

Generating Formal Software Architecture Descriptions from Semi-Formal SysML-Based Models: A Model-Driven Approach

Camila Araújo[1,2(✉)], Thais Batista[2], Everton Cavalcante[2], and Flavio Oquendo[3]

[1] State University of Rio Grande do Norte, Natal, Brazil
camilaaraujo@uern.br
[2] Federal University of Rio Grande do Norte, Natal, Brazil
{thais,everton}@dimap.ufrn.br
[3] IRISA-UMR CNRS/Université Bretagne Sud, Vannes, France
flavio.oquendo@irisa.fr

Abstract. The critical nature of many complex software-intensive systems requires formal architecture descriptions for supporting automated architectural analysis regarding correctness properties. Due to the challenges of adopting formal approaches, many architects have preferred using notations such as UML, SysML, and their derivatives to describe the structure and behavior of software architectures. However, these semi-formal notations have limitations regarding the sought support for architectural analysis. This paper presents an approach to bridge the rigor of formal architecture descriptions and the ease of use of SysML-based notations widely used elsewhere. The main concern is providing formal semantics to SysADL, a SysML-based language to describe software-intensive system architectures. The formal semantics is provided by π-ADL, a formal architecture description language. A model-to-model transformation was defined and implemented to concretize the mapping between the elements of these languages and hence automatically generate formal architecture descriptions in π-ADL from SysADL. This paper describes a proof-of-concept to illustrate the mapping between SysADL and π-ADL and an exploratory study on the transformation performance.

Keywords: Model-driven development · Model transformation · Architecture description language · Formal verification · SysML

1 Introduction

The representation of software architectures is one of the main activities of an architecture-driven software development process as it allows anticipating important decisions regarding the system design. This activity results in architecture

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12951, pp. 394–410, 2021.
https://doi.org/10.1007/978-3-030-86970-0_28

descriptions, which can be used to communicate the software architecture among stakeholders and support its maintenance, evaluation, and evolution. Due to the critical nature of many software-intensive systems, architecture descriptions need to be expressed in formal notations as means of better supporting automated architectural analysis [2,6]. The architectural analysis aims at precisely verifying if their designed architectures can meet important properties, e.g., correctness.

As a consequence of the significant learning curve, lack of knowledge among stakeholders, and weak support concerning tooling and guidance on the use of formal approaches, software architects have primarily preferred to use semi-formal notations such as UML, SysML or their derivatives as means of describing structure and behavior of software architectures. This choice has been mainly motivated by their lower learning curve, visuality, and general-purpose scope [13, 18,19]. However, this type of notation has known limitations regarding the sought support for automatically verifying architectural properties.

This paper presents a model-driven approach to bridge the rigor of formal architecture descriptions and the ease of use of SysML-based notations widely used elsewhere. More specifically, the proposed approach intends to provide a formal semantics to SysADL [17], a SysML-based architectural language that combines typical constructs of architectural languages with the use of the popular diagrammatic notation based on the SysML Standard for modeling software-intensive systems. The formal semantics is provided by π-ADL [16], a language based on π-calculus to formally describe software architectures. On the one hand, SysADL was chosen for this work as it is aligned with the ISO/IEC/IEEE 42010 International Standard for architectural descriptions [10]. On the other hand, π-ADL was chosen since it allows formally describing both structure and behavior of dynamic software architectures, unlike most existing architectural languages [4].

The transformation of an architecture description in SysADL to a corresponding one in π-ADL allows for its further formal verification, which is already available for π-ADL architecture descriptions [5]. A mapping process was conceived to establish the relationship between SysADL and π-ADL and their elements. The mapping was concretized by a model-to-model transformation implemented in the ATLAS Transformation Language (ATL) [1,11] upon Eclipse-based frameworks, which are also used in the SysADL [12] and π-ADL [5] tools. In this paper, the proposed approach is validated with an Automated Guided Vehicle (AVG) System [9]. This paper also reports the results of computational experiments to analyze the performance and scalability of the transformation.

The remainder of this paper is structured as follows. Section 2 provides the background of this work. Section 3 details the transformation from SysADL to π-ADL, illustrated with samples of the AVG system. Section 4 reports the exploratory study to analyze the performance of the proposed transformation. Section 5 discusses related work. Section 6 contains some concluding remarks.

2 Background

2.1 SysADL

SysADL [17] is a SysML-based language designed to support multiple view modeling, cross-view checking, validation, and execution of software architectures. This language reconciles the rigorous semantic of architectural languages with the use of popular diagrammatic notation based on the SysML Standard for modeling software-intensive systems. Moreover, SysADL is supported by the SysADL Studio tool [12], which enables software architects to describe software architectures either visually and/or textually.

SysADL defines three software architecture viewpoints for a system, namely (i) *structural*, (ii) *behavioral*, and (iii) *executable*. The structural viewpoint defines the architectural elements composing the structure of a system (*components*, *ports*, *connectors*) and relationships among them. SysADL requires declaring all elements before creating their instances in Block Definition Diagrams (BDDs). The Internal Block Diagram (IBD) is used to specify how instances of components and connectors form the configuration of architectures.

The behavioral viewpoint details the behavior of (i) components and connectors through *activities*, *actions*, *constraints* and (ii) ports through *protocols*. Activity instances are described in the Activity Diagram by instantiating actions and flows. Activities or actions may have validation constraints specified through expressions in the OMG Action Language for Foundational UML (ALF). Constraints can be also expressed using the Parametric Diagram.

The executable viewpoint represents the concretization of both structural and behavioral viewpoints by simulating the architecture behavior at runtime. The primary purpose of the simulation is to validate the behavior logic regarding the satisfaction of requirements and analysis of architecture functionalities. In the executable viewpoint, it is possible to specify details of each action by using ALF statements and define and instantiate elements. An ALF engine interprets the executable instances to execute the architecture.

Figure 1 illustrates an excerpt from an architectural description in SysADL. Figure 1(a) specifies the *ClientPT* port as a composite of the input port *aC* and the *qC* output port, as well as the *ServerPT* port as a composite of the *aS* input port and the *qS* output port. In SysADL, ports must be defined before being used in the definition of components and connectors.

Figure 1(b) depicts the definition of the *ClientServerCN* connector that uses instances of the previously defined *ClientPT* and *ServerPT* ports. The corresponding IBD specifies the internal configuration of this connector as using the *ClientServerQueryCN* and *ClientServerAnswerCN* connectors to link the *cP* and *sP* composite ports. Figure 1(c) shows the definition of the *ClientServerARCH* architecture as a composition of instances of the previously defined *ClientCP* and *ServerCP* components and *ClientServerCN* connector.

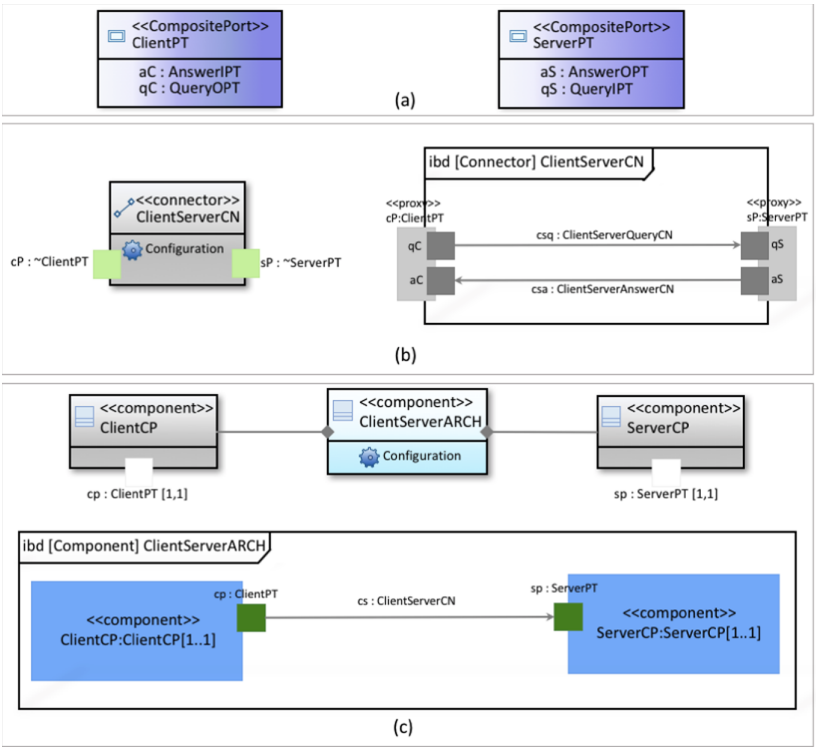

Fig. 1. Excerpt of a client-server architecture as described in SysADL.

2.2 π-ADL

π-ADL [16] is a formal architectural language based on the π-calculus process algebra to describe software architectures under both structural and behavioral viewpoints. The structural viewpoint describes a software architecture in π-ADL, through *components*, *connectors*, and their *composition* to form an architecture. In terms of behavioral viewpoint, both components and connectors comprise a behavior that expresses an architectural element's interaction and internal calculation and uses connections to connect and transmit values.

Figure 2 shows a part of the specification of an architectural description in π-ADL. Figure 2(a) specifies the *ClientServerCN* connector with the *aC* and *qS* input connections and the *qC* and *aS* output connections, followed by their protocol declaration. The behavior of the connector is defined as a composition that instantiates the *ClientServerQueryCN* and *ClientServerAnswerCN* connectors along with unifications that specify the information flow through connections.

Figure 2(b) shows the specifications of the *ClientCP* and *ServerCP* components. The behaviors of *ClientCP* and *ServerCP* components were specified as unobservable, which expresses an internal behavior not externally observable. In Fig. 2(c), the *ClientServerARCH* architecture is shown as a composition of

instances of the *ClientCP* and *ServerCP* components and the *ClientServerCN* connectors, along with the unifications that bind these architectural elements.

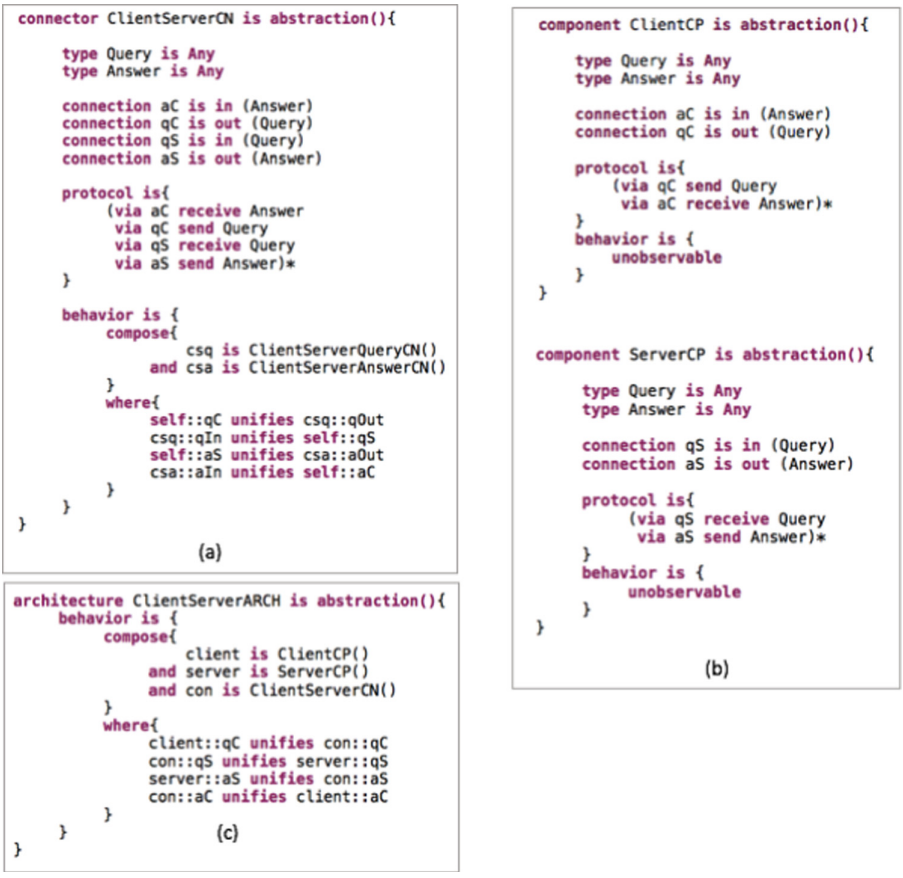

Fig. 2. Excerpt of a client-server architecture as described in π-ADL.

2.3 Model-to-Model Transformation Process

The Model-Driven Engineering (MDE) approach relies on models as central elements to software development. One of the keys to this approach is performing model-to-model (M2M) transformations [15], which consist in specifying transformation rules that make it possible to produce target models from source models. Figure 3 illustrates a model transformation process. Transformation rules are responsible for define how to map elements of the source model to corresponding ones in the target model.

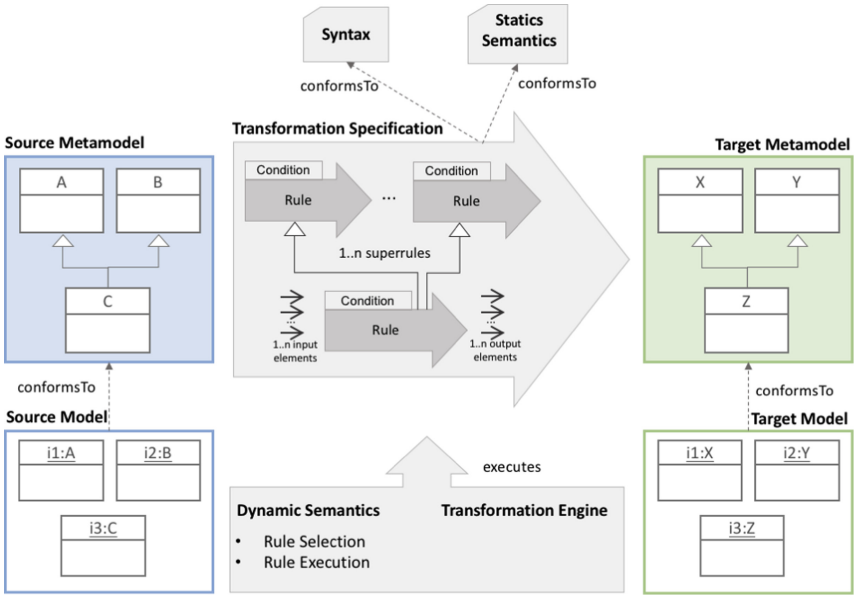

Fig. 3. A model-to-model transformation pattern (adapted from Wimmer et al. [22]).

Both source and target models must conform to their respective metamodels since the transformation rules are based on the abstract syntax of the metamodels. The transformation specification must also comply with the syntax and semantics of the language used for implementing the transformation. This work uses the SysADL and π-ADL metamodels as source and target metamodels, respectively. The goal is transforming an input architecture description in SysADL (conformed to its metamodel) and producing an architecture description in π-ADL (conformed to its metamodel) as output. The mapping process from SysADL to π-ADL and transformation algorithms are shown in Sect. 3.

3 Mapping SysADL Architecture Descriptions to π-ADL

This section describes how to generate π-ADL architecture descriptions from the description of SysADL structural elements. Section 3.1 describes the correspondences between the elements of SysADL and π-ADL. Section 3.2 describes some algorithms to transform an architecture description in SysADL into another one in π-ADL. Section 3.3 presents a process used to perform the translation between the models. Section 3.4 illustrates the proposal by specifying part of the architecture of the Automated Guided Vehicle (AVG) system in SysADL and showing how to translate it to π-ADL.

3.1 Correspondences Between SysADL and π-ADL

Table 1 summarizes the relationships between the main elements of SysADL and π-ADL, each one detailed as follows.

Table 1. Correspondences summary between SysADL and π-ADL elements.

SysADL	π-ADL
Component	Component
Connector	Connector
Architecture	Architecture
Port	Connection declarations
Primitive value types	Basic types
Value type	Any type
Data type	View type
Configuration	Behavior (composition)
Connector binding	Connection unification
Delegation	Connection unification

Component. In SysADL, components are structural elements that represent system functionalities and can be instantiated by other components or an architecture. A simple component performs computation using data available in its ports. A composite component performs computation through the constituent components in their internal structure. In π-ADL, components are created as an abstraction that can be instantiated within the architecture's specification.

Connector. Connectors are responsible for the interaction among components by defining which ports can be connected and how the interaction between connected components occurs. A connector can be either a simple element connecting two or more ports or a composite element embedding other connectors and behavior on its own. In π-ADL, connectors manage interactions among components with constituent elements and/or an internal behavior.

Architecture. The main element of an architecture description is the configuration that specifies the architecture itself. In SysADL, an architecture has a configuration that indicates its structural organization in terms of components and connectors. In π-ADL, an architecture is specified as a composition of component and connector instances.

Port. A SysADL port is an interaction point between a component and other architectural elements. It represents how data flow out from an output port or flow in to an input port. A composite port is composed of multiple port instances. The concept of port is not directly provided in the π-ADL version we have used in this work, but it is supported through the concept of connection

representing interaction points as well as binding between interaction points. In this way, data flow by typed connections representing the interaction points of communication channels used to transmit (send/receive) values. Protocols ensure that value types to be transmitted via connections comply with their respective declarations and the order of operations to perform.

Data Types. Both SysADL and π-ADL have four primitive data types, namely Integer, Real, String, and Boolean. These primitive data types are mapped exactly to their counterparts. Composite data types from SysADL are mapped to the View constructed type provided by π-ADL, which is composed of labeled field data. A value type from SysADL is mapped to π-ADL according to its intensity type, and is generally typed as the union *Any* type.

Configuration. The SysADL configuration refers to the structural organization of the architecture on how component and connector instances are linked to each other. A configuration can represent either the structure of a composite component or the overall software architecture. In π-ADL, a composition behavior expresses how independent sub-behaviors in parallel can form a composite behavior associated with the architecture or the composite element. It is possible to instantiate components and connectors to represent either the structure of architectural elements or the general architecture.

Connector Binding. Binding component instances does not happen directly in the sense that there must be a connector instance (linking component–connector ports) between two components. In π-ADL, a component's connection can be attached to a connector's connection to enable these elements to communicate, expressed through the unification of connections.

Delegation. In a SysADL configuration, ports of internal components are not visible to the external composite component. When port data must be visible, the port is linked to a proxy port via a binding connector expressed by delegation. Each port must be of the same data type and direction. In the transformation to π-ADL, SysADL delegations are converted to connection unifications by using the `self` clause to refer to the current component.

3.2 Transformation Rules from SysADL to π-ADL

An M2M transformation can be accomplished by implementing the mapping rules in a transformation language. This section presents a transformation algorithm in pseudocode, independently from the transformation language used to implement it.

Algorithm 1 shows the starting point of the transformation, i.e., the translation of the root element of a SysADL architecture description (*Model*) into the root element of a π-ADL architecture description (*ArchitectureDescription*). The *RuleModel2Architecture* procedure receives *SysADL!Model* as input and produces *PIADL!ArchitecturalDescription* as a result. The *CONS*, *COMPS*, and *ARCHS* sets (lines 3–5) respectively hold all connector, component, and architecture instances from the *SysADL!Model*. The *archElements* attribute of the

Algorithm 1. SysADL to π-ADL main transformation rule

```
1: procedure RuleModel2Architecture(SysADL!Model)
2:     ad : PiADL!ArchitectureDescription
3:     CONS ← ⋃ CN ∈ SysADL!ConnectorDef
4:     COMPS ← ⋃ CP ∈ SysADL!ComponentDef
5:     ARCHS ← ⋃ AC ∈ SysADL!ArchitectureDef
6:     ad.archElements ← ∅
7:     for each CN ∈ CONS do
8:         conn ← RuleConnectorDef2Connector(CN)
9:         ad.archElements ← ad.archElements ∪ conn
10:    for each CP ∈ COMPS do
11:        comp ← RuleComponentDef2Component(CP)
12:        ad.archElements ← ad.archElements ∪ comp
13:    ad.archs ← ∅
14:    for each AC ∈ ARCHS do
15:        arch ← RuleArchitectureDef2Architecture(AC)
16:        ad.archs ← ad.archs ∪ arch
17:    ad.behavior ← UNOBSERVABLE
18:    return ad
```

π-ADL architecture description element must contain the definition of all architectural elements, namely connectors and components. The procedure iterates over the sets of existing components and connectors from the SysADL model and respectively transform them into component and connectors in π-ADL according to the *RuleComponentDef2Component* and *RuleConnectorDef2Connector* auxiliar procedures (lines 7–12). The *archs* attribute of the π-ADL architecture description element must contain the definition of architectures. The last loop (lines 14–16) iterate over the set of architecture definitions from the SysADL model and transform them into the corresponding π-ADL element through the *RuleArchitecturerDef2Architecture* auxiliar procedure. The definition of these auxiliar procedures and an implementation of them are available at https://bit.ly/3huzjTe.

3.3 Mapping Process

Figure 4 illustrates the technical process to generate an architecture description in π-ADL from a SysADL model. The proposed transformation is supported by tools and artifacts complying with the Xtext Eclipse framework, namely (i) SysADL Studio, the tool that supports architectural modeling and execution of SysADL models [12], and (ii) the π-ADL toolchain, which performs syntactic and semantic analysis of architecture descriptions in π-ADL [5]. This infrastructure allows verifying if an architecture description in SysADL is correct according to the syntactic rules of the language and automatically generates a π-ADL file according to the ATL transformation rules. ATL has been chosen as a transformation language due to its maturity, tool support, and community support.

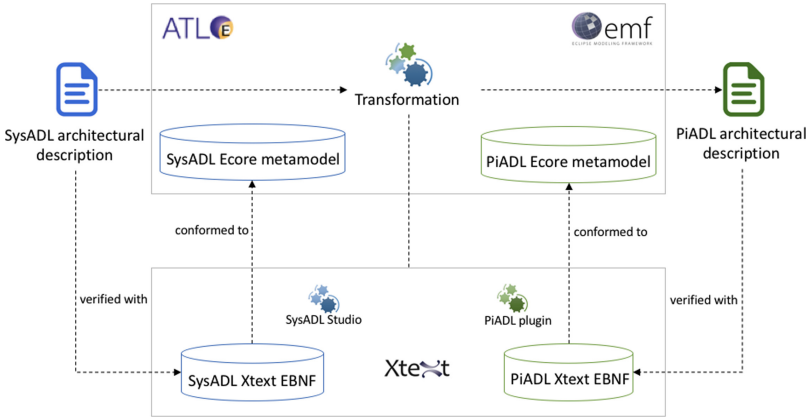

Fig. 4. Process for generating a π-ADL architecture description from a SysADL model.

The SysADL and π-ADL Ecore metamodels are parameters to the ATL transformation. The ATL Module checks for semantic errors in the transformation while the ATL compiler is automatically called for each ATL file, thus compiling transformations to the ASM assembly language [1]. ASM bytecode is then executed by the ATL Virtual Machine, which is specialized in handling models and provides a set of instructions for model manipulation.

3.4 An Illustrative Example: The AVG System

In this section, our proposed approach is illustrated through the Automated Guided Vehicle (AVG) system [9], showing how its architecture description in SysADL can be automatically translated to a corresponding specification in π-ADL by following the correspondences drawn in Sect. 3.1. AGV is a real-time system composed of a *motor*, which is commanded to start/stop moving and sends started/stopped responses, and an *arrival sensor* to detect when AVG has arrived at a station. If the station is the destination, then the AGV stops; otherwise, it continues to the next station. Moreover, there is a *robot arm* for loading and unloading a part onto and off of the AGV. The AGV system interacts with two other existing systems by receiving commands from the *Supervisory System* and sending vehicle acknowledgments to it, as well as sending the vehicle status to an external *Display System*.

Figure 5 shows an IBD corresponding to the configuration the *IAGVSystemCP* component in SysADL. This configuration is defined as the connection of the instances of *RobotArm*, *ArrivalSensor*, *VehicleControl*, and *Motor* components via instances of *commandArm*, *locationVehicle*, *commandMotor*, and *notificationMotor* connectors. The *in_outData* composite port and the *sendStatus* output port are connected to the *VehicleControl* component instance.

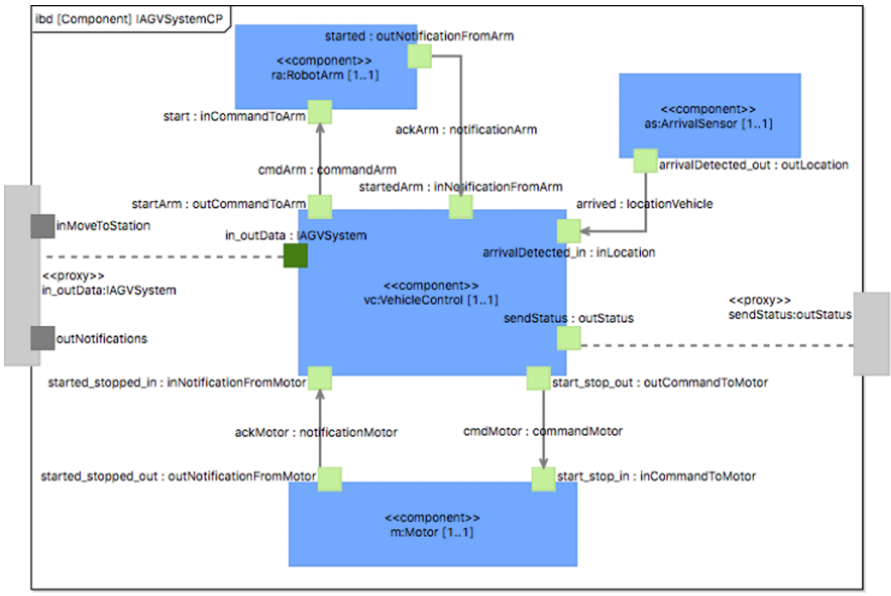

Fig. 5. Configuration of the *IAGVSystemCP* component in SysADL.

Listing 1.1 shows an ATL implementation of the transformation of component definitions in SysADL to component declarations in π-ADL, through the *lazyComponentDef2Component* rule. The attribute mapping of these elements is shown in lines 6–19. Except for the *name* attribute, all attributes are mapped by calling other ATL rules.

The result of the transformation is an architecture description in π-ADL, such as the excerpt shown in Listing 1.2. The *IAGVSystemCP* component ports have been transformed into connections with their respective protocol (lines 2–10). The configuration was transformed into a composition behavior where components (lines 13–16) and connectors (lines 17–21) were instantiated with their bindings (lines 23–32). Finally, connections involving the component ports (delegations in SysADL) were transformed into unifications (lines 33–35) with the use of the *self* clause. Once the transformation to π-ADL has been realized, it is possible to formally verify software architecture properties through the π-ADL toolchain [5,20]. The complete architecture description of the AVG system in SysADL and the resulting one in π-ADL are available at https://bit.ly/3huzjTe.

4 Evaluation

The effective adoption of formal verification approaches by the software industry faces significant barriers, such as the scalability of solutions and the high learning curve of the mathematical formalisms inherent to the techniques that are not part of stakeholders' routines. In particular, scalability is critical when

```
lazy rule lazyComponentDef2Component {
  from
    sSC: SYSADL!ComponentDef (not sSC.oclIsTypeOf(SYSADL!
        ArchitectureDef))
  to
    pC: PIADL!Component (
      name <- (if(sSC.isBoundary=false) then sSC.name else
          'boundary' + sSC.name endif),
      typeDecl <- typeDecs -> iterate(tdecl; tdecls:
          OrderedSet(PIADL!TypeDeclaration) = OrderedSet{}
           | tdecls.including(thisModule.lazyTuple2TypeDec
          (tdecl))),
      connections <- sSC.ports -> iterate(con; cons:
          Sequence(PIADL!ConnectionDeclaration) = Sequence
          {} | cons.including(
      if (con.definition.oclIsTypeOf(SYSADL!
          CompositePortDef)) then
        con.definition.ports -> iterate(port; ports:
            Sequence(PIADL!ConnectionDeclaration) =
            Sequence{} | ports.including(thisModule.
            lazyCPortUse2ConDec(port)))
      else
        thisModule.lazySPortUse2ConDec(con)
      endif)),
    protDecl <- thisModule.lazyComp2ProtocolDec(sSC),
    behavior <- if (sSC.isBoundary = true or sSC.composite
        .oclIsUndefined()) then
      thisModule.lazyUnobservable2BehavDec(PIADL!
          Unobservable)
      else
        thisModule.lazyCDefConfig2CBehavDec(sSC)
      endif
  )
}
```

Listing 1.1. An ATL rule that implements the transformation of component definitions in SysADL into component declarations in π-ADL.

analyzing software architectures in industrial contexts since software systems are becoming even more complex with an increasing number of components and connectors. Therefore, evaluating the proposed transformation of SysADL architecture descriptions into π-ADL descriptions concerning scalability is a key point for supporting industrial applications.

The evaluation considered a set with six software architectures descriptions in SysADL. The smallest description had 10 elements and the largest one had 85 elements (see Table 2). In the transformation from SysADL to π-ADL, there is a significant increase in the rule's complexity when dealing with composite elements, such as ports, connectors, and components.

The ATL transformation was performed ten times for each test architecture and the observed execution time was registered. The experiments were performed on a computer with an Intel Core i5 1.6 GHz processor, 8 GB of RAM, and macOS High Sierra version 10.136 as the operating system. Table 3 shows the descriptive statistics of the experiment results.

```
component IAGVSystemCP is abstraction() {
  connection sendStatus is out Status
  connection inMoveToStation is in VehicleData
  connection outNotifications is out
      NotificationToSupervisory

  protocol is {
    (via sendStatus send Status
     via inMoveToStation receive VehicleData
     via outNotifications send NotificationToSupervisory)*
  }
  behavior is {
    compose {
      m is Motor()
      and aS is ArrivalSensor()
      and ra is RobotArm()
      and vc is VehicleControl()
      and arrived is locationVehicle()
      and ackArm is notificationArm()
      and cmdArm is commandArm()
      and ackMotor is notificationMotor()
      and cmdMotor is commandMotor()
    } where {
      aS::arrivalDetected_out unifies arrived::lOPT
      arrived::lIPT unifies vc::arrivalDetected_in
      ra::started unifies ackArm::naOPT
      ackArm::naIPT unifies vc::startedArm
      vc::startArm unifies cmdArm::caOPT
      cmdArm::caIPT unifies ra::start
      m::started_stopped_out unifies ackMotor::nmOPT
      ackMotor::nmIPT unifies vc::started_stopped_in
      vc::start_stop_out unifies cmdMotor::cmOPT
      cmdMotor::cmIPT unifies m::start_stop_in
      vc::sendStatus unifies self::sendStatus
      self::inMoveToStation unifies vc::inMoveToStation
      vc::outNotifications unifies self::outNotifications
    }
  }
}
```

Listing 1.2. Description of the *IAGVSystemCP* component in π-ADL.

The Shapiro-Wilk test was applied to verify if data followed a normal distribution at a significance level $\alpha = 0.05$. After confirming the sample normality (p-values $> \alpha$), the Pearson correlation coefficient r was calculated to statistically measure the degree of correlation between the amount of simple/composite structural elements and the execution of the ATL transformation from SysADL to π-ADL. The obtained values were $r = 0.9627$ for simple elements and $r = 0.9383$ for composite elements, thus indicating a strong correlation between the number of elements and the execution time in both cases (see Table 3).

Another important observation was that the growth trend of the execution time is polynomial. Equation 1 and Eq. 2 respectively represent the obtained polynomials for simple and composite elements, which were also used to estimate the execution times shown in Table 4. It is possible to observe that the trans-

Table 2. Architecture descriptions considered in the experiments.

Architecture description	Types	Ports		Connectors		Components		Architectures	Total
		S	C	S	C	S	C		
A1	5	12	2	6	1	6	2	1	35
A2	13	16	2	8	1	11	2	1	54
A3	12	8	0	4	0	7	3	1	35
A4	5	2	0	0	0	2	0	1	10
A5	5	4	0	2	0	2	0	1	14
A6	20	24	6	11	3	17	3	1	85

S = simple element, C = composite element

Table 3. Descriptive statistics and correlation analysis of execution time measures (in seconds).

Architecture description	Simple elements	Composite elements	Median	Standard deviation
A1	24	6	0.0365	0.0116
A2	35	6	0.1015	0.0610
A3	19	2	0.0505	0.0088
A4	4	1	0.0045	0.0010
A5	8	1	0.0065	0.0025
A6	52	13	0.2110	0.0504
p-value (Shapiro-Wilk)	0.7699	0.2439		
Pearson coefficient r	0.9627	0.9383		

formation of an architecture containing only composite elements takes twice the execution time of architecture with only simple elements. When the architecture has thousands of elements, the execution time increases up to approximately 10 h, even considering an architecture with only simple elements.

$$y = 7E{-}05x^2 + 0.0003x + 0.0024, R^2 = 0.9841 \quad (1)$$

$$y = 0.0007x^2 + 0.007x + 0.0075, R^2 = 0.9011 \quad (2)$$

Besides scalability, the approach proposed in this paper aimed to tackle the inherent complexity of adopting formal architectural languages. The use of a SysML-based language such as SysADL and the automated transformation to a formal notation (in this case, π-ADL) have the potential of alleviating the learning curve typically required by a formal verification strategy. In this perspective, software architects could benefit from the proposed approach in that they can use a well-known, standardized notation to produce architecture descriptions that can be further formally verified with respect to architectural properties.

5 Related Work

Enoiu et al. [8] introduced a way of integrating architectural models with verification techniques implemented through the ViTAL tool. The approach allows expressing functional behavior in the EAST-ADL architectural language for

Table 4. Estimated execution time in terms of number of simple/composite elements.

Elements	Execution time	
	Simple	Composite
10^1	0.04 s	0.15 s
10^2	3.53 s	7.71 s
10^3	5.84 min	11.78 min
10^4	9.72 h	19.46 h

automotive embedded systems as timed automata models, which have precise semantics and can be formally verified. The proposal uses model-based techniques to allow for the automated transformation between different design models. In this perspective, the ViTAL tool allows transforming functional EAST-ADL models into the UPPAAL PORT tool to support model verification.

Taoufik et al. [21] proposed using model transformations to translate UML 2.0 architecture descriptions into Wright, a formal CSP-based architectural language, towards verifying the behavioral consistency of software architectures. The Wright descriptions are automatically translated into a CSP specification acceptable to the FDR2 model checker through the Wr2fdr tool.

Maraoui and Cariou [14] proposed a strategy for the formal verification of the composition of Web services through the integration of a mediation protocol. The approach provides an MDE process that guides developers through a series of transformations between models, including SysML, to obtain a formal code in the ACME architectural language (with Armani) towards verification.

Similar to this work, the studies previously mentioned define their approaches of assigning formal semantics to an architectural language by applying MDE techniques, but their concerns are limited to behavioral aspects only or are specific to an application domain. Oppositely, the approach presented in this paper deals with a general-purpose SysML-based architectural language that allows describing software architectures under both structural and behavioral viewpoints, independently from the application domain or architectural style.

6 Concluding Remarks

Formally expressing architecture descriptions in critical domains allows for automated verification of architectural properties of software-intensive systems. This paper presented the definition of a formal semantics for a SysML-based ADL, SysADL, through model transformations to a formal language based on the π-calculus process algebra, π-ADL. As π-ADL supports the formal verification of architectural properties, SysADL specifications transformed to π-ADL can be verified with tools available for this formal language. Note that the transformation models from SysADL to π-ADL are based on the denotational semantics of SysADL in terms of π-ADL.

The transformation from SysADL to π-ADL was implemented using ATL to validate the proposed approach. Furthermore, some experiments were performed to analyze the performance of the transformation and estimate its scalability. Obtained results showed a strong correlation between the number of simple and composite elements and the time spent to perform the transformation.

Ongoing work aims to extend the transformation from a behavioral viewpoint and compare the performance results of the ATL implementation with other transformation languages, such as QVTo [3] and Kermeta [7], to check for significant performance differences. Furthermore, the SysADL Studio tool is being integrated with the π-ADL toolchain to include, in addition to the automatic generation of π-ADL architecture descriptions, also their automatic formal verification of architectural properties, in particular to guarantee correctness.

References

1. ATL Developer Guide. https://wiki.eclipse.org/ATL/Developer_Guide
2. Araujo, C., Cavalcante, E., Batista, T., Oliveira, M., Oquendo, F.: A research landscape on formal verification of software architecture description. IEEE Access **7**, 171752–171764 (2019)
3. Barendrecht, P.: Modeling transformations using QVT Operational Mappings. Eindhoven University of Technology, The Netherlands (Apr), Tech. rep. (2010)
4. Cavalcante, E., Batista, T., Oquendo, F.: Supporting dynamic software architectures: from architectural description to implementation. In: 12th Working IEEE/IFIP Conference on Software Architecture, pp. 31–40. IEEE, USA (2015)
5. Cavalcante, E., Quilbeuf, J., Traonouez, L.-M., Oquendo, F., Batista, T., Legay, A.: Statistical model checking of dynamic software architectures. In: Tekinerdogan, B., Zdun, U., Babar, A. (eds.) ECSA 2016. LNCS, vol. 9839, pp. 185–200. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48992-6_14
6. Dias, F., et al.: Empowering SysML-based software architecture description with formal verification: from SysADL to CSP. In: Jansen, A., et al. (eds.) ECSA 2020, LNCS, vol. 12292, pp. 101–117. Springer, Switzerland (2020)
7. Drey, Z., Faucher, C., Fleurey, F., Mahé, V., Vojtisek, D.: Kermeta language - Reference Manual (2009). http://www.kermeta.org/docs/KerMeta-Manual.pdf
8. Enoiu, E.P., Marinescu, R., Seceleanu, C., Pettersson, P.: ViTAL: a verification tool for EAST-ADL models using UPPAAL PORT. In: 17th International Conference on Engineering of Complex Computer Systems, pp. 328–337. IEEE, USA (2012)
9. Gomaa, H.: Software modeling and design: UML, use cases, patterns, and software architectures. Cambridge University Press (2011)
10. ISO/IEC/IEEE 42010: Systems and Software Engineering - Architecture Description. ISO, Switzerland (2011)
11. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.M. (ed.) Satellite Events at the MoDELS 2005 Conference, LNCS, vol. 3844, pp. 128–138. Springer, Berlin (2006)
12. Leite, J., Batista, T., Oquendo, F., Silva, E., Santos, L., Cortez, V.: Designing and executing software architectures models using SysADL studio. In: 2018 IEEE International Conference on Software Architecture Companion, pp. 81–84. IEEE, USA (2018)

13. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6), 869–891 (2013)
14. Maraoui, R., Cariou, E.: A mediation based approach for formal verification of web services composition. In: 2017 International Conference on Engineering & MIS. IEEE, USA (2017)
15. Mens, T.: Model transformation: a survey of the state of the art. In: Babau, J.P., et al. (eds.) Model-Driven Engineering for distributed real-time systems, pp. 1–19. ISTE Ltd/John Wiley & Sons Ltd, United Kingdom/USA (2013)
16. Oquendo, F.: π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Software Engineering Notes **29**(3), 1–14 (2004)
17. Oquendo, F., Leite, J., Batista, T.: Software Architecture in Action: Designing and executing architectural models with SysADL grounded on the OMG SysML Standard. UTCS, Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44339-3
18. Ozkaya, M.: The analysis of architectural languages for the needs of practitioners. Softw. Pract. Experience **48**(5), 985–1018 (2018)
19. Ozkaya, M.: Do the informal and formal software modeling notations satisfy practitioners for software architecture modeling? Inf. Softw. Technol. **95**, 15–33 (2018)
20. Quilbeuf, J., Cavalcante, E., Traonouez, L.-M., Oquendo, F., Batista, T., Legay, A.: A Logic for the Statistical Model Checking of Dynamic Software Architectures. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 806–820. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_56
21. Taoufik, S.R., Tahar, B.M., Mourad, K.: Behavioral verification of UML2.0 software architecture. In: 12th International Conference on Semantics. Knowledge and Grids, pp. 115–120. IEEE, USA (2016)
22. Wimmer, M., et al.: Surveying rule inheritance in model-to-model transformation languages. J. Object Technol. **11**(2), 1–46 (2012)