and applications differ. On the tool level, Palladio is primarily tailored for performance simulation and analysis, enabling architects to assess system quality attributes such as scalability and responsiveness. In contrast, MontiArc is centered on the compositional development of software architectures and provides additional capabilities for formal verification to ensure the correctness of system interactions and communication. By integrating Palladio and MontiArc,

[2] "The MontiCore Grammar Library", https://github.com/MontiCore/monticore/tree/dev/monticore-grammar (accessed: June 17, 2025).

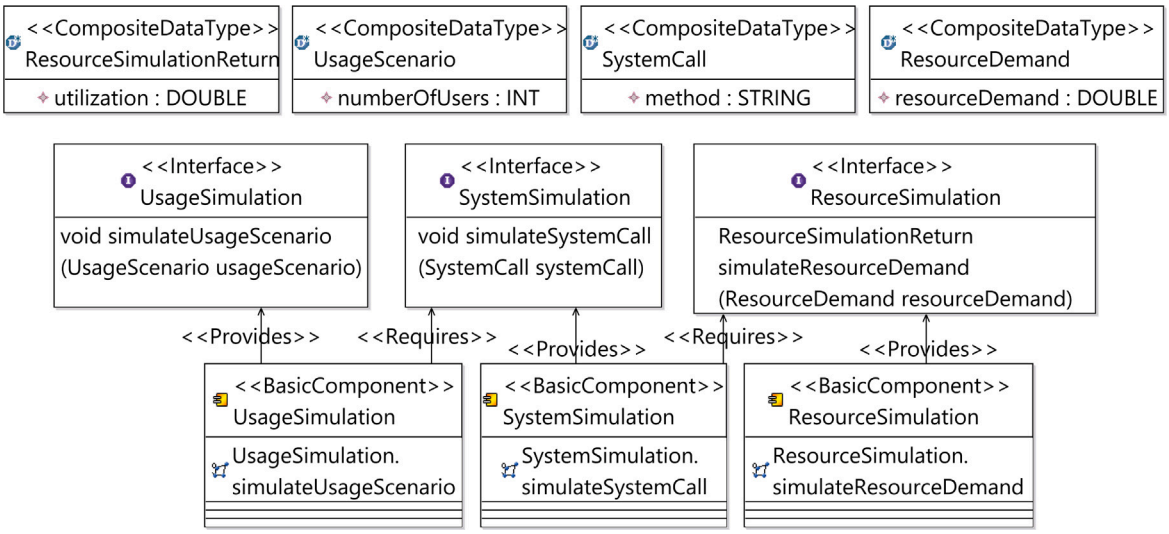


Fig. 3. Repository model of Slingshot.

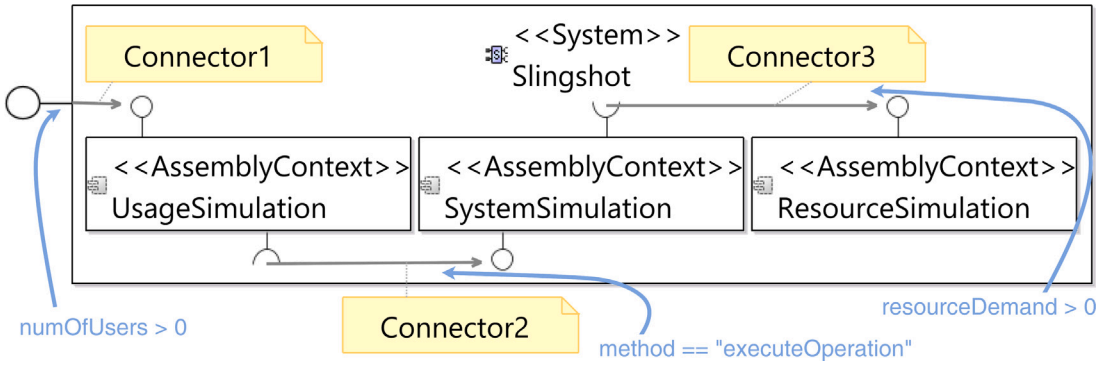


Fig. 4. Assembly model of Slingshot.

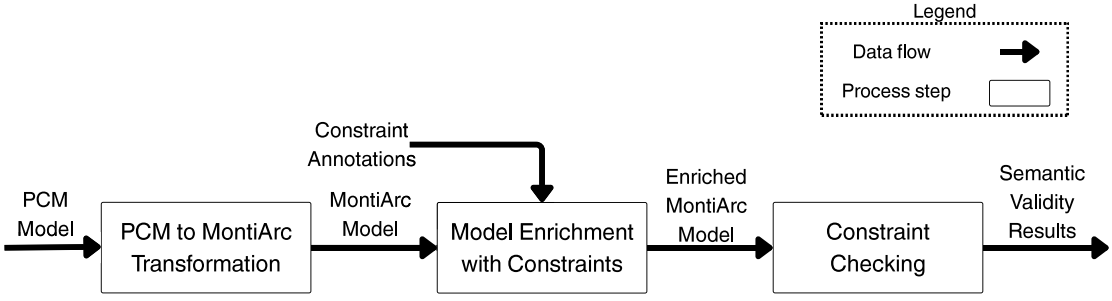


Fig. 5. Transformation process and artifacts.

the proposed approach leverages the strengths of both methods. This integration is further strengthened by allowing the specification of constraints within the Palladio approach, which can then be validated using MontiArc, ensuring semantic soundness between architectural design and composition verification.

5.1 Transforming PCM to MontiArc

Our transformation uses the Repository (Fig. 3) and the Assembly model (Fig. 4) of the PCM as inputs. The other model types of the PCM (i.e., Allocation, Deployment, and Usage) are not relevant to our approach. The Usage model is excluded because the analysis is static and therefore independent of how the system is used. Similarly, the Allocation of hardware resources and the Deployment of the software system to those resources are not relevant, as they do not affect the system's composition.

We transform PCM to MontiArc because these two frameworks serve different purposes. PCM is designed primarily for performance modeling with an emphasis on behavioral specifications and performance properties, whereas MontiArc focuses on architectural descriptions centered on hierarchical components and their interconnections. By transforming PCM models into MontiArc, we enable architectural analysis and validation using MontiArc's component-based framework, which better supports compositional reasoning and architectural constraint checking, which is a key contribution of our work.

The transformation is implemented using QVT Operational (QVTo),[3] which is a model transformation language well-suited for defining mappings between EMF-based models. QVTo was selected for its support of complex model navigation and strong integration with EMF, which aligns with the needs of our transformation logic. The purpose of this transformation is to make PCM architecture models usable within the MontiArc framework for further analysis. We use QVTo to transform PCM elements into corresponding MontiArc representations by generating valid textual MontiArc models, even though MontiArc itself is not based on EMF. Therefore, we developed an Xtext[4] grammar for MontiArc and used the Ecore model Xtext generates from this grammar as the transformation target. The grammar then allows us to save the transformed model as a valid textual MontiArc model.

The PCM input models are assumed to satisfy the following conditions to ensure the success of the transformation:

(a) Each component must have explicitly defined required and provided interfaces.
(b) Operation signatures must be expressible as data-type-based communication patterns suitable for MontiArc, typically by mapping method calls to port-based interactions.
(c) All parameter and return types used in operation signatures should be either basic types (e.g., int, boolean) or composite data types that can be represented as message types in MontiArc.

Our transformation is divided into three key steps. The first step is to extract and transform the data types from the Repository model into a MontiArc `ClassDiagram`. PCM's Repository model includes various primitive, collection-based, and composite data types, as well as interfaces that define methods with potentially multiple parameters and return types. Since MontiArc does not support method-based interfaces in the same way as PCM does, these must be transformed appropriately. Primitive data types like `int` or `char` can be used directly in MontiArc models and do not need their own definition, therefore they are transformed whenever necessary and not specifically in this step. However, composite data types (that are composed by attributes) require explicit representation in the MontiArc model. The transformation generates MontiArc `ClassDiagrams` based on collection and composite data types defined in the Repository model. Additionally, PCM interfaces with methods containing multiple parameters must be restructured, as MontiArc does not natively support method signatures. To resolve this, we aggregate these parameters of all interfaces of the Repository model into a single `CDClass`.

In addition, all interfaces of the Repository model are searched for methods with two or more parameters. These parameters will be summarized in a single `CDClass` in the MontiArc class diagram. Following this step, we have a MontiArc `ClassDiagram`, as shown in Listing 2 that contains all necessary data types for the description of the PCM model as a MontiArc model. Each PCM `CompositeDataType` in Fig. 3 is mapped to a class in MontiArc, with its attributes directly translated into typed fields. For example, the PCM data type `ResourceSimulationReturn`, which encapsulates a `utilization : DOUBLE` attribute, is translated into a MontiArc class with a corresponding field `double utilization` (line 15, Listing 2). This transformation is illustrated in Listing 3. The mapping function `compositeDataTypeToCDClass` takes a `CompositeDataType` as input and produces a `CDClass` element. It sets the class `name` based on the source entity name and applies a helper mapping to convert each `InnerDeclaration` (which represents an attribute declaration within a `CompositeDataType`) into a class member.

[3] "OMG QVT 1.3 Specification", https://www.omg.org/spec/QVT/1.3/ (accessed: June 17, 2025).
[4] "The Xtext Framework", https://eclipse.dev/Xtext/ (accessed: June 17, 2025).

```
1  classdiagram Slingshot {
2      public class UsageScenario {
3          int numberOfUsers;
4      }
5
6      public class SystemCall {
7          String method;
8      }
9
10     public class ResourceDemand {
11         double resourceDemand;
12     }
13
14     public class ResourceSimulationReturn {
15         double utilization;
16     }
17 }
```

Listing 2: Class diagram based on the repository model in Fig. 3.

```
1  mapping CompositeDataType::
       compositeDataTypeToCDClass() : CDClass {
2      name := self.entityName;
3      public := true;
4      members += self.
           innerDeclaration_CompositeDataType->
           map innerDeclarationToMember();
5  }
```

Listing 3: QVTo mapping for transforming CompositeDataType to CDClass.

In the next step, the PCM Assembly model from Fig. 4 is transformed. The output of this transformation step is shown in Listing 4 (see Appendix for the complete listing). The constraints are shown in gray because they are added in the third step of the transformation. The first challenge in this stage is transforming the system, which is the root element of an Assembly model, to the core component of the MontiArc model. Next, the `AssemblyContexts`, which represent instances of components specified in the repository (Fig. 3), are transformed into sub-components in MontiArc. A key complexity here is that PCM `AssemblyContexts` encapsulate components defined in the Repository, and their roles (i.e., required and provided interfaces) need to be correctly translated into MontiArc ports. For example, consider the `UsageSimulation` component. It provides the `UsageSimulation` interface and requires the `SystemSimulation` interface. In PCM, both are modeled as roles in the Assembly model, which are shown through the circle at the top for the provided role and the half-circle at the bottom for the required role (Fig. 4). Through the provided role the component receives a `UsageScenario` as input and through the required role the component outputs a `SystemCall`. Therefore, the component has one input port for `UsageScenario` while the port for `SystemCall` is an output port. The last part of this step is the transformation of PCM connectors, which define how assembly contexts communicate according to the roles of the encapsulated components, to MontiArc connectors.

```
1  component Slingshot {
2    port <<condition = "x.numberOfUsers > 0 &&
         x.numberOfUsers < 2147483647">> in
       UsageScenario usageScenario;
3    component UsageSimulation {
4      port <<condition = "x.numberOfUsers > 0"
         >> in UsageScenario usageScenario;
5      port <<delayed, condition = "x.method
         == \"executeOperation\"">> out
         SystemCall systemCall;
```

```
  }
  component SystemSimulation {
    port <<condition = "x.method == \"executeOperation\""
        >> in SystemCall systemCall;
    port <<condition = "x.resourceDemand > 1.0" >> out
        ResourceDemand resourceDemand;
    port in ResourceSimulationReturn
        resourceSimulationReturn;
  }
  component ResourceSimulation {
    port out ResourceSimulationReturn
        resourceSimulationReturn;
    port <<condition = "x.resourceDemand > 1.0" >> in
        ResourceDemand resourceDemand;
  }
  UsageSimulation usageSimulation;
  SystemSimulation systemSimulation;
  ResourceSimulation resourceSimulation;
  usageSimulation.systemCall ->
      systemSimulation.systemCall;
  resourceSimulation.
      resourceSimulationReturn ->
      systemSimulation.
      resourceSimulationReturn;
  systemSimulation.resourceDemand ->
      resourceSimulation.resourceDemand;
  usageScenario -> usageSimulation.
      usageScenario;
}
```

Listing 4: MontiArc model based on the repository (Fig. 3) and system (Fig. 4) model.

In the final stage, the generated MontiArc model is enriched with constraints that were specified as Ecore annotations in the additional input model. Listing 5 illustrates the result of the transformation, showing how constraints are integrated into the MontiArc model. Each annotation refers to a directed connector in the PCM Assembly model and contains different key value pairs that define specific conditions for ports. Taking Connector2 as an example, it connects the `UsageSimulation` and the `SystemSimulation`. The constraints with the keys containing `Source` are applied to `UsageSimulation` and the keys containing `target` are applied to `SystemSimulation`. The constraint `x.method == "executeOperation"` is applied to the output port of the `UsageSimulation` corresponding to Connector2 and to the input port of the `SystemSimulation` corresponding to Connector2.

```
<eAnnotations references="Slingshot.system
    #Connector1">
  <details key="Source:In" value="x.
      numberOfUsers > 0 &amp;amp;&amp;amp; x.
      numberOfUsers &lt; 2147483647"/>
  <details key="Target:In" value="x.
      numberOfUsers > 0"/>
</eAnnotations>
<eAnnotations references="Slingshot.system
    #Connector2">
  <details key="Source:Out" value="x.
      method == &quot;executeOperation&
      quot;"/>
  <details key="Target:In" value="x.method
      == &quot;executeOperation&quot;"/>
</eAnnotations>
<eAnnotations references="Slingshot.system
    #Connector3">
  <details key="Source:Out" value="x.
      resourceDemand > 0.0"/>
  <details key="Target:In" value="x.
      resourceDemand > 0.0"/>
</eAnnotations>
```

Listing 5: Ecore model of the constraints.

Fig. 6 gives an overview of the model elements involved from both Palladio and MontiArc and the complex relationships the transformation has to take into account. Due to the different underlying concepts of both modeling approaches, elements from MontiArc might be generated based on one or multiple different metamodel elements from Palladio. Enriching already generated elements based on information from other metamodel elements is also necessary for some elements, e.g., when a constraint from an annotation is applied to a port. The right side of the figure shows the model elements in Palladio, while the left side displays the elements in MontiArc. For example, a component from the Repository in Palladio, as well as an `AssemblyContext` from the system, are both transformed into a Component element in MontiArc. The mapping from an `AssemblyContext` in Palladio to a `Component` in MontiArc starts by assigning the component's name based on the `AssemblyContext`'s `entityName`. It then iterates over the provided and required roles of the encapsulated component and transforms them into MontiArc ports. These ports are then added to the resulting component. This process ensures that the interface structure from Palladio is preserved and represented appropriately in the MontiArc model. The QVTo transformation code implementing this mapping is provided in Appendix.

5.2 Semantic checking of transformed constraints

We present a solution to automatically ensure the semantic soundness of PCM models by performing constraint checking after transforming them into MontiArc. The core idea is to verify that a modeled system behaves as intended by checking whether its architectural constraints are satisfied. An analysis component comes with its assumptions about its inputs and outputs. Thus, every component can guarantee to act as intended – through its output guarantees – only if the aforementioned assumptions are satisfied. By proving that the required assumption is satisfied by the output guarantee of the connected component, we can conclude that the connection is also semantically valid. To exemplify this on our very simple running example from Listing 4, the output constraint of the `UsageSimulation` on port `systemCall` is guaranteed then the assumption on the input of the `SystemSimulation` also has to be satisfied for the connection to be valid.

To achieve this, constraints are embedded as stereotypes over ports of C&C architectural components. This employs a reduced form of the assumption-guarantee formalism (Broy and Stølen, 2001) that is easily automatable. The assertion paradigm is similar to the Eiffel programming language (Meyer, 1990) but employed at the architectural level. The restriction in question is that the constraints are not expressed through formulas handling potentially infinite port flow histories but only instant values present at ports. This approach uses static model analysis and restricts the types available at ports performing dependent typing similar to Swamy et al. (2016). The validation process consists of the following steps: (1) Parsing the generated MontiArc model and processing constraint pairs defined with the `condition` stereotype iterating over their connections. Processing constraints starts by parsing the constraints into the MontiCore expression framework (Hölldobler

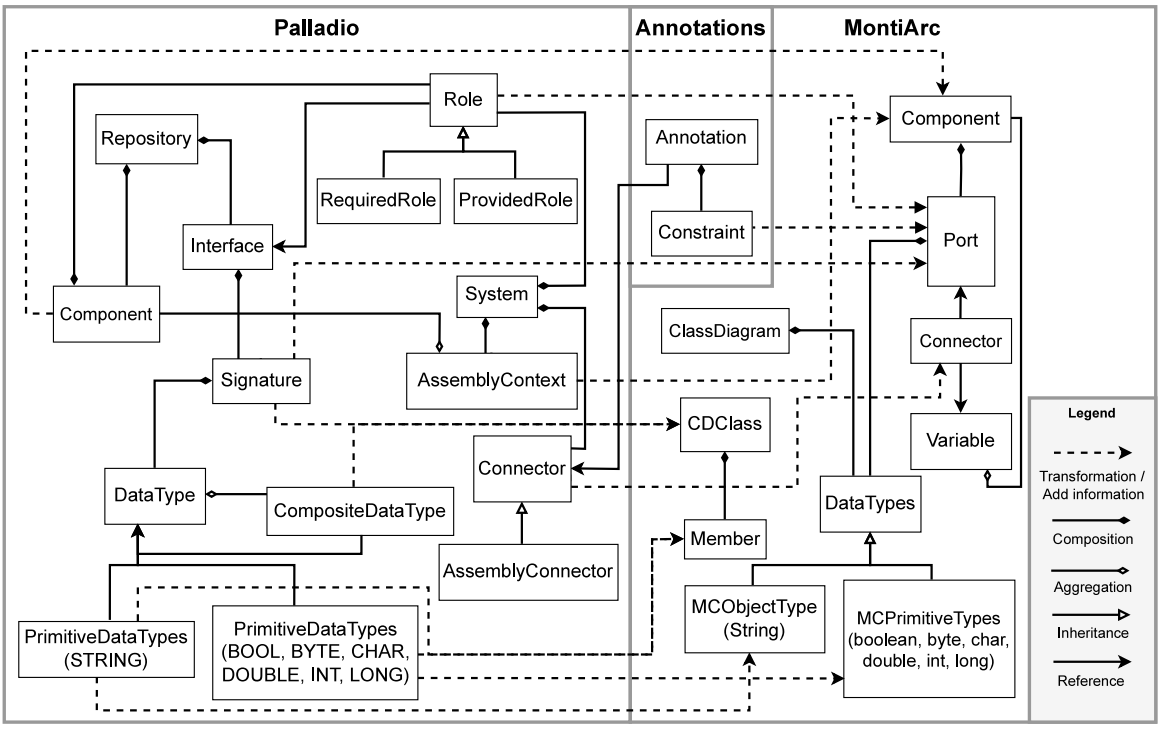


Fig. 6. Mapping of transformed elements from Palladio to MontiArc.

et al., 2021). (2) Translating constraints into SMT formulas performed in concordance to Rumpe et al. (2024) . (3) Employing the Z3 SMT solver to check if the translated implication over the constraints is valid.

We employ SMT checking over the chosen OCL subset to automatically prove that the connections between two components are potent enough to guarantee an implication. As explained by Rumpe et al. (2024), OCL can be directly mapped to SMT with minimal limitations due to their large overlap. Additionally, SMT solvers offer direct regular expression support. Our implementation reuses an OCL to SMT translator developed by Rumpe et al. (2024) restricted on the set of MontiCore OCL expressions. The specific OCL variant supported by the translator is described in Rumpe (2016). Our restriction of the OCL language is introduced to ensure that the generated constraints remain simple, decidable, and efficiently solvable by standard SMT solvers. In particular, our approach only handles quantifier-free constraints. Furthermore, when complex objects are transmitted on ports, their datatypes must be represented and translated into SMT, a process also devised by Rumpe et al. (2024). Our approach uses implementation-focused class diagrams and objects diagrams as presented in Rumpe (2016). For constraint validation, we use Microsoft's Z3 SMT solver (De Moura and Bjørner, 2008) to check the SMT-Lib formulas derived from our transformation (Barrett et al., 2010). SMT-lib and Z3 were chosen as a input/output language and solver pair because of its standardized use when checking constraints with SMT when it comes to the former and its wide usage in different applications for the latter (Bjørner and Jayaraman, 2015; Bjørner, 2018).

To formulate checkable constraints, we build implications over the assumptions and the guarantee obligations. Since port datatypes are instances of classes, we introduce an SMT sort for each class. Listing 6 presents the declaration of the `UsageScenario` sort and corresponding function `numberOfUsers`, which represents its attribute. This follows the class diagram generated for the PCM model in Fig. 3. The return type of the function is represented by the SMT-Lib built-in `Int` sort.

```
(declare-sort UsageScenario)
(declare-fun numberOfUsers (UsageScenario) (
    Int))
(declare-fun x () UsageScenario)
(assert (=> (and (< (numberOfUsers x)
    2147483647")
                (> (numberOfUsers x) 0))
            (> (numberOfUsers x) 0)))
```

Listing 6: Translated Constraints in SMT-Lib.

The translation of the constraints takes place by introducing a variable `x` and declaring it in the respective sort of the port's type, i.e., the sort generated for the class, here `UsageScenario`. The implication and logical operators used in constraints are then 1-to-1 mapped to SMT-Lib. The result of the checker is then constructed with a classic approach, i.e., to check validity we negate the implication and check for unsatisfiability. The result is then produced to the standard output with a statement about the validity of the implication along with references to the affected ports. Listing 7 presents an output of the constraint checker for the valid Slingshot model, referencing the connected ports in question. If the negated implication is satisfiable, then the implication is not valid. Listing 8 presents the output of the constraint checker if we modify the target constraint in Listing 4 to require a positive number of users. In this case, the tool detects the error but still processes all implications before halting. The tool presents a counter-example in the form of an object diagram, performing a retranslation of the SMT-Lib model into an object-oriented representation. This helps identify violations and incorrect constraints easily.

Formulating constraints using only arithmetic and string operations can be tedious. This limits the expressiveness of constraints to only pre-defined string operations combined with boolean logic between constraints. In particular, while writing string constraints the user also has to possess knowledge of string standard method library signatures

Table 1
Structure of the use-cases for the evaluation.

Change / Expression	Modify					
	Output guarantee			Input assumption		
	Refine	Refactor	Abstract	Refine	Refactor	Abstract
Boolean				6.3.2		
Regex-Kleene						6.4.4
Regex-MinMax	6.4.2			6.4.3		6.4.4

```
Constraint of Port usageScenario in
    Component Slingshot guarantees
    constraint of Port usageScenario in
    Component UsageSimulation
```

Listing 7: Output of the Constraint Checker for the Constraint-Enriched model.

```
[ERROR] Found error in port-constraint.
    Constraint of Port usageScenario in
    Component UsageSimulation does not
    follow from constraint of usageScenario
    in Component Slingshot.
Counterexample: objectdiagram x {
  usageScenario_0:UsageScenario {
    int numberOfUsers=1;
  };
}
```

Listing 8: Output of the Constraint Checker for a negative result in the model.

and has to understand their usage. This defers addressing the issue to the tooling provider, requiring code completion support. To better motivate the need for regular expression support, we revisit the running example from Listing 4. The `method` property for a `systemCall` can be constrained through a naming scheme only through a complex chain of string method calls. A regular expression simplifies this to a more concise regular expression membership operation. Thus, we introduced the option to use regular expressions to specify string constraints, as promised in C1. Even though regular expression dialects also require prior knowledge, these are standardized and in conformance to the Java specification.[5] To satisfy C1 we introduce a minimal language that extends the OCL with the regular expressions[6] framework provided with MontiCore. There are works that extend the OCL with regular expressions support. Lano (2021) provides a proposal for standardization of regular expressions in future OCL releases. Eclipse OCL also provides an implementation of regular expressions (Damus and Sánchez-Barbudo, 2002). These extensions, however, are very string-centric and complex in their matching capabilities, thus requiring prior knowledge of each extension standard library method signature inside the expressions to use. As these are not standardized in the OCL itself, we chose to minimally implement a very simple formalism where the regular expression can only express the language generated by the regular expression. Listing 9 presents the implementation of a language embedding operation through a MontiCore grammar with

```
grammar OCLwithRegEx extends
    de.monticore.ocl.OCLExpressions,
    de.monticore.ocl.SetExpressions,
    de.monticore.ocl.OptionalOperators,
    de.monticore.expressions.BitExpressions,
    de.monticore.expressions.
        ExpressionsBasis,
    de.monticore.regex.RegularExpressions {
  start Expression;

  @Override
  RegExLiteral implements Expression;
}
```

Listing 9: MontiCore Grammar that performs Language Embedding of regular expressions into OCL Expressions.

the goal of allowing regular expressions to be used in OCL expressions by providing an implementation of the hook-point non-terminal for expressions through regular expression literals, which can then be used in an OCL set expression formalism.

To provide a transformation from OCL with regular expressions to SMT-Lib, we only have to implement the transformation for regular expression elements, as MontiCore's modular expressions framework allows us to reuse the formerly presented transformation for OCL expressions as presented in Rumpe et al. (2024). Additionally, Z3 and SMT-Lib provide support for regular expressions natively. We formalize the bridging point by defining the language defined by a regular expression instance as a set. Thus, we map the aforementioned language as an SMT-Lib set and define its use through membership relations. We illustrate this for our running example, Listing 4. `x.method isin R"executeOperation"` is an adapted constraint using regular expressions, which expresses the constraint obligation that the property `method` of the variable `x` is in the set defined by the language of the regular expression `executeOperation`.

6 Evaluation

In this section, we address C3 by evaluating our approach, validating different constraints across a range of scenarios. The evaluation is designed to assess the quality of our approach in handling semantic constraints under varying conditions and identify any challenges or differences observed during the validation process. To conduct the evaluation, we design multiple scenarios inspired by our original case studies and add an additional Artifact Model Analysis case study to address C3 with respect to regular expression support. These scenarios simulate diverse contexts and constraints to assess our approach.

We outline our evaluation goals and metrics in Section 6.1, followed by a description of the evaluation design in Section 6.2. The first case study is discussed in Section 6.3, and the second case study is detailed in Section 6.4. In Section 6.5, we present the results of our evaluation. Threats to validity are discussed in Section 6.6, and limitations are addressed in Section 6.7. Finally, information on the availability of evaluation data is provided in the 'Data Availability' section.

[5] "The Java Regular Expression Language Specification", https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/regex/Pattern.html (accessed: January 23, 2025).

[6] "The MontiCore Regular Expression Library", https://github.com/MontiCore/monticore/blob/dev/monticore-grammar/src/main/grammars/de/monticore/regex/RegularExpressions.mc4 (accessed: January 23, 2025).

6.1 Evaluation goals and metrics

This evaluation aims to systematically evaluate our proposed approach across different scenarios for validating semantic constraints. To structure our evaluation, we employ the Goal-Question-Metric (GQM) approach (Caldiera and Rombach, 1994; Basili and Weiss, 1984). We aim to achieve the following objectives through examination of different configurations, contexts, and constraints. Scenarios are designed in accordance with the following goals.

- G1 Evaluate the accuracy of the approach in validating semantic constraints.
- G2 Evaluate the performance of the approach as the complexity of semantic constraints increases.

We evaluate the accuracy of our approach because it is a crucial factor in ensuring the correctness of component compositions during system design. If the results of the constraint checking do not accurately reflect whether a composition satisfies the intended constraints, the approach would be unreliable in practice. To calculate the accuracy of constraint checking, it is essential to systematically explore scenarios in which the composition may change. The methodology employed for systematically exploring scenarios is described in Section 6.2. Additionally, we evaluate the performance of the approach by analyzing the runtime of the constraint checker in terms of increasing constraint complexity. Our approach is a static analysis taking place at design time. Thus, the threshold where the performance of our tool is important is very permissive. But as SMT solving is generally a NP-hard problem and the general OCL satisfiability problem is undecidable (Franconi et al., 2019), the effectiveness of our tool could be reduced to very limited examples, if the constraints are complex enough. This assessment is important because the constraint checker plays a central role in our validation process. It is responsible for identifying counter-examples and validating the composition, which can be computationally expensive, along with potentially being the main bottleneck of our approach. As the complexity of constraints increases, it is necessary to understand how the runtime grows to ensure that the approach remains feasible and efficient for larger and more complex constraints.

To evaluate our goals, we aim to answer the following questions:

- Q1 How accurate is the approach in detecting violations and semantic constraint checking?
- Q2 How is the performance of the approach impacted as the complexity of the constraints increases?

To answer the question Q1, we employed the well-known metrics suggested by Metz (1978) to evaluate accuracy. The True Positive Fraction (TPF) is defined as $TPF = t_p/P$, where t_p represents the number of scenarios with constraint violations that were correctly identified, and P is the total number of actual constraint violation scenarios. The True Negative Fraction (TNF) is defined as $TNF = t_n/N$ where t_n is the number of identified scenarios without constraint violations, and N is the total number of actual scenarios without constraint violations. With these two metrics, we can calculate the accuracy of our approach for scenarios involving expected violations and those that do not.

In addition to these metrics, we also calculate the F1-score (Manning, 2009), which is a widely used metric that combines Precision and Recall. Precision is the proportion of predicted positives that are actually correct and is defined as $Precision = t_p/t_p + f_p$ where f_p refers to cases incorrectly identified as positive. Recall is the proportion of actual positives that are correctly identified, which is the same as the TPF. This is especially important because a high TPF or TNF alone may not fully represent the overall accuracy of the validation process. The F1-score is defined as $F_1 = 2 \cdot (\text{Precision} \cdot \text{Recall})/(\text{Precision} + \text{Recall})$.

To answer question Q2, we evaluate the execution time of the operations in the MontiArc constraint checker to assess the performance of our approach. This evaluation helps us understand how the system behaves with increasing constraint complexity, providing insights into its efficiency and computational cost.

6.2 Evaluation design

We aim to systematically evaluate our approach using case studies and defining scenarios. Case studies are especially useful for understanding complex phenomena in their real-life context (Runeson and Höst, 2009). Through these case studies, we are able to demonstrate the applicability of our approach under different conditions and its ability to address practical issues. On the basis of expert knowledge relevant to the case studies, we derive one relevant scenario for each case study, which then serves as the basis for generating a set of variations by altering relevant architectural or constraint-related parameters. For each original scenario, we systematically introduce variations by modifying, removing, or extending specific components of the case studies. Each component comes with its own constraints, thus any operation that changes the architecture also changes the constraints. Components may be modified to assess adaptability, removed to analyze the impact of their absence, or added to introduce new functionality and evaluate the model's ability to accommodate enhancements. These variations allow us to explore the flexibility of our approach across a range of scenarios. In particular, these evaluations help us explore how our approach handles replacements and restructurings of individual components without affecting the overall validity of the system. To do this, we start with a composition that is already valid, apply various substitutions and restructurings that deliver valid or invalid compositions, and then check whether the composition still holds up using our approach. We provide an expected result for each operation and compare the output of the tool with it to assess accuracy. Furthermore, by varying the constraint types, we ensure that our transformation is not limited to a single model or a fixed set of constraints but can be generalized across diverse situations.

In the following, we present two case studies from different domains along with their corresponding scenarios. For each scenario, we describe the relevant constraints that must be satisfied. We classify the results of constraint checking into the following categories in order to calculate accuracy metrics. A t_p (True Positive) is a case where the approach correctly identifies a constraint violation in a connection between ports that is indeed invalid. A t_n (True Negative) is a case where the approach successfully recognizes that all constraints are satisfied in a valid port connection, i.e., no constraints are violated. Based on these classifications, we calculate the TPF for all scenarios with expected violations and the TNF for all scenarios without any violations. Moreover, the F1-score is a single metric that shows how well a model correctly identifies positive cases while minimizing false positives and false negatives. A higher F1-score indicates better overall performance in identifying relevant results accurately.

To address Q2, we performed execution time performance testing for the newly added regular expression support (C1), as these are the only computationally hard problems while performing constraint solving over the set of transformed OCL constraints with the restrictions described in 5.2. Arithmetic constraints are also briefly touched upon in order to demonstrate their viability. We deemed the number of constraints not relevant for this particular evaluation, as interval constraint solving scales linearly with the number of constraints. Elaborating on the method for Q2, we decided to evaluate the time complexity of the operations of the MontiArc constraint checker, starting with regular expressions. We evaluate here how the runtime is influenced if the constraint checker requires to check valid connections in comparison to invalid connections where a counter-example is found. The performance testing was conducted on a machine running Windows 10 22H2 19045.5737, with a 13th Gen Intel(R) Core(TM) i7-1365U 1.80 GHz, having 32 GB LPDDR5 RAM 4800 MHz.

6.2.1 Introduction to case studies

Before explaining each case study in detail, we first examine the changes presented in Table 1. For brevity only selected changes are discussed in details. The full evaluation with all models can be found here.[7] This provides an overview of how scenarios are represented in the context of modifications. Replacement cases are particularly interesting, especially when evaluating regular expression constraints, as they better reflect the accuracy of our approach when constraints are modified like for like. For semantic soundness through constraint checking, replacing a component cannot be done without replacing the assumptions and promises associated with it. We can validate the modification if the approach finds the same result as the expected result.

Every column in Table 1 represents a modification relationship between a component and its replacement in relation to the datatypes used in the scenario. Every replacement scenario stands in relation with the original scenario, which can systematically be disseminated to the displayed operations. The crucial point of this distribution is that every replacing component is modeled only as a black box and without behavior. Thus, we have to focus on the constraints delivered with this black box and reason if the connections and implicit composition it is part of remain valid throughout replacement.

6.3 First case study: Slingshot

For our first case study, we examine Slingshot, a simulation framework previously introduced in Section 4. Slingshot is an event-driven simulator designed to be extensible and capable of simulating Palladio models. Its architecture facilitates communication between components through publishing and responding to events.

The Slingshot simulator consists of three analysis components capable of simulating the performance of models created using PCM. To illustrate its functionality, we model a simplified version of these components and their interactions as a core scenario, and then derive variations by modifying architectural aspects such as adding, removing, and modifying components. In addition, we define several semantic constraints on their incoming and outgoing ports, based on expert knowledge specific to this case study.

6.3.1 Original scenario – Main components with output guarantee

Datatype String & int

Case study & relevance Slingshot is designed to evaluate the performance behavior of the system under varying usage conditions. As shown in Fig. 7, the `usageScenario` serves as the input port to the `UsageSimulation` component. Each usage scenario models individual use cases of the system and includes a workload that describes usage intensity. The parameter `numberOfUsers` in this workload represents the population, which is meaningful only if it is greater than zero. This is enforced by a constraint that differentiates between closed and open workloads: if `closedWorkload` is true, then `numberOfUsers` must be greater than 0 and less than 2,147,483,647. Otherwise, for open workloads, the interval between requests must be greater than 0.0 (line 1, Listing 10). The `userRequest`, as the output of `UsageSimulation`, contains a `systemCall`, which is a string representing the name of the invoked method. This string must remain unchanged when it reaches the input port of `SystemSimulation`. Similarly, we define a constraint on the input to the `ResourceSimulation` component, which reflects the demand for resources simulated by the `SystemSimulation` component. Each component in the system produces at least one output, and constraints are also applied to these outputs. The outputs represent the simulation results, or performance predictions, which include `responseTime` and `resourceUtilization`. These results are considered valid only if their values are greater than or equal to zero. Any value below zero would violate the constraints and render the predictions invalid. In particular, the constraint on port `utilization` ensures that if both `totalTime` and `busyTime` are greater than zero, then `utilization` must be equal to `busyTime` divided by `totalTime`, and the resulting value must also be greater than zero. Listing 10 shows how these constraints are defined, respectively.

```
1  if x.closedWorkload == true then (x.
       numberOfUsers > 0 && x.numberOfUsers <
       2147483647) else x.interval > 0.0
2  ------------------------------------
3  x.method == " executeOperation "
4  ------------------------------------
5  x.resourceDemand > 0.0
6  ------------------------------------
7  (x.totalTime > 0 && x.busyTime > 0)
8    implies
9  (x.utilization == x.busyTime / x.totalTime
       && x.utilization > 0)
10 ------------------------------------
11 x.responseTime > 0.0
```

Listing 10: Constraint on ports UsageScenario, UserRequest, ResourceDemand, Utilization, and ResponseTime, respectively.

Result Listing 11 illustrates the output of our constraint checker tool applied to the constraints presented in Listing 10 and Fig. 7.

```
1 Constraint of Port usageScenario in
      Component Slingshot guarantees
      constraint of Port usageScenario in
      Component UsageSimulation
2 Constraint of Port resourceDemand in
      Component SystemSimulation guarantees
      constraint of Port resourceDemand in
      Component ResourceSimulation
3 Constraint of Port resourceSimulationReturn
      in Component ResourceSimulation
      guarantees constraint of Port
      resourceSimulationReturn in Component
      SystemSimulation
4 Constraint of Port usageSimulationReturn in
      Component UsageSimulation guarantees
      constraint of Port usageSimulationReturn
       in Component Slingshot
5 Models processed successfully!
```

Listing 11: Output of the constraint checker tool applied to the constraints presented in Fig. 7.

6.3.2 Modify ResourceSimulation component scenario – Separation of active and passive resources

Datatype String & int & boolean

Case study & relevance In the second scenario, we aim to examine the impact of modifying components on the validation of constraints. Specifically, we focus on decomposing the functionality of the `ResourceSimulation` component. Since there are two types of resources – active resources, such as CPUs and I/O devices, and passive resources, such as semaphores and threads – we divide the `ResourceSimulation` component into two distinct functions. One function is responsible for managing the demand for active resources, while the other handles the demand for passive resources, if applicable. As illustrated in Fig. 8, constraints are applied to the inputs of these

[7] https://github.com/FeCoMASS/Model-Transformation-for-Automated-Constraint-Validation

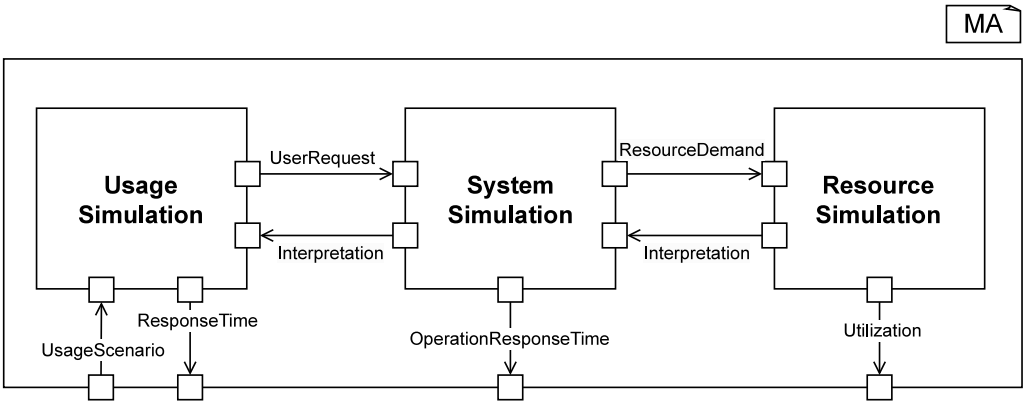


Fig. 7. Case Study — Slingshot: The original scenario with three analysis components.

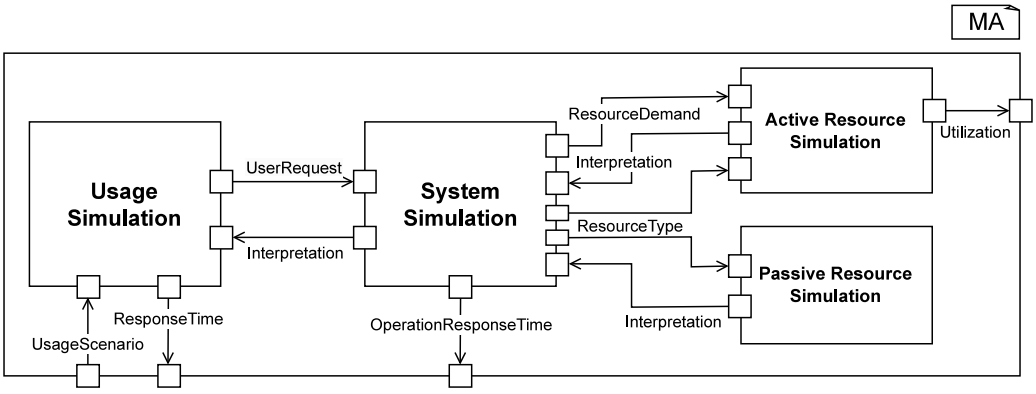


Fig. 8. Modification of the ResourceSimulation component by separating resources based on type.

separated components to ensure they correctly correspond to their respective resource types. This separation not only ensures proper validation of constraints for each resource category but also allows for more fine-grained analysis of how the system handles different types of resource demands. Listing 12 shows the constraint on the output of `SystemSimulation` on the `ResourceType` port and input to `ActiveResourceSimulation`, and the constraint on the output ports of `ResourceType` and input to `PassiveResourceSimulation`.

```
x.active == true && x.passive == false
x.active == true
x.active == false && x.passive == true
x.passive == true
```

Listing 12: Constraints on ports ResourceType.

Result Listing 13 illustrates the result of our tool applied to Fig. 8 and Listing 12.

```
Constraint of Port active in Component
    SystemSimulation guarantees constraint
    of Port active in Component
    ResourceSimulation
Constraint of Port passive in Component
    SystemSimulation guarantees constraint
    of Port passive in Component
    PassiveResourceSimulation
Models processed successfully!
```

Listing 13: Output of the Constraint Checker in Fig. 8.

6.3.3 Add NetworkSimulation component scenario — Add constraints for ResourceSimulation
Datatype String & int

Case study & relevance This scenario is inspired by Henss et al. (2013) to predict the performance of network-intensive distributed systems. The core idea of this work is the integration of Palladio with a network simulation tool to analyze the overall system performance. This composition enables the prediction of end-to-end response times in scenarios where network communication plays a critical role, without requiring explicit hardware-level simulations.

The network simulation is closely related to `ResourceSimulation`, as it incorporates critical factors such as network latency and throughput. By modeling these elements, the framework effectively captures the impact of network communication on system behavior. This integration provides a more comprehensive understanding of system performance by reflecting the dynamic interplay between computational resources and network constraints. Fig. 9 shows how this new component interacts with the `ResourceSimulation` component.

```
x.serverLoad > 0.0
x.latency > 0.0
```

Listing 14: New constraints for NetworkSimulation component.

Result Listing 15 shows the result of the constraint checker for the new constraints in Listing 14.

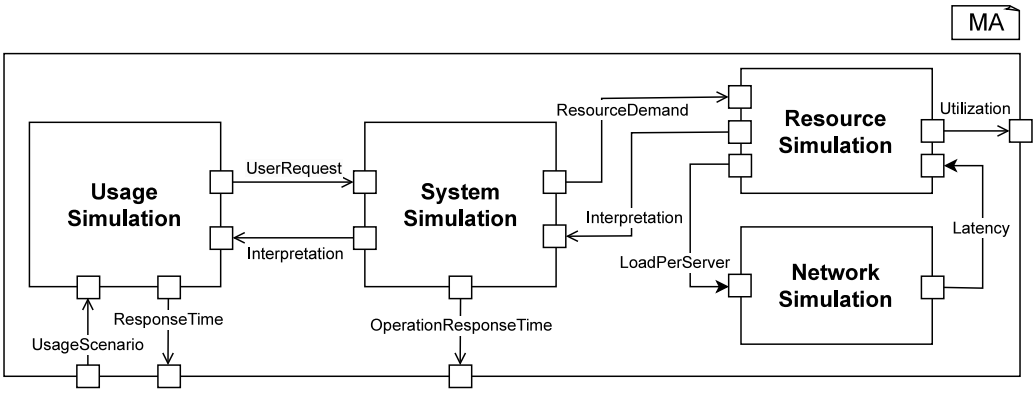


Fig. 9. Addition of the NetworkSimulation component to the original scenario.

```
1 Constraint of Port serverLoad in Component
      ResourceSimulation guarantees
      constraint of Port serverLoad in
      Component NetworkSimulation
2 Constraint of Port latency in Component
      NetworkSimulation guarantees constraint
       of Port latency in Component
      ResourceSimulation
3 Models processed successfully!
```

Listing 15: Output of the Constraint Checker in Fig. 9 for the new constraints.

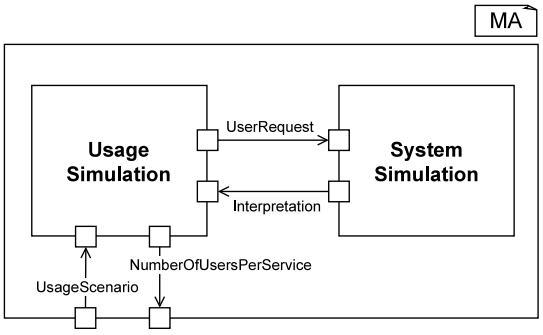


Fig. 10. Removal of the ResourceSimulation component.

6.3.4 Remove ResourceSimulation component scenario — Removal of related constraints

Datatype String & int

Case study & relevance In another scenario where the `ResourceSimulation` component is removed, the focus shifts to a simplified modeling approach involving only `UsageSimulation` and `SystemSimulation`. Without `ResourceSimulation`, the simulation no longer includes explicit hardware resource modeling such as CPU, memory, or disk usage. Despite the lack of direct resource modeling, it is still possible to calculate the total load on each service based on the user's behavior and the configuration of the internal system. This new configuration is depicted in Fig. 10. The new port must never have a value less than zero.

Result As shown in Listing 16, it is the output of the constraint checker for the constraint on the port `NumberOfUsersPerService`, which is required to be defined as the return value of the `UsageSimulation` component.

```
1 Constraint of Port usageSimulationReturn in
      Component UsageSimulation guarantees
      constraint of Port usageSimulationReturn
       in Component Slingshot
2 Models processed successfully!
```

Listing 16: Output of the Constraint Checker in Fig. 10 for the new constraint.

6.4 Second case study: Artifact model analysis

Artifact analysis is crucial in complex software development projects as it transforms implicit knowledge into an explicit understanding of the intricate relationships between artifacts, tools, and processes. The artifact tooling from this case-study is published in Greifenberg et al. (2020), Hillemacher et al. (2021) and Butting et al. (2018, 2017). Greifenberg et al. (2020) developed an artifact model which can be used as a basis for any artifact-based analysis, such as a simple check for double names in all software artifacts, and can be extended for more complex analyses, such as dependency analysis. Such analyses based on the artifact model contain in general a high degree of modularity, allowing us to evaluate the semantic soundness of the composition employing our approach. An extended overview of artifact analysis is given in this link.[8]

The case study presented in this section shows a simplified model of this artifact analysis. The evaluation consists of modeling an artifact analysis pipeline in its decomposed form using MontiArc, deriving semantic constraints from the appropriate literature and validating the composition using our approach. We contribute to this evaluation by demonstrating valid and invalid operations on analysis components shipped with their own constraints and comparing expected outputs with actual outputs of our approach.

6.4.1 Original scenario

Datatype Length-bounded regular expression & string

Case study & relevance This scenario presents the architecture model of the system in an underspecified form. This is then used as a baseline for architectural modifications to the system which then have to be validated using our approach. Fig. 11 displays a decomposed view of a software artifact analysis system. Artifacts in raw form flow into the system under analysis and are fed into extractor components. Each extractor displayed filters out its irrelevant artifacts. Functionally, an extractor component gathers data from the artifact's content relevant for an analysis and stores said data into a data structure. The processing

[8] https://www.se-rwth.de/research/Artifacts/.

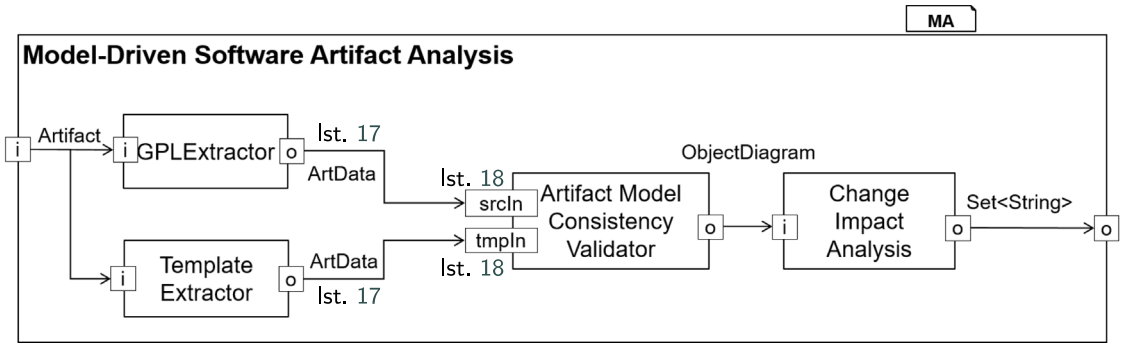


Fig. 11. Case Study — Artifact Model: Artifacts are processed by extractor components and validated for their consistency to the artifact model. The valid object diagram is then fed into a change impact analysis component. Conditions required for functional validity are expressed through stereotypes on ports.

of an artifact takes place in a sequential run with each `ArtData` representing a single object. These then have to be checked by an artifact model validator, which cumulates and merges all data it receives into an object diagram, and finds inconsistencies regarding conformance to the artifact model. The explicit analysis component evaluates this merged object diagram and produces the analysis results.

We chose this scenario as representative, because this displays a generally usable contract that is trivially checkable due to bounded regular expressions and does not require deep insight into the functionality of extractors, allowing for interpretation from an architectural viewpoint. This can only serve as a weak condition, thus covering additional cases through refining this scenario further by abstracting or refining the components and its provided constraints is needed.

```
x.id == "@" + x.simpleName + "!" + x.
    nameExtension &&
      x.simpleName isin R"[a-zA-Z]{1,20}" &&
      x.nameExtension isin R"[a-zA-Z]{1,20}"
```

Listing 17: Constraint on the GPLExtractor's Port o. Every object diagram artifact has an id and consists of the concatenation of the filename and the extesion.

```
 x isin R"@[a-zA-Z_0-9]{1,20}![a-zA-Z_$
    ]{1,20}"
```

Listing 18: Constraint on Validator's input Ports `srcIn` and `tmpIn`. A validator requires the identifier to have a simple alphanumeric format encompassed by two special characters.

The correctness of the composition is ensured by contracts over its components which the constraints here are based on. An extractor derives identificators for each processed artifact, based on its name, the file extension and two specific encoding characters `@ !`. Listing 17 displays the simple form of an ID, which is guaranteed by extractors. The Artifact Model validator requires that all IDs contain the encoding characters accompanied by alphanumerics. This assumption is displayed in Listing 18.

Result Listing 19 illustrates the output of our tool matching expected output for this trivial case, where the character set in the promise is a subset of that of the condition.

```
Constraint of Port o in Component
    GPLExtractor guarantees constraint of
    Port srcIn in Component Validator
```

Listing 19: Output of the Constraint Checker for both Extractor to Validator connections displayed in Fig. 11.

6.4.2 Refine output guarantee – Replace general extractor with a language specific extractor

Datatype Length-bounded regular expression & string

Case study & relevance A usual step when modeling any kind of analysis project is to refine the architecture of the system by introducing an implementation-driven architecture into the model. This can be done by modeling language specific extractors that handle the Java ecosystem – as used in the literature – with Java as the General Programming Language (GPL) and Freemarker as the template language. Obviously, a decomposition of one or both extractors presented in the original model can be performed while preserving the refinement relationship, but as this was covered in a previous scenario (Section 6.3.2), we chose a direct component-to-component replacement scenario. Fig. 12 displays such a replacement. The Java-specific promise guarantees that all outputs of the Java extractor start with a capital letter and have the Java-specific fixed file ending. The replacement of the template extractor with a Freemarker extractor then takes place analogously and is not displayed here for brevity.

Result Intuitively, the expectation is that a more specific extractor, by offering a stronger guarantee, validates the conditions required by the validator component. The promise of the Java extractor is more specific and in a refinement relation to the more general extractor displayed in the original scenario. The actual result indeed finds the implication valid, as shown in Listing 21.

6.4.3 Refine input assumption - Replacement with a case-sensitive restricted validator

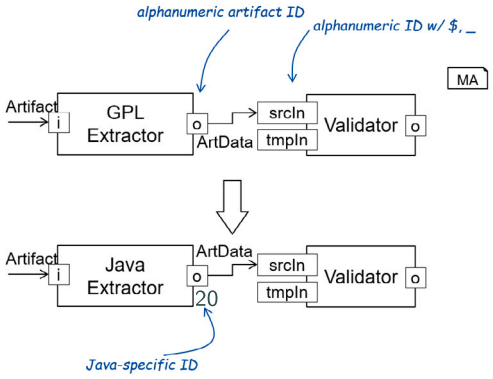
Datatype Length-bounded regular expression & string

Case study & relevance In this scenario we explore the situation where the validator component is replaced to model case-sensitive file systems. This is reflected in requiring camel case file endings without replacing any other component of the original scenario. To detail the functional specification of the validator, the validity of any extracted artifact can only be computed if the file endings are of the correct format. This is functionally another component refinement including its input condition. This scenario differs from the previous one by its expected result.

Result A strict refinement on the target of the connection does not necessarily guarantee that the implication holds. Thus, a counter-example is expected. Fig. 13 displays the component and its regular expression which encodes this in our approach. Because now the target of the implication is a stronger requirement than the source, our tool finds a counter-example provided in Listing 23.

6.4.4 Abstract input assumption - Replacement with an all-accepting validator with kleene operations

Datatype Length-bounded regular expression & string



(a) Semantic Refinement of a Promise Supplied by the General Extractor in listing 17 by Specifying Stronger Contracts Specific to Java.

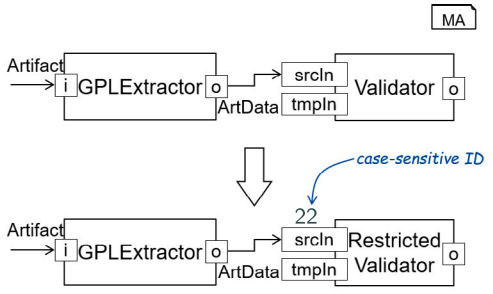
```
x.simpleName isin R"[A-Z][a-zA-Z0
    -9]{0,19}" &&
  x.nameExtension == "java" &&
  x.id ==  "@"+ x.simpleName + "!" + x
      .nameExtension
```

Listing (20) Refined Constraint.

```
Constraint of Port o in Component
    JavaExtractor guarantees constraint
     of Port srcIn in Component
    Validator
```

Listing (21) Output of the Constraint Checker for both Extractor to Validator connections displayed in Figure 11.

Fig. 12. Replacement operation of an extractor with a more specific one.



(a) Semantic refinement of assumed condition supplied by the artifact model validator by requiring a stronger assumption for case-sensitive file systems

```
x.id isin R"@[a-zA-Z0-9]{1,20}![a-z][a
    -zA-Z0-9]{0,19}"
```

Listing (22) Refined Constraint.

```
port GPLExtractor.o does not guarantee
     port ArtifactModelValidator.srcIn.
Counter example: objectdiagram x {
  objectDiagram_0:ObjectDiagram {
    String fullName="";
    String simpleName="k";
    String nameExtension="H";
    String id="@k!H";
  };
```

Listing (23) Output of the Constraint Checker for the Extractor to Refined Validator Connection as a Counter-Example.

Fig. 13. Replacement of a artifact model validator with a more specific one.

Case study & relevance This scenario presents an abstraction case where the validator is replaced by an all-accepting validator provided the two parts of the ID are encompassed by the characters presented in the original case, as shown in Fig. 14. This validator handles any naming schemes, also providing an abstraction of the associated assumption. Because a word regular expression could lead to catastrophic failures, we can obviously see that our approach is as strong as the constraints provided by the model. Kleene operations are put in use here. This represents a more convenient way of modeling constraints with regular expressions, by not specifying any bounds. While this has use cases in the real world, in the case of specifying constraints, this poses real issues.

Result An expected result for the single replacement of the validator – with its constraint in Listing 24 – delivers a positive result, which our tool certifies. In the case we also replace the output promise from the original scenario with a Kleene repetition as seen in Listing 25, even though through intuitive reasoning the result should stay the same, the tool runs in an endless loop. We chose to count this as a failed detection of a valid connection because it represents the user-friendly case.

6.4.5 Remove input assumption and merger of connections - Removal of an Artifact Model Validator

Case study & relevance This scenario explores the case where the Artifact Model validator is not present in the system. This effect offers no apparent explicit functional difference from a black-box perspective, but crucially it removes a component that processes artifact data streams and merges the connections in the pipeline.

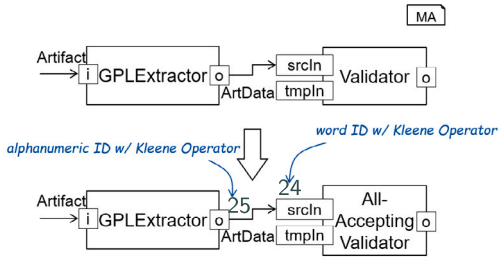
Result The connection produced by this operation modifies the architecture directly and evidently makes the types present on each connection end not match, as shown in Fig. 15. This is not caught directly by our SMT-centered approach, but rather trivially by MontiArc's context conditions which check weather port datatypes match. Listing 26 displays such a failure and an error-coded message.

```
0xC1110 Type mismatch, expected
    ObjectDiagram but provided ArtData
```

Listing 26: Output of the Constraint Checker for a Component Removal scenario where the merged connection ends port types are mismatched.

6.4.6 Add component - Addition of a merger component

Case study & relevance This scenario expresses an operation where the architecture was modified by adding a component specifically modeled to transfer artifact data from each individual artifact to a merged object diagram with aggregated data. Trivially, the wiring of the component was also designed in this addition. The newly-produced




```
x.id isin R"@.+!.+"
```

Listing (24) Abstracted Assumption.

```
x.simpleName isin R"[a-zA-Z0-9]+" &&
  x.nameExtension isin R"[a-zA-Z]+" &&
  x.id ==  "@"+ x.simpleName + "!" + x
      .nameExtension"
```

Listing (25) Kleene-Defined Guarantee.

Fig. 14. Replacement of an artifact model validator with a general validator.

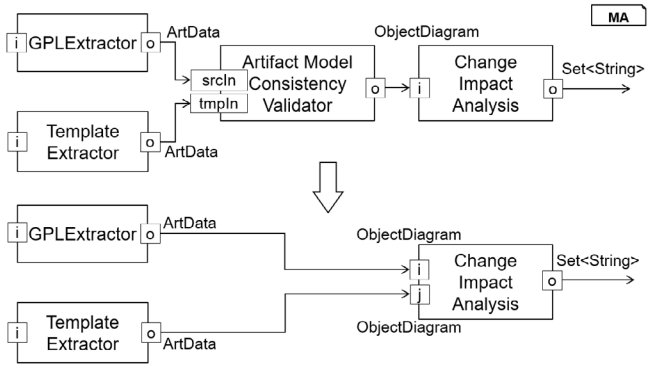


Fig. 15. Removal of an Artifact model validator component and merger of remaining connections.

result of the validation will be influenced by the new condition-promise pair shipped with the Merge component.

Result Trivially when modeling this connection, datatypes match with this modification by addition to the system architecture, as seen in Fig. 16. But as the validator's interface has been changed the old guarantee of the validator is obsolete. Thus, the merger component and the validator's promise require new constraints, making this scenario's result entirely dependent on the new constraints provided by the redesign, which was not presented with Q1 in mind.

6.5 Evaluation results

To evaluate the accuracy of our approach (Q1), we analyzed all defined scenarios under two conditions: (1) scenarios where no predefined violations exist in the constraints and (2) modified scenarios where constraints are deliberately altered to introduce expected violations. This allowed us to assess the accuracy of our constraint-checking mechanism in both cases. The results are presented in Table 2, where, for instance, S6.3.1 represents a scenario with no violations, while S6.3.1' denotes the modified version of S6.3.1 that includes an expected violation. In this case, the violation occurs because the input assumption of ports, which is defined by the constraints, does not align with the guarantee imposed on the output of the same ports.

For scenarios that contain violations, we can only calculate TPF, as there are no true negatives in these cases. The TPF value is 1.0, indicating that our approach correctly identified all scenarios with invalid constraints, and achieved the highest possible result. For scenarios that include only valid constraints, we calculate TNF, as there are no true positives in this case. This demonstrates that the approach successfully validates scenarios containing only the correct constraints. The analysis shows strong results for both TPF and TNF, confirming the overall quality and reliability of our approach in accurately distinguishing valid and invalid constraints.

As mentioned earlier in Section 6.2, we also calculate the F1-score to provide a more balanced evaluation of our approach. In our case, both Precision and Recall have a value of 1, since $t_p = 1$ and both f_p and f_n are 0. This indicates that our approach perfectly identifies valid and invalid constraints. Using the F1-score formula, $F_1 = 2 \cdot \frac{1 \cdot 1}{1+1} = 1$, we see that our approach achieves perfect effectiveness in validating constraints under the evaluated scenarios.

For Q2, Fig. 17 displays the execution time of the implication displayed in the scenario 6.4.1 with modifications to force valid checks or counter-examples. The blue plot represents the runtime measurement of finding counter-examples, while the red plot the runtime of validating the constraint. The constraints tested were selected by modifying the guarantee to limit it to a reduced number of repetitions. The x-axis represents the index `i` in the constraint`x isin [a-zA-Z_0-9]{1,i}`, more specifically, how many character repetitions do we allow in the guarantee. In the case of counter-example finding test, we keep the original constraint from Listing 18 unchanged. For the valid-constraint check, we chose to scale the assumption as well to force valid implications. So, while using the same indexing as above for the guarantee, we also scaled the assumption, as in `@[a-zA-Z0-9]{1,i}![a-zA-Z_$]{1,i}`, thus finding valid checks.

The result is clear that every regular expression can be checked up to single Kleene operations on both sides of the connection, as also seen in scenario 6.4.4. To be more precise, by introducing bounded regular expressions to any side of the connection, non-termination is avoided. The caveat is that the most powerful scenarios where Kleene operations are found on both sides of the implication do not terminate even in the simplest of character ranges. This was not presented in the plot. We performed the test for Kleene operations by replacing the bounds with the `Kleene plus` operation on both sides of the connection tested by the former aforementioned test.

Our findings suggest that finding counter-examples tends to scale linearly with the bounds of the repetition of characters with slight oscillations due to the Z3 optimizations. Inopportunely, checking implications that result in valid checks tends to scale exponentially to the point where valid implications with high character repetitions are unfeasible to check. Another point to consider is the performance of the regular expressions in comparison with simple arithmetic constraint checks on datatypes such as integers and floats. As expected, these scale at most, linearly with the distance between the compared values. A simple check where `x < 0` implies `x < $y` takes milliseconds even at the max distance where the integer `y` is the JVM upper bound. The execution time values for integers can be found in Fig. 18 and is in concordance to our assumptions, i.e. these being feasible to check independent of size.

6.6 Threats to validity

We discuss the threats to validity based on Runeson and Höst's guidelines (Runeson and Höst, 2009), which categorize them into Internal Validity, External Validity, Construct Validity, and Reliability.

Internal validity ensures that the observed causal relationships are not affected by any factors other than those intentionally considered in the study. In our study, we evaluate the accuracy and performance of our approach. The metrics related to accuracy are influenced by

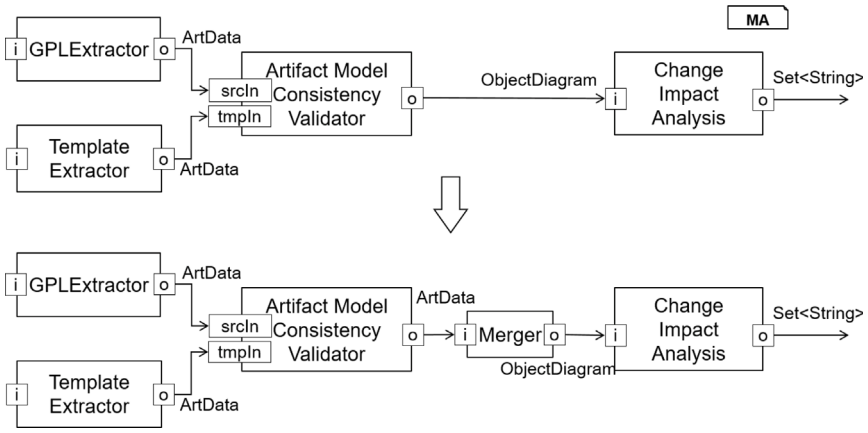


Fig. 16. Addition of a merger component aggregating all artifact data into an object diagram.

Table 2
Evaluation results for Q1 — Accuracy of our approach in detecting violations.

Scenarios	Violations detected	No violations detected	TPF	TNF	F1-score
Expected violations	S6.3.1', S6.3.2', S6.3.3', S6.3.4', S6.4.1', S6.4.2', S6.4.3	–	1.0	–	1.0
No violations	–	S6.3.1, S6.3.2, S6.3.3, S6.3.4, S6.4.1, S6.4.2	–	1.0	

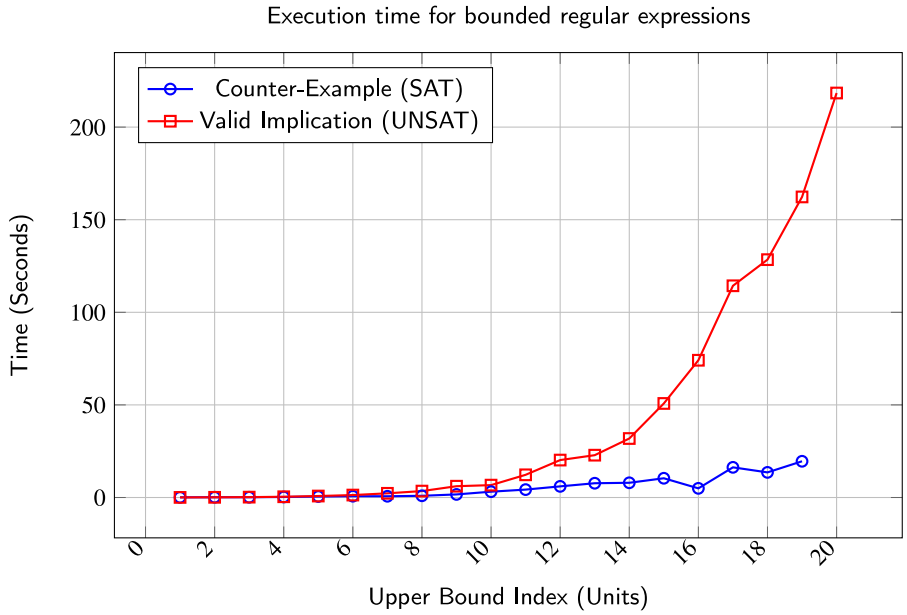


Fig. 17. Performance testing of regular expressions with bounded repetition of characters. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the selection of case studies and scenarios. We mitigate this factor by choosing two case studies that cover two different scopes of the analysis components and by systematically designing the scenarios to capture all major potential changes. However, we cannot completely eliminate this factor, as we were unable to identify additional case studies with various types of constraints. This limitation will be addressed in future work. There may also be biases related to the enrichment of models with semantic constraints, due to the varying definitions of constraints used by practitioners. However, we can exclude this factor, as the authors of this article are sufficiently familiar with the two selected case studies. Furthermore, any errors or inconsistencies introduced during the transformation from Palladio models to MontiArc models could potentially impact the semantic validation results. We exclude this threat, as we were able to model our scenarios and achieve the expected results in constraint checking with good accuracy. Moreover, the use of a simplified model of a use case for evaluation purposes may limit the generalizability of the results. Therefore, testing with real-world data will be essential to ensure that the approach is robust and reliable. This means that the real modeling of these case studies may differ slightly, as they are more complex than the versions presented in our

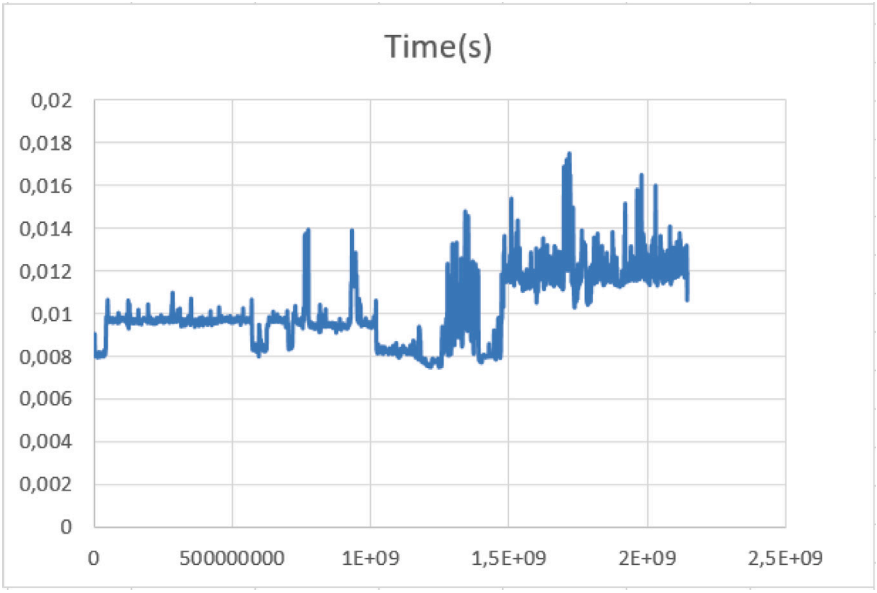


Fig. 18. Empirical measurement of a direct implication check where the distance between the assumption and the guarantee covers the JVM integer bounds.

study. However, since we have modeled the most critical parts of the use cases and incorporated the most important semantic constraints, this factor should not significantly impact the results, even in more complex models. Additionally, while our approach has been tested under controlled conditions, future studies should validate its effectiveness in more diverse and large-scale scenarios to further strengthen its reliability and applicability.

External validity assures that the findings can be generalized, making the results useful beyond the researcher who conducted the study. A potential bias may arise from the selection of the two use cases used to evaluate our approach. To mitigate this, we systematically defined different scenarios for each case study to cover a wide range of possible states and variations. Moreover, while our study presents exemplary results, it does not include a comprehensive empirical validation across a diverse set of models or real-world case studies. This limitation may affect the external validity of our findings, as the results may not be directly generalizable to all possible use cases. However, the selected cases are derived from related works and are considered representative of the application domain.

Construct validity ensures that the metrics used for answering the research questions are appropriate. The TPF and TNF methods were chosen for the accuracy calculation in our study because the analysis of semantic constraints yields binary results-whether the constraints are validated or not and whether violations have been detected or not. In this case, these metrics are sufficient. For the evaluation of scalability and performance, we chose to measure the time complexity of the constraint checker, as the performance of our approach primarily depends on this factor. By focusing on execution time, we ensure that our evaluation reflects the computational efficiency of the method in different scenarios.

Reliability ensures that the results of the evaluation are independent of the researchers. The creation of a model in Palladio largely depends on the design decisions made by the conducting researcher. However, we can argue that if the researcher has a good understanding of Palladio and MontiArc, they can create a reasonable model that produces results similar to ours. In general, we ensured reproducibility by providing the models for all scenarios, the transformation code, and the constraint-checking process. Conducting the study only requires running the provided models; therefore, the results of the conducted scenarios do not depend on the individual researcher and should yield consistent outcomes. Nevertheless, a potential threat to validity arises from the fact that modeling still involves subjective decisions, and variations in experience or interpretation might introduce minor differences. To mitigate this, we have provided clear instructions on how to use the tool and the necessary artifacts to ensure traceability and minimize inconsistencies.

6.7 Limitations

The most restrictive limitation of our approach is that it does not support a broad range of constraints, particularly those related to inter-message relations. For example, constraints such as ‘‘Value Increasing’’, which ensure that a value monotonically increases over time, are not currently supported, as these require encoding into SMT constraints limited port communication history. Additionally, our approach does not facilitate the aggregation of data across multiple channels of different components, which is crucial for analyzing complex interactions. In a similar manner, while we can enforce local constraints within individual components, enforcing global constraints that span across multiple components remains challenging. It affects scenarios where system-wide properties, such as timing dependencies, need to be validated. Additionally, Palladio does not support enumerations, and lists and sets are also unsupported as data types. QVTo also imposes limitations when transforming nested classes. These issues restrict the ability to fully leverage MontiArc’s capabilities for modeling constraints.

The MontiArc Language supports hierarchical composition, allowing systems to be structured with nested subcomponents; however, our approach does not utilize this capability. The lack of this capability restricts our ability to modularly define and manage some complex systems. Moreover, MontiArc incorporates automata-based behavior modeling, allowing dynamic component behavior to be specified formally. By doing so, it is possible to capture state transitions and interactions within a system more accurately. We are not currently able to formally specify and verify dynamic behaviors within our system model due to the lack of support for automata, but there is currently work being done to support automata to SMT encoding.

To the topic of validating our approach, the evaluation was primarily driven by accuracy and performance, as these metrics are critical for evaluating its effectiveness and efficiency. In contrast, usability focuses on tool usage and user experience, which is less relevant to our validation goals. Additionally, we assume that users have basic knowledge of defining models in Palladio. Given this level of expertise, usability metrics would not provide significant insights or meaningful improvements to our validation process. Therefore, usability was not a primary focus of our evaluation.

7 Conclusion & future work

In this paper, we presented an approach to automatically check semantic constraints by transforming software architecture models from the Palladio approach to the MontiArc architecture description language. We then performed constraint checking within MontiArc. Our approach leverages the assumption-guarantee formalism for constraint validation and enables the validation of constraints on both primitive data types and regular expressions. In contrast to related approaches, we bridge the gap between architecture analysis methods focused on performance prediction and semantic constraint checking. This enhances software architecture reliability by combining performance evaluation with semantic soundness while enabling thorough analysis without the need for repeated modeling of the same system in different formalisms.

We evaluated the accuracy of the approach in checking semantic constraints in both valid and invalid substitution scenarios, based on the defined constraints. To achieve this, we systematically derived multiple scenarios for each case study. The evaluation results demonstrate that the approach can accurately check valid implications that denote the semantic soundness of a composition, while effectively detecting violations in scenarios where the performed operation invalidates its semantic soundness. Additionally, we assess the performance of the approach by evaluating the runtime of the constraint checker as the complexity of constraints increases. The results indicate that our approach performs well in assessing arithmetic constraints. However, for regular expressions, performance does not scale linearly as complexity increases when assessing constraints in scenarios where the composition is semantically sound.

For future work, we plan to extend our approach to fully support the expressiveness of the PCM concepts within the MontiArc framework. This will involve incorporating additional features and enhancing the transformation process to accommodate more complex architectural elements. For facilitating the reuse of the MontiArc semantic soundness in this approach, we plan to retransform the resulting counter-examples and validation results into Palladio to be interpreted and to allow for optimizations at system design time.

Another future plan is to extend the approach to handle more complex, domain-specific constraints, such as inter-message relations constraints, in order to better capture the dynamic aspects of system behavior in software architecture models. To this end, we also plan to incorporate the more potent communication channel history-based assumption-guarantee formalism into our semantic soundness framework. Finally, we will conduct empirical studies with larger and more complex case studies to further validate the performance and effectiveness of the proposed approach, and to assess scalability.

CRediT authorship contribution statement

Bahareh Taghavi: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Data curation, Conceptualization, Investigation. **Sebastian Weber:** Writing – original draft, Software, Methodology. **Adrian Marin:** Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Conceptualization. **Bernhard Rumpe:** Writing – review & editing, Supervision. **Sebastian Stüber:** Writing – review & editing, Validation. **Jörg Henß:** Methodology. **Thomas Weber:** Writing – review & editing. **Robert Heinrich:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was funded by the DFG (German Research Foundation) – project number 499241390 (FeCoMASS), and supported by the Collaborative Research Center “Convide”- SFB 1608- 501798263, and supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF), KASTEL Security Research Labs, and supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF). This paper has been edited by our textician, Daniel Shea.

Appendix. Complete code listings for the running example

```
package cd2pojo.slingshot;
classdiagram Slingshot {
  public class SystemCall {
    String method;
  }
  public class SystemSimulationReturn {
    double utilization;
  }
  public class ResourceDemand {
    double resourceDemand;
  }
  public class UsageSimulationReturn {
    double responseTime; double utilization;
  }
  public class ResourceSimulationReturn {
    double utilization;
  }
  public class UsageScenario {
    int numberOfUsers;
  }
}
```

Listing 27: Class diagram derived from the Slingshot Repository model (see Fig. 3)

```
package slingshot;
import cd2pojo.slingshot.Slingshot.
    UsageScenario;
import cd2pojo.slingshot.Slingshot.
    UsageSimulationReturn;
import cd2pojo.slingshot.Slingshot.
    ResourceSimulationReturn;
import cd2pojo.slingshot.Slingshot.
    ResourceDemand;
import cd2pojo.slingshot.Slingshot.
    SystemCall;
import cd2pojo.slingshot.Slingshot.
    SystemSimulationReturn;
component Slingshot {
  port << condition = "x.numberOfUsers > 0
      && x.numberOfUsers < 2147483647" >> in
       UsageScenario usageScenario;
  port << condition = "x.responseTime > 1.0"
       >> out UsageSimulationReturn
      usageSimulationReturn;
  component UsageSimulation {
    port << condition = "x.responseTime >
        1.0" >> out UsageSimulationReturn
        usageSimulationReturn;
    port << condition = "x.numberOfUsers >
        0" >> in UsageScenario usageScenario
        ;
    port << delayed, condition = "x.method
        == \"executeOperation\"" >> out
        SystemCall systemCall;
```

```
    port in SystemSimulationReturn
        systemSimulationReturn;
  }
  component SystemSimulation {
    port out SystemSimulationReturn
        systemSimulationReturn;
    port << condition = "x.method == \"
        executeOperation\"" >> in SystemCall
         systemCall;
    port << condition = "x.resourceDemand >
        1.0" >> out ResourceDemand
        resourceDemand;
    port << condition = "x.utilization >
        1.0" >> in ResourceSimulationReturn
        resourceSimulationReturn;
  }
  component ResourceSimulation {
    port << delayed,  condition = "x.
        utilization > 1.0" >> out
        ResourceSimulationReturn
        resourceSimulationReturn;
    port << condition = "x.resourceDemand >
        1.0" >> in ResourceDemand
        resourceDemand;
  }
  UsageSimulation usageSimulation;
  SystemSimulation systemSimulation;
  ResourceSimulation resourceSimulation;
  systemSimulation.systemSimulationReturn ->
       usageSimulation.
      systemSimulationReturn;
  usageSimulation.systemCall ->
      systemSimulation.systemCall;
  resourceSimulation.
      resourceSimulationReturn ->
      systemSimulation.
      resourceSimulationReturn;
  systemSimulation.resourceDemand ->
      resourceSimulation.resourceDemand;
  usageScenario -> usageSimulation.
      usageScenario;
  usageSimulation.usageSimulationReturn ->
      usageSimulationReturn;
}
```

Listing 28: MontiArc model based on the Repository (Fig. 3) and System (Fig. 4) models.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi=
    "http://www.omg.org/XMI" xmlns:ecore="
    http://www.eclipse.org/emf/2002/Ecore"
    name="Constraints" nsURI="Constraints"
    nsPrefix="constraints">
  <eAnnotations references="Slingshot.system
      #Connector1">
    <details key="Source:In" value="x.
        numberOfUsers > 0 &amp;amp;&amp; x.
        numberOfUsers &lt; 2147483647"/>
    <details key="Target:In" value="x.
        numberOfUsers > 0"/>
    <details key="Source:Out" value="x.
        responseTime > 0.0"/>
    <details key="Target:Out" value="x.
        responseTime > 0.0"/>
  </eAnnotations>
  <eAnnotations references="Slingshot.system
      #Connector2">
    <details key="Source:Out" value="x.
        method == &quot;executeOperation&
        quot;"/>
    <details key="Target:In" value="x.method
         == &quot;executeOperation&quot;"/>
  </eAnnotations>
  <eAnnotations references="Slingshot.system
      #Connector3">
    <details key="Source:Out" value="x.
        resourceDemand > 0.0"/>
    <details key="Target:In" value="x.
        resourceDemand > 0.0"/>
  </eAnnotations>
</ecore:EPackage>
```

Listing 29: Ecore model of the constraints.

```
mapping AssemblyContext::assemblyContextToComponent() :
      Component {
  name := self.entityName;
  self.encapsulatedComponent__AssemblyContext.
       providedRoles_InterfaceProvidingEntity
    -> select(oclIsTypeOf(OperationProvidedRole))
    -> forEach(role) {
        role.oclAsType(OperationProvidedRole) -> map
            operationProvidedRoleToPort()
          -> forEach(port) { arcElements += port; };
    };
    // Process ALL Required Roles
    self.encapsulatedComponent__AssemblyContext.
        requiredRoles_InterfaceRequiringEntity
      -> select(oclIsTypeOf(OperationRequiredRole))
      -> forEach(role) {
          role.oclAsType(OperationRequiredRole) ->
              map operationRequiredRoleToPort()
            -> forEach(port) { arcElements += port;
                };
  };
}
```

Listing 30: QVTo mapping for transforming AssemblyContext to Component.

Data availability

All data used in the evaluation are available in our dataset: https://github.com/FeCoMASS/Model-Transformation-for-Automated-Constraint-Validation. It includes our complete tool with the transformation and constraint-checking process, as well as the full QVTo code used for the model transformation. Additionally, all the scenarios used for the evaluation are included.

References

Acerbis, R., Bongio, A., Brambilla, M., Tisi, M., Ceri, S., Tosetti, E., 2007. Developing ebusiness solutions with a model driven approach: The case of acer EMEA. In: International Conference on Web Engineering. Springer, pp. 539–544. http://dx.doi.org/10.1007/978-3-540-73597-7_51.

Allen, R., Vestal, S., Cornhill, D., Lewis, B., 2002. Using an architecture description language for quantitative analysis of real-time systems. In: Proceedings of the 3rd International Workshop on Software and Performance. pp. 203–210. http://dx.doi.org/10.1145/584369.584399.

Baier, C., Katoen, J.-P., 2008. Principles of Model Checking. MIT Press.

Barrett, C., Stump, A., Tinelli, C., et al., 2010. The smt-lib standard: Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories. Vol. 13, Edinburgh, UK, p. 14.

Basili, V.R., Weiss, D.M., 1984. A methodology for collecting valid software engineering data. IEEE Trans. Softw. Eng. (6), 728–738. http://dx.doi.org/10.1109/TSE.1984.5010301.

Bertolino, A., Mirandola, R., 2004. CB-SPE tool: Putting component-based performance engineering into practice. In: International Symposium on Component-Based Software Engineering. Springer, pp. 233–248. http://dx.doi.org/10.1007/978-3-540-24774-6_21.

Bjørner, N., 2018. Z3 and SMT in industrial R&D. In: Formal Methods: 22nd International Symposium, FM 2018, Held As Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings 22. Springer, pp. 675–678. http://dx.doi.org/10.1007/978-3-319-95582-7_44.

Bjørner, N., Jayaraman, K., 2015. Checking cloud contracts in microsoft azure. In: Distributed Computing and Internet Technology: 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings 11. Springer, pp. 21–32. http://dx.doi.org/10.1007/978-3-319-14977-6_2.

Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, http://dx.doi.org/10.1007/978-3-031-02549-5.

Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T.F., Weber, R., 1992. The Design of Distributed Systems: An Introduction to Focus. Citeseer.

Broy, M., Stølen, K., 2001. Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg, http://dx.doi.org/10.1007/978-1-4613-0091-5.

Broy, M., Stølen, K., 2012. Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer Science & Business Media.

Butting, A., Greifenberg, T., Rumpe, B., Wortmann, A., 2017. Taming the complexity of model-driven systems engineering projects. In: Part of the Grand Challenges in Modeling (GRAND'17) Workshop. URL: http://www.se-rwth.de/publications/Taming-the-Complexity-of-Model-Driven-Systems-Engineering.pdf.

Butting, A., Greifenberg, T., Rumpe, B., Wortmann, A., 2018. On the need for artifact models in model-driven systems engineering projects. In: Seidl, M., Zschaler, S. (Eds.), Software Technologies: Applications and Foundations. In: LNCS 10748, Springer, pp. 146–153. http://dx.doi.org/10.1007/978-3-319-74730-9_12, URL: http://www.se-rwth.de/publications/On-the-Need-for-Artifact-Models-in-Model-Driven-Systems-Engineering-Projects.pdf.

Cabot, J., Claris, R., Riera, D., et al., 2008. Verification of UML/OCL class diagrams using constraint programming. In: 2008 IEEE International Conference on Software Testing Verification and Validation Workshop. IEEE, pp. 73–80. http://dx.doi.org/10.1109/ICSTW.2008.54.

Caldiera, V.R.B.G., Rombach, H.D., 1994. The goal question metric approach. Encycl. Softw. Eng. 528–532.

Cardei, I., Fonoage, M., Shankar, R., 2008. Model based requirements specification and validation for component architectures. In: 2008 2nd Annual IEEE Systems Conference. IEEE, pp. 1–8. http://dx.doi.org/10.1109/SYSTEMS.2008.4519001.

Czepa, C., Tran, H., Zdun, U., Kim, T.T.T., Weiss, E., Ruhsam, C., 2017. On the understandability of semantic constraints for behavioral software architecture compliance: A controlled experiment. In: 2017 IEEE International Conference on Software Architecture. ICSA, IEEE, pp. 155–164. http://dx.doi.org/10.1109/ICSA.2017.10.

Czepa, C., Zdun, U., 2019. How understandable are pattern-based behavioral constraints for novice software designers? ACM Trans. Softw. Eng. Methodol. (TOSEM) 28 (2), 1–38. http://dx.doi.org/10.1145/3306608.

Damus, C., Sánchez-Barbudo, A., 2002. OCL documentation.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 337–340.

France, R., Rumpe, B., 2007. Model-driven development of complex software: A research roadmap. Futur. Softw. Eng. (FOSE '07) 37–54. http://dx.doi.org/10.1109/FOSE.2007.14, URL: http://www.se-rwth.de/publications/Model-driven-Development-of-Complex-Software-A-Research-Roadmap.pdf.

Franconi, E., Mosca, A., Oriol, X., Rull, G., Teniente, E., 2019. OCL FO: First-order expressive OCL constraints for efficient integrity checking. Softw. Syst. Model. 18 (4), 2655–2678. http://dx.doi.org/10.1007/s10270-018-0688-z.

Gacek, C., de Lemos, R., 2006. Architectural description of dependable software systems. In: Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective. Springer, pp. 127–142. http://dx.doi.org/10.1007/1-84628-111-3_7.

Greifenberg, T., Hillemacher, S., Hölldobler, K., 2020. Applied artifact-based analysis for architecture consistency checking. In: Ernst Denert Award for Software Engineering 2019. Springer, pp. 61–85, URL: http://www.se-rwth.de/publications/Applied-Artifact-Based-Analysis-for-Architecture-Consistency-Checking.pdf.

Greifenberg, T., Look, M., Roidl, S., Rumpe, B., 2015. Engineering tagging languages for DSLs. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems. MODELS, IEEE, pp. 34–43.

Haber, A., 2016. MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems. In: Aachener Informatik-Berichte, Software Engineering, Band 24, Shaker Verlag, URL: http://www.se-rwth.de/phdtheses/Diss-Haber-MontiArc-Architectural-Modeling-and-Simulation-of-Interactive-Distributed-Systems.pdf.

Haddad, S., Kordon, F., Pautet, L., Petrucci, L., 2011. Architecture description languages. In: Models and Analysis in Distributed Systems. John Wiley & Sons, Ltd, pp. 97–134. http://dx.doi.org/10.1002/9781118602638.ch5, Chapter 5.

Hamlet, D., Mason, D., Woit, D., 2004. Properties of software systems synthesized from components. In: Component-Based Software Development: Case Studies. World Scientific, pp. 129–158.

Harel, D., Rumpe, B., 2004. Meaningful modeling: What's the semantics of "Semantics"? IEEE Comput. J. 37 (10), 64–72, URL: http://www.se-rwth.de/staff/rumpe/publications20042008/Meaningful-Modeling-Whats-the-Semantics-of-Semantics.pdf.

Heinrich, R., Strittmatter, M., Reussner, R.H., 2021. A layered reference architecture for metamodels to tailor quality modeling and analysis. IEEE Trans. Softw. Eng. 47 (4), 775–800. http://dx.doi.org/10.1109/TSE.2019.2903797.

Heinrich, R., Werle, D., Klare, H., Reussner, R., Kramer, M., Becker, S., Happe, J., Koziolek, H., Krogmann, K., 2018. The palladio-bench for modeling and simulating software architectures. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 37–40. http://dx.doi.org/10.1145/3183440.3183474.

Henss, J., Merkle, P., Reussner, R.H., 2013. The OMPCM simulator for model-based software performance prediction. In: Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques. pp. 354–357.

Hillemacher, S., Jäckel, N., Kugler, C., Orth, P., Schmalzing, D., Wachtmeister, L., 2021. Artifact-based analysis for the development of collaborative embedded systems. In: Model-Based Engineering of Collaborative Embedded Systems. Springer, pp. 315–331, URL: http://www.se-rwth.de/publications/Artifact-Based-Analysis-for-the-Development-of-Collaborative-Embedded-Systems.pdf.

Hölldobler, K., Kautz, O., Rumpe, B., 2021. MontiCore Language Workbench and Library Handbook: Edition 2021. In: Aachener Informatik-Berichte, Software Engineering, Band 48, Shaker Verlag, URL: http://www.monticore.de/handbook.pdf.

Katić, J., Klinaku, F., Becker, S., 2021. The slingshot simulator: An extensible event-driven PCM simulator (poster).

Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B., 2020a. An approach for logic-based knowledge representation and automated reasoning over underspecification and refinement in safety-critical cyber-physical systems. In: Hebig, R., Heinrich, R. (Eds.), Combined Proceedings of the Workshops At Software Engineering 2020. Vol. 2581, CEUR Workshop Proceedings, URL: http://www.se-rwth.de/publications/An-Approach-for-Logic-based-Knowledge-Representation-and-Automated-Reasoning-over-Underspecification-and-Refinement-in-Safety-Critical-Cyber-Physical-Systems.pdf.

Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B., 2020b. MontiBelle - toolbox for a model-based development and verification of distributed critical systems for compliance with functional safety. In: AIAA Scitech 2020 Forum. American Institute of Aeronautics and Astronautics, URL: http://www.se-rwth.de/publications/MontiBelle-Toolbox-for-a-Model-Based-Development-and-Verification-of-Distributed-Critical-Systems-for-Compliance-with-Functional-Safety.pdf.

Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B., Schweiger, A., 2024. Model-driven development for functional correctness of avionics systems: A verification framework for sysml specifications. CEAS Aeronaut. J. 15 (4), http://dx.doi.org/10.1007/s13272-024-00762-6, URL: http://www.se-rwth.de/publications/Model-driven-Development-for-Functional-Correctness-of-Avionics-Systems-A-Verification-Framework-for-SysML-Specifications.pdf.

Klinaku, F., Stieß, S.S., Hakamian, A., Becker, S., 2025. An architectural view type for elasticity modeling and simulation—The Slingshot approach. J. Syst. Softw. 112432. http://dx.doi.org/10.1016/j.jss.2025.112432.

Koch, S., 2024. A Reference Structure for Modular Model-based Analyses. KIT Scientific Publishing, http://dx.doi.org/10.5445/KSP/1000167848.

Lano, K., 2021. Adding regular expression operators to OCL. In: STAF Workshops. pp. 162–168.

Ly, L.T., Rinderle-Ma, S., Göser, K., Dadam, P., 2012. On enabling integrated process compliance with semantic constraints in process management systems: Requirements, challenges, solutions. Inf. Syst. Front. 14 (2), 195–219. http://dx.doi.org/10.1007/s10796-009-9185-9.

Manning, C.D., 2009. An introduction to information retrieval.

Medvidovic, N., Taylor, R.N., 1997. A framework for classifying and comparing architecture description languages. ACM SIGSOFT Softw. Eng. Notes 22 (6), 60–76. http://dx.doi.org/10.1145/267896.267903.

Metz, C.E., 1978. Basic principles of ROC analysis. In: Seminars in Nuclear Medicine. Vol. 8, Elsevier, pp. 283–298. http://dx.doi.org/10.1016/S0001-2998(78)80014-2, no. 4.

Meyer, B., 1988. Eiffel: A language and environment for software engineering. J. Syst. Softw. 8 (3), 199–246. http://dx.doi.org/10.1016/0164-1212(88)90022-2.

Meyer, B., 1990. Lessons from the design of the eiffel libraries. Commun. ACM 33 (9), 68–88. http://dx.doi.org/10.1145/83880.84464.

Meyer, B., 1992. Applying'design by contract'. Computer 25 (10), 40–51. http://dx.doi.org/10.1109/2.161279.

Moriconi, M., Qian, X., Riemenschneider, R.A., 1995. Correct architecture refinement. IEEE Trans. Softw. Eng. 21 (4), 356–372. http://dx.doi.org/10.1109/32.385972.

Nipkow, T., Paulson, L.C., Wenzel, M., 2002. Isabelle/hol: A Proof Assistant for Higher-Order Logic. In: Lecture notes in artificial intelligence, vol. 2283, Springer, Berlin [etc.].

Pandey, R., 2010. Architectural description languages (ADLs) vs UML: A review. ACM SIGSOFT Softw. Eng. Notes 35 (3), 1–5. http://dx.doi.org/10.1145/1764810.1764828.

Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., 2016. Modeling and Simulating Software Architectures: The Palladio Approach. MIT Press.

Richters, M., Gogolla, M., 2000. Validating UML models and OCL constraints. In: International Conference on the Unified Modeling Language. Springer, pp. 265–277. http://dx.doi.org/10.1007/3-540-40011-7_19.

Ringert, J.O., Rumpe, B., 2011. A little synopsis on streams, stream processing functions, and state-based stream processing.. Int. J. Softw. Inform. 5 (1–2), 29–53.

Rumpe, B., 2016. Modeling with UML: Language, Concepts, Methods. Springer International, URL: https://mbse.se-rwth.de/.

Rumpe, B., Stachon, M., Stüber, S., Voufo, V., 2024. Semantic difference analysis with invariant tracing for class diagrams extended by OCL. In: Workshop on Model Driven Engineering, Verification and Validation, MODELS Companion '24: International Conference on Model Driven Engineering Languages and Systems. MoDeVVa, Association for Computing Machinery (ACM), pp. 1066–1075. http://dx.doi.org/10.1145/3652620.3687818, URL: http://www.se-rwth.de/publications/Semantic-Difference-Analysis-with-Invariant-Tracing-for-Class-Diagrams-Extended-by-OCL.pdf.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14, 131–164. http://dx.doi.org/10.1007/s10664-008-9102-8.

Steinberg, D., Budinsky, F., Merks, E., Paternostro, M., 2008. EMF: Eclipse Modeling Framework. Pearson Education.

Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., Yang, J., 2013. Secure distributed programming with value-dependent types. J. Funct. Programming 23 (4), 402–451. http://dx.doi.org/10.1017/S0956796813000142.

Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., et al., 2016. Dependent types and multi-monadic effects in F. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 256–270. http://dx.doi.org/10.1145/2837614.2837655.

Taghavi, B., Heinrich, R., Marin, A., Rumpe, B., Stüber, S., Weber, S., 2025. Semantic validation for slingshot simulator using MontiArc. In: Softwaretechnik-Trends Band 45, Heft 1. Gesellschaft für Informatik e.V..

Talcott, C., Ananieva, S., Bae, K., Combemale, B., Heinrich, R., Hills, M., Khakpour, N., Reussner, R., Rumpe, B., Scandurra, P., et al., 2021. Composition of languages, models, and analyses. Compos. Model-Based Anal. Tools 45–70. http://dx.doi.org/10.1007/978-3-030-81915-6_4.

Weber, S., Henß, J., Taghavi, B., Weber, T., Stüber, S., Marin, A., Rumpe, B., Heinrich, R., 2024. Semantics enhancing model transformation for automated constraint validation of palladio software architecture to MontiArc models. In: European Conference on Software Architecture. Springer, pp. 30–38. http://dx.doi.org/10.1007/978-3-031-71246-3_4.