UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO
DOUTORADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

Enriching SysML-Based Software Architecture Descriptions: A Model-Driven Approach

Camila de Araújo

Natal-RN, Brasil
2023

Camila de Araújo

Enriching SysML-Based Software Architecture Descriptions: A Model-Driven Approach

Tese de Doutorado apresentado ao Programa de Pós-Graduação em Sistemas e Computação do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Engenharia de Software

Supervisor: Thais Vasconcelos Batista

Natal-RN, Brasil
2023

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Araújo, Camila de.
Enriching SysML-based software architecture descriptions: a model-driven approach / Camila de Araújo. - 2023.
174f.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Programa de Pós-Graduação em Sistemas e Computação. Natal, RN, 2023.
Orientação: Profa. Dra. Thais Vasconcelos Batista.
Coorientação: Prof. Dr. Everton Ranielly de Sousa Cavalcante.

1. Computação - Tese. 2. Software architecture - Tese. 3. Architectural description language - Tese. 4. Model-driven development - Tese. 5. Formal verification - Tese. 6. SysADL - Tese. 7. Pi-ADL - Tese. I. Batista, Thais Vasconcelos. II. Cavalcante, Everton Ranielly de Sousa. III. Título.

RN/UF/CCET CDU 004

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

CAMILA DE ARAÚJO

"Enriching SysML-Based Software Architecture Descriptions: A Model-Driven Approach"

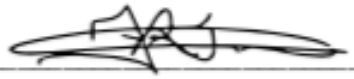
Esta Tese foi julgada adequada para a obtenção do título de Doutor(a) em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Prof. Dr. NÉLIO ALESSANDRO AZEVEDO CACHO
Coordenador do PPgSC

Banca Examinadora

Documento assinado digitalmente
gov.br EVERTON RANIELLY DE SOUSA CAVALCANT
Data: 03/04/2023 18:07:15-0300
Verifique em https://validar.iti.gov.br

Examinador(a) Interno(a): **Dr. EVERTON RANIELLY DE SOUSA CAVALCANTE**



Examinador(a) Externo(a): **Dr. FLAVIO OQUENDO**

Documento assinado digitalmente
gov.br LUCAS BUENO RUAS DE OLIVEIRA
Data: 31/03/2023 23:37:08-0300
Verifique em https://validar.iti.gov.br

Examinador(a) Externo(a): **Dr. LUCAS BUENO RUAS DE OLIVEIRA**

MARCEL VINICIUS MEDEIROS OLIVEIRA:02386943488
Assinado de forma digital por MARCEL VINICIUS MEDEIROS OLIVEIRA:02386943488
Dados: 2023.04.01 13:02:28 -03'00'

Examinador(a) Interno(a): **Dr. MARCEL VINICIUS MEDEIROS OLIVEIRA**

Documento assinado digitalmente
gov.br THAIS VASCONCELOS BATISTA
Data: 31/03/2023 13:46:38-0300
Verifique em https://validar.iti.gov.br

Presidente: **Dr.ª THAIS VASCONCELOS BATISTA**

Documento assinado digitalmente
gov.br CAMILA DE ARAUJO SENA
Data: 03/04/2023 20:22:00-0300
Verifique em https://validar.iti.gov.br

Discente: **CAMILA DE ARAÚJO**

Natal, 31 de março de 2023

To my mother, Chagas (in memoriam), and grandmother, Antonia (in memoriam),
strong women who founded my path until here.
To Sofia, my daughter, who inspires me to become a better person every day.
To Demóstenes, my husband, who holds my hand every step of the way.

Acknowledgements

"No man is an island, entire of itself; every man is a piece of the continent, a part of the main" says the quote from John Donne's poetry, recognizing that human life is built of conviviality, social relationships, and why not affections?! I often say that the Ph.D. process is very lonely because no one is exactly on the same boat, traveling the same path as us. However, this often reminds us how much people are necessary and how they can make our journey lighter and more pleasant. I was fortunate to have good friends on this journey, and I would like to thank everyone who contributed in some way to this process and who was with me, showing that happiness is also found in the journey, not just in the destination.

I thank God, who sustains me when my weaknesses manifest themselves and allows me to live this journey and achieve my goals, making me victorious even in the face of my wavering faith.

I thank my mother, Chagas (in memoriam), my grandparents Antônia (in memoriam) and Francisco (in memoriam), and my aunts Cacá, Dalva, and Nevinha, who spared no effort in providing me with a good education, imparting values such as ethics, dedication, discipline, and respect. These teachings guide my life and accompany me wherever I go. You are the voice in my conscience that directs me on every path.

I thank my daughter Sofia and my husband Demóstenes, who did not choose to go through the Ph.D. process, but had to sacrifice many important things during this journey of mine. We cannot turn back time and experience the moments we missed in these past years, but I promise that I will do my best to ensure that the moments we share from now on are unique. I will dedicate quality time to our family, and together, we will create many joyful memories.

To my supervisor, Professor Thais Batista, I am grateful for all the support you have provided throughout this journey. You have gone above and beyond in offering me assistance, not just in academic matters. Thank you for your availability, patience, care, advice, and friendship. I have learned so much from you. Thank you for everything.

I thank Professor Everton Cavalcante, my co-supervisor, for his assistance and close monitoring of this work. The dedication and rigor with which he carries out his duties inspire me.

I would like to express my gratitude to Professor Flávio Oquendo for his significant contributions to the progress of this work. His valuable advice and precise suggestions at various stages of its development are deeply appreciated.

I extend my thanks to professors Lucas Bueno and Marcel Oliveira for accepting the invitation to be a part of the examination committee and for their valuable contributions in enhancing this thesis. I would like to particularly thank Professor Marcel for

his close assistance with the denotational semantics of SysADL, dedicating many hours of his time to clarify my doubts and correct function definitions.

I am grateful to Eduardo Silva and Lidiane Santos for their assistance with SysADL and SysADL Studio. The knowledge and guidance you provided were instrumental and crucial to the success of this work.

I express my gratitude to my study colleagues, who were essential companions during the initial disciplines of the doctoral program: Guilherme Freitas, Francimaria Santos, Roberta Cintia, Douglas Rolim, Cesar Perdigão, Stefano Loss, and Rafael Jullian. Your presence was vital in navigating the stormy seas at the beginning of this journey!

I want to thank my friends for our diverse and spontaneous conversations: Bartira Rocha, Chico Dantas, and Isaac Oliveira. Your support, coffee breaks (both in person and remotely), engaging conversations, and friendship have made everything better. Thank you for always being there!

To my dear friend Lyrene Fernandes, I am grateful for our enjoyable conversations that span from philosophical subjects to the most delightful trivialities. These shared moments provided an essential escape from the tensions of this process. Thank you for everything!

Thanks to my friends at the Department of Computer Science at UERN, who provided tremendous support and shared my workload, allowing me to solely focus on my doctorate for a period of time. I truly appreciate their support.

I extend my gratitude to Maihana Cruz, psychologist, for her invaluable help, teachings, and care for my mental health throughout this process. Your support played a pivotal role in reaching a successful conclusion. Thank you!

Finally, I would like to thank all my family and friends, who supported me and have been an integral part of this process.

"Still lying underneath the stormy skies,
She said: Oh, I know the sun must set to rise!"
Coldplay

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 support for architectural analysis, particularly formal verification. This work investigates how to formally support SysML-based architecture descriptions to enable the formal verification of software architectures. As a result of this research, the main contribution is proposing a model-driven approach (MDD) that provides formal semantics to a SysML-based architectural language, SysADL, through a seamless transformation of SysADL architecture descriptions to the corresponding formal specifications in π-ADL, a well-founded theoretically language based on the higher-order typed π-calculus. The proposal implementation involves the execution of a four-phase process: (i) Model-to-Model (M2M) transformation of SysADL models into π-ADL model; (ii) Model-to-text (M2T) transformation of π-ADL models into π-ADL source; (iii) corresponding executable architecture generation, and architecture validation; and (iv) property verification. The work has other associated contributions to support the 4-phase process: (i) a denotational semantics to SysADL in π-ADL; (ii) a definition of a process to support the automated transformation of SysADL models into π-ADL models; (iii) The validation of the π-ADL architecture generated by the MDD transformation to demonstrate that it is in accordance with the original SysADL architecture; and (iv) the verification of formal architectural properties analyzing execution traces. The proposal was implemented and validated using a Flood Monitoring System architecture.

Keywords: Software architecture, Architectural description language, Model-driven development, Formal verification, SysADL, π-ADL.

Abstract

A natureza crítica de muitos sistemas complexos de uso intensivo de software requer descrições formais de arquitetura para dar suporte à análise arquitetural automatizada em relação às propriedades de correção. Devido aos desafios de adotar abordagens formais, muitos arquitetos têm preferido usar notações como UML, SysML e seus derivados para descrever a estrutura e o comportamento das arquiteturas de software. No entanto, essas notações semi-formais têm limitações quanto ao suporte à análise arquitetural, principalmente à verificação formal. Este trabalho investiga como oferecer suporte formal a descrições de arquitetura baseadas em SysML para permitir a verificação formal de arquiteturas de software. Como resultado desta pesquisa, a principal contribuição é propor uma abordagem orientada a modelos (MDD) que fornece semântica formal para uma linguagem de arquitetura baseada em SysML, SysADL, por meio de uma transformação contínua das descrições da arquitetura SysADL para as especificações formais correspondentes em π-ADL, uma linguagem teoricamente bem fundamentada baseada no π-calculus de tipo de ordem superior. A implementação da proposta envolve a execução de um processo de 4 fases: (i) transformação modelo-a-modelo (M2M) de modelos SysADL em modelo π-ADL; (ii) transformação de modelo para texto (M2T) de modelos π-ADL em código π-ADL; (iii) geração de arquitetura executável correspondente e validação da arquitetura; e (iv) verificação de propriedades. O trabalho tem outras contribuições associadas para dar suporte ao processo de 4 fases: (i) uma semântica denotacional para SysADL; (ii) a definição de um processo para suportar a transformação automatizada de modelos SysADL em modelos π-ADL; (iii) A validação da arquitetura π-ADL gerada pela transformação MDD para mostrar que está de acordo com a arquitetura original SysADL; e (iv) a verificação de propriedades arquiteturais formais usando rastros de execução. A proposta foi implementada e validada utilizando uma arquitetura de Sistema de Monitoramento de Enchentes.

Palavras-chave: Arquitetura de software, Linguagens de descrição arquitetural, Desenvolvimento dirigido a Modelos, SysADL, π-ADL.

List of Figures

List of Tables

List of Algorithms

Listings

List of abbreviations and acronyms

ADL	Architecture Description Language
ALF	Action Language for Foundational UML
ATL	Atlas Transformation Language
BDD	Block Definition Diagrams
BLTL	Bounded Linear-time Temporal Logic
CML	COMPASS Modelling Language
CSP	Communicating Sequential Process
CTL	Computation Tree Logic
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
FDR	Failures Divergence Refinement
FMS	Flood Monitoring System
GCSL	Goal Contracts Specification Language
IBD	Internal Block Diagrams
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Organization for Standardization
LTL	Linear Temporal Logic
M2M	Model-to-Model
M2M	Model-to-Text
MDD	Model-Driven Development
MDE	Model-Driven Engineering
OMG	Object Management Group
PAT	Process Analysis Toolkit

PORT	Partial Order Reduction Technique
PoSM	Port State Machines
RML	Reactive Module Language
RTCA	Radio Technical Commission for Aeronautics
SMC	Statistical Model Checking
SMV	Symbolic Model Verifier
SysML	Systems Modelling Language
TCTL	Timed Computation Tree Logic
UFRN	Universidade Federal do Rio Grande do Norte
UML	Unified Modeling Language
XML	Extensible Markup Language

Contents

1 Introduction

Since its inception, software architecture has emerged, intending to provide a basic understanding of the large-scale structures of software systems. The advances in the study of software architectures have matured to encompass a broad set of analysis notations, tools, and techniques, offering concrete guidance for complex software design and development (SHAW; CLEMENTS, 2006).

The international standard ISO/IEC/IEEE 42010 (ISO/IEC/IEEE 42010, 2007) describes software architecture as the fundamental conception of a system in terms of its constituent elements and their relationships with each other and the environment, as well as the principles that guide the system design and evolution.

According to Georgas et al. (GEORGAS; DASHOFY; TAYLOR, 2006), good architectural practices, when applied throughout the lifecycle, can increase understanding of a software system and the development process used to create it, ensuring that qualities of particular relevance are met, and reduce the overall cost of software engineering.

The above-mentioned works converge to the fact that software architectures are essential in successfully developing software-intensive systems, playing a critical role in almost all aspects of the software development lifecycle as requirements specification, system design, implementation, reuse, maintenance, and runtime adaptation (GARLAN, 2003). This way, the description resulting from the architectural modeling activity can be used as an essential artifact to validate and achieve business objectives and quality requirements.

System Architectural Descriptions are work products of software architecture and are expressed in different ways by architects and developers (ISO/IEC/IEEE 42010, 2007), which may even use natural language documents or abstract "box and arrow" diagrams. While these informal forms of documentation are helpful in many ways, they often need more rigor and precision to comprehensively describe architectures.

According to Georgas et al. (GEORGAS; DASHOFY; TAYLOR, 2006), opting for an appropriate level of rigor and precision in architectural modeling is a key enabler of the value of architecture in activities other than design. Good models provide a basis for architectural analysis, understanding and communicating about the design of a system, and guide the evolution of a system.

The Architecture Description Languages (ADLs) are notations specifically designed to allow the rigorous and precise specification of software architectures, with semi-formal languages based on UML/SysML being a clear preference of the software architect community, according to the findings in the study of Malavolta (MALAVOLTA et al., 2013).

One of the advantages of adopting a formal ADL is the possibility of carrying

out activities involving analysis in software architectures, such as verifying architectural properties, that is, identifying essential properties of the system through architectural models even before its implementation (TAYLOR; MEDVIDOVIC; DASHOFY, 2010). This issue has been a challenge in the context of software-intensive systems. Performing such activities as early as possible is essential to avoid inaccuracies, inconsistencies, and undesirable issues that can propagate into later stages of the development process when a correction will undoubtedly be costly.

This is even more essential when dealing with the critical nature of many complex systems whose intended architecture must be verified concerning their correctness and fulfillment of required behavior and properties of interest, as prescribed in security standards for critical software, such as RTCA-DO178C for avionics (RTCA/DO-178C, 2012), IEC-62304 for medical systems (CORONATO, 2018), and IEC-62279 (IEC 62279:2015, 2015) for railway systems, which strongly recommend the adoption of formal verification techniques as a means to ensure the security of these systems.

Considering the importance of formal verification in the context of software architecture as a way to help ensure quality in current software systems, formal architectural descriptions are highly desirable as a means of better supporting automated architectural analysis, acknowledged as an important activity in the software industry (MALAVOLTA et al., 2013; OZKAYA, 2018b). The main advantage of adopting a formal approach is precisely determining if a software system can satisfy properties and constraints related to requirements and check the accuracy and correctness of architectural designs.

1.1 Problem Statement

In the context of software-intensive systems, software architecture modeling has been considered an essential activity in determining the success of such systems and contributing to their quality. This activity results in architecture descriptions, artifacts that can be used to communicate the software architecture among stakeholders, and support its maintenance, evaluation, and evolution. An architectural description is the central element of automated architectural analysis, enabling to accurately verify that the designed architectures can meet essential properties to the system.

In the literature, some studies focus on investigating the needs of the software industry concerning architectural languages (MALAVOLTA et al., 2013; OZKAYA, 2018a). Most users of architectural languages: (i) need to perform some analysis in architectural descriptions; (ii) prefer automatic analysis with tool support; (iii) become disinterested in the architectural analysis due to the abstraction of architectural descriptions and the complexity of their formal descriptions. Nonetheless, the relevance of formal architecture descriptions is well known, mainly for supporting automated architectural analysis.

Software architects generally prefer to use semi-formal notations such as Unified

Modeling Language (UML) (OMG, 2001), Systems Modeling Language(SysML) (OMG, 2001), or their derivatives to describe the structure and behavior of software architectures. This choice was primarily driven by the lower learning curve, visuality, and general-purpose scope. However, this type of notation has known limitations regarding architectural representation. SysADL (OQUENDO; LEITE; BATISTA, 2016) is a SysML-based architectural language that addresses this limitation by joining standard constructs of architectural languages and the popular diagrammatic notation based on the SysML Standard for modeling software-intensive systems. SysADL complies with the ISO/IEC/IEEE 42010 International Standard for architectural descriptions (ISO/IEC/IEEE 42010, 2007).

SysML-based notations have limitations regarding automatically checking architectural properties, lack of well-defined semantics, and gaps in complex design decisions. Formal approaches fill these gaps and better support architectural analysis. Among the formal software architecture languages, π-ADL (OQUENDO, 2004) is a formal language based on the π-calculus (MILNER, 1999) process algebra to specify dynamic software architectures. This work uses π-ADL as a formal ADL as it allows formally describing both structure and behavior of dynamic software architectures, unlike most existing architectural languages (CAVALCANTE, 2016). As a formal language, π-ADL suffers from the same problems of other formal ADLs: practitioners do not adopt them arguing the steep learning curve of formalisms (MALAVOLTA et al., 2013).

Previous work on a systematic literature review (ARAUJO et al., 2019) highlighted some significant lacks regarding the formal verification of software architecture descriptions, specially in the context of SysML-based architectures. Some of these lacks identified in the literature and addressed in this doctoral research are described in the following.

Lack of approaches proposing a seamless formal enrichment of non-formal ADLs. A formal enrichment is required for a non-formal language, such as SysADL, to support an architectural analysis process. Most existing studies in the area encourage the use of some formalisms to provide formal support and specify properties to be verified, such as temporal logic, process algebra, or other mathematical formalisms. However, formally describing software architectures is very complex, with a high learning curve for architects (MALAVOLTA et al., 2013). There is a gap concerning proposals with a seamless formal enrichment of non-formal languages, especially those that describe dynamic architectures, to avoid the need of learning formalisms.

Lack of support for the formally verifying properties in SysML-based languages enabling dynamic software architectures. The software industry has adopted SysML as a standard for describing software-intensive system architectures, especially for its simplicity and low learning curve compared to formal ADLs. Architects working with architecture modeling for critical systems are interested in performing different types of analysis in the architecture, preferably in an automatic way (MALAVOLTA et al., 2013). However, non-formal languages are not directly suitable for formal architectural

analysis.

The literature finds some approaches on associating the ease of UML-based or SysML-based descriptions with the possibility of formal property verification using some mathematical formalism (LIMA et al., 2017; OZKAYA; KOSE, 2018a), or a formal language (TAOUFIK; TAHAR; MOURAD, 2016). However, they often do not consider essential features such as general-purpose architectural languages handling formal specification and tool support.

Lack of tools for specifying and verifying properties on architectures in an integrated environment. Tools provide essential support to software architects as they contribute to advance the architecture design practice by supporting a syntactically correct model, reusing standard architectural elements, and executing simulation at design time. In the context of formal property verification of software architectures, a set of tools has an equally relevant supporting role since formal verification in manual settings is an extremelly hard and error-prone task, especially considering the size of current architectures that have thousands of components. Therefore, an important gap to be filled refers to providing integrated environments and tools, support mechanisms, and technologies able to transparently dealing with architectural description and verification of architectural properties.

1.2 Goals and Research Questions

This work focuses on the formal enrichment of a SysML-based architectural description language, aiming to reconcile the easy-to-use architectural modeling expressiveness of a SysML-based ADL (SysADL) with the formal support of an ADL (π-ADL), relieving the architect from the burden of explicitly dealing with formal specifications.

The main objective of this work is to define an approach to add formal semantics to SysADL in order to support formal property verification of the software architectural description. As presented below, we have outlined a set of research questions to help us achieve this goal.

RQ1: Can a Model-Driven Development (MDD)-based approach enable a seamless formal enrichment of SysADL models transforming them to π-ADL models?

This RQ investigates whether an MDD-based approach (via model-to-model transformation (M2M) and model-to-text transformation (M2T)) enables a seamless formal enrichment of SysADL. The idea is to establish a mapping process between SysADL and π-ADL, to define the correspondences between the elements of the languages. This MDD transformation based on the denotation semantics established between SysADL and π-ADL models ensures the semantic equivalence between the models. The main interest behind the idea of transforming an architecture description in SysADL to a corresponding one in π-ADL is enabling its further formal verification, which is already available for

π-ADL (CAVALCANTE et al., 2016).

RQ2: Is it possible to generate a valid executable architecture from the π-ADL model resulting from the MDD-based transformation of a SysADL model?

This RQ aims to investigate whether, from an MDD-based transformation of SysADL into π-ADL, it is possible to generate π-ADL architectures that are valid and, therefore, can be executed. The understanding that underlies this investigation is that when generating an architecture that can be executed, it is possible to show that it is correct by observing that the execution of its behavior follows what was predicted from the original SysADL architecture. From a less rigorous point of view, this fact would confirm that the models transformation is correct. Intending to answer this question, the plan is to automatically generate an executable architecture in Go Language (GOOGLE, 2009) from the architectural description in π-ADL generated from the initial model in SysADL. Go is a general-purpose programming language designed to address building scalable distributed systems and handling multicore and networked computer architectures as required by current systems. Go is a natural choice of implementation language for π-ADL architectures because both are based on the same underlying formalism, π-calculus. These similarities promote a direct relationship between the elements of these languages, easing the automatic generation of the Go source code from an architectural description in π-ADL, thus contributing to minimizing the risk of architectural deviations and allowing the validation of the architecture itself.

RQ3: Can an architectural property expressed from a SysADL model and its respective transformation into π-ADL be verified by equivalent traces of these models' execution?

This RQ investigates whether model transformation between SysADL and π-ADL models generates equivalent architectures. The idea is to observe if an architectural property can be verified in the SysADL and π-ADL architectures, respectively origin and target of the MDD-based transformation, achieving equivalent execution traces. In this way, together with verification by semantic inspection, it would be possible to show that the transformation is complete, correct and that it is possible to verify properties in the generated architecture.

1.3 Expected Contributions

The main contribution of this work is to define an MDD approach that provides a seamless transformation of SysADL architecture descriptions to the corresponding formal specifications in π-ADL, which can be formally verified. The proposed approach comes up with four main associated contributions, each one summarized in the following.

1. **A denotational semantics to SysADL** The first contribution brought by this work is the definition of the denotational semantics of SysADL using π-ADL, in-

tending to show the transformation of syntactic elements from SysADL to π-ADL is feasible. Through this process, SysADL would have a semantic formal.

2. **The definition of a process to support the automated transformation of SysADL models into π-ADL models.** The second contribution is subdivided into three smaller processes that complement each other to support the automatic transformation of an architectural description in a SysADL model to an architectural description in π-ADL. The three pieces of the process are explained below.

   a) *The definition of algorithms to transform SysADL models into π-ADL models.* The first piece to contribute to the proposed process is the definition in pseudocode of the algorithms that allow mapping the architectural elements of SysADL into elements of π-ADL, defined in accordance with the denotational semantics of SysADL as a function of π-ADL. From these algorithms, it is possible to implement a SysADL transformation using any model transformation language or another non-model-based strategy.

   b) *An automated process to model-to-model transformation (M2M) from SysADL to π-ADL.* The second part of contributing to model transformation concerns the implementation of algorithms for transforming the architectural elements of SysADL models into π-ADL models. The automated M2M transformation process between SysADL and π-ADL was implemented using the ATL transformation language in an Eclipse environment. As process output, there is an XML file containing a π-ADL model with the architectural description corresponding to the initial SysADL Model.

   c) *An automated process to execute automated transformation for π-ADL code generation from π-ADL model (M2T).* The third piece of this contribution is the automated process implemented to perform the M2T transformation of the model containing the architectural description π-ADL in XML format into a source code π-ADL. The transformation engine was implemented in the Acceleo model transformation language in an Eclipse environment. With this tool, the process of transforming a model containing an initial SysADL architectural description into a source file π-ADL is completed, containing the initial architecture transformed into π-ADL.

3. **The validation of the π-ADL architecture generated by the MDD transformation to show that it is in accordance with the original SysADL architecture**. The third contribution, focusing on validating the architecture resulting from the MDD transformation process from SysADL to π-ADL, is achieved through a process that involves four steps: (i) the automatic generation of source code in the Go programming from the resulting architectural description in π-ADL (CAVAL-

CANTE, 2016); (ii) the insertion of Go code to implement the functional behavior of the architecture; (iii) the compilation and the execution of the source code to execute the architecture; and (iv) the comparison of the execution results with respect to the definitions of the original SysADL architecture.

4. **The verification of formal architectural properties using execution traces.** The fourth contribution is the verification of formal architectural properties by observing the execution traces. By comparing the execution traces of the original SysADL model and the transformed model π-ADL, it is also possible to demonstrate the equivalence between them.

1.4 Thesis Outline

The remainder of this doctoral thesis is organized as follows. Chapter 2 establishes the basic concepts for understanding this work: software architecture, MDD Transformation, SysADL, and π-ADL. Chapter 3 presents the model-driven approach to transforming SysADL into π-ADL in two steps: an M2M transformation and an M2T transformation. Chapter 4 describes the process adopted to generate executable architecture in Go from the π-ADL model resulting from the MDD transformation. Chapter 5 presents verification simulations of architectural properties from the SysADL and π-ADL models, showing that the source and target architectures are equivalent. Chapter 6 presents some related work. Chapter 7 revisits the contributions achieved and presents perspectives for future work.

Three appendices are also provided as additional material. Appendix A details the architectural modeling in SysADL of the Flood Monitoring System (FMS), the running example of the thesis. Appendix B presents the SysADL and π-ADL grammatical specification. Appendix C shows the denotational semantic of SysADL in π-ADL function. Finally, Appendix D presents the challenges and limitations of the work implementation.

2 Background

This chapter presents the main concepts related to this work. Initially, Section 2.1 introduces the running example used in this thesis: the Flood Monitoring System (FMS). Section 2.2 describes concepts related to software architecture. Sections 2.3 and 2.4 introduce the SysADL and π-ADL languages, respectively, their main elements, and how they can describe software architectures. Section 2.5 presents Model-Driven Engineering approach. Section 2.6 reports previous work that is a baseline for our approach.

2.1 FMS System: a Running Example

This section introduces the Flood Monitoring System (FMS), an example used throughout this thesis to illustrate the proposed approach. This example was chosen because it is a system with various components, dynamic situations, and complexity that can be used to exemplify the different parts of this work.

Floods are natural disasters known for their potentially destructive impact, which can ruin natural resources, cause casualties and harm the environment. The occurrence of floods is a problem in several countries worldwide, especially in urban regions crossed by rivers, when associated with extreme precipitation events, becoming a challenge for security and regional development. Regardless of their magnitude, floods pose a risk and must be detected.

In this context, a system to monitor the flow and level of water in rivers, together with the support of meteorological information, can help issue warnings and emergency alerts. Early warnings are essential to reduce the scale and cost of damage caused by floods, which have a clear correlation with both the depth of the flood and the amount of early warning given (ROURE et al., 2006). A monitoring system would offer up-to-date and accurate data to guide decisions on the necessary actions according to current conditions, ensuring better risk management, contingency plans, traffic reorganization in the vicinity of flooded areas, and coordination of efforts, rescue measures, and decisions that could mitigate the impacts of such an event.

The FMS modeling used in this work is a system typically based on a network composed of wireless *sensors* that measure the water level in flood-prone areas near a river, a *Gateway* station that analyzes measured data and can trigger alerts when a flood condition is detected. Communication between these elements takes place via wireless network connections using *ZigBee*[1], wireless communication protocol for IoT devices, focusing on low-power devices (HAQUE; ABDELGAWAD; YELAMARTHI, 2022).

[1] Connectivity Standards Alliance (CSA-IoT): <https://csa-iot.org/all-solutions/zigbee>

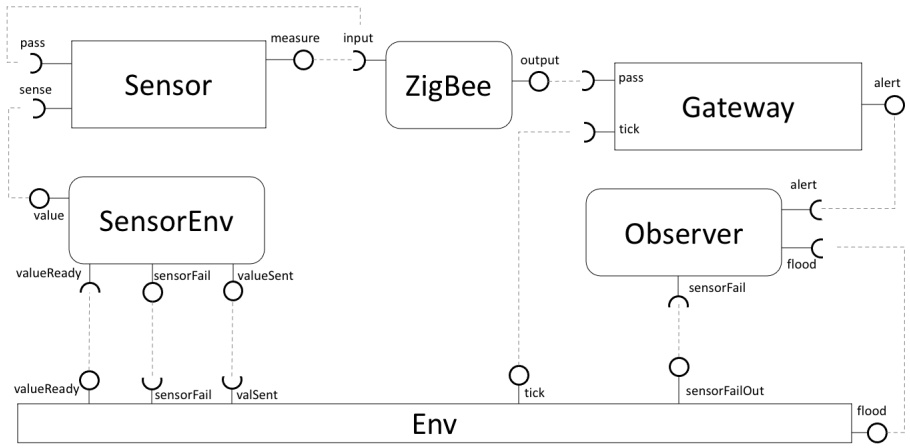


Figure 1 – Flood Monitoring System

Figure 1 shows the architecture of the system, containing one of each architectural element: *Sensor, SensorEnv, ZigBee, Gateway, Observer, and Env*. The elements *Sensor*, *SensorEnv*, and *ZigBee* can be added and removed during system execution, according to the need for river monitoring coverage.

The *sensor* communicate via *ZigBee* connectors and a *gateway* component receives all measurements to assess the current risk. Each measurement from a sensor is propagated to its neighbors via ZigBee connectors until it reaches the gateway. The environment is modeled through the *Env* component and the *SensorEnv* connector. *Env* is responsible for synchronizing the model by defining cycles corresponding to the frequency with which the sensors take measurements. A cycle consists of: (i) signaling to *Gateway* that a new cycle has started; (ii) updating the state of the river; (iii) flag each *SensorEnv* connector to deliver a new measurement to *Sensor*; and (iv) wait for each connector *SensorEnv* to confirm that a new measurement has been delivered.

A version of the Flood Monitoring System presented in this section was modeled in SysADL to be used to validate the proposed approach. The FMS SysADL Model is presented in detail in the appendix A.

2.2 Software Architecture

Bass et al. (BASS; CLEMENTS; KAZMAN, 2012) describe the software architecture of a system as the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both. More broadly, the 42010 standard "Systems and Software Engineering - Architecture description" defines software architecture as: "The fundamental properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution." (ISO/IEC/IEEE 42010, 2007)

An architecture acts as a bridge between requirements and implementation. It

represents the initial mapping of requirements to computational components, making it possible to assess the feasibility of meeting the requirements with the proposed project, still at a high level (KOGUT; CLEMENTS, 1994). Decisions made at the architecture level directly affect the achievement of business goals, functional and quality requirements and play a fundamental role in determining the system's overall quality (NAKAGAWA; OQUENDO; BECKER, 2012).

It is essential to distinguish between architecture and design concepts. Perry (PERRY; WOLF, 1992) said that the architectural elements, their interactions, and constraints satisfy the requirements and serve as a basis for the design. In contrast, design is concerned with modularization and the detailing interfaces of the design elements, algorithms, and the data types needed to support the architecture and satisfy the requirements.

Software systems, in general, are too complex to be understood all at once, so abstraction is an important concept when thinking about their architectures. Furthermore, software development is a process that involves many different people with different backgrounds using various notations, so this complex environment needs to be considered to maintain a coherent architecture and communication among the stakeholders. In this context, the architectural description appears as a relevant communication tool.

The ISO/IEC/IEEE 42010 Standard defines architecture description as an artifact to describe a system's fundamental concepts or properties in its environment embodied in its elements, relationships, and the principles of its design and evolution. An architecture description includes one or more architectural views, addressing one or more of the concerns of system stakeholders. The viewpoint is a fundamental concept in software architecture, and all stakeholder interests are mapped in viewpoints. A viewpoint defines the kinds of models that direct the diagrams used to represent its corresponding views. Such models are what a stakeholder "sees" when looking at the system from a specific viewpoint (OQUENDO; LEITE; BATISTA, 2016). The ISO/IEC/IEEE 42010 Standard also emphasizes that multiple views are essential to cover all the stakeholders' concerns and detail the architecture from different perspectives.

2.2.1 Architecture Description Language

According to the 42010 standard, an architecture description language (ADL) is any form of expression used in architecture descriptions. Often an ADL is supported by automated tools to aid the creation, use, and analysis of its models.

In terms of software architectural description, there are two main lines of work: (i) one based on the definition of new languages for describing the architecture of software-intensive systems, the so-called Architecture Description Languages (ADLs). Another is based on the use of general-purpose modeling languages, in particular, Unified Modeling Language (UML) (OMG, 2011) and recently Systems Modeling Language (SysML) (OMG, 2001).

Medvidovic (MEDVIDOVIC; TAYLOR, 2000) defines an ADL as a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation. ADLs use graphical and/or textual notations to express components and architectural relationships, provide both a concrete syntax and a conceptual framework for characterizing architectures, and generally have tooling support for architecture creation, analysis, and simulation.

The ISO/IEC/IEEE 42010 Standard specifies a conceptual model of architecture description, but it does not define the main building blocks and primary elements of an architecture description (OQUENDO; LEITE; BATISTA, 2016). However, Medvidovic and Taylor (MEDVIDOVIC; TAYLOR, 2000), based on a survey of architecture description notations and approaches, identified that ADLs capture aspects of software design centered around a system's components, connectors, and configuration. Figure 2 illustrates the main architectural concepts explained below.

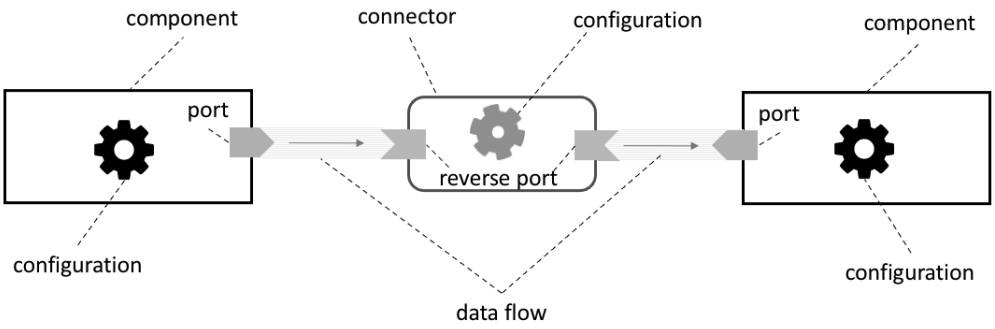


Figure 2 – Main architectural concepts

Components are architectural elements that provide functionalities in a system. Components may be *simple*, implementing a simple functionality, or as large as an entire application, *composite*, that encompasses other architectural elements. A component defines *ports* that are interaction points between it and the other architectural elements. Ports explicitly specify the data type that a component provides and requires, which is essential to support the component's composability. A port can be composed of other ports. In that case, it is called a *composite port*.

Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. It provides the mechanism for binding components together and specifies the kind of component's port that can be bound to it. A connector can be a *simple* element connecting two or more ports or even a *composite* element composed of others.

Configurations describe the topology for identifying which components are part of software architecture and how they are connected through connectors. This way, the architecture configuration defines a connected graph of components and connectors that describes the architecture. This information is needed to determine whether appropriate

components are connected and their interfaces match.

Figure 3 shows a sample of some architectural elements that are part of the FMS system:

(a) *SensorEnvConnectorCP* and *ZigBeeConnectorCP* components;

(b) *mvCN* and *CmH2OCN* connectors; and

(c) a configuration that groups instances of the *SensorCP*, *ZigBeeConnectorCP*, and *GatewayCP* components, linked by two instances of the *CmH2OCN* connector (*c1* and *c2*).

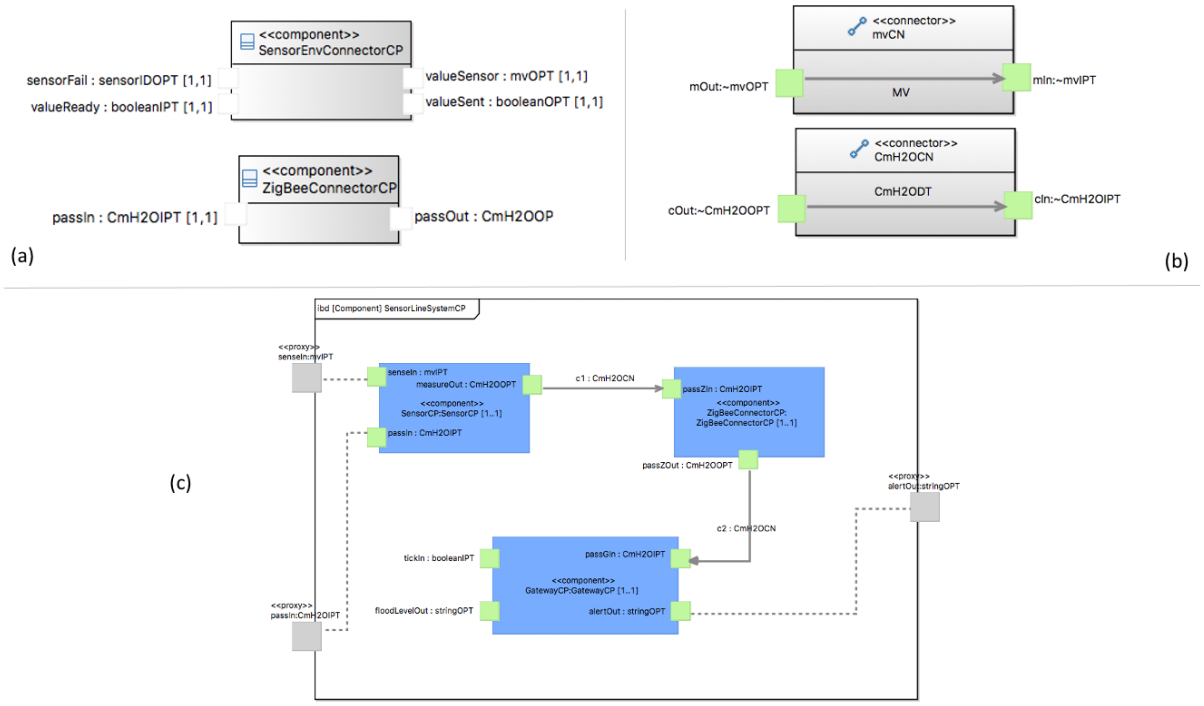


Figure 3 – Sample of architectural elements

2.3 The SysADL Architecture Description Language

SysADL is an ADL based on SysML (Systems Modeling Language) (OMG, 2001) that combines the rigorous semantics of ADLs with the principles of systems engineering used by SysML, a standard language used by the industry to model software-intensive systems (OQUENDO; LEITE; BATISTA, 2016). SysADL includes the definition of multiple architectural views, validation, and execution of the architecture. Furthermore, SysADL complies with the ISO/IEC/IEEE 42010 standard (ISO/IEC/IEEE 42010, 2007), which represents an international standard on the requirements of an ADL.

SysADL defines three architectural viewpoints, which provide the constructs for describing different architecture views from these viewpoints to communicate the software

architecture to the involved stakeholders. The SysADL viewpoints, *structural*, *behavioral*, and *executable*, will be detailed in the next sections.

2.3.1 Structural viewpoint

The structure of a software architecture refers to the way in which the architectural elements are organized to achieve the required functionalities and system qualities (OQUENDO; LEITE; BATISTA, 2016). The structural view describes the building blocks of architecture from a conceptual point of view in terms of components, connectors, and configurations. SysADL uses *Block Definition Diagrams (BDD)* and *Internal Block Diagrams (IBD)* to build this view.

BDDs are used to define components, connectors, ports, and types of architecture. An IBD shows how components and connectors bind each other, defining the architecture's configuration or the internal structure of composite components and connectors.

The elements used in the structural viewpoint are:

Value Types. Value Types specify the different data types in the architecture description, as the data flow through the ports, for example. In SysADL, there are primitive data types (*Real*, *Integer*, *Boolean*, and *String*), user-defined Value Types, Enumerations, and Data Types. Figure 4a shows some examples of value types in SysADL: primitive type Boolean, *riskEN* Enumeration composed of five values (*Low*, *LowMedium*, *Medium*, *MediumHigh*, and *High*), and *CmH2ODT* Data Type composed of three attributes (*idS*, and *mvSensor*).

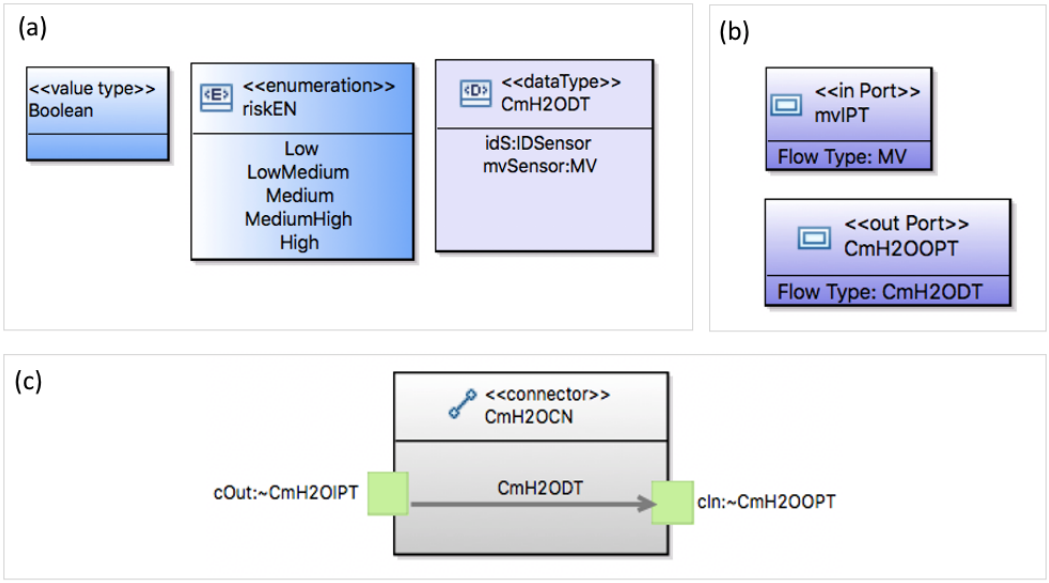


Figure 4 – Structural elements from a SysADL model.

Ports. A port specifies an interaction point between a component and other architectural elements. It represents how the data flow to a component (*in* ports) from another (*out* ports). A port can be composed of other ports, it is called a *composite port*. Figure

4b shows the following simple ports: *mvIPT* (input) and *CmH2)DT* (output). SysADL still brings the notion of a conjugate port, a port of the same type as an originally defined port but with a flow in the opposite direction. To distinguish them from the others, we add a tilde "∼" before the port type, like those used in the CmH2OCN connector shown in Figure 4c.

Connectors. A connector refers to a software element that represents the communication between components. A connector supports interaction between components. It binds ports of the connected components, allowing data to flow between them. A connector can be a simple element connecting two ports or a composite element composed of other connectors and has a configuration. Figure 4c shows a simple connector *CmH2OCN* that links the conjugated ports ∼*cOut* (output) and ∼*cIn* (input). The arrow and label indicate, respectively, data flow and data type.

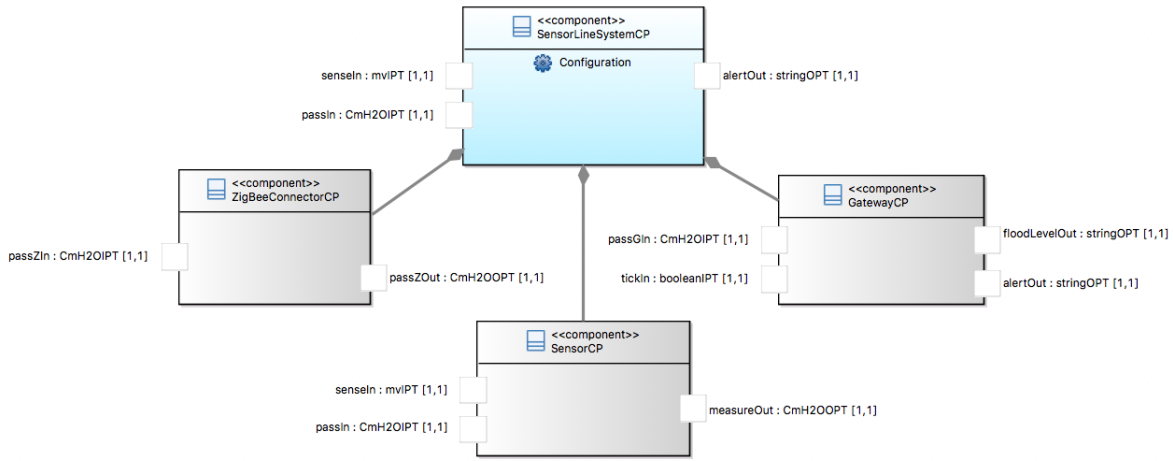


Figure 5 – *Components* in a SysADL model.

Components. A Component specifies its structure in terms of ports. A component can have input ports to obtain required data and output ports to produce provided data. A component performs services by consuming data in its in ports and providing the results in its output ports. A component can have several input and output ports. In SysADL, components can be: (i) *boundary*, whose behavior is unkown; (ii) *composite*, when its behavior is defined instantiation and connection of other components; and (iii) *simple*, when an activity defines its behavior. Figure 5 shows the structural definition a composite component: *SensorLineSystemCP*(input ports - *senseIn(mvIPT), passIn(CmH2OIPT)*; output port - *alertOut(stringOPT)*), composed of three simple components:

- *ZigBeeConnector*(input port - *passZIn (CmH2OIPT)*; output port - *passZOut (CmH2OOPT)*),
- *SensorCP* (input ports - *senseIn (mvIPT), passIn (CmH2OIPT)*; output port - *measureOut (CmH2OOPT)*),

- *GatewayCP*(input ports - *passGIn(CmH2OIPT), tickIn(Boolean)*; output ports - *foodIn(stringOPT), alertOut(stringOPT)*);

In the structural excerpt in Figure 5, the lines indicate that the componenxts *ZigBeeConnectorCP*, *SensorCP*, and *GatewayCP* are instantiated in the *configuration* of the *SensorLineSystemCP* component to compose it.

Architecture. The ISO/IEC/IEEE 42010 standard assumes that an architecture description expresses at least one architecture. It means that an architecture description may describe one or many architectures. In SysADL, architecture is a composite component, which behavior is defined of the instantiation and connection of other components.

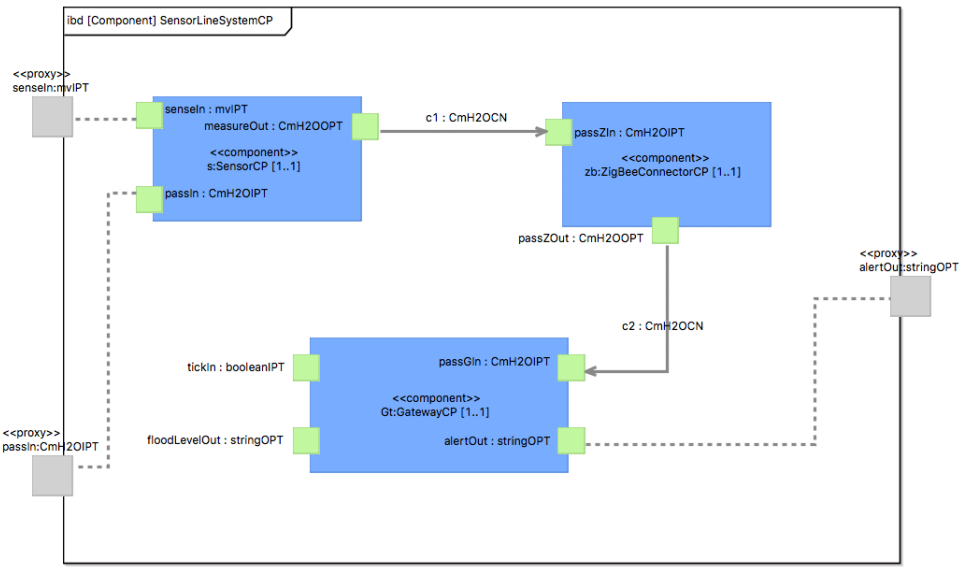


Figure 6 – *Configuration* element in a SysADL IBD

Configuration. Configuration is a concept that refers to the structural organization for identifying which components are part of a software architecture and how they are connected through connectors. A configuration can represent the structure of a composite component or the overall software architecture of a system. It uses components and connectors previously defined. In SysADL, a configuration is described using an IBD, as shown in Figure 6. This figure illustrates the *SensorLineSystemCP* IBD, with the instance of the components *SensorCP(s)*, *ZigBeeConnectorCP(zb)*, and *GatewayCP(Gt)*, and the respective connections of their ports through instances of the connectors *c1(CmH2OCN)*,and *c2(CmH2OCN)*. Internal component ports are not visible from the outside. The *senseIn (mvIPT)*, *passIn (CmH2OIPT)*, and *alertOut (stringOPT)* internal ports need to be externally visible; therefore, they are bound to a proxy port via a binding connector. SysADL uses binding connector to link an internal port to a proxy port. In this way, a proxy port specifies the capabilities of internal ports visible through external connectors.

2.3.2 Behavioral viewpoint

Identifying components, connectors, and the interaction between them, as represented by the structural view of architecture, is not enough to define a software architecture. These definitions say nothing about how components and connectors behave and how these interaction behaviors imply system functionality. To complement the architectural description in these aspects, it is necessary to embrace this description for the aspects foreseen in the behavioral viewpoint.

A software architecture's behavior refers to how the structural elements perform activities and interactions to achieve the required system functionality. The behavior of a component defines its functional behavior, while the behavior of a connector defines the interaction behavior regulating communication between components. The behavior of a port defines the protocol that governs the dataflow interaction from or to that port. Finally, the behavior of a configuration emerges from the composition of the individual behaviors of components and connectors.

SysADL provides its behavioral viewpoint by defining the following behavioral constructs: *activity, action*, and *equations*. An *activity* expresses the behavior of a component or connector as a sequential control flow of actions. *Actions* are simple behaviors that execute from beginning to end receiving parameters and returning a result, may be internal actions or express the send and receive data. *Equations* define the semantics of actions by expressing their pre and post-conditions, responsible for validating an expected behavior, specified through ALF expressions (Action Language for Foundational UML) (OMG, 2017).

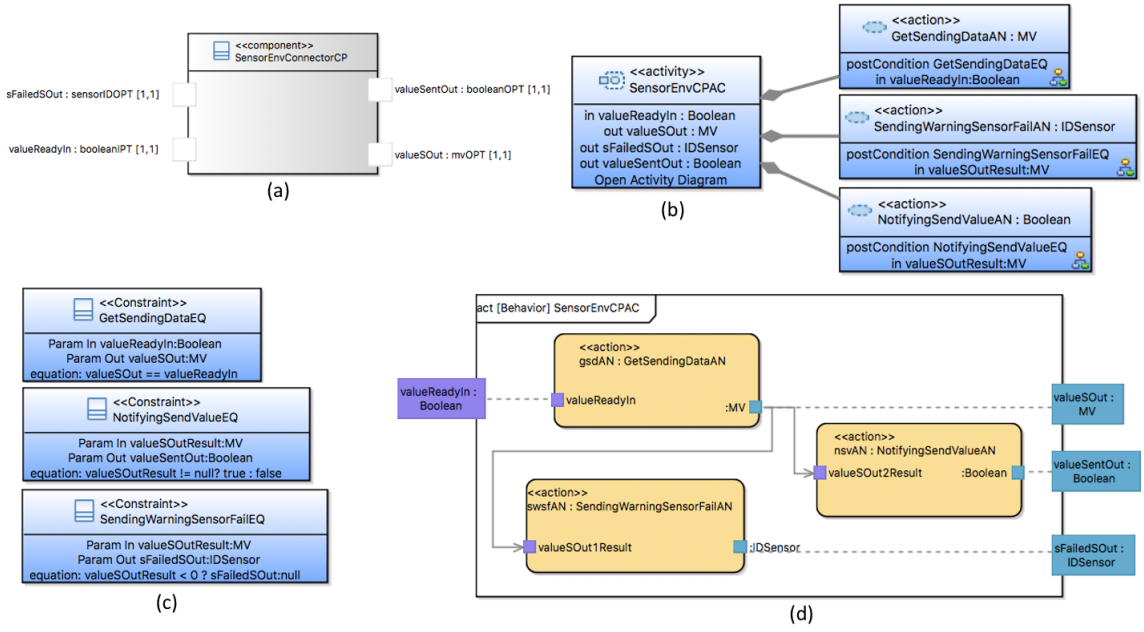


Figure 7 – Excerpts from the behavioral specification of an architect in SysADL

Figure 7 shows excerpts from the behavioral specification of the *SensorEnvCon-*

nectorCP component (Figure 7a) that has two input ports (*valueReadyIn(booleanIPT)*) and three output ports (*sFailedSOut(sensorIDOPT)*, *valueSentOut(booleanOPT)* and *valueSOut (mvOPT)*).

Figure 7b shows the definition of the *SensorEnvConnectorCPAC* activity, whose input and output parameters exactly reflect the input and output ports of the component associated with it (*SensorEnvConnectorCP*). This figure also shows the specification of three actions: (*GetSendingDataAN*, *SendingWarningSensorFailAN* and *NotifyingSendValueAN*) and the respective constraints (*GetSendingDataEQ*, *SendingWarningSensorFailEQ* and *NotifyingSendValueEQ*) that define the equations used as post-conditions of the actions in Figure 7c.

Finally, Figure 7d shows the diagram associated with *SensorEnvConnectorCPAC* activity with instantiation of actions, control, *pins* that represent parameters and specify a data stream, and data flow between actions.

2.3.3 Executable viewpoint

In the SysADL executable view, it is possible to specify details of each action at the implementation level through ALF statements but independent of any implementation platform or programming language. Figure 8 presents an excerpt of an executable BDD, with the declaration of an executable element corresponding to the *NotifyingSendValueAN* actions, declared in the behavioral BDD (Figure 7b). An ALF engine must interpret executable instances to execute the architecture. The execution of a SysADL architecture model allows simulating the system operations to verify and analyze its functionalities.

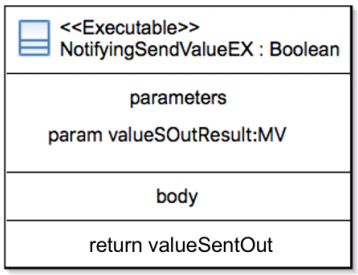


Figure 8 – Excerpt from the executable specification of an architect in SysADL

The illustrated excerpts in this section are from the architectural modeling of the FMS. The full version is available in appendix A.

2.4 The π-ADL Architecture Description Language

π-ADL (OQUENDO, 2004) is a formal, well-founded theoretical language based on the higher-order typed π-calculus (MILNER, 1999), intended to describe software architectures under both structural and behavioral viewpoints, providing formally founded

constructs for architecture description while achieving computational completeness and high expressiveness. π-ADL deals with dynamicity concerns by architectural-level primitives to specify programmed reconfiguration operations, that is, predicted and pre-planned changes described at design time and triggered at runtime by the self-system under a specific condition or event (CAVALCANTE et al., 2016). π-ADL is automated by tools, i.e., a specification and verification toolset providing support for automated checking.

This section presents the π-ADL language and the main elements of an architectural description in π-ADL.

2.4.1 Architectural Concepts

From the structural viewpoint, software architecture is described in π-ADL in terms of *components*, *connectors*, and their composition to form the system, i.e., *architecture* as a configuration of components and connectors. *Components* represent the specification of computational elements in a software system, whereas connectors manage interactions among components. Components and connectors can be composed to construct composite elements (components or connectors). *Connections* are basic interaction points. Their architectural role is to provide communication channels between two architectural elements. From the behavioral viewpoint, both components and connectors comprise a behavior that expresses an architectural element's interaction and internal computation and uses connections to send and receive values between architectural elements.

Figure 9 illustrates the main architectural concepts of π-ADL. From a black-box perspective, only connections of components/connectors and values passing through connections are observable. From a white-box perspective, internal behaviors of such elements are also observable.

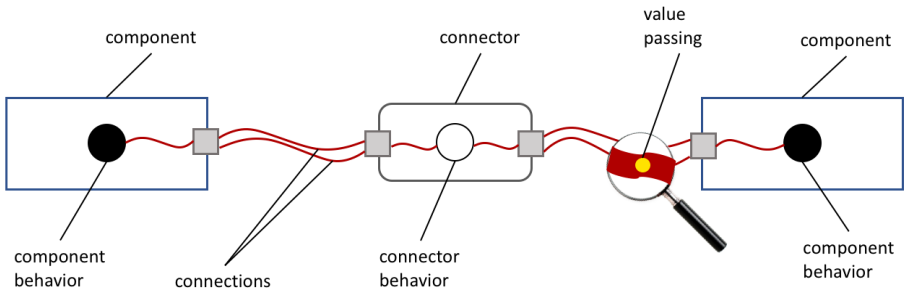


Figure 9 – Architectural concepts in π-ADL- from (CAVALCANTE, 2016)

2.4.2 Type System

This section presents the atomic and composite data types defined in *first-order* π-*ADL* (π-ADL $_{FO}$), extended version of π-*ADL base* with constructs data types, used in this work.

Base types. The base value types are used to express atomic values. Table 1 shows the base types defined in π-ADL $_{FO}$.

Table 1 – Base types defined in π-ADL $_{FO}$

Type	Syntatic representation	Definition
Natural	`Natural`	Natural numbers
Integer	`Integer`	Integer numbers
Real	`Real`	Real numbers
Boolean	`Boolean`	Boolean logical values
String	`String`	Character strings
Any	`Any`	Generic type

Constructed types. The π-ADL $_{FO}$ provides constructors for defining composite types. Table 2 presents these constructed value types, each one described as follows.

Table 2 – Constructed types defined in π-ADL $_{FO}$

Type	Syntatic representation	Definition
Tuple	`tuple`$[T_1, T_2, ..., T_n]$	Tuple $(v_1, v_2, ..., v_n)$ in which each v_i is of type T_i
View	`view`$[l_1 : T_1, l_2 : T_2, ..., l_n : T_n]$	Labeled form of a tuple $(v_1, v_2, ..., v_n)$ in which each v_i has a label l_1 and is of type T_i

Figure 10 shows the definition of the *CommandToArm* type that can receive any value since it was declared as *Any*, and of the constructed types *Location* and *VehicleData*, both of the *View* are constituted by the fields *loc(String)*; and *destination(Location)*, *command(CommandToArm)*, respectively.

```
type CommandToArm is Any
type Location is view[
    loc: String
]
type VehicleData is view[
    destination: Location,
    command: CommandToArm
]
```

Figure 10 – Types Declaration in π-ADL

2.4.3 Behavior constructs

π-ADL provides behavior constructs to represent the internal behavior of architectural elements, making use of connections to send and receive values. Behaviors defined in π-ADL come from existing operators provided by π-calculus, in which channels (connections in π-ADL) are used to transmit values between interacting processes, i.e., behaviors of architectural elements. Table 3 describes the behavior constructors in π-ADL, similarly to the basic π-calculus constructs.

Prefixing actions. Behaviors act by performing actions, which are expressed through prefixes. As shown in table 3, a prefix can be an output or input prefix or a silent prefix: (i) the *input prefix* describes the ability to receive a value v via the connection c;

Table 3 – Behavior constructs defined in π-ADL

Behavior construct	**Syntatic representation**	**Definition**
Input prefixing action	`via` c `receive` v	Receive value v via connection c
Output prefixing action	`via` c `send` $s : T$	Send value s of type T via connection c
Silent prefixing action	`unobservable`	Unobservable internal action
Parallel composition in parallel	`compose` B_1 `and` B_2 ... `and` B_n	Execute behaviors $B_1, B_2, ..., B_n$
Non-deterministic choice	`choose` B_1 `or` B_2 ... `or` B_n	Choose to execute either behavior B_1 or $B_2, ..., B_n$
Inert process	`done`	Nothing to do

(ii) *output prefix* expresses the capacity to send a value s of type T via the connection c; and (iii) the *silent prefix* expresses the ability to act be unobservable, i.e., invisibly, internally, silently.

Composition behavior. The parallel composition expresses a behavior's ability to compose sub-behaviors capabilities in parallel, each one independently progressing from the others in its execution thread. The sub-behaviors interact with each other via shared connections, as shown in:

```
compose{
    via x send v
    and
    via x receive y:Integer
}
```

The sub-behaviors of this composition behavior interact with each other via the connection x. The first one sends a value v that the second sub-behavior receives as the value *integer* y.

Choice behavior. The choice behavior expresses the ability of a behavior to execute alternative sub-behaviors. When a given sub-behavior acted, the others are no longer available. Thus only one choice sub-behavior is executed, as shown in:

```
choose{
    via input receive v:Integer
    via output send (v+1)
    or
    via alt receive s:Integer
    db = s
}
```

Two alternative options express this choice behavior, (1) receiving an *integer* value v via the *input* connection and then sending the increment of v via the *output* connection, or (2) receiving an *integer* value s via the *alt* connection and then assigning it to a variable called *db*. The selection criterion for choosing the block to be executed is non-deterministic.

Inert (nil) process. The behavior of an inert process represents the end of execution of behavior, i.e., it does not execute any further actions.

2.4.4 Describing the FMS system in π-ADL

Figures 11 and 12 show part of a Flood Monitoring System architectural description in π-ADL. Figure 11 shows the specification of the *SensorEnveConnectorCP*, and Figure 12 shows the specification of the architecture *FMSa1ARCH*. In Figure 11, the *SensorEnvConnectorCP* component behavior defines *GetSendingDataAN*, a function that receives a *valueReadyIn (Boolean)* as input and returns a *MV*, in addition, the functions SendingWarningSensorFailAN, and NotifyingSendValueAN. In Figure 12 ,the *ClientServerARCH* architecture is shown as a composition of instances of the *SensorCP*, *ZigBeeConnectorCP*, *SensorEnvConnectorCP*, *GatewayCP*, *ObserverConnectorCP*, and *EnvCP* components. The instance of the 10 connectors (c1-c10) *CmH2OCN*, *mvCN*, *booleanCN*, *IDSensorCN*, and *stringCN* types, along with the unifications that bind these architectural elements.

```
component SensorEnvConnectorCP is abstraction(){
    type MV is  Any

    connection valueReadyIn is in   (Boolean)
    connection valueSentOut is out   (Boolean)
    connection valueSOut    is out   (MV)
    connection sFailedSOut  is out   (Integer)

    protocol is {
    ( via valueReadyIn  receive   Boolean
     via valueSentOut  send   Boolean
     via valueSOut  send   MV
     via sFailedSOut  send   Integer
    )*
    }


    behavior is {
        GetSendingDataAN is function(valueReadyIn : Boolean) : MV{
            //statements
            return 0
        }
        SendingWarningSensorFailAN is function(valueSOut1Result : MV) : IDSensor{
            //statements
            return 0
        }
        NotifyingSendValueAN is function(valueSOut2Result : MV) : Boolean{
            //statements
            return 0
        }
    }
}
```

Figure 11 – Excerpt of a *SensorEnvConnectorCP* Component as described in π-ADL

2.5 Model-Driven Engineering

The Model-Driven Engineering (MDE) approach emerged in the context where the systems development industry need more abstraction to deal with increasingly complex and rapidly evolving systems. MDE provides common concepts for understanding the basics of models and metamodels (SEIDEWITZ, 2003). A model is a simplified representation of a system built with an intended purpose, and a metamodel is a description of a modeling language allowing the development of modeling and analysis tools (FLOREZ; SÁNCHEZ; VILLALOBOS, 2013).

```
architecture FMSa1ARCH is abstraction(){
    behavior is {
        compose {
                S1 is SensorCP()
            and Zb1 is ZigBeeConnectorCP()
            and EnvS1 is SensorEnvConnectorCP()
            and Gt is GatewayCP()
            and Obs is ObserverConnectorCP()
            and Env is EnvCP()
            and c1 is CmH2OCN()
            and c2 is CmH2OCN()
            and c3 is mvCN()
            and c4 is booleanCN()
            and c5 is IDSensorCN()
            and c6 is booleanCN()
            and c7 is booleanCN()
            and c8 is IDSensorCN()
            and c9 is stringCN()
            and c10 is booleanCN()
        } where {
                S1::measureOut unifies c1::cOut
                c1::cIn unifies Zb1::passZIn
                Zb1::passZOut unifies c2::cOut
                c2::cIn unifies Gt::passGIn
                EnvS1::valueSOut unifies c3::mOut
                c3::mIn unifies S1::senseIn
                Env::valueReadyOut unifies c4::bOut
                c4::bIn unifies EnvS1::valueReadyIn
                EnvS1::sFailedSOut unifies c5::sidOut
                c5::sidIn unifies Env::sFailedEIn
                EnvS1::valueSentOut unifies c6::bOut
                c6::bIn unifies Env::valueSentIn
                Env::tickOut unifies c7::bOut
                c7::bIn unifies Gt::tickIn
                Env::sFailedEOut unifies c8::sidOut
                c8::sidIn unifies Obs::sFailedOIn
                Gt::alertOut unifies c9::sOut
                c9::sIn unifies Obs::alertIn
                Env::floodOut unifies c10::bOut
                c10::bIn unifies Obs::floodIn
        }
    }
}
```

Figure 12 – Excerpt of a FMS architecture as described in π-ADL

The basic principle behind MDE is that everything is a model. It provides a generic approach to deal with all software artifacts used and produced during the software development life-cycle. Even the languages used to specify the models can be considered models too, referred to as metamodels (MENS, 2013).

The belief that everything is a model and adopting a model-centric view requires techniques and tools that allow manipulation and reasoning about such models. In this context, model transformation is one of the pillars of MDE (SANCHEZ; FLOREZ, 2018).

2.5.1 Model-to-Model Transformation Process

The Model-Driven Development (MDD) approach relies on models as central elements to software development. One of the keys to this approach is *model-to-model transformations* (M2M) that consist of specifying rules that make it possible to produce target models from source models. Figure 13 illustrates a model transformation process. Transformation specification rules are responsible for defining how to map elements of the source model to corresponding ones in the target model.

Both source and target models must conform to their respective *metamodel* since the transformation rules are based on metamodel. In turn, the transformation specification must also comply with the syntax and semantics of the language used for implementing the transformation.

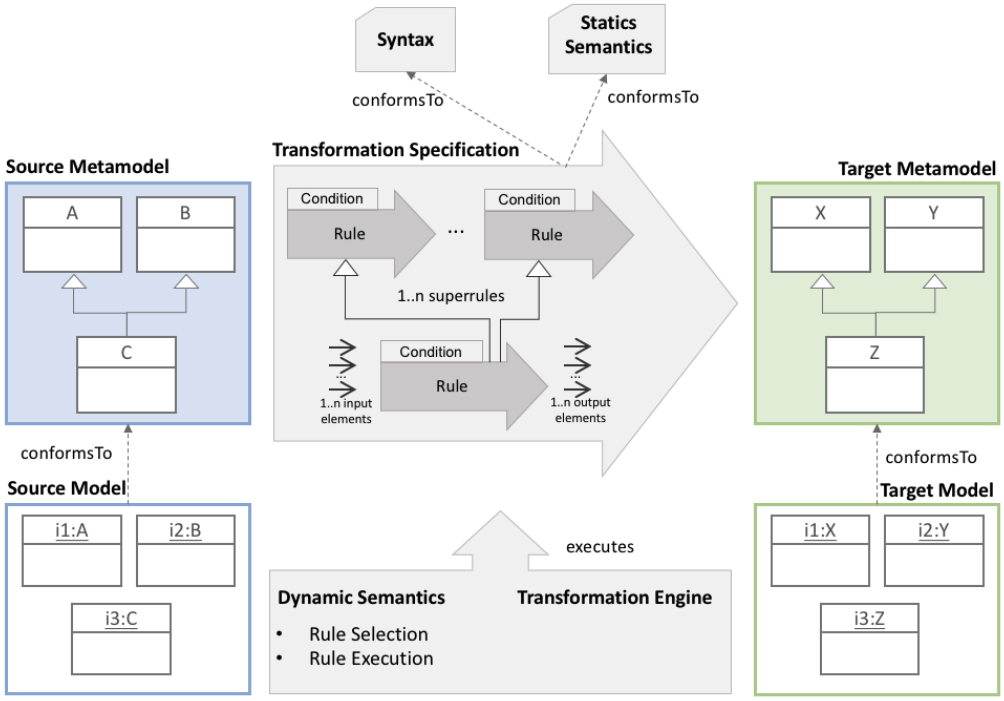


Figure 13 – A model-to-model transformation pattern - adapted from Wimmer (WIMMER et al., 2012)

Figure 14 shows on the left an excerpt from the SysADL metamodel in a class that defines the "ComponentDef" element, while on the right, a section from the π-ADL metamodel that represents the "Component" element is shown. Still in the image, the lines superficially demonstrate the correspondence between the attributes of the different models. In the M2M Transformation, rules must be established that effectively transform the elements of the source model into elements in the destination model.

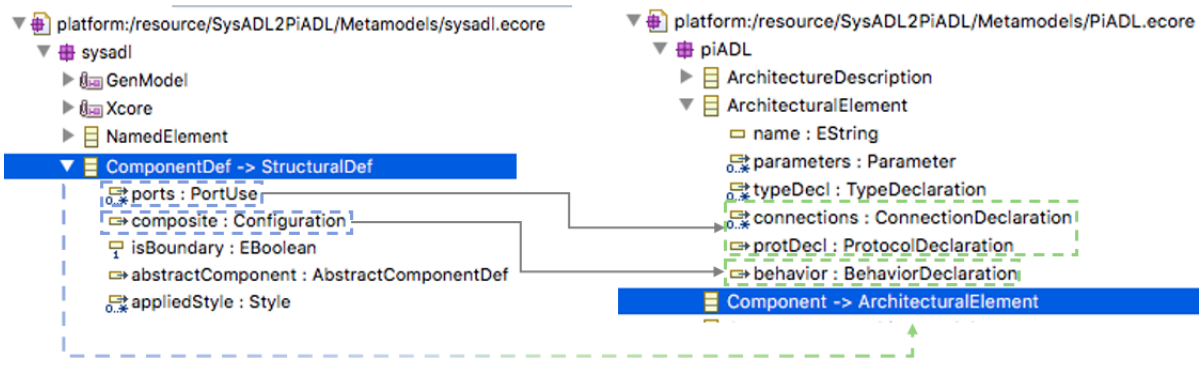


Figure 14 – Excerpt of a correspondence from the SysADL metamodel (left) and π-ADL metamodel (right)

2.5.2 Model-to-Text Transformation Process

Another key to the MDD approach is model-to-text (M2T) transformations, which generate a source code from a model. This M2M transformation use template-based technology. In this context, a template consists of the target text containing a metacode to access variable information (CZARNECKI; HELSEN, 2006).

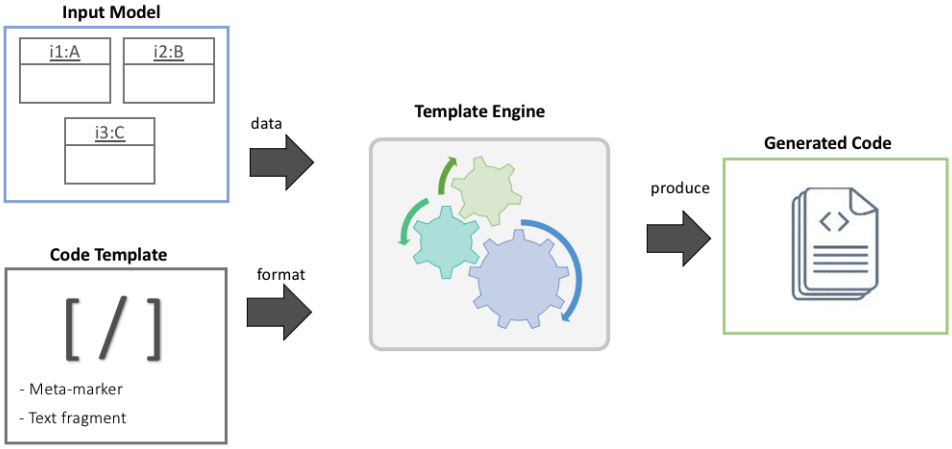


Figure 15 – A model-to-text transformation template-based pattern

Figure 15 shows a summary of the M2T transformation process. The meta-markers and text fragments are organized by the format expressed in the code template. This design considers the syntax rules of the target language. The template engine produces the target source code following the code template guidelines, replacing the meta-markers with the input model data.

Figure 16 shows on the left a fragment of a π-ADL model describing a component called "ZigBeeConnectorCP," while on the right, a π-ADL code is displayed with the definition of the "ZigBeeConnectorCP" element generated from a template-based M2T transformation.

2.6 A Formally Founded Framework for Dynamic Software Architectures

This section presents a work that is used by the approach proposed in this work. Cavalcante (CAVALCANTE et al., 2016) proposed a formal framework for dynamic software architectures involving: (i) π-ADL (OQUENDO, 2004), a formal language for describing software architectures under structural and behavioral perspectives; (ii) the specification of programmed dynamic reconfiguration operations; (iii) automatic generation of source code from architectural descriptions, and; (iv) a statistical verification approach to formally express and verify properties in dynamic software architectures.

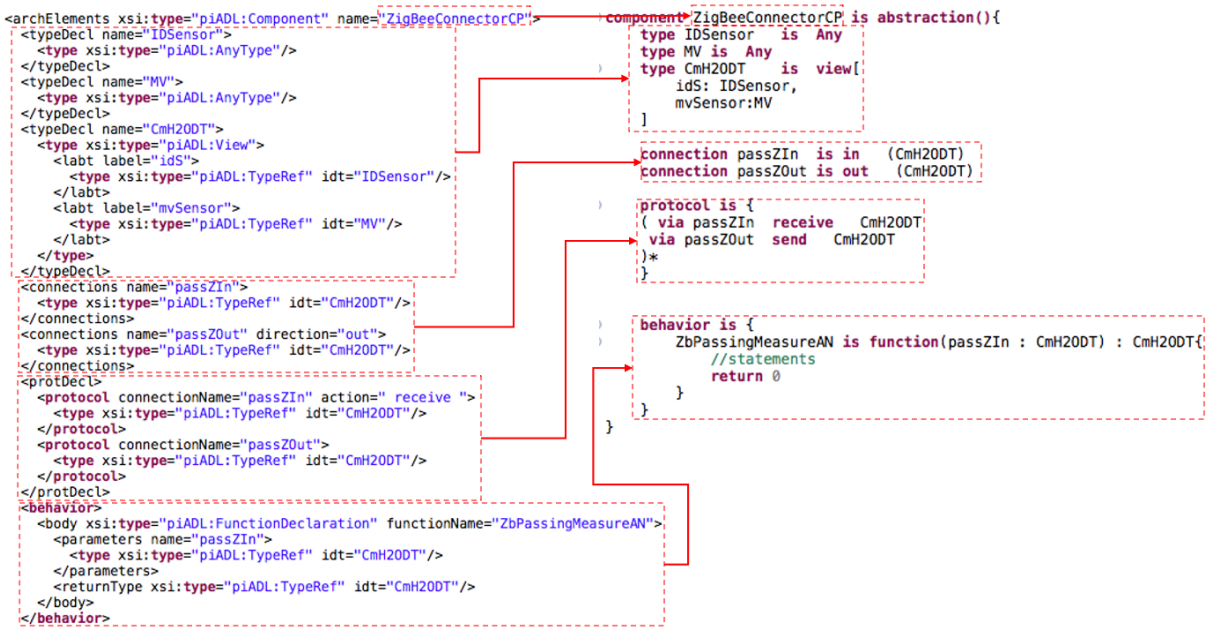


Figure 16 – Excerpt from the π-ADL model (left) and code by M2T Transformation

The main contributions brought by the proposed framework are four. (i) the π-ADL language was extended with architectural-level primitives to describe programmed dynamic reconfigurations; (ii) architectural descriptions in π-ADL are translated to implementation source code in the Go programming language[2], thus helping to minimize architectural; (iii) a new logic called DynBLTL (QUILBEUF et al., 2016) is used to express properties in dynamic software architectures formally; and, (iv) Statistical Model Checking (SMC) based tooling was built to automate checking of architectural properties while seeking to reduce effort, computational resources, and time to accomplish this task. SMC is a probabilistic, simulation-based approach that consists of building a statistical model of finite executions of the system under verification and deducing the probability of satisfying a given property within confidence bounds (CAVALCANTE et al., 2016).

PLASMA[3] (JEGOUREL; LEGAY; SEDWARDS, 2012) is an SMC tool that presents itself as a platform that allows users to create custom SMC plug-ins on top of it. PLASMA offers three alternative modes for SMC: (i) a probabilistic Monte Carlo algorithm, (ii) a Monte Carlo algorithm with Chernoff confidence limits, and (iii) sequential hypothesis testing. However, one of the most important features of PLASMA is building plug-ins on top of the platform so that users can take advantage of the PLASMA environment to create custom statistical model checkers. Cavalcante (CAVALCANTE, 2016) reports that in addition to its efficiency and performance results, such flexibility was one of the main reasons that motivated the choice of PLASMA as the basis for the development of the specification and verification toolchain of properties of dynamic software architectures

[2] The Go programming language. Available at: <https://golang.org/>
[3] Plasma Lab - <https://project.inria.fr/plasma-lab/>

described in π-ADL.

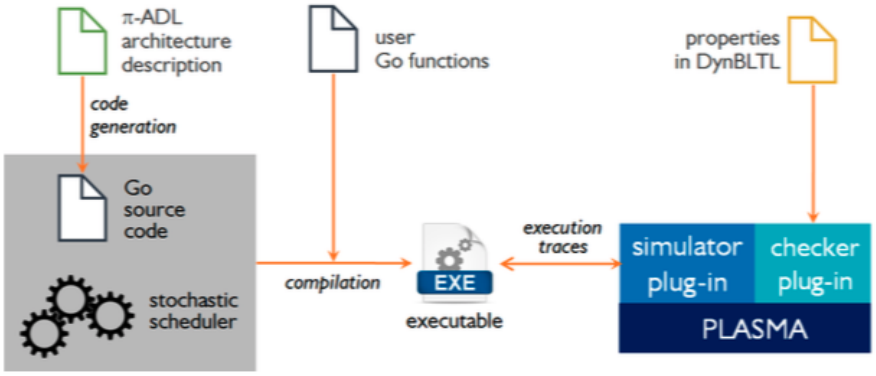


Figure 17 – Overview of the developed SMC-based toolchain for verifying properties expressed in DynBLTL regarding dynamic software architectures described in π-ADL (CAVALCANTE, 2016)

Figure 17 provides an overview of the developed SMC-based toolchain for verifying properties of dynamic software architectures. The inputs for the process are an architecture description in π-ADL and a set of properties specified in DynBLTL. The architecture description in the π-ADL language is translated towards generating source code in the Go programming language. Instrumenting the Go source code with these actions is necessary to generate execution traces upon the compilation and execution of such a code. These execution traces are provided as input to the simulator plug-in, which uses the SMC algorithms originally implemented in PLASMA.

2.7 Conclusion of the Chapter

This chapter presented the basic concepts related to this work (i) Flood Monitoring System - the running example, (ii) software architecture, describing its elements; (iii) the SysADL language, describing its viewpoints and diagrams; (iv) the π-ADL language, its concepts and architectural elements; (v) basic principles of M2M and M2T transformations of the Model-Driven Engineering approach.

The next chapter details the process of transforming SysADL models into π-ADL, with the final objective of obtaining a source code of an architectural description according to the π-ADL grammar for further analysis of this architecture such as property verification.

3 Transforming SysADL into π-ADL: a Model-Driven Approach

Aiming to achieve the main goal and answer the research questions of this study, we present a proposal to add formal semantics to SysADL to support the formal property verification of dynamic software architecture description. The proposal uses a Model-driven approach that assigns SysADL with the formal support of π-ADL, according to the structure and elements illustrated in Figure 18 and briefly described below.

Aiming to answer RQ1, we define a model-to-model (M2M) transformation between SysADL and π-ADL, specifying rules to produce a target model (π-ADL) from a source model (SysADL) by mapping their respective elements as described in their meta-models. The transformation algorithms and implementation are built by following the definition of the denotational semantics of SysADL as a function of π-ADL, as presented in section 3.2 and summarized in Figure 18. This figure shows that the architectural description in SySADL is provided as input (1) to a mechanism that automates their transformation into π-ADL models (2), based on denotational semantics.

The process continues aiming to answer RQ2, the architectural description transformed in π-ADL (3) is input to a toolchain that will transform the architectural description in π-ADL into an executable architecture in Go Language (GOOGLE, 2009) (4). Details of this phase are shown in Chapter 4.

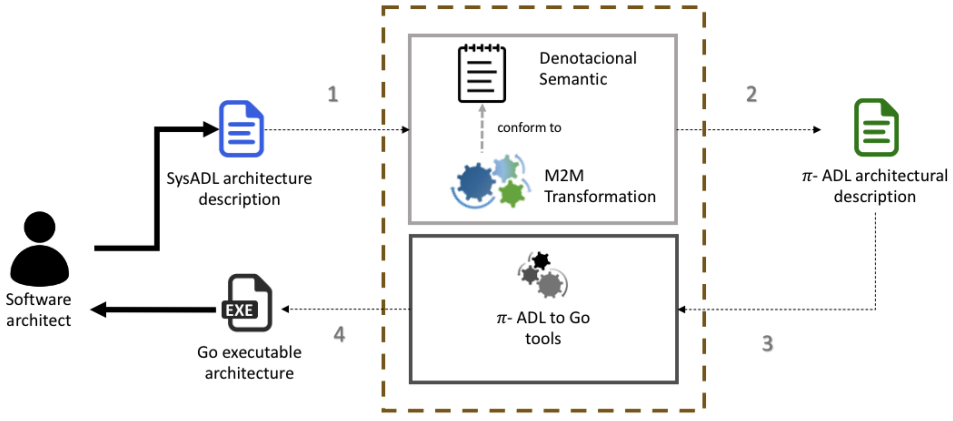


Figure 18 – Process for generating a π-ADL architecture description from a SysADL model.

The rest of this Chapter will cover the details of the process to map the SysADL architecture descriptions to π-ADL. For that, the structure of this Chapter is as follows. Section 3.1 describes the correspondence between SysADL and π-ADL. Section 3.2 presents a denotational semantics of SysADL in function of π-ADL. Section 3.3 describes the algorithms to transform an architecture description in SysADL under the structural

and behavioral viewpoint into another one in π-ADL. Section 3.4 presents the M2T process used to perform the transformation to generate π-ADL code. Section 3.5 presents the implementation process used to perform the transformation between the models. Finally, Section 3.6 presents the conclusion of this Chapter.

3.1 Correspondences between SysADL and π-ADL

We performed an initial mapping based on the syntax and semantics of the languages in order to realize how the architectural elements of SysADL corresponded to those of π-ADL. Table 4 summarizes the findings of the relationship between the main elements of SysADL and π-ADL under the structural and behavioral viewpoint, each one detailed as follows.

The main elements of the SysADL structural viewpoint are: *components, connectors, architectures, ports, value types(primitives, user-defined, and data type), configuration, connector bindings*, and *delegations*.

The main elements of the behavioral viewpoint are: *activities, actions, pins*, and *flow activities*.

Table 4 – 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
Activity	Behavior
Action	Function
Pin	Parameters
Flow activity	Input/output prefix

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 with a behavior on its own. In π-ADL, connectors manage interactions among components with constituent elements and/or an internal behavior. Connectors in SysADL can be composed of other connectors and can have configurations, but in the approach presented in this thesis, we chose to use only simple connectors with simple ports and whose only function is to connect simple ports.

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 from an output port of a component to an input port of another component. A composite port is composed of multiple port instances. The π-ADL version used in this work is supported through the concept of connection representing interaction points as well as binding between interaction points. In this way, data flow by connections representing the 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 in which the sending/receiving operations must be performed.

Primitive data types. Both SysADL and π-ADL have four primitive data types, namely *Integer, Real, String,* and *Boolean.* These data types are mapped 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. Value types from SysADL are mapped to π-ADL according to the intensity type, the *Any* type, which expresses a generic type.

Value type. In SysADL, a value type can used to add to a data type a physical dimension and unit. In the transformation to π-ADL, the user-defined value type is mapped to the *Any* type, which expresses a generic type.

Structured data type. In SysADL, data types are structured types defined using attributes. These data types are mapped to the View constructed type provided by π-ADL, which is composed of labeled field data.

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

Connector binding. Binding components 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 need to 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.

Activity. In SysADL, activities specify the behavior of simple, non-boundary components. An activity describes the behavior of an architectural element expressing its parameters and in its body by calling basic actions that execute the behavior and the control and data flow between actions. In π-ADL, activities are represented through a set of behavior clauses containing functions defined from actions and input/output prefixes expressing data flows between functions.

Action. In SysADL, an action refers to an atomic behavior that receives input parameters, computes them, and results in the output parameter. An action is defined by its parameters, pre-conditions expressed in terms of input parameters, and post-conditions described by input and output parameters. An action can be used to specify a call to simple behavior. It is only used inside an activity. In π-ADL, actions are represented as function declarations with the definition of its input parameters, return type, statements block.

Pin. SysADL pins are representations of parameters and specify a stream of data. Input pins represent input parameters, while output pins represent output parameters. In π-ADL, pins are expressed through the function's input parameters and input/output prefixes representing the data flow between functions.

Flow activity. SysADL flows represent both control and data flows in actions. They link input and output pins by describing how data and control flows concurrently progress from an action to another. In π-ADL, these flows are transformed into (i) input prefixes when it comes to input parameters and (ii) output prefixes to indicate that the output parameter will receive a function return.

3.2 Denotational Semantics

Programming languages have three main characteristics: (i) the appearance and structure of its sentences (Syntax), (ii) the assignment of meanings to the sentences (Semantics), and (iii) the usability of the language (Pragmatics) (SCHMIDT, 1986). Archi-

tectural description languages, such as SysADL and π-ADL, also have well-known syntax, semantics, and pragmatics, briefly discussed in sections 2.3 and 2.4. However, there are more formal ways to define the syntax and semantics of a language than those presented so far.

The language syntax specification has been thoroughly studied, and the Extended Backus-Naur form (EBNF) is widely used for defining syntax. The EBNF specification (ISO 14977, 1996) of SysADL and π-ADL grammar are described in Appendix B.2 and B.3, respectively.

The definition of Denotational Semantics is frequently used to express the semantics of languages. Denotational semantics is a mathematical approach that uses the definition of functions to define the meaning of programs. With formal semantics, we give programs meaning by mapping them into some abstract but precise domain of objects (SLONNEGER; KURTZ, 1995). The goal of denotational semantics is to provide a mathematical model of the meaning of programs, in terms of their inputs and outputs, without referring to their implementation or execution.

A sentence in a programming language or the syntactic construction of a system is defined in terms of its constituent parts in its EBNF specification. Analogously, denotational semantics has as a fundamental principle that its definitions must be compositional, i.e., the denotation of each construct is defined in terms of the denotations of its constituent parts (CIOBANU; TODORAN, 2017).

π-ADL takes its roots in previous work concerning the use of π-calculus as the semantic foundation for architecture description languages (CHAUDET et al., 2000; CHAUDET; OQUENDO, 2000). The document developed by Oquendo et al. (OQUENDO et al., 2002) presents the abstract syntax and formal semantics of the ArchWare Architecture Description Language, predecessor project of π-ADL.

π-ADL has its formal semantics well described as a domain-specific extension of the higher-order typed π-calculus, and achieves Turing completeness (OQUENDO, 2004). Thus, we define the denotational semantics of SysADL in π-ADL, aiming to show that the transformation of syntactic elements from SysADL to π-ADL is feasible, and through this process, SysADL would have formal semantics. The denotational semantic of SysADL using π-ADL is described in Appendix C.

3.3 M2M Transformation from SysADL to π-ADL

Based on the denotational semantics of SysADL in π-ADL, we define language-agnostic M2M transformation algorithms. This section presents five algorithms, namely:

1. *RuleModel2Architecture*, SysADL to π-ADL main transformation rule;
2. *RuleConnectorDef2Connector* transforms a SysADL connector in a π-ADL connec-

tor;

3. *RuleComponentDef2Component* transforms a SysADL component into a π-ADL component;

4. *RuleComponent2BhDeclaration* realizes behavior transformation of a SysADL component into a π-ADL behavior; and

5. *RuleArchitectureDef2Architecture* defines how the SysADL architecture is transformed into a π-ADL architecture.

For the pseudocode representation of the algorithms in this section, some notations expressed below are used:

- *Metamodel!Cl*: This expression identifies the domain of elements (objects) manipulated in the algorithm, informing the *class* and *metamodel* to which the object belongs, e.g., *SysADL!ComponentDef* (A *ComponentDef* element defined in the SysADL metamodel);

- $SET \leftarrow \forall\ ELEM \in MM!Class$: Creates an ordered set called *SET* where all *ELEM* elements included is an element of class *Cl* of the *MM* Metamodel, e.g., $CONS \leftarrow \forall\ CN \in SysADL!ConnectorDef$ creates the *CONS* ordered set, including in it all objects that belong to the *ConnectorDef* class of the SyADL metamodel

- $SET \leftarrow \bigcup_{(ELEM.att1.att2)} SET$: Creates an ordered set called *SET* including all *att2* elements. Note that *att2* is an attribute of *att1* elements, which is an attribute of *ELEM*.

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 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 RuleConnectorDef2Connector* auxiliar procedures (lines 7-12). In a similar way, the *archs* attribute of the π-ADL architecture description element must contain the definition of architectures. The last loop (lines 14-16) iterates over the set of architecture definitions of the SysADL model and transforms them into the corresponding π-ADL element through the *RuleArchitecturerDef2Architecture* auxiliar procedure.

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
```

The flowchart in Figure 19 exhibits an overview of the sequence of activities implemented by Algorithm 1.

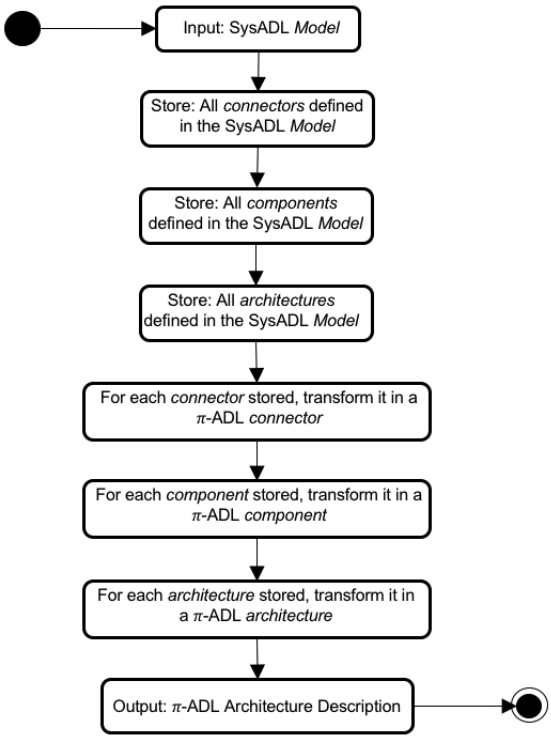


Figure 19 – Transformation flowchart from SysADL *Model* to π-ADL *ArchitectureDescription*.

Algorithm 2 Transformation rules for connectors

```
1:  procedure RuleConnectorDef2Connector(CN)
2:      Result cnp as PIADL!Connector
3:      cnp.name ← CN.name
4:      TYPES ← ⋃(CN.ports.definition) TYPES
5:      cnp.typeDecls ← ∅
6:      for each tp ∈ TYPES do
7:          TYPES ← TYPES ∪ dependences(tp)
8:          tempType ← RuleTypeDef2TypeDeclaration(tp)
9:          cnp.typeDecls ← cnp.typeDecls ∪ tempType
10:     CONPS ← ⋃(CN.ports) CONPS
11:     cnp.connections ← ∅
12:     cnp.protDecl ← ∅
13:     for each cnp ∈ CONPS do
14:         tempCon ← RulePortUse2ConnectionDecl(cnp)
15:         cnp.connections ← cnp.connections ∪ tempCon
16:         tempPT ← RulePortDef2ProtocolDeclaration(cnp)
17:         cnp.protDecl ← cnp.protDecl ∪ tempPT
18:     if CN.Composite = ∅ then
19:         cnp.behavior ← UNOBSERVABLE
20:     else
21:         cnp.behavior ← RuleConnector2CompositionBhv(CN)
```

An implementation of *RuleConnectorDef2Connector* to turn a *CN*, a *SYSADL! ConnectorDef*, into *cnp*, a *PIADL!Connector* is shown in algorithm 2. The *name* attribute of *cnp* directly receives the *CN.name* (line 3), without having to call other mapping rules. This happens when the attributes are of the same type. Otherwise, other rules or methods are required. The *TYPES* variable (line 4) receives all instances of the types of port definition that the *CN* connector uses, it will be important for declaring the types used by the *CN* connector. This declaration is made in the *For* loop (lines 6-9) that iterates through all types of *TYPES*: initially, *TYPES* will be added by other types returned by the *dependencies (tp)* function, this happens because the *DataType* types in SysADL are composed of attributes that can be from others types that needs to be declared. After calculating the type dependencies (line 7), each type $tp \in TYPES$ will be transformed to π-ADL by *RuleTypeDef2TypeDeclaration*. In this thesis, we chose to use only simple connectors with simple ports and whose only function is to connect simple ports. Thus, the *RuleConnector2CompositionBhv(CN)* was not implemented.

As explained in Section 3.1, each SysADL port will be transformed into a π-ADL connection, which requires a protocol declaration. These transformations are performed in the *For* loop (lines 13-17), which iterates over all the ports used by the *CN* connector: (i) adding to the *connections* attribute the result of the transformation of each port by *RulePortUse2ConnectionDecl*, and (ii) adding to the *protDecl* attribute created from the result of using *RulePortDef2ProtocolDeclaration* for each port. Finally, the *behavior* attribute

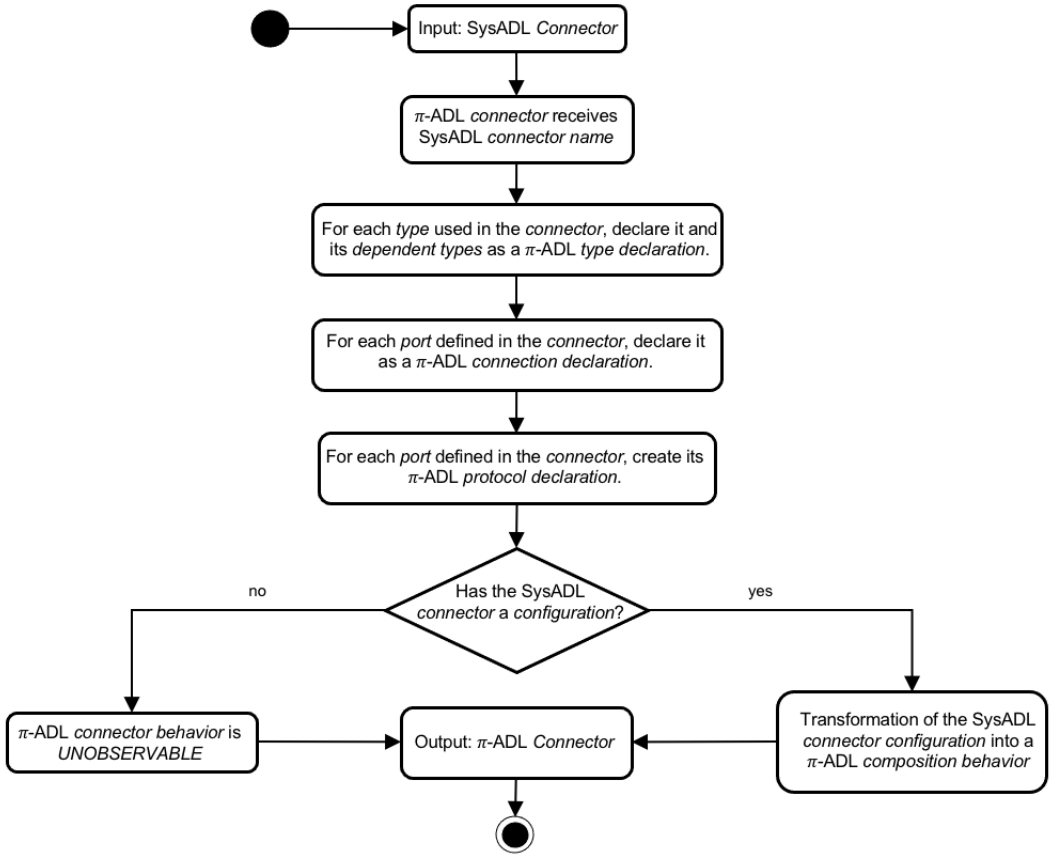


Figure 20 – Transformation flowchart from SysADL *ConnectorDef* to π-ADL *Connector*

will have a value of *UNOBSERVABLE* (line 19) if the *CN* connector's composite field has no defined *configuration*; otherwise, the *RuleConnector2CompositionBehavior* will transform the *CN* connector information into a *Composition Behavior Declaration*(line 21). The flowchart in Figure 20 exhibits an overview of the sequence of activities implemented by Algorithm 2.

Algorithm 3 presents the initial point of transformation of the SysADL component definition into the π-ADL component through the *RuleComponentDef2Component* rule. The name of the target component (*name* attribute) is assigned with the same name of the source component (line 3). The $TYPES$ set (line 4) aggregates all instances of the types within a port definition, which is important for declaring the types used by the component. The loop in lines 6-9 iterates through all types of the $TYPES$ set: initially, types returned by the *dependencies* function are added to $TYPES$. This is due to the fact that SysADL data types may comprise attributes of other types that need to be declared. After calculating the type dependencies, each type $tp \in TYPES$ is transformed into π-ADL by executing the *RuleTypeDef2TypeDeclaration* rule.

Each SysADL port is transformed into a π-ADL connection, which requires a protocol declaration. These transformations are performed in the loop defined in lines 13-17, which iterates over all the ports used by the component. It (i) adds the result of the transformation of each port from the *RulePortUse2ConnectionDecl* rule to the

Algorithm 3 Transformation rule for component

```
1:  procedure RuleComponentDef2Component(CP)
2:      Result cmp as PIADL!Component
3:      cmp.name ← CP.name
4:      TYPES ← ⋃(CP.ports.definition) TYPES
5:      cmp.typeDecls ← ∅
6:      for each tp ∈ TYPES do
7:          TYPES ← TYPES ∪ dependences(tp)
8:          tempType ← RuleTypeDef2TypeDeclaration(tp)
9:          cmp.typeDecls ← cmp.typeDecls ∪ tempType
10:     COMPS ← ⋃(CP.ports) COMPS
11:     cmp.connections ← ∅
12:     cmp.protDecl ← ∅
13:     for each ccp ∈ COMPS do
14:         tmpCom ← RulePortUse2ConnectionDecl(ccp)
15:         cmp.connections ← cmp.connections ∪ tmpCom
16:         tmpPT ← RulePortDef2ProtocolDeclaration(ccp)
17:         cmp.protDecl ← cmp.protDecl ∪ tmpPT
18:     ACTS ← ⋃(CP.name) ACTS
19:     if CP.Composite = ∅ then
20:         if CP.name ∈ ACTS then
21:             cmp.behavior ← RuleComponent2BhDeclaration(CP)
22:         else
23:             cmp.behavior ← UNOBSERVABLE
24:     else
25:         cmp.behavior ← RuleComponent2CompositionBh(CP)
```

connections attribute, and (ii) adds the result of the transformation of each port from the *RulePortDef2ProtocolDeclaration* rule to the *protDecl* attribute.

Finally, the algorithm checks if the component is composite (line 19). If so, the *RuleComponent2CompositionBh* transform the component information into a composition behavior declaration (line 25). If the component has no configuration defined, then another test is required to check if it has an associated activity in the $ACTS$ set (line 20). If so, the *RuleComponent2BhDeclaration* rule transforms the component's activity into a behavior declaration by executing the Algorithm 4 (line 21). Otherwise, the component is considered as a boundary component and hence its behavior is stated as $UNOBSERVABLE$ (line 23). The flowchart in Figure 21 exhibits an overview of the sequence of activities implemented by Algorithm 3.

Algorithm 4 describes the transformation rule for activities. It defines the *RuleComponent2BhDeclaration* rule to create a π-ADL behavior declaration for a SysADL component definition. As the behavior of a SysADL component is not defined within it, the algorithm needs to call the *getActivity* function to obtain the respective behavior definition for that component (line 4).

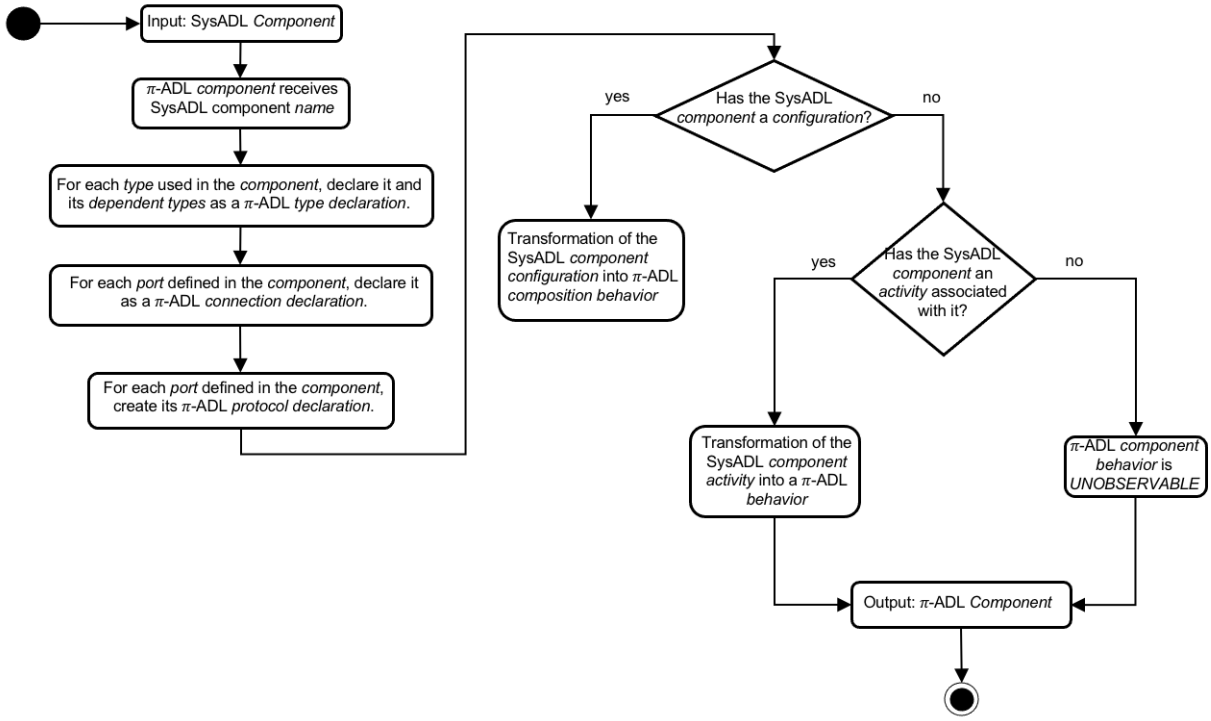


Figure 21 – Transformation flowchart from SysADL *ComponentDef* to π-ADL *Component*

The body of the π-ADL behavior declaration is built in two main steps. First, the $ACTIONS$ set receives all instances of actions used by the activity (line 5) and each action $an \in ACTIONS$ is transformed into π-ADL by executing the *RuleActionUse2FunctionDeclaration* rule and assigned to the body of the behavior declaration (lines 7-8). Next, the $FLOWS$ and $PARAM\_IN$ sets respectively receive all instances of activity flows and input pins used by the activity (lines 9-10). The loop in lines 11-16 iterates over all activity flows $fl \in FLOWS$ by checking if its source is within $PARAM\_IN$ (line 12). If so, the behavior declaration receives the transformation result of the activity flow by the *RuleActivityRelation2InputPrefix* rule (line 13), otherwise it receives the transformation result of the activity flow by the *RuleActivityRelation2OutputPrefix* (line 15). The flowchart in Figure 22 exhibits an overview of the sequence of activities implemented by Algorithm 4.

Algorithm 5 shows the *RuleArchitectureDef2Architecture*, which transforms *AC*, an *SYSADL!Architecture*, into *arc*, a *PIADL!Architecture*. The *name* attribute of *arc* receives the *AC* name whereas *arc.behavior* receive *RuleArchitecConfig2CompositionBehav* result, which transforms the *AC* component information into a *Composition Behavior Declaration*. The flowchart in Figure 23 exhibits an overview of the sequence of activities implemented by Algorithm 5.

The algorithms shown in this section were implemented using Atlas Transforma-

Algorithm 4 Transformation rule for component activities

```
1: procedure RuleComponent2BhDeclaration(CP)
2:     Result cmpBh as PIADL!BehaviorDeclaration
3:     cmpBh.body ← ∅
4:     actCP ← getActivity(CP)
5:     ACTIONS ← ⋃(actCP.actions) ACTS
6:     for each an ∈ ACTIONS do
7:         tmpBF ← RuleActionUse2FunctionDeclaration(an)
8:         cmpBh.body ← cmpBh.body ∪ tmpBF
9:     FLOWS ← ⋃(actCP.body.flows) FLOWS
10:    PARAM_IN ← ⋃(actCP.inParameters) PARAM_IN
11:    for each fl ∈ FLOWS do
12:        if fl.source ∈ PARAM_IN then
13:            tmpBF ← RuleActivityRelation2InputPrefix(fl)
14:        else
15:            tmpBF ← RuleActivityRelation2OutputPrefix(fl)
16:        cmpBh.body ← cmpBh.body ∪ tmpBF
```

Algorithm 5 Transformation rule for architectures

```
1: procedure RuleArchitectureDef2Architecture(AC)
2:     Result arc as PIADL!Architecture
3:     arc.name ← AC.name
4:     arc.compose ← RuleArchitecConfig2CompositionBehav(AC)
```

tion Language(ATL)[1](ATLAS Group LINA & INRIA Nantes, 2005). Listing 3.1 shows an excerpt of an ATL rule responsible for transforming a SysADL element of the *ComponentDef* class into an element of the *Component* class in π-ADL, in our project. It is possible to see that values are assigned to the element attributes of the Component class (π-ADL): name (lines 6-10), typeDecl (lines 11-13), connections (lines 14-24), protDecl (line 25), and behavior (lines 26- 31). More information about the definition of auxiliary procedures and their implementation are available at <https://bit.ly/3pYsobE>.

The result of the ATL transformation of SysADL into π-ADL is an Extensible Markup Language (XML) file with an architecture description in π-ADL, as can be seen in Figure 24, which shows an excerpt of the result of the M2M transformation of the FMS. The image shows an association between the result of the transformation of a FMS component (left) and π-ADL metamodel (right). Arrow (a) indicates that the result generated a *piADL:Component*, and that in the π-ADL metamodel, the *Component* class implements the *ArchitecturalElement* class and its attributes: *name*, *parameters*, *typeDecl*, *connections*, *protDecl*, and *behavior*. Still, on the left side, the arrows (b), (c), and (d), respectively, show that the *typeDecl*, *connections*, and *protDecl* attributes are from the *TypeDeclaration*, *ConnectionDeclaration*, and *ProtocolDeclaration* classes and therefore have other attributes inherited from these classes. It is possible to note that the model

[1] ATL Developer Guide - <https://wiki.eclipse.org/ATL/Developer_Guide>

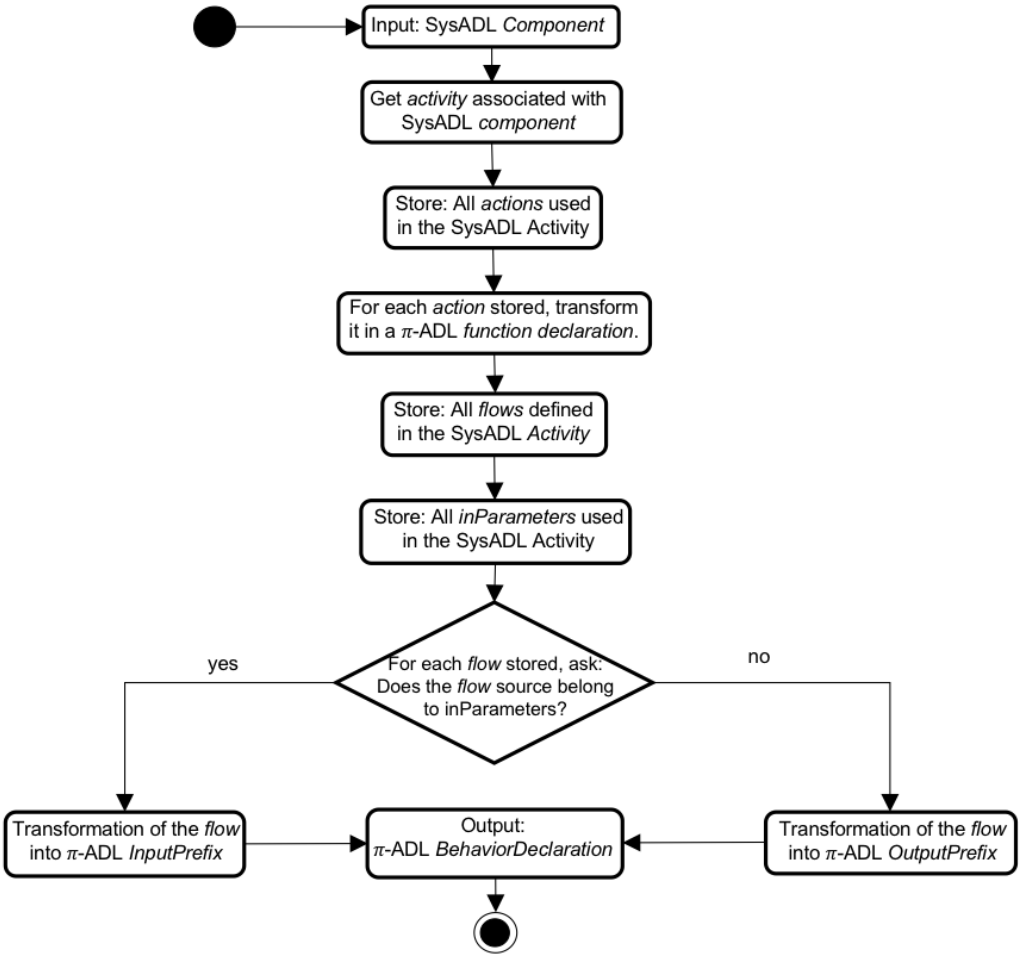


Figure 22 – Transformation flowchart from SysADL *Activity* to π-ADL *BehaviorDeclaration*

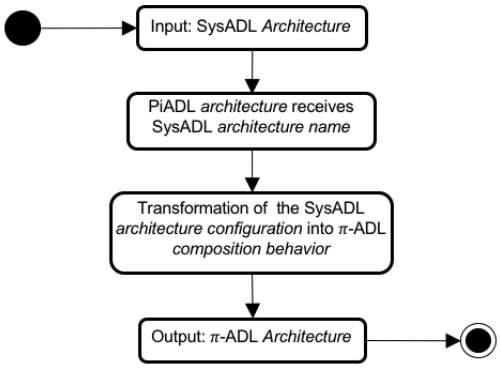


Figure 23 – Transformation flowchart from SysADL *Architecture* to π-ADL *Architecture*

generated in XML (left) has meta marks (tags) with the exact attributes labels of the metamodel (right), as indicated by arrows 1 to 5:

1. *name(EString)*: indicating the name of *piADL:Component*;
2. *typeDecl(TypeDeclaration)*: it has the *name* and *type* tags;

```
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 3.1 – An ATL rule that implements the transformation of component definitions in SysADL into component declarations in π-ADL.

3. *connections(ConnectionDeclaration)*: it has the *name*, *direction*, and *type* tags;

4. *protDecl(ProtocolDeclaration)*: It is composed of *protocol(ProtocolAction)* that have *connectionName*, *action* and *type* tags;

5. *behavior(BehaviorDeclaration)*: it has a body attribute that is a *BehaviorClause*, whose attributes do not appear in the image on the right.

3.4 M2T Transformation from π-ADL Model to π-ADL code

The result of the M2M transformation from SysADL to π-ADL is generated in XML. However, to verify architectural properties using Plasma for π-ADL (or other equivalent tool), generating the π-ADL source code is necessary. For this reason, one more step of model transformation was needed, this time, an M2T transformation to transform a π-ADL model expressed in XML into π-ADL source code.

The XML file resulting from the transformation between the SysADL and π-ADL models is already generated with the meta-markers from the π-ADL model classes and

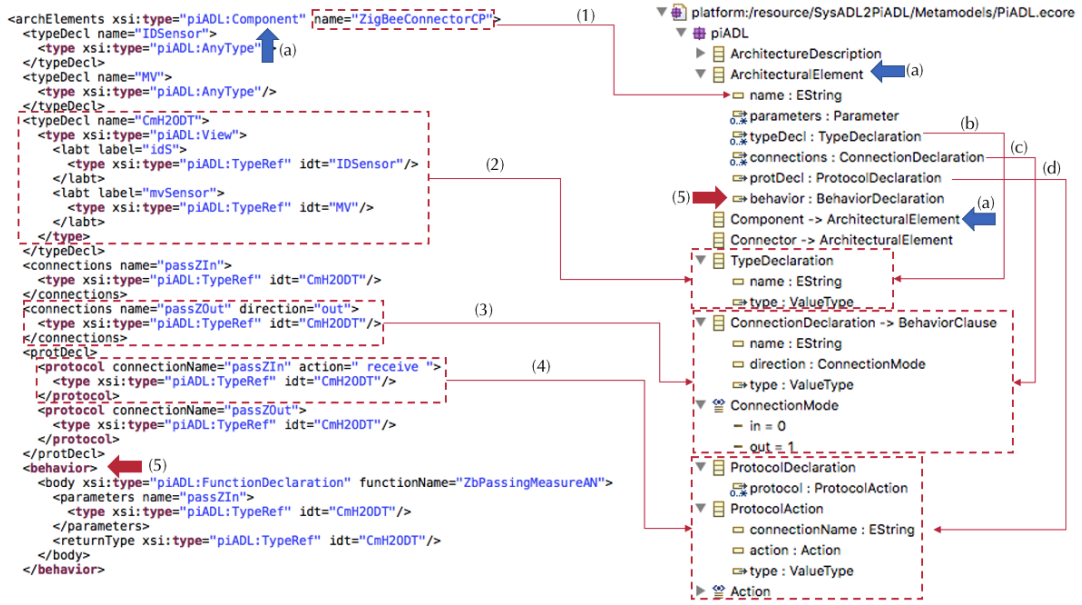


Figure 24 – *Component* element in a π-ADL model in XML representation (left) and the corresponding π-ADL (right).

attributes, so it is only necessary to create a form to map the text in the XML standard in the source code π-ADL.

The M2T transformation in this approach was implemented using Acceleo[2](OBEO, 2011), which is a template-based technology to create custom code generators from data sources available in Eclipse Modeling Framework[3](EMF) format.

As can be seen in Figure 24, the XML file resulting from the M2M transformation contains the meta-markers and sufficient data to reconstruct an architectural description in π-ADL source code from a template defined in Acceleo, following the π-ADL grammar syntax rules (in Appendix B.3). Figure 25 compares the XML model (left) and the π-ADL code resulting from the M2T transformation. The labeled arrows show the associations between the source model and the generated code:

1. The language reserved words, separators, parentheses, and brackets are positioned according to the guidelines in the grammar rules of π-ADL through the Acceleo template. The first arrow indicates that the Component's name is marked with the name attribute of the *archElements* tag;

2. The *typeDecl* tags have the *name* and *type* attributes, and the type as *piADL:View* still contains the *label* and *idt*(label type) attributes. In the example shown in the image, it was defined the types *IDSensor(Any)*, *MVSensor(Any)*, and *CmH2ODT(view)* composed of *idS(IDSensor)* and *mvSensor(MV)*;

[2] Acceleo - <https://www.eclipse.org/acceleo/>
[3] EMF - <https://www.eclipse.org/modeling/emf/>

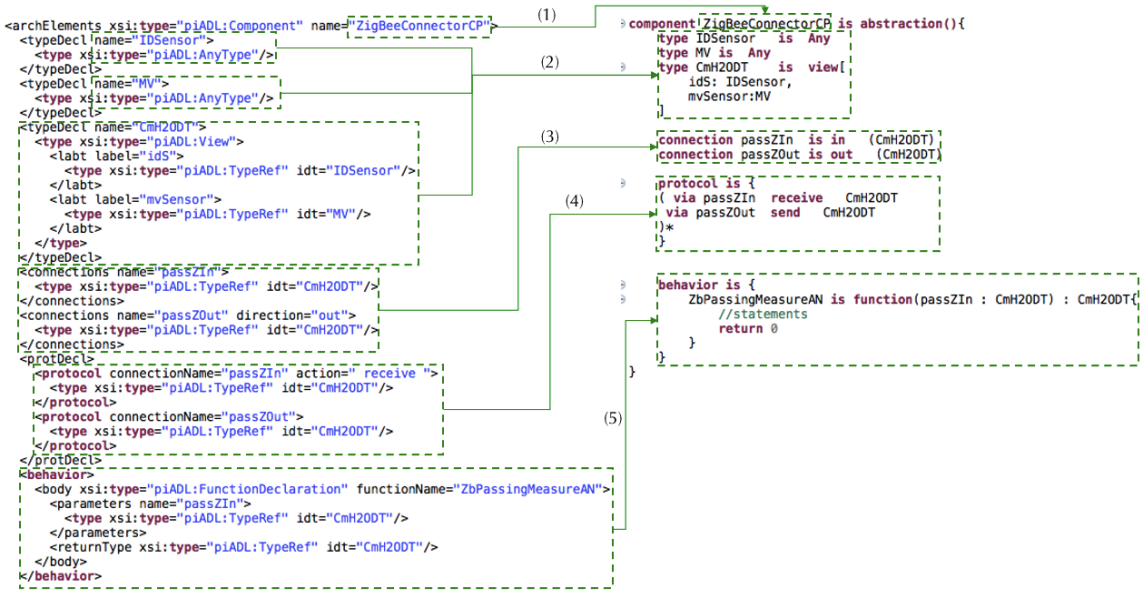


Figure 25 – Component element in a π-ADL model in XML representation (left) and the corresponding π-ADL codel (right)

3. The *connections* tags with the attributes *name*, *type*, and *direction* originate definitions of connections *passZIn* and *passZOut*, respectively *in* and *out* direction, and both *CmH2ODT* type;

4. The *protocol* tags have *connectionName*, *action*, and *type* attributes. In the example, it defines the protocols for *passZin* and *passZout* connections;

5. The behavior tag was defined as a *piADL:FunctionDeclaration* with *FunctionName*, *parameters* (*name* and *type*), and *returnType* attributes.

3.5 Implementation of the Mapping Process

Figure 26 illustrates the process to generate an architecture description in π-ADL from a SysADL model. In Phase One, the proposed transformation is supported by tools and artifacts complying with the Xtext Eclipse framework[4], namely (i) SysADL Studio, which supports architectural modeling and execution of SysADL models (LEITE et al., 2018), and (ii) the π-ADL plugin, which performs syntactic and semantic analysis of architecture descriptions in π-ADL (CAVALCANTE; OQUENDO; BATISTA, 2014). This infrastructure allows checking if an architecture description in SysADL is correct according to the language's syntactic rules and automatically generates a π-ADL file according to the ATL transformation rules.

The SysADL and π-ADL Ecore metamodels are ATL transformation parameters, and the ATL Module checks for semantic errors in the transformation. The ATL compiler

[4] Xtext - <https://www.eclipse.org/Xtext/>

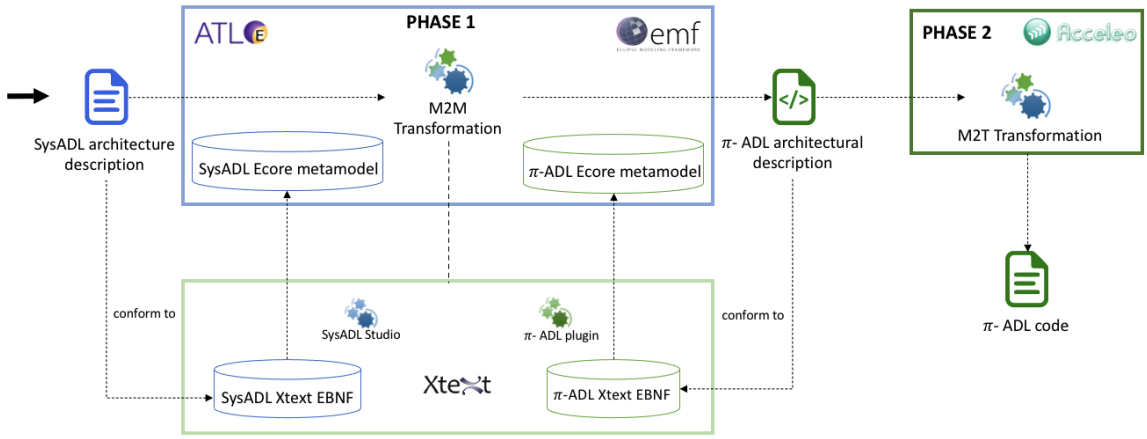


Figure 26 – Process for generating a π-ADL architecture description from a SysADL model

is automatically called for each ATL file, thus compiling transformations to the ASM assembly language. 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.

In Phase Two, the proposed transformation was supported by a template developed in the Acceleo tool, allowing code generation from the artifact resulting from Phase One of the approach. For the generated code to comply with the syntactic rules of the π-ADL language, it was necessary to align the template with the rules of the π-ADL grammar, the same used in Phase One of the process.

Appendice A detail the modeling of the SysADL architecture of the FMS. The artifacts used/generated in the M2M and M2T transformations to generate π-ADL architectural description from the SysADL model, are available in <https://bit.ly/3pYsobE>.

3.6 Conclusion of the Chapter

This chapter presented details about the SyADL to π-ADL Model transformation process, such as (i) Mapping SysADL Architecture Descriptions to π-ADL elements; (ii) SyADL denotational semantics in π-ADL; (iii) SysADL to π-ADL M2M algorithms transformation; and (iv) π-ADL model to π-ADL code M2T transformation.

The next chapter shows the details of generating an executable architecture from the transformed π-ADL model to show that the generated model expresses an equivalent architecture where it can observe the same execution.

4 Validation of the π-ADL Architecture

Software architectures are generally used to capture essential design decisions at a higher level of abstraction. However, they can also be used to derive system implementation. Relating a software architecture to its implementation is a mapping problem, where the challenge is to ensure that the architectural intent is reflected in the built systems (ZHENG; TAYLOR, 2012).

The design of software systems architecture is part of software development and provides a basis for achieving non-functional requirements (BEHNAMGHADER et al., 2017). The architecture must be incorporated into the implementation to meet these requirements. Sometimes, as explained by De Silva and Balasubramaniam (De Silva; BALASUBRAMANIAM, 2012), architectural erosion happens when the implementation of a software system diverges from its designed architecture. Architectural erosion can happen during software evolution, but it can also happen during the initial system implementation (GARLAN; SCHMERL, 2004). Architectural deviation can impede the evolution of systems and inhibit the achievement of architectural design objectives.

There are several strategies to minimize, prevent, or even repair architectural erosion, as mapped in the work of De Silva and Balasubramaniam (De Silva; BALASUBRAMANIAM, 2012). Among the categories of techniques to minimize this problem, some of them incorporate methods and tools available to transform architectural models into implementation, as happens in the automatic generation of code from the architectural description.

Considering reducing this gap between architecture and implementation, providing an automatic means to allow the generation of source code from an architectural description would be interesting. Following this way, this work realize the automatic generation of an executable architecture in the Go language (GOOGLE, 2009) from the automatically generated architectural description in π-ADL. With the automatic generation of an executable version of the architecture, it is possible to show that the architectural description is valid by observing that the execution of its behavior follows what was foreseen in the original design of the architecture.

Aiming to answer RQ2, we define a new phase in the process presented so far, where an executable architecture in the Go Language will be generated from the π-ADL model resulting from the transformation. Figure 27 updates the process shown in Figure 26, introducing phase 3. In this new phase, the π-ADL architectural model resulting from the M2M transformation of SysADL into π-ADL is used as input for a minor process that involves four steps: (i) the automatic generation of source code in the Go programming

from the resulting architectural description in π-ADL (CAVALCANTE, 2016); (ii) the insertion of Go code to implement the functional behavior of the architecture; (iii) the compilation and the execution of the source code to execute the architecture; and (iv) the comparison of the execution results concerning the definitions of the original SysADL architecture.

This chapter presents the details regarding the generation of an executable architecture from the transformed π-ADL model. It is structured as follows. Section 4.1 shows the correspondences between the main architectural elements of π-ADL and Go languages. Section 4.2 presents the technical details concerning the steps of phase 3, in charge of generating an executable architecture from the architectural description in π-ADL. Section 4.3 discusses the details of the architecture validation, using the FMS architecture generated by the process as an example. Section 4.4 contains the conclusions of the Chapter.

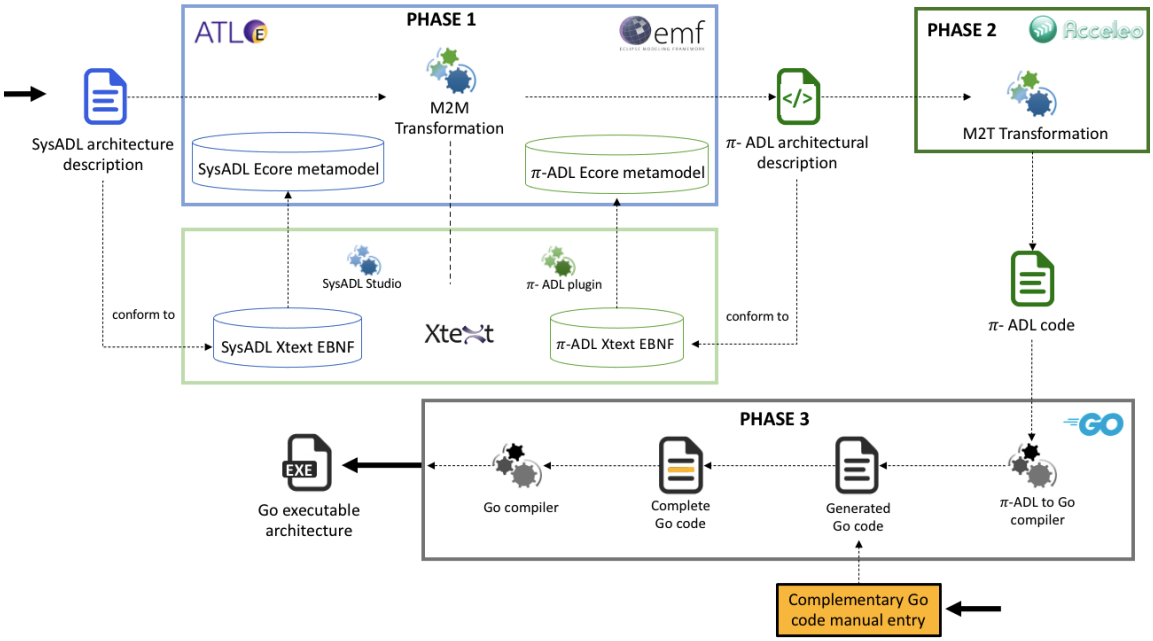


Figure 27 – Updating the process - Generating Go Language executable architecture from the π-ADL architecture description

4.1 Mapping a π-ADL software architecture into Go code

Some aspects were essential to decide which strategy to use to generate an executable architecture from an architectural description in our approach: (i) the possibility of deriving executable architecture from π-ADL, once the architectural description is in π-ADL; (ii) the provision of a tool to perform automated code generation, aiming to mini-

mize the previously mentioned architectural erosion problem; (iii) the possibility of easily integrate our approach with an automated property verification tool in the future.

Considering the mentioned aspects, we decided to use part of the methodology and tooling defined in Cavalcante (CAVALCANTE, 2016) to map the architecture described in the π-ADL model to an executable architecture in the Go language.

Cavalcante's work (CAVALCANTE, 2016) filled an existing gap between the architectural and implementation levels, helping to minimize the risk of architectural deviations and allowing validation of the architecture itself from source code generation in the Go programming language from the architecture descriptions expressed in π-ADL. Furthermore, Cavalcante's tool produces a stochastic executable model from π-ADL architecture descriptions, thus allowing for property verification, realizing SMC as a plugin for Plasma Lab (PLASMA, 2022).

The tool presented by Cavalcante's work offers two essential services to our strategy: (i) The syntactic checker validates a π-ADL source code generated by MDD transformations at phases 1 and 2 of our process, i.e., confirming that our approach generates an architectural description conform to π-ADL metamodel and grammar; (ii) The compile generates source code in the Go language, making the architectural execution feasible. With this observable architecture, it is possible to validate its compliance with the specification in the original SysADL architecture.

The π-ADL and Go common formal basis is the processes algebra of π-calculus, a genuine choice to generate an executable architecture from π-ADL. Go and π-ADL has a direct relationship between some language elements, such as the use of connections in π-ADL and channels in Go as a means of communication and synchronization between concurrent processes. Table 5 summarizes the correspondences between the main architectural level elements of π-ADL and the implementation level elements of Go, each described below.

Table 5 – Correspondences summary between the main architecture-level elements of π-ADL and implementation-level elements of Go (CAVALCANTE, 2016)

π-ADL	**Go**
Architectural abstraction (component, connector, architecture)	Function (goroutine)
Behavior of architectural abstraction	Body of function (goroutine)
Connection	Channel
Connection declaration	Instantiation of channel map
Instantiation of architectural element	Call to goroutine
Unification of connections	Special-purpose function (goroutine)
Coordinating behavior	Main function

Architectural abstraction and their behavior. The architectural elements, Components and Connectors, and the Architecture are defined in π-ADL as abstractions over behaviors. In Go, these elements are represented by functions called goroutines, which are functions that can be executed concurrently and communicate using typed

channels. Thus, it is observed that goroutines are equivalent to the notion of communicating processes in π-calculus. The respective architectural abstraction name and the required parameter list define such functions' signatures. In turn, the body of these functions comprises the respective behavior of the element.

Connections. In π-calculus, interactions between concurrent processes occur by communication channels to synchronize such processes through sending and receiving data. Similarly, π-ADL uses connections, abstractions representing communication channels between architectural elements. Using typed connections, components and connectors can send (output connections) and receive (input connections) any value from existing types and the connections themselves. Like π-calculus, Go uses typed channels to send and/or receive values between concurrent processes (goroutines to be synchronized) so that connections in π-ADL are mapped to channels in Go. The type of values transmitted over a channel is specified in the connection statement.

Connection declaration. The connections defined in the structure of architecture components and connectors are represented in Go by channel maps. Channel maps are an associative, unordered, non-sequential array of <string, channel> pairs that take the names of connections declared as keys and map these keys to channel objects that represent the connections themselves.

Instantiation of architectural element. In π-ADL, components and connectors are instantiated within the architecture by a simple instruction that only involves informing the name to which the instance will be referred within the architecture and of which architectural element it is an instance. In the Go language, the instantiation of components and connectors in architecture involves two steps: (1) to create channel maps that represent the set of connections that compose such architectural elements; and (2) to launch of the goroutines that respectively represent these architectural elements within the function associated with the architecture. In goroutine calls, the channel maps representing the created instance are provided as a goroutine parameter.

```
component SensorCP is abstraction(){
     type MV is Real
     connection senseIn  is in(MV)
     behavior is {
          unobservable
     }
}

architecture FMSarch is abstraction(){
     behavior is{
          compose{
               S1 is SensorCP()
               //other declared instances
          }where{
               //unification of connections
          }
     }
}
```

```
func SensorCP(conn map[string]interface{}){
     //behavior implementation
}

func FMSarch(){
     S1 := map[string]interface{}{
          "senseIn" : make(chan float64),
     }
     //other declared instances
     go SensorCP(S1)
}
```

Figure 28 – Correspondence of architectural elements instantiation in π-ADL and Go

Figure 28 illustrates an example of the excerpt from the *FMSarch* architecture. In this Figure, the left side exhibits the architectural description in π-ADL, while the right side shows the equivalent source code in Go. The *FMSarch* architecture comprises a *SensorCP* component and other architectural elements suppressed from the image for simplicity. Creating an instance of the *SensorCP* component (*S1*) first involves creating the channel map representing the set of connections declared for this component (*senseIn*). This component is then executed by calling the *SensorCP* goroutine and providing the previously created channel map as an input parameter.

Unification of connections. To realize the connection between two architectural elements π-ADL defines *unification*, that links the connections of both elements so that they can communicate and transport values, connections, or even architectural elements. In Go, this process is implemented by a special-purpose function called *unify* (CAVALCANTE, 2016), which receives the connections to be unified as parameters. Such a function reads the contents of the sending channel (output connection) and writes them to the receiving channel (input connection).

Coordinating behavior. A coordinating behavior performs an application (similar to a call) to enable the execution of an architecture. In Go, this behavior is represented by the main function, the first function called when a Go program is executed. In this way, the main function calls the goroutine, which represents the architecture itself, which in turn calls the goroutines associated with instances of architectural elements (components and connectors).

4.2 Executable architecture generation procedure

The π-ADL language has an open-source toolset to support the formal description, analysis, refinement, code generation, and evolution of software architectures (OQUENDO et al., 2004; OQUENDO, 2006; ARCHWARE, 2022). These tools allow the compilation of architectural models into their executable representations and the formal analysis and evolution of such models. Among the tools available to perform executable architecture generation from π-ADL, our approach uses the more recent tool since its creation. It supports the evolutions of the π-ADL language over the years and the requirements of new generation software systems (e.g., distribution, large scale, concurrency, and dynamism) (CAVALCANTE, 2016).

This section describes the process of producing source code in Go from architecture descriptions in π-ADL using the unidirectional generative approach developed by Cavalcante (CAVALCANTE, 2016).

4.2.1 π-ADL textual editor and syntatic checker

The textual editor used is based on the Eclipse platform and allows architects to make architectural descriptions using π-ADL, in addition to being able to automatically generate the implementation code in Go. A screenshot of the tool is shown in Figure 29.

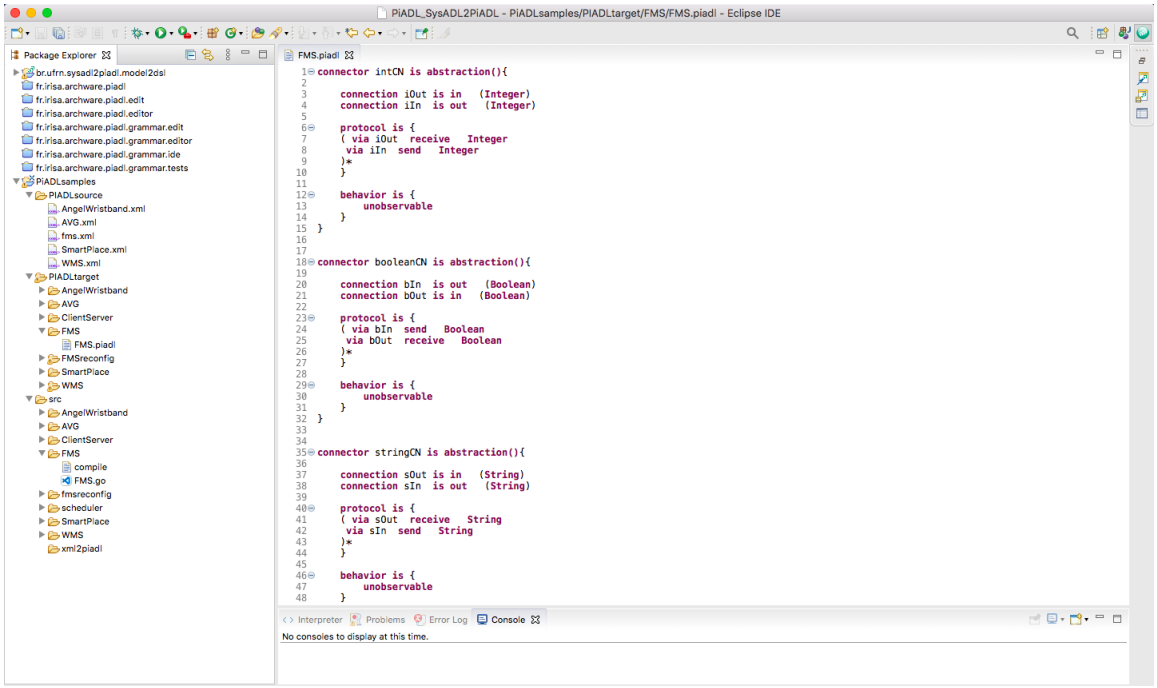


Figure 29 – Screenshot of the plugin Eclipse-based π-ADL textual editor

The π-ADL plugin infrastructure is integrated into the Eclipse development environment and provides the π-ADL text editor with useful features such as (CAVALCANTE, 2016):

- syntax highlighting, a helpful feature for distinguishing keywords from identifiers;
- syntactic and semantic validation of architecture descriptions;
- interpretation of expressions, which is useful for type checking purposes;
- statement/expression based on the syntactic rules;
- automatic build on save for automatically generating Go source code from the architectural description in π-ADL (if they are correct according to the language's syntactic and validation rules) when saved in the language editor; and,
- other practical features more aimed at software architects designing an architectural description, such as auto-formatting, to show error and warning alerts, thereby enabling architects to detect and fix errors and potential problems early, and content assistance, which provides suggestions on how to complete a sentence;

As mentioned before, the use of this tool has two essential roles in our approach: (1) to demonstrate that the π-ADL source code generated from the transformation of the architectural description into SysADL is syntactically correct, showing that the approach is valid and correctly generates the architecture code in π-ADL; and (2) to generate an executable version of the architecture, showing that through our MDD-based approach it is possible to generate valid π-ADL architectures that can even be executed.

4.2.2 Code generation procedure to the Go Language

The main objective to generate code in the Go language is to derive an executable architecture whose execution makes it possible to observe the architecture behavior and validate if it complies with the architecture requirements.

As shown in Figure 27, the output of our process in phase 2 is a textual file containing the π-ADL code of the architecture initially described in SysADL. In order to generate the respective Go code, the architecture must be verified as correct by the syntactic analyzer that is part of the generator tool. In addition to parsing the textual architecture description against the rules defined in the π-ADL grammar, the code generator also uses some validators to parse a π-ADL architecture description semantically.

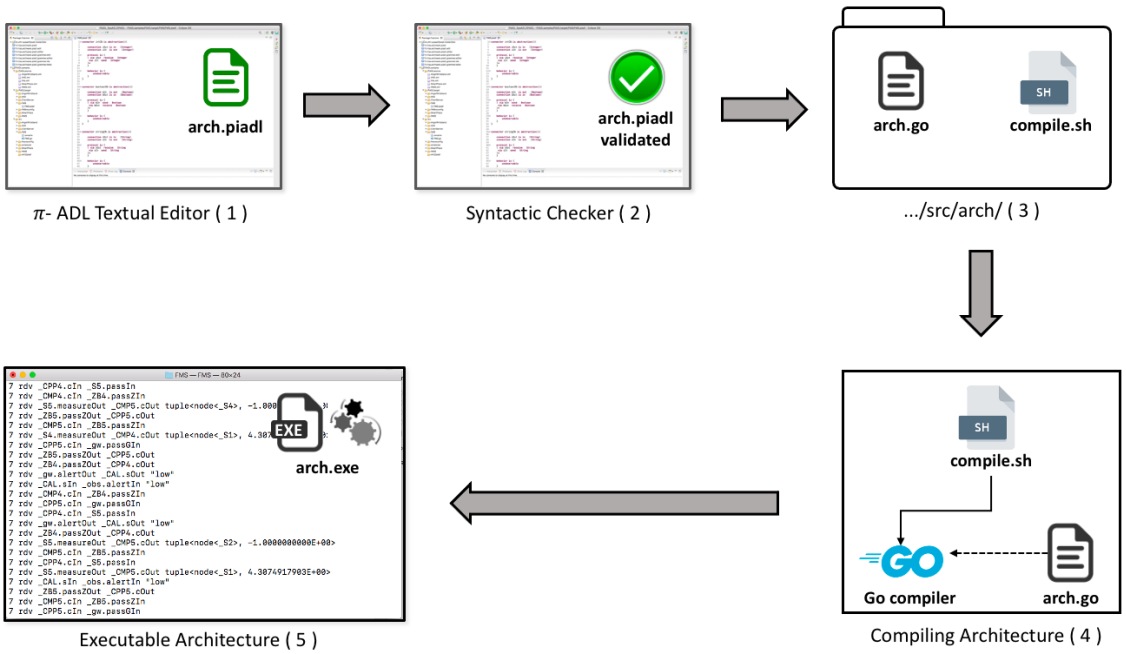


Figure 30 – Executable architecture generation procedure

For the tool to carry out all the necessary analyses and then automatically generate the Go code, the sequence of steps are as follows, starting with the π-ADL tools (text editor and syntax checker), as illustrated in Figure 30:

1. The first step consists in opening the π-ADL architecture description in the π-ADL textual editor;

2. In this step the syntactic checker on the tool validates the π-ADL source code;

3. In this step if the file is valid, the π-ADL tool automatically generates two files in the same directory: a Go file with the architecture, and a script called *compile*;

4. The next step involves running the *compile* script, that call the Go compile[1] (version 1.13.15), which receives the π-ADL architecture description file;

5. Finally, the executable architecture is generated.

Figure 31 shows an excerpt of the Go code generated from the π-ADL description of the *ZigBeeConnectorCP* component. This component is implemented in Go by the *ZigBeeConnectorCP* function, which receives a channel map (*conn*) representing its declared connections as a parameter. In the *ZigBeeConnectorCP* function, the value received by the *passZIn* input channel is assigned to a variable (*m*) to be sent by the *passZOut* output channel.

```
component ZigBeeConnectorCP is abstraction(){
    type IDSensor  is  String
    type MV   is  Real
    type CmH2ODT   is  view[
        idS: IDSensor,
        mvSensor:MV
    ]

    connection passZIn  is in (CmH2ODT)
    connection passZOut is out (CmH2ODT)

    protocol is {
    (via passZIn  receive CmH2ODT
     via passZOut  send   CmH2ODT)*
    }

    behavior is {
        via passZIn receive m : CmH2O
        via passZOut send m
        behavior()
    }
}
```

```
func ZigBeeConnectorCP(conn map[string]interface{}){
    m := <- conn["passZIn"].(chan CmH2O)
    conn["passZOut"].(chan CmH2O) <- m
    go ZigBeeConnectorCP(conn)
}
```

Figure 31 – Description of the ZigBeeCP in π-ADL (left) and corresponding code in Go (right)

The model-driven approach proposed in this thesis focuses on the transformations of elements from the structural and behavioral point of view of an architectural description in SysADL. For this reason, the consequent transformations for π-ADL and Go are limited to these elements. Thus, it is necessary to insert instructions with functional behavior in Go intertwined with the automatically generated code to complete the executable behavior of the architecture.

Figure 32b shows a part of the Go code generated from the π-ADL description of the *GatewayCP* component (fig. 32a). This component is implemented in Go by the *GatewayCP* function, which receives as a parameter a channel map representing its declared connections (excerpt omitted for image simplification). In the figure, delimited by

[1] Go Compiler available at https://go.dev/dl/

a frame, there is the piece of code manually inserted into the automatically generated code (excerpt marked with square bracket) by Cavalcante's tool (CAVALCANTE, 2016).

```
component GatewayCP is abstraction(){
     type IDSensor  is  Any
     type CmH2ODT   is  view[
          idS: IDSensor,
          mvSensor:MV
     ]
     connection passGIn  is in   (CmH2ODT)
     connection tickIn   is in   (Boolean)
     connection alertOut is out   (String)

     behavior is {
          StoreInfoSensorsAN is function(passGIn : CmH2ODT){
               unobservable
          }
          SendingAlertLevelAN is function() : String{
               unobservable
          }

          via tickIn receive tick:Boolean
          if(tick)then{

               via passGIn receive m:CmH2ODT
               via alertOut send SendingAlertLevelAN()
          }
          behavior()
     }
}
```

(a)

```
func GatewayCP(){
     tick := <- conn["tickIn"].(chan boolean)
     if(tick){
          m := <- conn["passGIn"].(chan CmH2ODT)
          StoreInfoSensorAN(m)
          conn["alertOut"].(chan string) <-
          SendingAlertLevelAN()
     }
     go GatewayCP ()
}

func SendingAlertLevelAN() String {
    sum:=0.0
    nbSensors:=0.0
    for _,val := range lastLevels{
        sum+=val
        nbSensors+=1
    }
    if sum/nbSensors > 4 {
        return "high"
    } else {
        return "low"
    }
}
```

(b)

Figure 32 – GatewayCP component in π-ADL (left) and corresponding complete Go Code (right)

4.3 Validation of the architecture

To validate the process related to the approach proposed in this thesis, we generated an executable version of the FMS previously presented in Section 2.1. Its architectural description through SysADL diagrams is available in Appendix A; The SysADL textual versions, resulting from the M2M transformation (XML), resulting from the M2T transformation(π-ADL code) are available in the <https://bit.ly/3pYsobE>. The executable architecture resulting from phase 3 of the process, with generation in Go code and insertion of complementary code, can be seen in <https://bit.ly/3pYsobE>.

The configuration version *FMSa5ARCH* of FMS architecture to generate executable architecture has 18 components:

- 5 *SensorCP*: S1, S2, S3, S4, and S5;
- 5 *SensorEnvConnectorCP*: SE1, SE2, SE3, SE4, and SE5;
- 5 *ZigBeeConnectorCP*: ZB1, ZB2, ZB3, ZB4, and ZB5;
- 1 *GatewayCP*: gw;
- 1 *EnvCP*: EnvCP;
- 1 *ObserverConnectorCP*: obs;

Forty connectors of various types are used to interconnect these components. This section focuses on the connectors used to illustrate the validation, which is listed below. Other details about the FMS architecture are available in Appendix A. Figure 33 illustrates a connector subset of the *FMSa5ARCH* are used to demonstrate the architecture validation. They are described below:

- (a) *CPP5(CmH2OCN)* to connect output *passZOut(ZB5)* to input *passGIn(gw)*;
- (b) *CAL(stringCN)* to connect output *alertOut(gw)* to input *alertIn(obs)*;
- (c) *CT (booleanCN)* to connect output *tickOut(EnvCP)* to input *tickIn (gw)*.

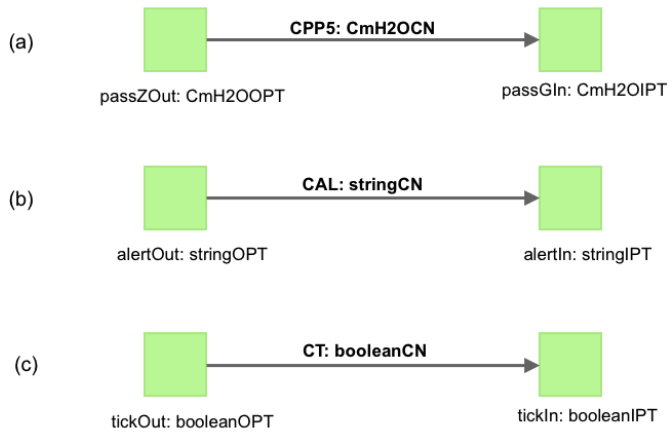


Figure 33 – SysADL connectors subset of *FMSa5ARCH*

We separate an excerpt of generated FMS architecture to exemplify how to realize the validation of the architectural behavior according to original SysADL architecture. It is nececessary to know the SysADL architectural description of the FMS parts used in the sample. In this way, Figure 34 presents details about the structure and behavior of the *GatewayCP* component.

Figure 34(a) shows a part of the structural definition of the *GatewayCP* component with input ports *passGIn (CmH2OIPT)*, *tickIn (booleanIPT)*, and output port *alertOut (stringOPT)*. Figure 34(b) shows a resume of the *GatewayCPAC* activity behavioral diagram with the *SendingAlertLevelAN* action. The action receives a *tickIn* information, performs a calculation, and decides to send a *"high"* or *"low"* alert level, which is specified by the *SendingAlertLevelEQ* constraint in Figure 34(c), in accordance with the Go code showed in Figure 32(b).

The architecture configuration in Figure 35 shows the simplified configuration where the *GatewayCP* behavior is executed. In addition to the *Gateway(gw)* component, the configuration excerpt involves other elements such as *ZigBeeConnectionCP(ZB)*, *EnvCP(EnvCP)*, *ObserverConnectorCP(obs)*, and connectors *booleanCN(CT)*, *CmH2OCN*

(CPP5), and *stringCN(CAL)*. The components had their structures simplified in the image, suppressing some ports.

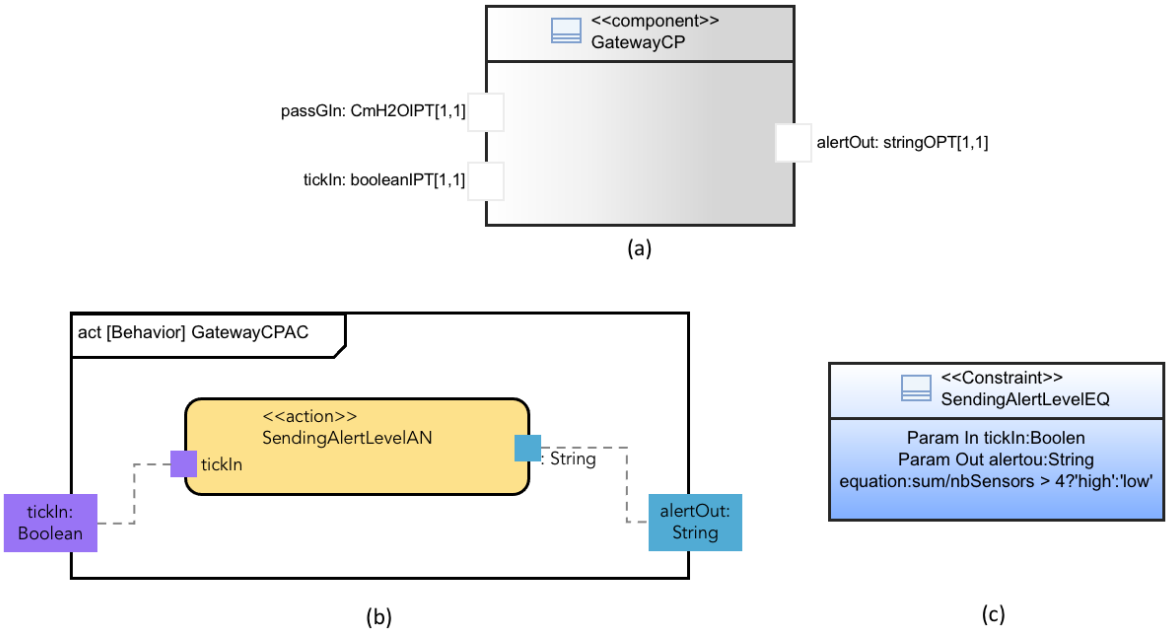


Figure 34 – *GatewayCP* component - SysADL Structure and Behavior definition

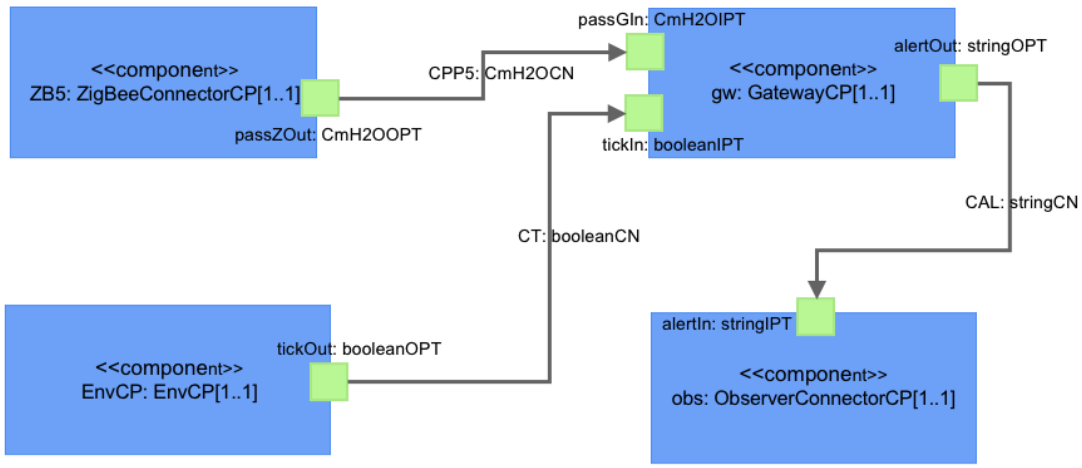


Figure 35 – IBD Excerpt of *FMSa5ARCH* architecture configuration

Figure 36 presents the actions sequence in *GatewayCP* behavior execution, simulating the send values between the components.

- *EnvCP* sends a *tick (true)* signal to *GatewayCP* to notify that a new synchronization cycle has started;
- *ZigBeeConnectionCP* send a *measure (S4, 4.38)* of the volume of water to *GatewayCP*;

- *GatewayCP* calculates the alert level (*"low"*); and
- Send a message (*"low"*) to *ObserverConnectorCP* with the current alert level;

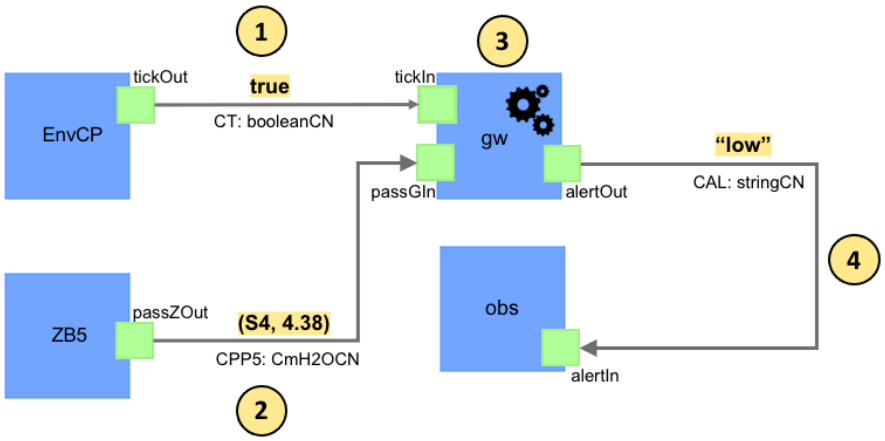


Figure 36 – The sequence of actions in executing the *GatewayCP* behavior

After carrying out the executable architecture generation process detailed in Section 4.2, a test execution of the *FMSa5ARCH* architecture produced an output whose execution log excerpt is shown in Figure 37. The sequence resembles the one illustrated in Figure 36. The highlighted section shows the following sequence of communication:

1. Via communication channel between *ZB5 (passZOut)* and connector *CPP5 (cIn)*, *ZB5* send a sensor measurement to the connector *CPP5*;
2. Via channel between *CPP5 (cOut)* and *gw (passGIn)*, *CPP5* forwarded a sensor measurement to the gateway *gw*; *gw* calculates the alert level *"low"* and sends
3. via connection *alertOut* to *CAL* connector;
4. Finally, *CAL* forwarded to*obs* the *"low"* string.

Inspecting the above excerpt architecture *FMSa5ARCH* execution log (fig. 37), it was possible to observe that the observable structure and behavior comply with that specified in SysADL(fig. 34(a), and 34(b, c), respectively). It is also noticed that the executable architecture reproduces the SysADL configuration shown in Figure 35, and the respective π-ADL and Go code versions (fig. 32). Then, it is possible to say that this architecture excerpt is validated because its behavior complies with the expected behavior specified in the original SysADL architecture.

The Annex A shows an excerpt of an output file with the result of executing the *FMSa5ARCH*. The complete output file version is available at <https://bit.ly/3pYsobE>.

```
1 rdv _SE5.valueSentOut _CSV5.bOut true
1 rdv _CMP4.cIn _ZB4.passZIn
1 rdv _SE1.sFailedSOut _CFS1.sidOut 0
1 rdv _S4.measureOut _CMP4.cOut tuple<node<_S3>, -1.0000000000E+00>
1 rdv _SE4.valueSentOut _CSV4.bOut true
1 rdv _ZB5.passZOut _CPP5.cOut
1 rdv _CPP5.cIn _gw.passGIn
1 rdv _gw.alertOut _CAL.sOut "low"
1 rdv _CSV1.bIn _EnvCP.valueSentIn true
1 rdv _CFS3.sidIn _EnvCP.sFailedEIn 0
1 rdv _CSV5.bIn _EnvCP.valueSentIn true
1 rdv _CFS1.sidIn _EnvCP.sFailedEIn 0
1 rdv _CSV4.bIn _EnvCP.valueSentIn true
1 rdv _CFS5.sidIn _EnvCP.sFailedEIn 439
1 rdv _CVS1.mIn _S1.senseIn
1 rdv _CAL.sIn _obs.alertIn "low"
1 rdv _SE1.valueSentOut _CSV1.bOut true
```

Figure 37 – Excerpt execution log of architecture *FMSa5ARCH*

4.4 Conclusion of the Chapter

This chapter presented details about the generation of executable architecture in Go language from π-ADL architectural description. The strategy used has a sequence of four steps: (i) the automatic generation of source code in the Go programming from π-ADL architecture using Calvalcante's tool (CAVALCANTE, 2016); (ii) the insertion of Go code to complementary to implement the functional behavior of the architecture; (iii) the compilation and the execution of the source code to execute the architecture; and (iv) the comparison of the execution results concerning the definitions of the original SysADL architecture.

The next chapter discusses the details of verifying architectural properties using execution traces of architecture, investigating the equivalences of SysADL and π-ADL architectures traces.

5 Verification of Architectural Properties

As mentioned earlier, one of the objectives of adopting a formal ADL is the possibility of carrying out activities involving the analysis of software architectures. Perry and Wolf (PERRY; WOLF, 1992) identify the types of analysis of software architectures that they consider fundamental to be supported and described as: (i) consistency of architectural style constraints; (ii) satisfaction of architectural styles by an architecture; (iii) consistency of architectural constraints; (iv) satisfaction of the architecture by the design; (v) establishment of dependencies between architecture and design, and architecture and requirements; and (vi) determination of the implications of changes in architecture or architectural style on design and requirements, and vice versa. In the survey by Ali (ALI et al., 2018), the interviewed professionals believe that consistency of architectural constraints would be useful in supporting various software development activities such as auditing, evolution, and quality attribute assurance.

Considering the essential role that the definition of architectural constraints has in the architecture design and the importance of their consistency for the correctness and quality assurance of the attributes of the system under development, this chapter shows how it is possible to verify architectural constraints from the architectural description in a source model in SysADL, comparing with the verification carried out in the target model π-ADL. The goal is to show how it is possible to verify architectural properties considering the approach proposed in this thesis. In addition, it also shows that this process's target and source architectures are equivalent to each other, thus responding to RQ3 expressed in Chapter 1.

According to (HOJAJI et al., 2019), a model execution tracing captures relevant information about the execution of an executable model, such as execution states, events that occur during execution, changes in an execution state, inputs processed, and outputs produced. Zoor et al. (ZOOR; APVRILLE; PACALET, 2021) mentions that trace analysis is a promising solution, providing relevant information about the system execution. Traces are collected by simulating a model or running the system. The analysis of execution traces is a well-established technique in the literature, with several objectives, such as. Trace analysis is a powerful approach to understanding and optimizing a system's behaviors, debugging it, performing model verification (HOJAJI et al., 2019), or performing other verifications (KEMPER; TEPPER, 2009; DEANTONI et al., 2010). Tracing the execution of a model is the foundation that supports various dynamic Verification and Validation (V&V) activities at the model level, such as debugging, model verification, testing, trace analysis for program comprehension, and other dynamic analysis tasks. These

techniques rely primarily on execution traces as a representation of system behavior. The analyses of traces can provide valuable insights into past events and actions. However, it is important to acknowledge the limitations of trace analysis, particularly in determining whether something has been rejected. To manage these limitations, it is necessary to complement trace analysis with additional information sources.

The verification in the process thesis is carried out manually in this thesis without tool support. But in Section 5.4, we discussed the possibilities for verification with tool support and that no structural change is necessary for our process to realize this possibility.

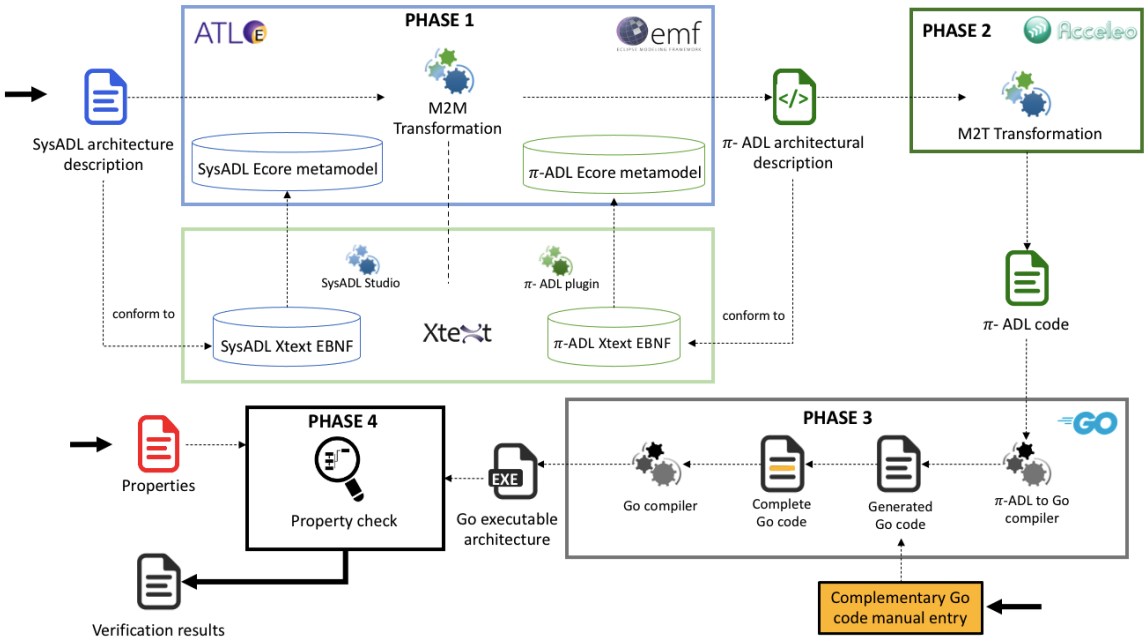


Figure 38 – The process to formally enrich SysADL and check properties

This Chapter details property verification in architectures through the approach proposed in this thesis. Section 5.1 discusses the Model Checking technique, one of the most used formal techniques for verifying properties. Section 5.2 discusses architectural properties and their types. Section 5.3 presents a proof of concept showing how properties expressed about software architecture can be verified in our approach. Section 5.4 presents some model checking support tools, showing relevant characteristics in using them. Finally, Section 5.5 contains the conclusions of the Chapter.

5.1 Model Checking

Formal verification of architectural properties through the model checking technique uses architectural models as a formal way of specifying system requirements, making it possible to attest that the program's behavior is in accordance with the correction specification. The basis of model checking is verifying that the specified properties are satisfied for that architectural model. The specification of properties is usually written in a logic

language, but it can be used with languages like OCL or even behaviors specified in state diagrams.

According to Clark et al. (CLARKE; EMERSON; SIFAKIS, 2009), model checking is a technique for verifying finite state concurrent systems. One benefit of this restriction is that verification can be automatically performed. The model checker typically uses an exhaustive search of the finite state space of the system to determine if some specification (property of the system) is true or not.

In the model checking context, state exploration is systematically exploring all possible states of a system or model to verify its correctness with respect to a given specification or property (BAIER; KATOEN, 2008). In this way, the model checking process is an exhaustive exploration of the state space of a system to see if an error state is achievable. If it happens, concrete counter-examples are produced whose execution takes the system to the error state.

As it is a technique that involves exhaustively traversing the state transition graph of the system for searching error states, it is entirely plausible to think of an automated approach to carry out the model checking process, which involves three fundamental parts: (i) an *executor* that receives the model with the state transition rules and the input states, and generates all reachable states; (ii) a *simulator* that generates all possible execution traces from the reachable states; and, (iii) a *verifier* that checks whether the predicate is satisfied and, if not, generates this state and path as a counterexample when traversing the execution traces for a given predicate. Figure 39 shows a generic model checking process for architectural descriptions, where a model checking tool receives the architectural model and the specified properties and performs the verification of the architectural properties, returning a "yes" from the provided model if it satisfies the given specifications and, if not, it generates a counterexample.

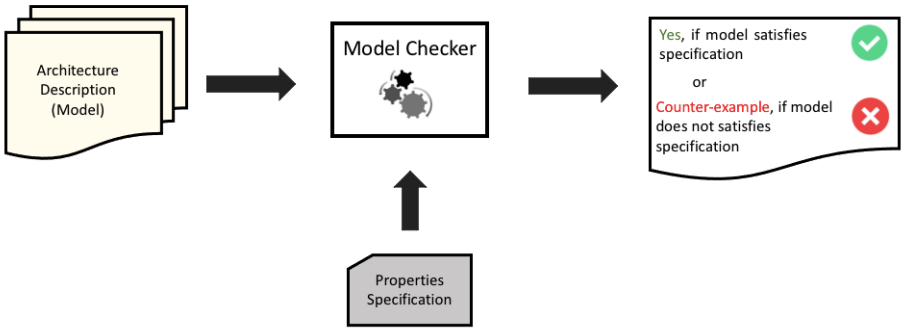


Figure 39 – Model Checking Process

It is essential to understand about the model checking process because we perform a similar manual process, in this thesis, to verify architectural properties as a means of validating the proposed approach shown in the figure 38, using the FMS architecture and properties specified for the model described in Section 5.3. Furthermore, in Section 5.4,

we will discuss alternatives to insert a model checking process supported by tools.

5.2 Architectural properties

Properties refer to statements or specifications that describe a system's desired behavior or requirements, representing how a system should operate. Such a specification aims to capture different functional and non-functional attributes to support various forms of validation and verification (DARSIE, 2012). These properties are typically expressed in a formal language and are used to verify whether a given system model satisfies the desired behavior.

Properties are an important part of the model checking process, as they provide the formal specification of the expected behavior of a system. Properties are typically checked against the system model using an automated tool, which explores all possible system states and transitions to determine whether the properties are valid.

There is a close relationship between a system's properties and quality attributes. The quality attributes refer to the desired characteristics of a software system. Such attributes are often used to assess the quality of a software system and its ability to perform its intended purpose. The architecture of a software system can be evaluated based on its properties, which can be classified into various categories. According to the product quality model defined in ISO/IEC 25010 (ISO/IEC, 2013), eight quality characteristics are considered, as shown in Figure 40.

Figure 40 – Software product quality model(ISO/IEC, 2013)

Not all these characteristics of the product quality model can be verified using formal techniques such as Model Checking. However, properties often verified using formal techniques are related to these model characteristics, such as correctness, safety, consistency, deadlock-freedom, livelock-freedom, reachability, etc. As a proof of concept of our approach, in this chapter some properties associated with the functional correctness of the FMS architecture will be specified and verified.

5.3 FMS architectural properties

This section presents a sample verification of 2 properties related to the functional correctness of the FMS as proof of concept of the approach proposed so far. The properties are related to the behavior of the *ZigBeeConnectorCP* and *SensorCP*.

ZigBeeConnectorCP Behavior. As detailed in Appendix A, the *ZigBeeConnectorCP* component was described in the structural view of SysADL with an input port *passZIn(CmH2ODT)* and an output port *passZOut(CmH2ODT)*. In the transformation to π-ADL, the *ZigBeeConnectorCP* component defines the *passZIn(CmH2ODT)* and *passZOut(CmH2ODT)* connections, respectively, analogous to its input and output ports described in SysADL. The defined behavior for *ZigBeeConnectorCP* has one single action *ZbPassingMeasureAN*, that receives a volume measurement from a previous *SensorCP* via *passZIn* port and forwards the volume measurement value to the next *SensorCP* or *GatewayCP* via *passZOut*.

Figure 41 illustrates the sequence of steps involved in the behavior of *ZigBeeConnectorAC* activity. Figure 41 illustrates activity without data on its pins. Figure 41b shows that the activity received data on the *passZin* pin. After receiving the input pin data, the action starts *ZbPassingMeasureAN* - Fig. 41c -, and when it finishes, the *passZOut* output pin receives data - Fig. 41d. After that, the activity returns to the initial state of waiting for data — Fig. 41f. Note that the execution of an activity is triggered when all input pins receive data. After consuming the data available on the input pins, the activity is ready to receive new data to run again. Execution of activities may result in possibly transformed data being sent through output pins.

The specification of the *ZigBeeConnectorCP* behavior was expressed by the *ZigBeeConnectorPC* protocol shown below, and in the contraint shown in Figure 42. The rule states that several times (none, once, or many times), the value received in *passZIn* will be sent through *passZOut*.

```
protocol ZigBeeConnectorPC (passZIn:CmH2ODT): passZOut:CmH2ODT{
    several send passZIn via passZOut
}
```

Listing 5.1 – Protocol ZigBeeConnectorPC

Figure 43 illustrates an excerpt from the FMS architecture containing a component *S1(SensorCP)* that has its output port *measureOut(CmH2OOPT)* sending information to the input port *passZIn* of the component *ZB1(ZigBeeConnectorCP)* through a connector *CMP1(CmH2OCN)*. The *ZB1(ZigBeeConnectorCP)* component also has its output port *passZOut* sending information to the *passGIn* input port of the *GW(GatewayCP)* component through a *CPP1(CmH2OCN)* connector.

The *ZigBeeConnectorCP* has an elementary behavior, defined in the same way as

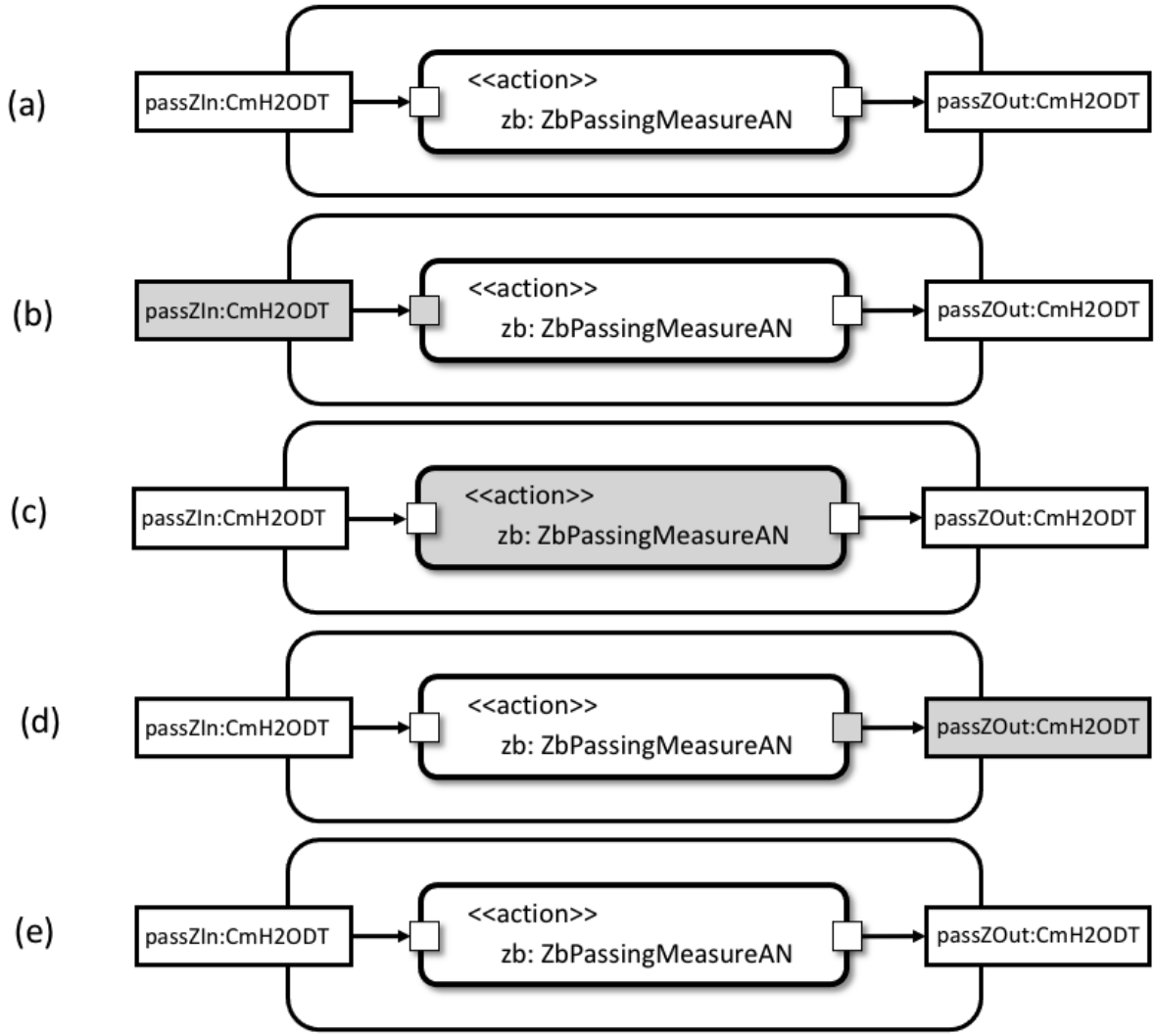


Figure 41 – *ZigBeeConnectorCP* Active Protocol

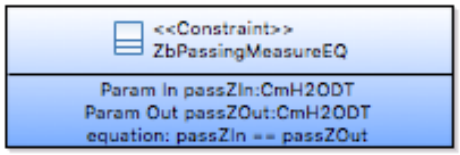


Figure 42 – *ZigBeeConnector* Constraint

the FMS architecture connectors. It receives a value and forwards the received value. As explained by Oquendo et al. (OQUENDO; LEITE; BATISTA, 2016), a SysADL action refers to a simple, atomic behavior that receives input parameters, computes them, and provides a result to the output parameter. An effort executes atomically when all in pins receive data and ends when it puts data in the out pins, without interruption.

In this way, the behavior of the *ZigBeeConnectorCP* has only three possible states, as shown in the state graph in Figure 44. In the initial state (0), both pins are without

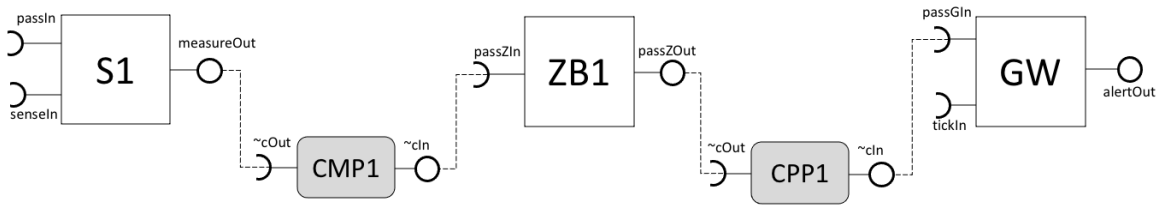


Figure 43 – Excerpt from the FMS architecture

values (`NULL`); When the *passZIn* input pin receives a value(`<S1,3.63>`), state (1) is reached; finally, the value(`<S1,3.63>`) is passed to the *passZOut* output pin, state (2) is reached, and the action ends in a valid state, showing that the architectural description *satisfies* the specified behavior.

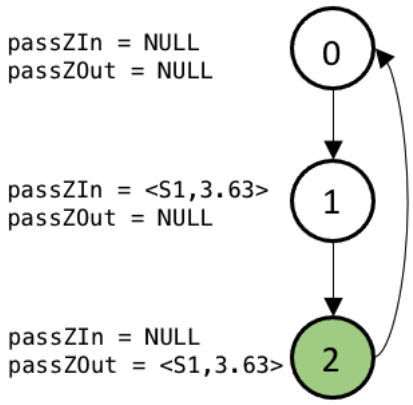


Figure 44 – ZigBeeConnectorCP behavior state graph

Figure 45 shows execution traces related to the excerpt of the architectural configuration shown in Figure 43, comparing the execution of the architecture in SysADL (left) and in π-ADL (right). Analyzing the numbered markings in the figure:

1. This snippet shows the communication between sensor `S1` and connector `CMP1`. `S1` sends the value `(S1, 3.6325)` through the `measureOut` output port/connection to the `cout` input port/connection of `CMP1`. In SysADL (left), we see a send action followed by a receive, which happens atomically, as previously explained. In π-ADL (left), there is an `rdv` (*rendez-vous*) action, i.e., the sequential execution of an output prefixing action followed by an input prefixing action in π-ADL. The π-ADL `rdv` action is equivalent to the pair of SysADL actions `send-receive`.

2. On the left, the excerpt shows the sending of the value `(S1, 3.6325)` through the output port `cIn` of the connector `CMP1`, being received by the ZigBee `ZB1` input port `passZIn` , an action equivalent to the `rdv` π-ADL action shown on the right

3. In the left excerpt, the Zigbee `ZB1` sends the value `(S1, 3.6325)` through port `passZOut` to port `cOut` of connector `CPP1`. Analogously, on the right, a connection

is formed between connections `ZB1.passZOut` and `CPP1.cOut` to send the value `(S1, 3.6325)`.

4. Finally, the snippets in SysADL(right) and π-ADL (left) perform the last sending/receiving of the data `(S1, 3.6325)` between the output port/connection `CPP1.cIn` and the port gateway port/connection `GW.passZIn`.

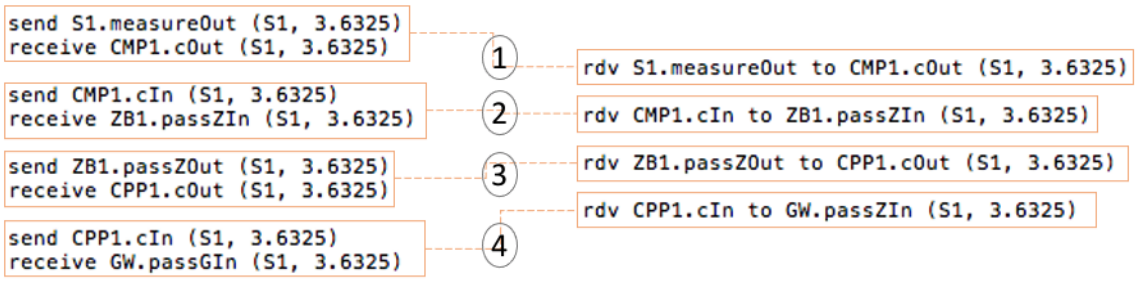


Figure 45 – Execution traces SysADL (left) and π-ADL (right) - ZigBeeConnectorCP Behavior

Analyzing the execution snippets simulated in SysADL and π-ADL in Figure 45, we observe that the traces are equivalent. To make this observation more visible, Figure 46 shows the state graph with a snippet of the states reachable by this snippet of execution. The states are named with the ports/connections values, and the transitions are named with the actions performed to reach the target state. Moreover, we use the same states to show that the actions in SysADL and π-ADL lead to the same states, being the architectures, therefore, equivalent.

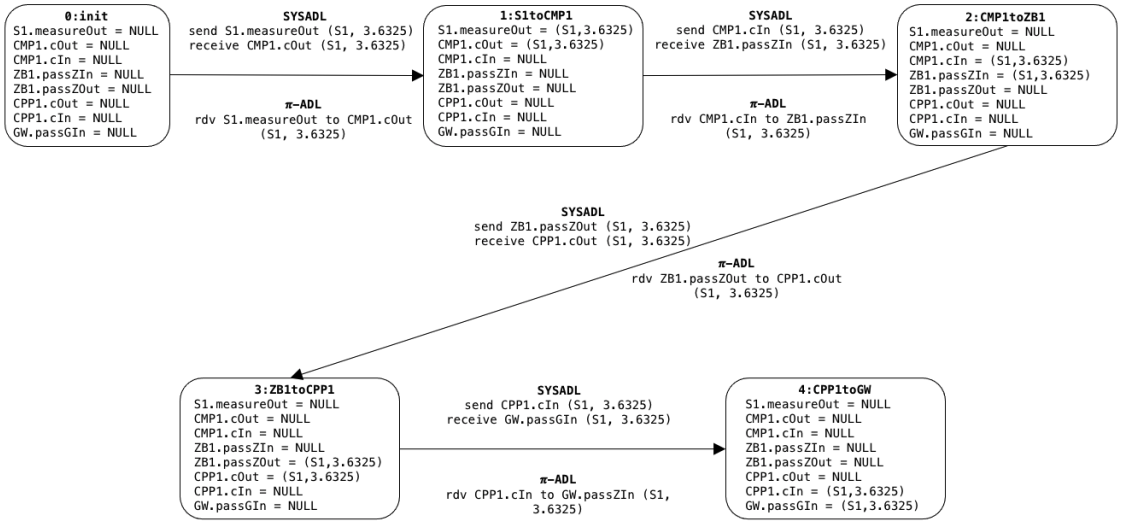


Figure 46 – FMS Partial State Graph

SensorCP Behavior. As detailed in Appendix A, the *SensorCP* component was described in the structural view of SysADL with two input port *senseIn(MV)*, and *passIn(CmH2ODT)* and an output port *measureOut(CmH2ODT)*, as shown in Figure 47.

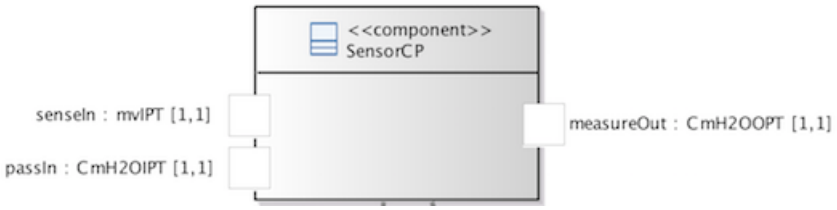


Figure 47 – *SensorCP* structural definition

In the transformation to π-ADL, the *SensorCP* component defines the *senseIn(MV)*, *passIn(CmH2ODT)* and *measureOut(CmH2ODT)* connections, respectively, analogous to its input and output ports described in SysADL.

The defined behavior for *SensorCP* has two actions:

- (i)*PassingMeasureAN*, modeling the behavior where the *SensorCP* receives a measurement from the local physical sensor in *MV* and forwards the measurement in *CmH2ODT* to the next *ZigBeeConnectorCP* in the network via measureOut, and
- (ii) *PassingMeasureAN*, that receives a volume measurement from a previous *ZigBeeConnectionCP* via *passIn* port and forwards the volume measurement value to the next *ZigBeeConnectionCP* via *measureOut*.

Figure 48 illustrates the sequence of steps involved in the behavior of the *SensorCPAC* activity. Figure 48a illustrates activity without data on its pins. Figure 48b shows that the activity received data on the *senseIn* pin. After receiving the data from the input pin, the action starts *LocalMeasuringAN* - Fig. 48c -, and when it finishes, the output pin *measureOut* receives the data - Fig. 48d, sending it to the connector. After that, the activity returns to a state waiting for data — Fig. 48e. Figure 48f shows when the data arrives at the input pin *passIn*. After receiving the data from the input pin, the action starts *PassingMeasureAN* - Fig. 48g -, and when it finishes, the output pin *measureOut* receives the data - Fig. 48h, sending it to the connector. As discussed earlier, the execution of an activity is triggered when all input pins receive data. After consuming the data available on the input pins, the activity is ready to receive new data to run again. Executing activities can result in possibly transformed data being sent through output pins.

The desired specification for *SensorCP* behavior was expressed by the SensorPC protocol shown below. The rule says that several times (none, once or many times), will be sent via *measureOut*, an alternation between the value measured by the local sensor, received via *senseIn*, and the value measured in another network point, forwarded by the previous ZigBee and received via *passIn*.

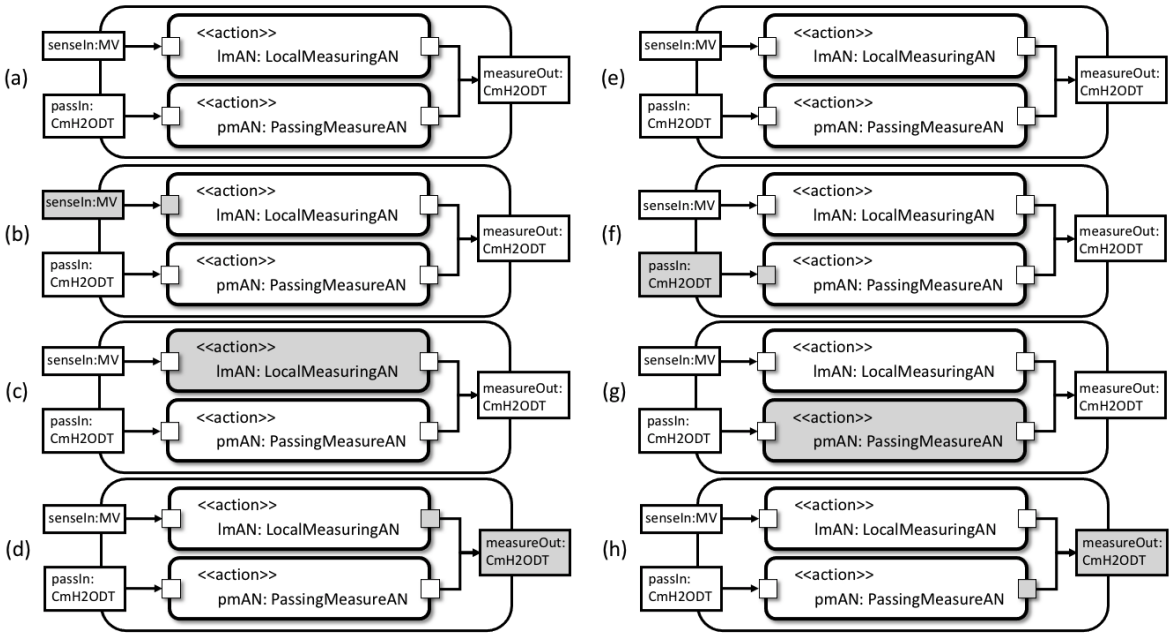


Figure 48 – *SensorCP* Active

```
protocol SensorPC (passIn:CmH2ODT, senseIn:MV): measureOut:CmH2ODT{
  several(send <self,senseIn> via measureOut | send passIn via measureOut)
}
```

Listing 5.2 – Protocol ZigBeeConnectorPC

The excerpt from the FMS architecture shown in Figure 43 shows how information from a *SensorCP* propagates along the sensor network until it reaches the *GatewayCP*. The section shown has only a single *SensorCP (S1)* and a single *ZigBeeConnectorCP (ZB1)*, so there would never be a transfer of value via *passIn*, coming from a previous sensor in the network. However, Figure 49 shows, in a simplified way, an excerpt of the architecture configuration of the FMS with three *SensorCP* and three *ZigBeeConnectorCP*; the connection between the previous sensor in the network and the ZigBee happens via connector *CMP*, and the connection between the output of ZigBee and the next sensor on the network happens via the *CPP* connector. The last ZigBee in the network (*ZB3*) is connected to the *GatewayCP(GW)*. In this way, it is possible to understand how extensive networks would be, with more sensors, ZigBee, and consequently more connectors.

When comparing the image of Figure 48, which activity diagram shows the information arriving at the *senseIn* or *passIn* pins and activating, respectively, *LocalMeasuringAN* or *PassingMeasureAN*, with the protocol shown in Listing 5.2, we can see that there are no guarantee that there is an alternate execution between the actions *LocalMeasuringAN* or *PassingMeasureAN*. From the way the behavior of *SensorCP* was modeled, the only information that arrives to start an action or another are the input pin *senseIn* and *passIn*. There is no "memory" of which action was triggered previously for the

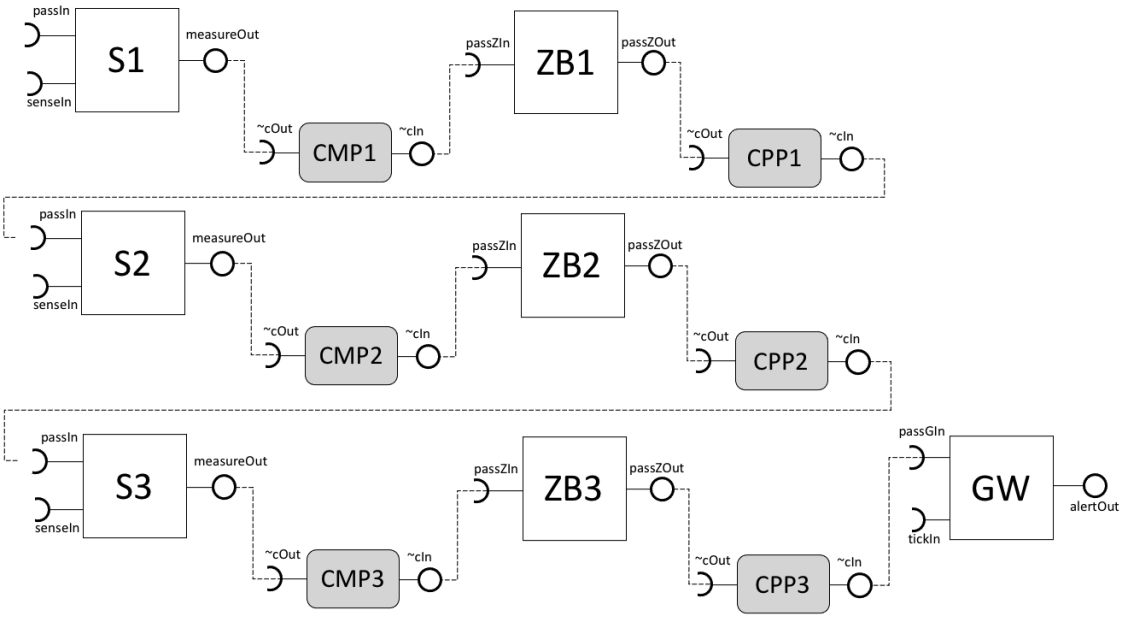


Figure 49 – Excerpt from the FMS architecture with 3 SensorCP

toggle in this execution. Figure 50 illustrates a state graph showing the transitions that should happen to guarantee this alternation and that, in this case, a *flag* variable would be necessary to signal that the action can be activated in order to attend the defined protocol.

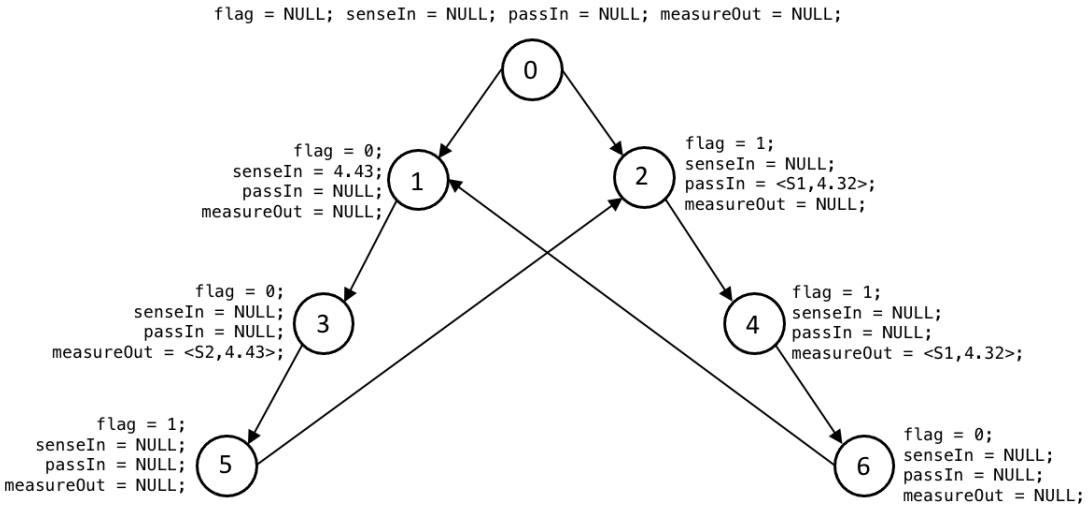


Figure 50 – Expected state-chart for SensorCP behavior

In Figure 50, the initial state (0) shows that no pin has a value, just like the `flag` variable. The odd states are reached when the measurement is propagated by the local sensor (*LocalMeasuringAN*), while the even states are reached when the measurement of a previous sensor in the network is forwarded (*PassingMeasureAN*). Traversing the odd branch of the graph: state `1` is reached when a measure arrives in `senseIn` and `flag = 0`; after that, state `3` is reached to mark that `measureOut` receives the value to be passed

on, and the `flag` remains at `0`; and finally, state `5` marks the end of execution of the *LocalMeasuringAN* action and the value of `flag = 1` means that now only the branch with even states can be activated. Analogously, traversing the even branch of the graph: state `2` is reached when a measure arrives at `passIn` and `flag = 1`; after that, state `4` is reached to mark that `measureOut` receives the value to be passed on, and the `flag` remains at `1`; and finally, state `6` marks the end of the execution of the *PassingMeasureAN* action, and the value of `flag = 0` means that only the branch with odd states can be activated.

Figure 51 shows execution traces related to the excerpt of the architectural configuration shown in Figure 49, comparing the execution of the architecture in SysADL (left) and in π-ADL (right). Analyzing the numbered markings in the figure:

1. This excerpt shows the communication performed by the `CVS3` connector, connecting *SensorEnvConnectorCP (SE3)* to *SensorCP(S3)*. `CVS3` sends the value `(4.4313)` through the output port/connection `mIn` to the input port/connection `senseIn` of `S3`. SysADL (left) shows a send action followed by a receive action, which happens atomically, as explained earlier. In π-ADL (left), there is a `rdv` (*rendez-vous*) action, that is, the sequential execution of an output prefixing action followed by an input prefixing action in π-ADL. The π-ADL `rdv` action is equivalent to the SysADL `send-receive` action pair.

2. On the left, the snippet shows the sending of the value `(S3, 4313)` through the output port `measureOut` of the sensor `S3`, being received by the Connector `CMP3` input port `cOut`, an action equivalent to the `rdv` π-ADL action shown on the right.

3. In the snippet on the left, the connector `CPP2` sends the value `(S1, 4.1547)` through port `cIn` to the port `passIn` of sensor `S3`. Analogously, on the right, a connection is formed between the connections `CPP2.cIn` and `S3.passIn` to send the value `(S1, 4.1547)`.

4. On the left, the snippet shows the sending of the value `(S3, 4.1547)` through the output port `measureOut` of the sensor `S3`, being received by the Connector `CMP3` input port `cOut`, an action equivalent to the `rdv` π-ADL action shown on the right.

5. In the snippet on the left, the connector `CPP2` sends the value `(S2, 2.5233)` through port `cIn` to the port `passIn` of sensor `S3`. Analogously, on the right, a connection is formed between the connections `CPP2.cIn` and `S3.passIn` to send the value `(S2, 2.5233)`.

6. Finally, on the left, the snippet shows the sending of the value `(S2, 2.5233)` through the output port `measureOut` of the sensor `S3`, being received by the Connector `CMP3` input port `cOut`, an action equivalent to the `rdv` π-ADL action shown on the right.

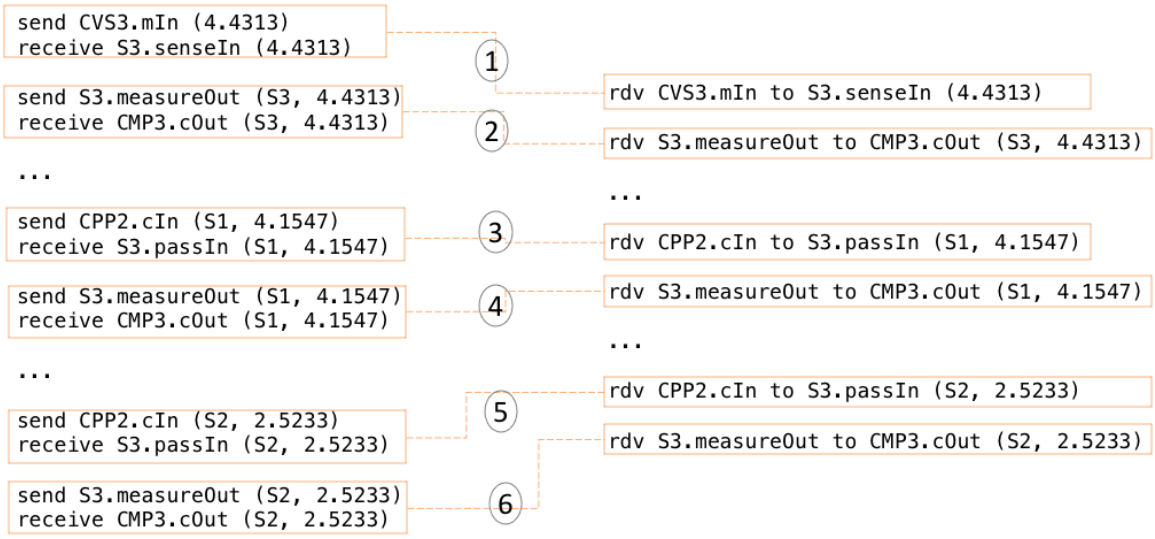


Figure 51 – Execution traces SysADL (left) and π-ADL (right) - SensorCP Behavior

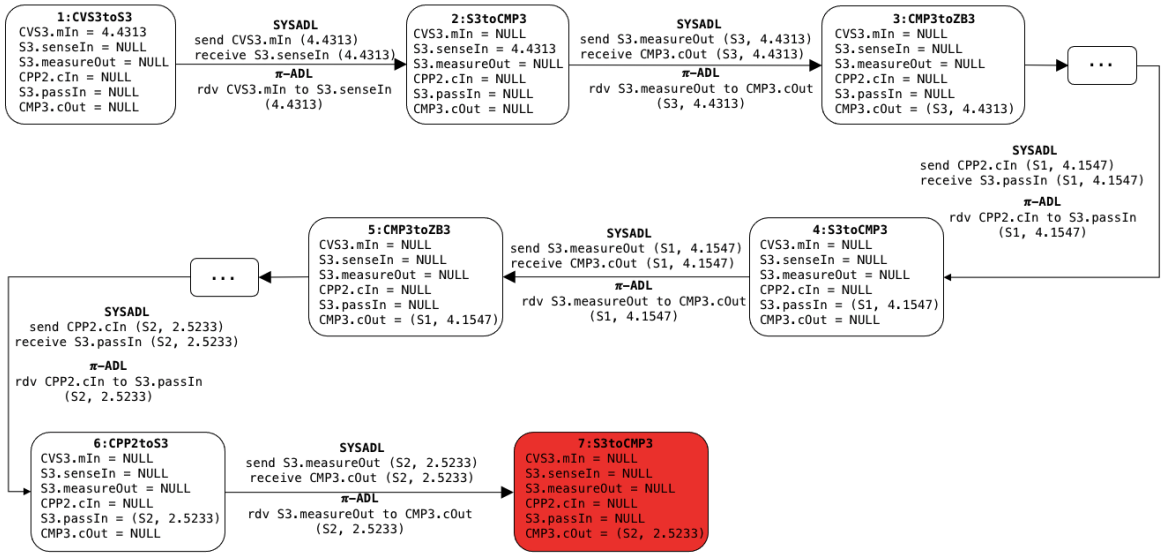


Figure 52 – FMS Partial State Graph - *SensorCP* Behavior

Analyzing the simulated execution snippets in SysADL and π-ADL in Figure 51, we observe that the traces are equivalent, showing that there is equivalence between the SysADL and π-ADL architectures. However, in this case, it is possible to observe that the modeled FMS architecture does not satisfy what is specified in the protocol in listing 5.2 through Figure 52, which shows the state graph with an excerpt of the states reachable by this execution snippet. The last state, the only one in color, shows that at this point, the execution reaches a state where the condition specified by the protocol is not accepted; that is, there is no alternation between the actions *LocalMeasuringAN* and *PassingMeasureAN*. In the graph, states are named with port/connection values, and transitions are named with actions taken to reach the destination state. Furthermore, we

use the same states to show that actions in SysADL and π-ADL lead to the same states, thus the architectures are equivalent.

This section presented a sample of how the verification of the architectural properties that specify the expected behavior of the *ZigBeeConnectorCP* and *SensorCP* components can be performed. The behaviors were expressed in the *ZigBeeConnectorPC* and *SensorPC* protocols using the SysADL syntax. The properties were verified through the analysis of execution traces and partial graphs of states. The property specified by the *ZigBeeConnectorPC* protocol was satisfied by the architecture modeled for the FMS, unlike the *SensorPC* protocol, which was not satisfied by the architecture. In this case, an execution trace was presented showing a processing sequence that leads to a violation of the protocol.

5.4 Tools for Model Checking

There are a variety of support tools for Model Checking available in the literature, and four fundamental aspects to be considered when choosing which tool to use: (i) the languages that are accepted for specifying the model to be checked; (ii) the languages used to specify the properties; (iii) model checking technique that the tool uses, such as explicit states, symbolic, partial order reduction, statistical model checking, among other possibilities, which concern technical aspects related to memory usage and performance; finally, (iv) if the tool uses some technique to minimize a classic problem for the model checking technique which is state-explosion. This section presents five model-checking support tools: (i) FDR (BROWNLEE, 2000; ROBINSON et al., 2013), (ii) NuSMV (CIMATTI et al., 1999), (iii) PAT (LIU; SUN; DONG, 2011), (iv) PLASMA (BOYER et al., 2013; JÉGOUREL; LEGAY; SEDWARDS, 2012), and (v) SPIN (HOLZMANN, 1997), showing their main features. Table 6 presents an overview of these tools at the end of the section.

FDR (Failures-Divergences Refinement). FDR (Failures-Divergence Refinement) is an explicit state model checker for state machines with foundations in the theory of concurrency based on CSP (ROSCOE, 2010). It gradually builds the state-transition graph, compressing it using state-space reduction techniques while checking properties, which also makes it an implicit state model checker (FRAPPIER et al., 2010). FDR is used to verify the correctness of concurrent systems and check whether a given CSP model satisfies a set of specified safety and liveness properties. According to the user manual FDR (ROBINSON et al., 2013), the original version of FDR was written in 1991 by Formal Systems (Europe) Ltd, and a completely revised version of FDR2 was released in the mid-1990s by the same organization. The current version of the tool is FDR4, first released in October 2016, after FDR3, which was first released in 2013. The University of Oxford released both versions. As is typical with model checking tools, FDR can automatically generate counterexamples when a property is violated, helping users identify the

problem's cause. Template checking is limited to systems specified in the CSP, which may not be suitable for all types of systems. FDR's focus on security and liveness properties means there may be better tools for checking other properties such as performance or reliability.

SPIN. The SPIN model checking tool is a widely used software checking tool developed at Bell Labs in the 1980s. It introduced the classic approach to real-time LTL model checking. Model specifications are written in a high-level specification language called Promela (PROcess MEta LAnguage), which describes a system's behavior, while properties are specified in LTL (FRAPPIER et al., 2010). The tool checks the logical consistency of a specification and reports on deadlocks, race conditions, different types of incompleteness, and unwarranted assumptions about the relative speeds of processes. According to Frappier (FRAPPIER et al., 2010), SPIN uses the on-the-fly model checking technique that avoids the explicit construction of the state space graph by exploring only those parts necessary to verify the desired property. Instead of constructing the entire graph upfront, the on-the-fly approach builds the graph incrementally, only exploring transitions relevant to the property being checked. However, this implies that the transitions are (re)calculated for each property to be checked. Therefore, if there are n properties to check, a transition is potentially computed n times, depending on the optimizations.

NuSMV. The NuSMV is a model checker based on the SMV (Symbolic Model Verifier) tool, developed at the University of California, Berkeley. This class of model checkers verifies temporal logic properties with implicit techniques (FRAPPIER et al., 2010), capable of verifying both finite-state and infinite-state systems, and it can handle systems with discrete and continuous variables. NuSMV uses symbolic language to represent the specification based on CTL and LTL logic. The model checker allows writing property specifications in CTL or LTL and choosing between BDD(Binary Decision Diagram)-based symbolic model checking and bounded model checking. The BDD is a data structure used in model checking to efficiently represent and manipulate boolean formulas. BDDs represent a system's state space because they can compactly represent boolean functions, making them well-suited for model-checking large systems. Bounded Model Checking is based on reducing the bounded model checking problem to a propositional satisfiability problem, using a propositional SAT solver to find an assignment that satisfies the problem.

PLASMA. PLASMA is a compact, flexible platform to enable a formal analysis of multiple modeling semantics on a single platform and allows others to integrate model-checking technology into their own software (JÉGOUREL; LEGAY; SEDWARDS, 2012), offering support for statistical model checking of stochastic models. PLASMA-lab accepts properties described as bounded linear temporal logic (BLTL) extended with custom temporal operators based on concepts such as minimum, maximum, and mean of a variable over time (BOYER et al., 2013). The PLASMA integrated development

Tool	**Model Language**	**Property Language**	**MC Technique**	**Handling with the state-explosion problem**
FDR	CSP	CSP	Explicit state	Gradual construction of the state transition graph, and use state space reduction techniques.
NuSMV	SMV description language	LTL, CTL	Symbolic MC, Bounded MC	BDD-based symbolic and Bounded model checking.
PAT	LTS, TTS, MDP	CTL, LTL, CTL*	Partial Order Reduction	POR, symmetry reduction, Process counter abstraction, and parallel MC.
PLASMA	π-ADL, RML, Adaptive RML, Bio, MATLAB, Simulink, SystemC, and LLVM	B-LTL, A-BLTL, GCSL, Observer, Nested BLTL, and DynBLTL	SMC	Statistical Model Checking
SPIN	PROMELA	LTL	Explicit state	On-the-fly technique

Table 6 – Model checking tools overview

environment facilitates distributed simulation and works with multiple user-defined language plug-ins, and serves as the basis to develop the toolchain for specifying and verifying properties of dynamic software architectures described in π-ADL (CAVALCANTE, 2016). According to Plasma Lab (PLASMA, 2022), in addition to π-ADL, it supports modeling in RML, adaptive RML, Bio (Biological Systems), MATLAB, Simulink, SystemC, and LLVM (Low-Level Virtual Machine). Plasma also supports several constraints and property specification languages (PLASMA, 2022) such as B-LTL (Bounded Linear-time Temporal Logics), A-BLTL (an extension of B-LTL for adaptive systems), GCSL (Goal Contracts Specification Language), Observer, Nested BLTL, and DynBLTL (CAVALCANTE, 2016).

PAT(Process Analysis Toolkit). PAT implements various model-checking techniques catering to properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking, and probabilistic model checking. To achieve good performance, advanced optimization techniques are implemented in PAT, e.g., partial order reduction, symmetry reduction, process counter abstraction, and parallel model checking. System models can be specified in many different ways, as this is the basis of the PAT tool approach, to compile different modeling languages into a set of common semantic models, such as LTS (Labeled Transition System), TTS (Timed Transition System), and MDP (Markov Decision Process) (LIU; SUN; DONG, 2011). System properties can be specified in different forms. A property can be specified in a dedicated language like temporal logic or the same language used to specify the system model frequently associated with CTL, LTL, and CTL* logic.

According to a preliminary analysis, the PLASMA tool would be the first candidate for automated support to verify properties for phase 4 of the process proposed in our approach. This choice is natural, as the PLASMA tool has a version that accepts the π-ADL language as a model specification language. To use the other tools, a greater effort would be needed to make them compatible with other model specification languages. We invested some efforts in trying to perform statistical model checking with the PLASMA tool on π-ADL architectures transformed from SysADL models with our approach, however, we were not successful in these attempts. The tool has no official operational support to fix problems and limitations of the current version. New attempts toward compatibility with PLASMA are part of future work.

5.5 Conclusion of the Chapter

This chapter discussed details about the formal verification of architectural properties through the model checking technique and what kind of properties can be expressed on a software architecture, also making a summary of the main automated tools to support the technique. This chapter also shows, through a manual and analytical approach, the verification of properties expressed on the architecture of FMS. The next chapter discusses some related works that also propose approaches for formalizing non-formal ADL.

6 Related Work

This chapter presents a summary of a systematic review (ARAUJO et al., 2019), which aims: (i) to provide an overview of formal verification strategies in software architecture descriptions; (ii)to understand features of languages used to describe software architectures and properties of interest to be formally verified; and, (iii) to understand how software architecture formal verification has been conducted in this context. In addition to the studies found by the systematic mapping, this chapter also discusses some related and complementary works found until December 2022.

Section 6.1 summarizes a systematic review performed at the beginning of our doctoral project. Section 6.2 presents works related to approaches involving the formalization of non-formal ADL. Section 6.2 discusses works related to approaches involving the use of other formalization strategies that do not only involve MDD-based transformations. Section 6.4 shows related work that uses another target formalism for SysADL formalization. Finally, Section 6.5 summarizes the approaches discussed in this chapter.

6.1 Systematic Review

The systematic review (ARAUJO et al., 2019) was carried out in December 2018, searching for primary studies with approaches to the formal verification of software architecture descriptions, intending to answer the following research questions (RQs):

RQ1: What are the languages used for architecture description towards supporting verification, their characteristics, and supported views?

RQ2: What are the languages used to specify architectural properties, their characteristics, and supported views?

RQ3: Which type of formal verification is addressed by the existing approaches?

To retrieve primary studies was used an automated searching process performed over four popular electronic publication databases: ACM Digital Library [1], IEEEXplore [2], ScienceDirect [3], and Scopus [4]. For the automated search, a string was defined to encompass terms such as *architecture description language*, *formal language*, and *software architecture*. At the end of the search process, the study proceeds with 39 primary studies.

[1] http://dl.acm.org
[2] http://ieeexplore.ieee.org
[3] http://www.sciencedirect.com
[4] http://www.scopus.com

The analyses of the selected studies resulted in several findings about the current state of the art on the formal verification of architecture descriptions. Some systematic review findings are summarized below:

- There is no reference language used to formally describe software architectures and specify architectural properties, thus resulting in a plethora of languages and notations;
- Visual and semi-formal notations have appeared as means of supporting software architecture description despite requiring an additional process to translate the produced models towards formal verification;
- The support for multiple viewpoints (in particular, the structural and behavioral viewpoints) has been considered as a relevant concern and hence addressed by both architecture description and property specifications;
- Properties related to reliability, concurrency, and real-time characteristics have been addressed in the studies as a reflex of the relevance of these concerns in many critical systems;
- Model checking has been the most used technique to support the formal verification of architectural properties despite its well-known limitations concerning scalability;

The systematic review was essential to provide a research landscape about the possible ways to deal with formal verification and property verification in software architecture description, to identify literature lacks, and to guide this doctoral research.

Among the studies selected in the systematic review and other recent research, we selected some studies related to the choices we made for our approach. These works present some relevant topics for this study. They are divided into three categories, all of them encompassing properties verification: (i) approaches involving the formalization of UML-based ADLs and properties verification (Sec. 6.2); (ii) approaches involving formalization of other architectural strategies and properties verification (Sec. 6.3); and, (iii) approaches involving formalization of SysADL and properties verification (Sec. 6.4). Section 6.5 summarizes related work, tabulating some important features of these approaches.

6.2 Formalization of UML-based ADLs and properties verification

This section presents works that aim to enrich a non-formal ADL with formalism to perform some formal analysis from architectural descriptions. Section 6.2.1 shows the work of Kang et al. (KANG et al., 2013), which presents a methodology to analyze

properties from architectural descriptions in EAST-ADL. Section 6.2.2 introduces the work of Taoufik et al. (TAOUFIK; TAHAR; MOURAD, 2016), proposing a formalization of UML2.0/PoSM through Wright/CSP. Section 6.2.3 presents the approach of Ozkaya and Kose (OZKAYA; KOSE, 2018b), suggesting the formalization of SAwUML through ProMeLa, for verifying properties in the SPIN tool. Finally, Section 6.2.4 compares and discusses the approaches described in the previous sections.

6.2.1 Enabling Formal Analysis in EAST-ADL

Kang et al. (KANG et al., 2013) propose a methodology for formal analysis and verification of architectural models described in a domain-specific non-formal ADL to automotive embedded systems.

The strategy suggests performing an MDD-based with the M2M transformation from EAST-ADL to Intermediate Component Model (ICM), followed by another M2M transformation from ICM to UPPAL PORT, as shown in Figure 53. The approach provides automated analysis support through the ViTAL tool. Each function model's functional and timing behavior is specified as Timed Automata models (TA), which have precise semantics and can be formally verified with ViTAL to prove that the EAST-ADL system model fulfills the specified real-time requirements and behavioral constraints. To allow the specification of timing properties, the approach and tool support timing properties in TCTL specification.

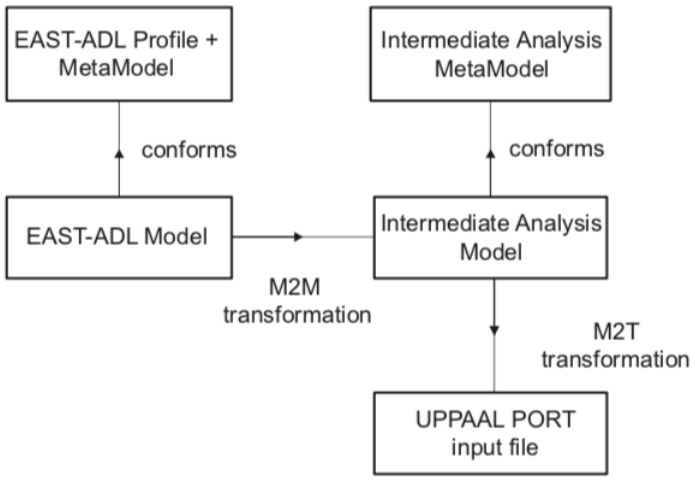


Figure 53 – Model export from EAST-ADL to UPPAAL PORT (KANG et al., 2013)

The work validates the proposal using a Brake-By-Wire (BBW) system modeled in EAST-ADL concerning particular functional and timing requirements. As a sample of timing verification, the work verifies a *DelayConstraint* property that ensures the brake reaction delay specified in the BBW model. Regarding functional verification, the sample given is a functional requirement of the system related to the *slip rate*, which verifies that

if the slip rate variable exceeds a specified value, the brake actuator is released no brake is applied.

Kang et al. (KANG et al., 2013) present formal modeling and verification techniques at an early stage of developing safety-critical automotive products initially described in the EAST-ADL architectural language. The work affirms the using TA formalism to facilitate the capture of the flow of execution within each functional block and the complex interactions between components and improve the modeling, verification, and behavior analysis capacity of EAST-ADL.

6.2.2 Behavioral verification of UML2.0/PoSM

Taoufik et al. (TAOUFIK; TAHAR; MOURAD, 2016) propose a methodology to verify the behavioral consistency of UML2.0/PoSM software architectures involving the UML to Wright formal ADL translation and a translation of Wright to CSP. Using the Wr2fdr tool, these Wright descriptions are automatically translated to a CSP specification acceptable by the FDR2 model-checker, a process that can be seen in the Figura 54

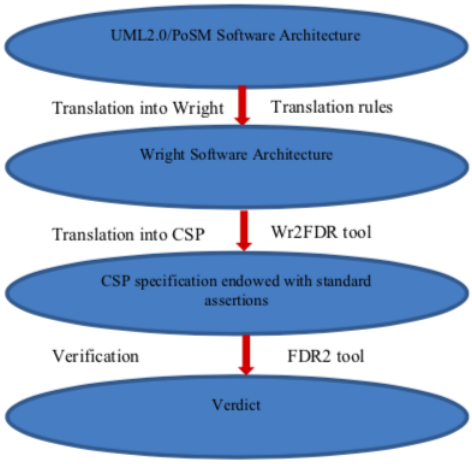


Figure 54 – Verification approach (TAOUFIK; TAHAR; MOURAD, 2016)

The strategy proposes performing the model transformation approach to translate the UML2.0 architectural elements into the Wright specification as an intermediate modeling language. Using the Wr2fdr tool, it becomes possible to translate a Wright specification into another in CSP, allowable by the FDR2 model-checker. The Wr2fdr tool even generates a set of contracts to connect behavioral compatibility and compatibility between the interfaces of a component and the component itself.

The work validates the proposal using an ATM/Bank system in UML2.0, with ports and components behavior specified by state machines using the PoSM profile, aiming to ensure the detection of the non-consistency.

Taoufik et al. (TAOUFIK; TAHAR; MOURAD, 2016) mentioned that the main work contribution is to detecting the non-consistency of UML2.0/PoSM software architec-

ture behavioral properties, covering the component and connector behavioral consistency properties and port/role compatibility. The verification of these properties is entrusted to FDR2.

6.2.3 SAwUML formal analysis using SPIN

Ozkaya and Kose (OZKAYA; KOSE, 2018b) suggest an approach to formal analysis in architectural descriptions specified in SAwUML, a UML-based architectural description language proposed to improve UML for the high-level, precise specifications of the structural and behavioral design decisions and their formal analysis.

The strategy combines UML's component and sequence diagrams under the same notation set. It extends the formal Design-by-Contract approach for the behavior specifications of the port methods drawn in the sequence diagram. To this purpose, SAwUML is supported with a modeling tool for specifying the integrated structural and behavioral design decisions and any system-level properties in the linear temporal logic. The tool can automatically translate the software architectures in SAwUML into a formal ProMeLa model. In the sequence, the SPIN model checker can verify the resulting ProMeLa models for the predefined properties. The architecture of this toolset is illustrated in Figure 55.

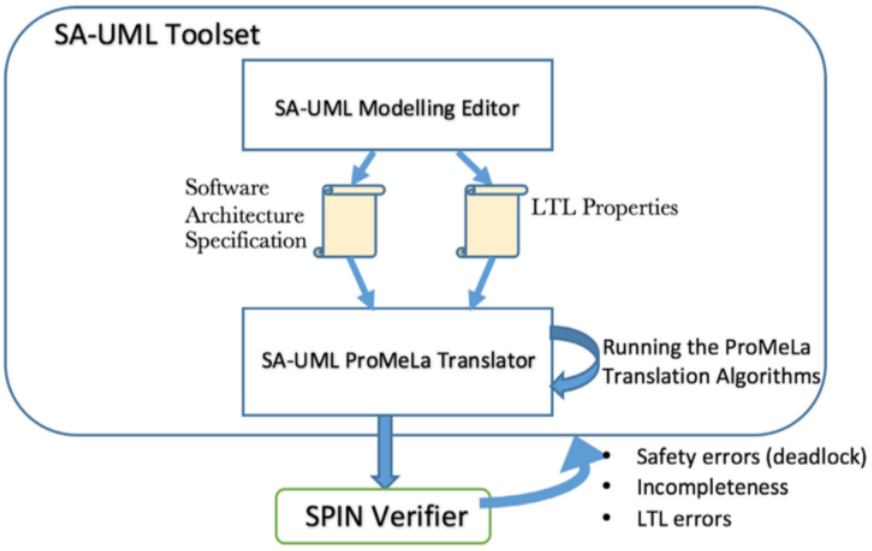


Figure 55 – SAwUML's tool architecture. (OZKAYA; KOSE, 2018b)

The work validates the proposal using the gas station system. The gas station architecture is visually specified using SAwUML's notation set, and then the visual specification is translated into a formal ProMeLa model for formal verification. Thus, deadlock and incompleteness properties are verified in the SPIN tool.

This work's principal contribution was proposing a new software architecture modeling language UML-Based, SAwUML, allowing that software systems to have been its behavior components specified in terms of high-level components with the Design-by-

Contract approach. In addition, it offers an integrated tool supporting the specifying of software architectures and any system-level properties in the form of LTL, automated translation to the formal ProMeLa language, which enables the formal verification of software architectures using the SPIN model checker.

6.2.4 Discussions

The proposals presented in this Section 6.2 and our approach have the common objective of enriching non-formal ADLs to make it possible to carry out architectural analysis, such as verifying properties from the architectural description. However, there are some conceptual differences between the proposed approaches. Below, we review some key aspects to consider about these approaches.

Architectural Description Languages. To perform a formal analysis of architectural description, the decisions about the languages used to specify the structural and behavioral aspects of the architecture, and the properties to be verified, are crucial and guide the entire strategy to be used. The three works presented use UML-based ADL: (i) EAST-ADL (KANG et al., 2013), (ii) UML2.0/PoSM (TAOUFIK; TAHAR; MOURAD, 2016) and (iii) SAwUML (OZKAYA; KOSE, 2018b). In this way, the approaches use UML diagrams to describe the structure of the architectures. Our proposal performs the architectural description in the SysML-based ADL, SysADL, and therefore uses inherited SysML diagrams to specify the architecture. Another important point about the ADLs chosen is that EAST-ADL (KANG et al., 2013) is a specific-purpose language for describing the architecture of automotive embedded systems. In contrast, the others are general-purpose languages, including SysADL.

All approaches use an auxiliary mechanism to the ADLs and are different in representing the behavioral aspects of the architecture. Kang et al. (KANG et al., 2013) utilize Timed Automata (TA), and Taoufik et al. (TAOUFIK; TAHAR; MOURAD, 2016) use Port State Machines (PoSM). Both approaches resemble state machines, which model states and the transition between them. Ozkaya and Kose (OZKAYA; KOSE, 2018b) employ a UML sequence diagram and a Design-by-Contract strategy. Our approach uses SysML activity, parametric diagrams, and Object Constraint Language (OCL), despite the limitations imposed by the fact that OCL does not have temporal operators.

Formalization Approach Different strategies can implement the semantic enrichment of non-formal languages. In the related works mentioned in the section 6.2, the works by Taoufik et al. (TAOUFIK; TAHAR; MOURAD, 2016), and Ozkaya and Kose (OZKAYA; KOSE, 2018b) performed a translation of the architectural specifications originally described in UML2. 0/PoSM and SAwUML, respectively, for their target formalisms, UPPAL-PORT and Wright ADL/CSP, respectively.

While Kang et al. (KANG et al., 2013)'s approach, like ours, chose to perform MDD-based transformations, a strategy that eliminates the problems of non-compliance

with the target formalism. Kang et al. have as their target the transformation to a language compatible with the UPPAL-PORT tool. Our approach targets π-ADL formal ADL that helps keep the same abstractions to describe architectural elements, smoothing the learning curve for software architects. Furthermore, our work defines the denotational semantics of SysADL in terms of π-ADL, validating that the original and target architectures are equivalent.

Another point to be mentioned is that none of the strategies, except ours, used the automatic derivation of an executable version of the architecture to help validate the original architecture's requirements.

Properties Verification. Another important decision is how properties will be expressed. This choice is closely related to the choice of the tool that perform the architectural analysis. It can even use a strategy to choose multiple property specification languages. Kang et al. (KANG et al., 2013) approach expresses the system design requirements through Timed Computation Tree Logic (TCTL) statements and can be verified by UPPAAL PORT, which executes model checking. The study by Taoufik et al. (TAOUFIK; TAHAR; MOURAD, 2016) uses Communicating Sequential Processes (CSP) to express architectural properties, with model checking performed by the FDR2 tool. Ozkaya and Kose (OZKAYA; KOSE, 2018b) define the use of Linear Temporal Logic (LTL) to express properties for verification using the Spin model checking tool.

The approaches by Kang et al. and Ozkaya and Kose use temporal logic to specify properties. This strategy can limit the verification of properties in systems that do not involve timing. The approach by Taoufik et al. uses CSP, which has known limitations regarding properties for dynamic architectures. Therefore, defining a logic also limits the possibilities of expressing and verifying properties. Our study uses SysADL's abstractions to specify the properties, protocol definition, and OCL restrictions. The development community widely accepts the OCL as it has been used with UML for a long time. We consider the tool support for formal verification of properties, the π-ADL choice as the underlying formalism brings the possibility of using property specification in other formalisms already accepted to the piadl toolset (PLASMA, 2022), such as B-LTL (its extensions A-BLTL, DynBLTL), GCSL, and RML language. These verification possibilities using a tool and exploring different types of logic are one of the actions for the future continuation of this work.

6.3 Formalization of other architectural strategies and properties verification

This section focuses on works that use other strategies, such as refinement and semantic rewriting, in addition to MDD-based transformations to formalize formal software architectures to perform properties verification based on architecture descriptions.

Section 6.3.1 shows the work of Lima et al. (LIMA et al., 2017), which means to analyze and refine design models specified using SysML through the definition of a SysML semantic based on COMPASS Modeling Language (CML). Section 6.3.2 shows the work of Rademaker et al. (RADEMAKER; BRAGA; SZTAJNBERG, 2005), which presents a methodology to analyze properties from architectural descriptions in CBabel ADL. Section 6.3.3 compares and discusses these approaches.

6.3.1 Integrated semantics about SysML models using refinement

Lima et al. (LIMA et al., 2017) presents an approach that provides a means to analyze and refine models of architectural descriptions in SysML. The semantics are defined using state-rich process algebra, the COMPASS Modeling Language (CML).

The presented strategy provides an architectural description using SysML diagrams to specify the structure and behavior of the architecture, where usage guidelines are defined to allow the construction of meaningful CML models. Thus, block diagrams define the system and its components, internal block diagrams define the relationship between them, each operation is implemented as either a state machine or an activity, but not both, and sequence diagrams define the scenarios of the system. The formalization of the SysML models is a denotational semantics from the SysML metamodel constructs to abstract CML syntax constructs. According to the defined semantics, Atego's Artisan Studio plugin was implemented to automate an M2T transformation from SysML to CML. Figure 56 shows a workflow overview of the approach through the analysis process supported by the CML toolset: (1) use of Artisan Studio to build SysML models; (2) CML plug-in can be used to generate an associated CML model; (3-4) integration with the Symphony tool to allow animation of the architecture; and, (5-6) integration with FDR3 tool to perform model checking of properties.

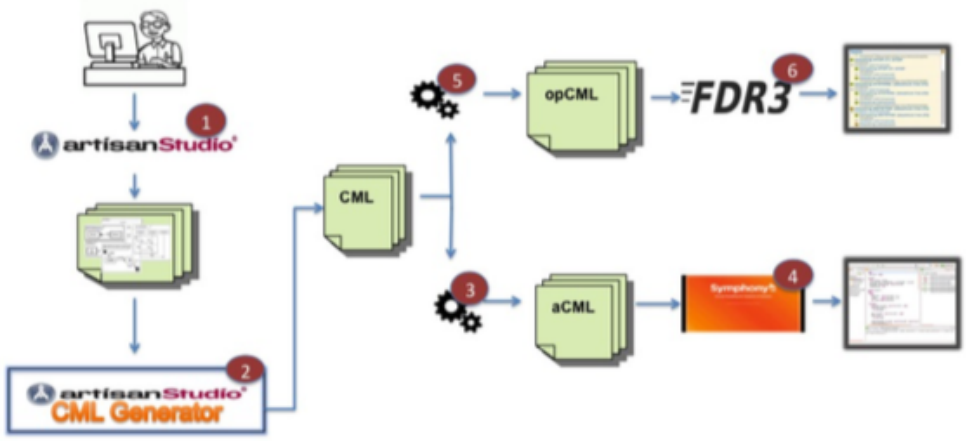


Figure 56 – SysML/CML - Overview of the modelling approach workflow. (LIMA et al., 2017)

Refinement is used in the context of the approach to assess behavior preservation, enabling reasoning at various levels of abstraction and about model transformations that can occur between abstract and concrete models. An example, if machines are built as part of the conceptual and low-level design models, it can ensure that the state machine in the low-level design respects the requirements built into the conceptual diagrams. Refinement is the main reasoning technique available for CML and explores the approach to reasoning about SysML models through their CML semantics.

The work is validated using a case study of a leadership election protocol, a critical component of an industrial application. The main requirement is that there is precisely one leader. This requirement is captured in a conceptual design specified by an abstract SysML model. The implementation uses a distributed architecture with no centralized control specified by an alternative, more concrete SysML model.

Lima et al. affirm that the significant work's contribution is a framework for reasoning using refinement about systems specified by collections of SysML diagrams: (i) guidelines of usage for construction of meaningful SysML models; (ii) a state-rich process algebraic semantics for SysML models, in particular, a CML semantics; and (iii) applications of the CML model in reasoning at the diagrammatic level.

6.3.2 A Rewriting Semantics for CBabel

Rademaker et al. (RADEMAKER; BRAGA; SZTAJNBERG, 2005) presents a rewrite logic semantic-based approach of the CBabel software architecture description language. The motivation is to add new features and improve the coordination aspects (in distributed systems) handled at the software architecture level and can be specified in high-level contracts in CBabel ADL. Rewrite logic is a logic and semantic framework to which several computational models, logics, and specification languages have been mapped to the Maude tool, a meta-tool with several analysis tools embedded in the system, such as the LTL model checker and Clavel's Inductive Theorem Prover.

The strategy foresees the mapping of concepts and architectural elements of CBabel for logic rewriting, as well as the coordination contracts that specify the interaction flow within a connector. Given the rewritten CBabel descriptions and logical semantics, the CBabel tool produces object-oriented Maude modules for each CBabel component and loads these modules into the Full Maude module database. Thus, the Full Maude tool performs model checking of properties specified in Maude syntax.

The work validates the proposal using a producer-consumer-buffer architecture analyzing three properties: (i) race condition, (ii) deadlock, and (iii), assuming the buffer is limited, overflow and underflow, that is, the producer should not add more items than the buffer may hold, and the consumer should not remove an item from an empty buffer.

The main contribution of Rademaker et al. (RADEMAKER; BRAGA; SZTAJNBERG, 2005) is to implement the logical rewrite of CBabel ADL, improving the aspects

that meet the coordination requirements, offering an automated approach with architectural analysis model checking by the Full Maude tool.

6.3.3 Discussions

The proposals presented in Section 6.3 show approaches provide formal support for ADLs to allow formal properties analysis, like our approach. However, there are some differences in the conception of these approaches, especially regarding the formalization strategies and other more specific objectives.

Architectural Description Languages. The two works presented use general-purpose ADLs specifying the structure and behavior of architectural descriptions: (i) SysML (LIMA et al., 2017), and (ii) CBabel (RADEMAKER; BRAGA; SZTAJNBERG, 2005). Rademaker et al. do not use SysML-based ADL approaches, such as Lima et al. and ours. Strategies using SysML-based ADLs have more significant potential for adoption by the software architects community since the formal approaches that contemplate SysML-based ADL and UML-based have more acceptance because they are the most used languages in the software industry (MALAVOLTA et al., 2013).

Formalization Approach. The works mentioned in Section 6.3 use different strategies to add formalism to the chosen ADLs. The work by Lima et al. (LIMA et al., 2017) performs an automatic MDD-based transformation (M2T) to transform SysML into CML, and refinement strategies to improve the transformation behavior, minimizing architectural erosion. The strategy of Rademaker et al. (RADEMAKER; BRAGA; SZTAJNBERG, 2005) involves the semantic rewrite of CBabel and its implementation in the Maude tool.

Properties Verification The study by Lima et al. (LIMA et al., 2017) uses Communicating Sequential Processes (CSP) to express architectural properties, with model checking performed by the FDR3 tool. The Rademarker et al. (RADEMAKER; BRAGA; SZTAJNBERG, 2005) approach expresses properties using Maude Syntax and performs model checking with the help of the Full Maude tool.

Specific Objectives. Unlike the other works shown in this chapter, the approaches by Lima et al. (LIMA et al., 2017) and Rademaker et al. (RADEMAKER; BRAGA; SZTAJNBERG, 2005) express specific goals in addition to enriching ADLs with formalism and enabling architectural analysis. Lima et al. (LIMA et al., 2017) aims to provide guidelines for building more meaningful SysML models, and the use of refinement was intended to preserve the behavior of the architecture in the model. Rademaker et al. (RADEMAKER; BRAGA; SZTAJNBERG, 2005) also aimed to improve coordination aspects (distributed systems) based on the semantic rewriting of CBabel.

6.4 Formalization of SysADL and properties verification

In addition to our work, we found another one that proposes the formal enrichment of SysADL using a different approach and another underlying formalism.

Dias et al. (DIAS et al., 2020) proposes an approach for empowering SysML-based ADLs with formal verification support founded on model checking. Therefore, it defines the semantics of SysADL in terms of Communicating Sequential Process (CSP) (ROSCOE, 2010). The strategy suggested performs the automated model transformation from SysADL architecture descriptions to CSP using a plug-in to the Eclipse-based SysADL Studio tool.

The process described by the approach was defined following three steps: (i) transformation of SysADL architectural description in CSP using the plug-in to the Eclipse/SysADL Studio tool; (ii) formal specification of properties in CSP; (iii) the plug-in to the Eclipse/SysADL Studio tool interact with FDR4 refinement checker to automatically verify if the model satisfies properties.

The work validates the proposal using a room temperature control system (RTC). The SysADL architectural description is transformed in CSP, CSP properties about the architecture are specified, and the integration of tools (plug-in to the Eclipse/SysADL Studio + FDR4) verifies if the model satisfies (i) deadlock-freedom, (ii) livelock-freedom, (iii) absence of miracles, and (iv) the compliance of the execution model with the behavioral model.

The proposal of Dias et al. is similar to our work regarding the interest in formally enriching a SysML-based language, specifically SysADL. Nonetheless, these works have three main conceptual differences mentioned below.

The master thesis of Dias (DIAS, 2021) extended the previous work with the possibility of formally defining the security properties to be checked in terms of SysADL Parametric Diagrams and adding a traceability feature to the plug-in developed for the SysADL Studio tool, which automates the translation of SysADL architecture models to CSP and transparently interacts with the FDR4 model checker to verify the desired properties automatically.

Target formalism. The use of CSP brings the advantage of being a formalism widely used in the formal verification community, with the support of consolidated tools for automatic verification, such as PAT and FDR4. However, CSP does not provide any built-in support for time or timing constraints, which may be a limiting factor for use in some domains. Original π-ADL does not support temporal notions. However, the study of Cavalcante (CAVALCANTE, 2016) defined the logic DynBLTL allowing specification of dynamic properties, with time notion.

Property specification language. Dias et al. (DIAS et al., 2020) also use CSP as the standard property specification language of the approach. Expressing properties

in CSP requires non-trivial knowledge of typical users, which can be a limiting factor in adopting the strategy in the software industry. Our approach does not limit the logic used to write the properties. This thesis uses SysADL's resources to specify the properties, protocol definition, and OCL restrictions. Although, the π-ADL choice as the underlying formalism brings the possibility of using property specification in other formalisms already accepted to the π-ADL toolset (PLASMA, 2022). The possibility of verifying properties expressed through the parametric diagram is an advantage of Dias' master work. It makes the process of expressing properties easier for the architect who already uses these diagrams and would not need to learn another approach.

Transformation strategy. Our decision to use an MDD-based strategy, through transformation to π-ADL, provides a scalable, automated transformation that maintains the semantics and context related to the domain of architectural software descriptions once that π-ADL is also an ADL.

6.5 Analysis of related work

After presenting and discussing the related works presented in this chapter, this section summarizes the main characteristics of each approach (including ours) through two tables.

Table 7 recapitulates the choices made for specification: (i) architecture(ADLs); (ii) scope of the ADL domain (general or specific); (iii) specification of structural aspects; (iv) specification of behavioral aspects; and, (v) language for specifying properties.

Table 8 summarizes the choices related to the approach to formalization and verification of properties: (i) strategy used for formalization; (ii) adjacent formalism target of formalization; (iii) If the approach foresees the derivation of executable architecture for validation of architectural requirements; (iv) type of proposed analysis (model checking, theorem prove); and, the tool used for automatic analysis.

Work	ADL	Domain	Structural	Behavioral	Properties
(KANG et al., 2013)	EAST-ADL	specif	UML diagrams	Timed Automata	TCTL
(TAOUFIK; TAHAR; MOURAD, 2016)	UML2.0/PoSM	general	UML diagrams	PoSM	CSP
(OZKAYA; KOSE, 2018b)	SAwUML	general	UML diagrams	UML diagrams, Design-by-Contract	LTL
(RADEMAKER; BRAGA; SZTAJNBERG, 2005)	CBabel	general	CBabel	CBabel	Maude syntax
(LIMA et al., 2017)	SysML	general	SysML diagrams	SysML diagrams	CSP
(DIAS et al., 2020)	SysADL	general	SysML diagrams	SysML diagrams, OCL	CSP, OCL
Our work	SysADL	general	SysML diagrams	SysML diagrams, OCL	*

Table 7 – Analysis of related works - Specification Languages

Work	Formalization Approach			Properties Verification	
	Strategy	**Target**	**Executable**	**Method**	**Tool**
(KANG et al., 2013)	MDD transformation	ICM, UPPAL-PORT	no	MC	UPPAL-PORT
(TAOUFIK; TAHAR; MOURAD, 2016)	translation	Wright ADL, CSP	no	MC	FDR2
(OZKAYA; KOSE, 2018b)	translation	ProMeLa	no	MC	SPIN
(RADEMAKER; BRAGA; SZTAJNBERG, 2005)	rewriting semantic	CBabel	CBabel	MC	Full Maude
(LIMA et al., 2017)	MDD transformation, refinement	CML	no	MC	FDR3
(DIAS et al., 2020)	Translation	CSP	no	MC	FDR4
Our work	MDD transformation	π-ADL	yes		

Table 8 – Analysis of related works - Formal Approach

7 Final Remarks

Software architectures have a significant role in developing software-intensive systems to achieve business objectives and quality requirements. Architecture Description Languages (ADL) (ISO/IEC/IEEE 42010, 2007) support architecture modeling by providing abstractions for systematic reasoning about system components and their connections early in development. Architectural descriptions can be used as a basis for the design or runtime and for the software architecture's documentation, maintenance, evaluation, and evolution.

Software systems and their architecture have rapidly evolved and became more complex. The architecture of these systems is subject to constant changes triggered by specific states, events, or user requests. The growing complexity of systems is proportional to users' dependence on them, requiring modern software systems to be continuously available, breaking the traditional separation between runtime and development time (WEYNS et al., 2015).

In this context, carrying out activities involving analysis of software architectures, such as checking architectural properties, has been challenging in software-intensive systems, i.e., identifying essential properties of the system using architectural models even before its implementation (TAYLOR; MEDVIDOVIC; DASHOFY, 2010). Performing such an activity as early as possible is essential to avoid incorrectness, inconsistencies, and undesirable issues that can propagate until later phases of the development process when a correction will be undoubtedly costly.

In this way, formal architecture descriptions is highly desirable to support automated architectural analysis better, being recognized as an important activity in the software industry (MALAVOLTA et al., 2013; OZKAYA, 2018b). However, software architects generally prefer to use semi-formal notations such as Unified Modeling Language (UML) (OMG, 2001), Systems Modeling Language(SysML) (OMG, 2001) or their derivatives to describe the structure and behavior of software architectures (MALAVOLTA et al., 2013), which are approaches with known limitations regarding architectural analysis because they are non-formal approaches. For this reason, it is not uncommon in the literature to find approaches that somehow combine non-formal ADLs with formalization strategies to allow the most diverse types of formal architectural analysis (KANG et al., 2013; TAOUFIK; TAHAR; MOURAD, 2016; OZKAYA; KOSE, 2018b; RADEMAKER; BRAGA; SZTAJNBERG, 2005; LIMA et al., 2017; DIAS et al., 2020).

Nonetheless, a systematic literature review (ARAUJO et al., 2019) highlighted some significant lacks regarding the formal verification of software architecture descriptions, related to SysML-based architectures, and approaches proposing a seamless formal enrichment of non-formal ADLs.

Considering this previously discussed scenario, this work focused on the formal enrichment of a SysML-based architectural description language, aiming to reconcile the easy-to-use architectural modeling expressiveness of a SysML-based ADL (SysADL) with the formal support of an ADL (π-ADL), relieving the architect of the limitation of dealing explicitly with formal specifications.

7.1 Revisiting the Research Questions

The main objective of this work was to define an approach to add formal semantics to SysADL to support formal property verification of software architectural description. To guide the execution of this work, three research questions were defined in Chapter 1, which, together with the answers obtained, are shown below:

RQ1: *Can a Model-Driven Development (MDD)-based approach enable a seamless formal enrichment of SysADL models transforming them to π-ADL models?*

This work implemented the transformation of software architectures modeled in SysADL into an architectural description in π-ADL, using an MDD-based approach that included the definition of the denotational semantics of SysADL in terms of π-ADL. This approach showed that it is possible, via a transformation between models, to have a formal description of a SysML-based architecture. The materialization of this transformation was implemented from the M2M transformation (SysADL for XML representation of the π-ADL model), followed by an M2T transformation (XML representation of the π-ADL model for source-code π-ADL), showing that it is possible to enable a seamless formal enrichment of SysADL models transforming them to π-ADL models using an MDD-based approach. The details involved in this process were shown in Chapter 3.

RQ2: *Is it possible to generate a valid executable architecture from the π-ADL model resulting from the MDD-based transformation of a SysADL model?*

The proposed approach defines a process in 4 steps, starting from an architectural description in SysADL until the verification of properties on this architecture transformed into π-ADL. Step 3, mainly, comprises generating an executable architecture from the transformed π-ADL model. The generation of the executable architecture was carried out with the help of the tool defined in work by Cavalcante (CAVALCANTE, 2016). It followed a 4-step process until the validation of the execution results concerning the requirements of the original SysADL architecture: (i) the automatic generation of source code in the Go programming from the resulting architectural description in π-ADL (CAVALCANTE, 2016); (ii) the insertion of Go code to implement the functional behavior of the architecture; (iii) the compilation

and the execution of the source code to execute the architecture; and (iv) the comparison of the execution results concerning the definitions of the original SysADL architecture. The implementation of phase 3 of the process showed that it is possible to generate a valid executable architecture from the π-ADL model resulting from an MDD-based transformation of a SysADL model. The details involved in this process were shown in Chapter 4.

RQ3: *Can an architectural property expressed from a SysADL model and its respective transformation into π-ADL be verified by equivalent traces of these models' execution?*

The final step, defined in the proposed approach, showed how it is possible to express properties on SysADL architectures and verify if these properties and restrictions on the architecture are satisfied by the architecture defined in SysADL and translated into π-ADL. To show that this verification is possible, an execution traces analysis approach was used, and the comparison of execution traces of the architectures in SysADL and π-ADL was used to verify if the state graph generated by the execution traces is the same for both architectures. Chapter 5 showed the details involved in this process.

7.2 Revisiting the Contributions

The contributions of this thesis are related to the formal enrichment of SysADL, defining an MDD-based approach, and providing a seamless transformation of SysADL architecture descriptions to the corresponding formal specifications in π-ADL, which can be formally verified.

The main contributions of this work are fourfold:

1. a denotational semantics to SysADL as a function of π-ADL;
2. the definition of a process to support the automated transformation of SysADL models into π-ADL models, the execution of the architecture and the formal verification of properties;
3. the validation of the π-ADL architecture generated by the MDD transformation to show that it is in accordance with the original SysADL architecture; and
4. the verification of formal architectural properties using execution traces.

A denotational semantics to SysADL The definition of the denotational semantics of SysADL as a function of π-ADL, shows that the transformation of syntactic elements from SysADL to π-ADL is feasible. Through this process, SysADL would have semantics.

The definition of a process to support the automated transformation of SysADL models into π-ADL models, the execution of the architecture and the formal verification of properties. It is defined as a function of three minor processes that complement each other to support the automatic transformation of an architectural description in a SysADL model to an architectural description in π-ADL: (i) the definition of algorithms to transform SysADL models into π-ADL models; (ii) an automated process to model-to-model transformation (M2M) from SysADL to π-ADL.; and (iii) an automated process to realize the transformation for π-ADL code generation from π-ADL model (M2T). The next steps of the process (execution and properties verification) are summarized below. The defined process is generic and, therefore, can be used in other contexts, adapting it to other languages.

The validation of the π-ADL architecture generated by the MDD transformation to show that it is in accordance with the original SysADL architecture. Focus on validating the architecture resulting from the MDD transformation process from SysADL to π-ADL, which is carried out through a process that involves four steps: (i) the automatic generation of Go source code from the resulting architectural description in π-ADL; (ii) the insertion of Go code to implement the functional behavior of the architecture; (iii) compilation and execution of the source code to execute the architecture; and (iv) the comparison of the execution results against the definitions of the original SysADL architecture.

The verification of formal architectural properties using execution traces. The verification of the formal architectural properties through observing the execution traces, comparing the execution traces of the original SysADL model and the transformed model π-ADL, an action that allows demonstrating the equivalence between them.

7.3 Future Work

Despite the contributions of this work described in Section 7.2, there are other directions for ongoing and future work. Some of these directions are intended to continue the research developed in this thesis, and others are planned to resolve some of the approach's limitations. Some possible future works are described below.

Integrate a model checking tool for property verification to the process defined in the approach of this work. In this work, we use analysis of execution traces to verify properties as a proof of concept. However, we understand the approach's limitations in a Model Checking process involving an exhaustive strategy for generating and verifying execution traces. To carry out the choice and integration of this tool some smaller steps are involved, such as (i) a systematic mapping of the available tools, considering features such as model and property specification languages, control strategies for the state explosion problem, and support for tool use; (ii) study and definition of a

taxonomy for categorization of types of properties; (iii) performance tests and use of the tool; (iv) integration to the process.

Extend execution viewpoint transformation from SysADL to π-ADL. By complementing the transformation of SysADL models with elements from the executable point of view, a complete architectural description would be generated concerning its functional behavior, including being propagated to the step of generating the executable architecture in Go for architecture validation (phase 3), considerably reducing the amount of code to be manually inserted, and contributing to less architectural erosion between the involved stages.

Extend the approach to handling with dynamics architecture. Dealing with dynamicity issues in architecture is a gap in works involving formal verification of architectural properties. π-ADL and its tools have already been extended to support these issues. Although SyADL has been developed with a base for the representation of dynamicity, these elements still need to be consolidated in language implementation. So we are interested in providing SysADL effectively with dynamicity to extend our approach with dynamic aspects.

Implementation of a strategy that would allow the formal verification of behaviors of SysADL architectures described from sequence or state machine diagrams. SysADL implements some SysML diagrams but it does not include the sequence and state machine diagrams. The extension of SysADL in this direction would be advantageous, as it would simplify the property verification process for software architects, who, in addition to using a SysML-based language to carry out the architectural description, would use a known diagram to specify the desired behavior, avoiding the use of more formal logics.

Endowing SysADL with a strategy to support environment configuration. It would be advantageous to have a strategy that supports environment configuration, an architectural style, for example, that facilitates the modeling of architectures that need to share environment information between the various elements of the model, such as sensors and actuators, allowing self-reference and passage of entire behaviors between architectural elements.

Study on scalability. Scalability is intrinsic when a solution involves software architecture and formal verification. We consider essential an additional examination that can test, evaluate and measure the scalability of our approach.

Bibliography

ALI, N. et al. Architecture consistency: State of the practice, challenges and requirements. *Empirical Software Engineering*, v. 23, p. 1–35, 02 2018. Cited on the page 77.

ARAUJO, C. et al. A research landscape on formal verification of software architecture description. *IEEE Access*, v. 7, p. 171752–171764, 2019. Cited 3 times on the pages 23, 94, and 108.

ARCHWARE. 2022. Disponível em: <https://www-archware.irisa.fr/software/pi-adl-toolset/>. Cited on the page 68.

ATLAS Group LINA & INRIA Nantes. *ATL: Atlas Transformation Language - User Manual*. [S.l.], 2005. Cited on the page 58.

BAIER, C.; KATOEN, J. *Principles of model checking*. [S.l.]: MIT Press, 2008. ISBN 978-0-262-02649-9. Cited on the page 79.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 3rd. ed. [S.l.]: Addison-Wesley Professional, 2012. ISBN 0321815734. Cited on the page 29.

BEHNAMGHADER, P. et al. A large-scale study of architectural evolution in open-source software systems. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 22, n. 3, p. 1146–1193, jun 2017. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-016-9466-0>. Cited on the page 64.

BOYER, B. et al. Plasma-lab: A flexible, distributable statistical model checking library. In: . Berlin, Heidelberg: Springer-Verlag, 2013. (QEST'13), p. 160–164. ISBN 9783642401954. Cited 2 times on the pages 90 and 91.

BROWNLEE, N. Formal systems (europe) ltd: Failures-divergence refinement–fdr2 user manual. *Blount MetraTech Corp. Accounting Attributes and Record Formats http://www.ietf. org/rfc/rfc2924. txt*, 2000. Cited on the page 90.

CAVALCANTE, E.; OQUENDO, F.; BATISTA, T. Architecture-based code generation: From π-ADL architecture descriptions to implementations in the Go language. In: AVGERIOU, P.; ZDUN, U. (Ed.). *ECSA 2014*. Switzerland: Springer, 2014. (LNCS, v. 8627), p. 130–145. Cited on the page 62.

CAVALCANTE, E. et al. Statistical model checking of dynamic software architectures. In: TEKINERDOGAN, B.; ZDUN, U.; BABAR, A. (Ed.). *ECSA 2016*. Switzerland: Springer International Publishing AG, 2016, (LNCS, v. 9839). p. 185–200. Cited 4 times on the pages 25, 38, 44, and 45.

CAVALCANTE, E. R. D. S. *A Formally Founded Framework for Dynamic Software Architectures*. Tese (Doutorado) — Université de Bretagne Sud ; Universidade federal do Rio Grande do Norte (Natal, Brésil), jun. 2016. Cited 14 times on the pages 23, 27, 38, 45, 46, 66, 68, 69, 72, 76, 92, 104, 109, and 167.

CHAUDET, C. et al. Architecture-driven software engineering: specifying, generating, and evolving component-based software systems. *IEE Proceedings-Software*, IET, v. 147, n. 6, p. 203–214, 2000. Cited on the page 51.

CHAUDET, C.; OQUENDO, F. π-space: a formal architecture description language based on process algebra for evolving software systems. In: IEEE. *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. [S.l.], 2000. p. 245–248. Cited on the page 51.

CIMATTI, A. et al. Nusmv: A new symbolic model verifier. In: HALBWACHS, N.; PELED, D. (Ed.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 495–499. ISBN 978-3-540-48683-1. Cited on the page 90.

CIOBANU, G.; TODORAN, E. N. Denotational semantics of membrane systems by using complete metric spaces. *Theoretical Computer Science*, v. 701, p. 85–108, 2017. ISSN 0304-3975. At the intersection of computer science with biology, chemistry and physics - In Memory of Solomon Marcus. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0304397517306382>. Cited on the page 51.

CLARKE, E. M.; EMERSON, E. A.; SIFAKIS, J. Model checking: Algorithmic verification and debugging. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 52, n. 11, p. 74–84, nov 2009. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/1592761.1592781>. Cited on the page 79.

CORONATO, A. *Engineering high quality medical software: Regulations, standards, methodologies and tools for certification*. United Kingdom: IET, 2018. Cited on the page 22.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. *IBM Systems Journal*, v. 45, n. 3, p. 621–645, 2006. Cited on the page 44.

DARSIE, J. *Statistical Software Properties: Defifinition, Inference and Monitoring*. Dissertação (Mestrado) — University of Nebraska, 2012. Cited on the page 80.

De Silva, L.; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, v. 85, n. 1, p. 132–151, 2012. ISSN 0164-1212. Dynamic Analysis and Testing of Embedded Software. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121211002044>. Cited on the page 64.

DEANTONI, J. et al. Rt-simex: Retro-analysis of execution traces. In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2010. (FSE '10), p. 377–378. ISBN 9781605587912. Disponível em: <https://doi.org/10.1145/1882291.1882357>. Cited on the page 77.

DIAS, F. *FormAr: Software Architecture Formalization for Critical Applications*. Dissertação (Mestrado) — UFRN, 2021. Cited on the page 104.

DIAS, F. et al. Empowering SysML-based software architecture description with formal verification: From SysADL to CSP. In: JANSEN, A. et al. (Ed.). *ECSA 2020*. Switzerland: Springer Nature Switzerland AG, 2020, (LNCS, v. 12292). p. 101–117. Cited 4 times on the pages 104, 106, 107, and 108.

FLOREZ, H.; SÁNCHEZ, M. E.; VILLALOBOS, J. Embracing imperfection in enterprise architecture models. In: CITESEER. *PoEM (Short Papers)*. [S.l.], 2013. p. 8–17. Cited on the page 41.

FRAPPIER, M. et al. Comparison of model checking tools for information systems. In: *IEEE International Conference on Formal Engineering Methods*. [S.l.: s.n.], 2010. Cited 2 times on the pages 90 and 91.

GARLAN, D. Formal modeling and analysis of software architecture: Components, connectors, and events. In: BERNARDO, M.; INVERARDI, P. (Ed.). *Formal Methods for Software Architectures, Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003, Bertinoro, Italy, September 22-27, 2003, Advanced Lectures*. Springer, 2003. (Lecture Notes in Computer Science, v. 2804), p. 1–24. Disponível em: <https://doi.org/10.1007/978-3-540-39800-4_1>. Cited on the page 21.

GARLAN, D.; SCHMERL, B. Using architectural models at runtime: Research challenges. In: OQUENDO, F.; WARBOYS, B. C.; MORRISON, R. (Ed.). *Software Architecture*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 200–205. ISBN 978-3-540-24769-2. Cited on the page 64.

GEORGAS, J. C.; DASHOFY, E. M.; TAYLOR, R. N. Architecture-centric development: A different approach to software engineering. *XRDS*, Association for Computing Machinery, New York, NY, USA, v. 12, n. 4, p. 6, aug 2006. ISSN 1528-4972. Disponível em: <https://doi.org/10.1145/1144359.1144365>. Cited on the page 21.

GOOGLE. *The Go programming language*. 2009. Disponível em: <https://golang.org/>. Cited 3 times on the pages 25, 47, and 64.

HAQUE, K. F.; ABDELGAWAD, A.; YELAMARTHI, K. Comprehensive performance analysis of zigbee communication: An experimental approach with xbee s2c module. *Sensors*, v. 22, n. 9, 2022. ISSN 1424-8220. Cited 2 times on the pages 28 and 120.

HOJAJI, F. et al. Model execution tracing: A systematic mapping study. *Softw. Syst. Model.*, Springer-Verlag, Berlin, Heidelberg, v. 18, n. 6, p. 3461–3485, dec 2019. ISSN 1619-1366. Disponível em: <https://doi.org/10.1007/s10270-019-00724-1>. Cited on the page 77.

HOLZMANN, G. The model checker spin. *IEEE Transactions on Software Engineering*, v. 23, n. 5, p. 279–295, 1997. Cited on the page 90.

IEC 62279:2015. *Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*. Switzerland: IEC, 2015. Cited on the page 22.

ISO 14977. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. [S.l.], 1996. Cited on the page 51.

ISO/IEC. Iso / iec 25010 : 2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. In: . [S.l.: s.n.], 2013. Cited 2 times on the pages 11 and 80.

ISO/IEC/IEEE 42010. *Systems and Software Engineering – Architecture Description*. Switzerland, 2007. Cited 5 times on the pages 21, 23, 29, 32, and 108.

JEGOUREL, C.; LEGAY, A.; SEDWARDS, S. A platform for high performance statistical model checking – plasma. In: FLANAGAN, C.; KÖNIG, B. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 498–503. ISBN 978-3-642-28756-5. Cited on the page 45.

JÉGOUREL, C.; LEGAY, A.; SEDWARDS, S. A platform for high performance statistical model checking - PLASMA. In: FLANAGAN, C.; KÖNIG, B. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Springer, 2012. (Lecture Notes in Computer Science, v. 7214), p. 498–503. Disponível em: <https://doi.org/10.1007/978-3-642-28756-5_37>. Cited 2 times on the pages 90 and 91.

KANG, E.-Y. et al. A methodology for formal analysis and verification of east-adl models. *Reliability Engineering & System Safety*, v. 120, p. 127–138, 2013. ISSN 0951-8320. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0951832013001725>. Cited 8 times on the pages 95, 96, 97, 99, 100, 106, 107, and 108.

KEMPER, P.; TEPPER, C. Automated trace analysis of discrete-event system models. *IEEE Trans. Software Eng.*, v. 35, n. 2, p. 195–208, 2009. Disponível em: <https://doi.org/10.1109/TSE.2008.75>. Cited on the page 77.

KOGUT, P.; CLEMENTS, P. Features of architecture description languages. In: *In Proceedings of the Eighth International Workshop on Software Specification and Design*. [S.l.: s.n.], 1994. p. 16–25. Cited on the page 30.

LEITE, J. et al. Designing and executing software architectures models using SysADL Studio. In: *2018 IEEE Int. Conf. on Software Architecture Companion*. USA: IEEE, 2018. p. 81–84. Cited 2 times on the pages 62 and 121.

LIMA, L. et al. An integrated semantics for reasoning about sysml design models using refinement. *Softw. Syst. Model.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 16, n. 3, p. 875–902, jul. 2017. ISSN 1619-1366. Cited 6 times on the pages 24, 101, 103, 106, 107, and 108.

LIU, Y.; SUN, J.; DONG, J. S. Pat 3: An extensible architecture for building multi-domain model checkers. In: *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering*. USA: IEEE Computer Society, 2011. (ISSRE '11), p. 190–199. ISBN 9780769545684. Disponível em: <https://doi.org/10.1109/ISSRE.2011.19>. Cited 2 times on the pages 90 and 92.

MALAVOLTA, I. et al. What industry needs from architectural languages: A survey. *IEEE Trans. on Software Engineering*, v. 39, n. 6, p. 869–891, Jun. 2013. Cited 5 times on the pages 21, 22, 23, 103, and 108.

MEDVIDOVIC, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, v. 26, n. 1, p. 70–93, 2000. Cited on the page 31.

MENS, T. Model transformation: A survey of the state of the art. In: _____. *Model-Driven Engineering for Distributed Real-Time Systems*. [S.l.]: John Wiley & Sons, Ltd, 2013. cap. 1, p. 1–19. ISBN 9781118558096. Cited on the page 42.

MILNER, R. *Communicating and Mobile Systems: The π-Calculus*. USA: Cambridge University Press, 1999. ISBN 0521658691. Cited 2 times on the pages 23 and 37.

NAKAGAWA, E. Y.; OQUENDO, F.; BECKER, M. Ramodel: A reference model for reference architectures. In: *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. USA: IEEE Computer Society, 2012. p. 297–301. ISBN 9780769548272. Cited on the page 30.

OBEO. *Acceleo - User Manual*. [S.l.], 2011. Cited on the page 61.

OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.6*. 2001. Disponível em: <https://www.omg.org/spec/SysML/1.6/>. Cited 4 times on the pages 23, 30, 32, and 108.

OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1*. 2011. Disponível em: <http://www.omg.org/spec/UML/2.4.1>. Cited on the page 30.

OMG. *The Action Language for Foundational UML Specification*. 2017. Disponível em: <https://www.omg.org/spec/ALF>. Cited on the page 36.

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*, v. 29, n. 3, p. 1–14, maio 2004. Cited 4 times on the pages 23, 37, 44, and 51.

OQUENDO, F. π-Method: A model-driven formal method for architecture-centric software engineering. *SIGSOFT Softw. Eng. Notes*, Association for Computing Machinery, New York, NY, USA, v. 31, n. 3, p. 1–13, may 2006. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/1127878.1127885>. Cited on the page 68.

OQUENDO, F. et al. The archware adl: Definition of the abstract syntax and formal semantics. *ARCHWARE European RTD Project IST-2001-32360. Deliverable D*, v. 1, 2002. Cited on the page 51.

OQUENDO, F.; LEITE, J.; BATISTA, T. *Software Architecture in Action: Designing and executing architectural models with SysADL grounded on the OMG SysML Standard*. Switzerland: Springer International Publishing, 2016. Cited 6 times on the pages 23, 30, 31, 32, 33, and 82.

OQUENDO, F. et al. Archware: Architecting evolvable software. In: *European Workshop on Software Architecture*. [S.l.: s.n.], 2004. Cited on the page 68.

OZKAYA, M. The analysis of architectural languages for the needs of practitioners. *Software: Practice and Experience*, v. 48, n. 5, p. 985–1018, maio 2018. Cited on the page 22.

OZKAYA, M. Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling? *Information and Software Technology*, v. 95, p. 15–33, Mar. 2018. Cited 2 times on the pages 22 and 108.

OZKAYA, M.; KOSE, M. A. SAwUML - UML-based, contractual software architectures and their formal analysis using SPIN. *Computer Languages, Systems and Structures*, Elsevier Ltd, v. 54, p. 71–94, 2018. ISSN 14778424. Cited on the page 24.

OZKAYA, M.; KOSE, M. A. Sawuml - uml-based, contractual software architectures and their formal analysis using SPIN. *Comput. Lang. Syst. Struct.*, v. 54, p. 71–94, 2018. Disponível em: <https://doi.org/10.1016/j.cl.2018.04.005>. Cited 7 times on the pages 96, 98, 99, 100, 106, 107, and 108.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, Association for Computing Machinery, New York, NY, USA, v. 17, n. 4, p. 40–52, out. 1992. ISSN 0163-5948. Cited 2 times on the pages 30 and 77.

PLASMA. 2022. Disponível em: <https://project.inria.fr/plasma-lab/>. Cited 5 times on the pages 66, 92, 100, 105, and 168.

QUILBEUF, J. et al. A logic for the statistical model checking of dynamic software architectures. In: MARGARIA, T.; STEFFEN, B. (Ed.). *ISoLA 2016*. Switzerland: Springer, 2016, (LNCS, v. 9952). p. 806–820. Cited on the page 45.

RADEMAKER, A.; BRAGA, C.; SZTAJNBERG, A. A rewriting semantics for a software architecture description language. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 130, p. 345–377, 2005. Cited 6 times on the pages 101, 102, 103, 106, 107, and 108.

ROBINSON, T. G. et al. *Failures Divergences Refinement (FDR) Version 3*. [S.l.], 2013. Disponível em: <https://www.cs.ox.ac.uk/projects/fdr/>. Cited on the page 90.

ROSCOE, A. *Understanding Concurrent Systems*. 1st. ed. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 184882257X. Cited 2 times on the pages 90 and 104.

ROURE, D. D. et al. Floodnet - improving flood warning times using pervasive and grid computing. 01 2006. Cited 2 times on the pages 28 and 120.

RTCA/DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*. USA: RTCA, 2012. Cited on the page 22.

SANCHEZ, D.; FLOREZ, H. Model driven engineering approach to manage peripherals in mobile devices. In: SPRINGER. *International Conference on Computational Science and Its Applications*. [S.l.], 2018. p. 353–364. Cited on the page 42.

SCHMIDT, D. A. *Denotational Semantics: A Methodology for Language Development*. USA: William C. Brown Publishers, 1986. ISBN 0697068492. Cited on the page 50.

SEIDEWITZ, E. What models mean. *IEEE software*, IEEE, v. 20, n. 5, p. 26–32, 2003. Cited on the page 41.

SHAW, M.; CLEMENTS, P. The golden age of software architecture. IEEE Computer Society Press, Washington, DC, USA, v. 23, n. 2, p. 31–39, mar 2006. ISSN 0740-7459. Disponível em: <https://doi.org/10.1109/MS.2006.58>. Cited on the page 21.

SLONNEGER, K.; KURTZ, B. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. 1st. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201656973. Cited on the page 51.

TAOUFIK, S. R.; TAHAR, B. M.; MOURAD, K. Behavioral verification of UML2.0 software architecture. In: *12th Int. Conf. on Semantics, Knowledge and Grids*. USA: IEEE, 2016. p. 115–120. Cited 8 times on the pages 24, 96, 97, 99, 100, 106, 107, and 108.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. *Software Architecture: Foundations, theory, and practice*. USA: John Wiley & Sons, Inc., 2010. Cited 2 times on the pages 22 and 108.

WEYNS, D. et al. Design for sustainability = runtime adaptation ⋃ evolution. In: . New York, NY, USA: Association for Computing Machinery, 2015. (ECSAW '15). ISBN 9781450333931. Cited on the page 108.

WIMMER, M. et al. Surveying rule inheritance in model-to-model transformation languages. *Journal of Object Technology*, v. 11, n. 2, p. 1–46, ago. 2012. Cited on the page 43.

ZHENG, Y.; TAYLOR, R. N. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In: *Proceedings of the 34th International Conference on Software Engineering*. [S.l.]: IEEE Press, 2012. (ICSE '12), p. 628–638. ISBN 9781467310673. Cited on the page 64.

ZOOR, M.; APVRILLE, L.; PACALET, R. Execution trace analysis for a precise understanding of latency violations. In: *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. [S.l.: s.n.], 2021. p. 123–133. Cited on the page 77.

APPENDIX A – Flood Monitoring System in SysADL

Flooding is a problem in many places worldwide, especially in urban regions crossed by rivers; when associated with extreme events, that becomes a challenge for security and regional development. Regardless of their magnitude, floods pose a risk and must be addressed. In this context, a system to monitor the flow and level of water in rivers, together with the support of meteorological information, can help issue warnings and emergency alerts. Early warnings are essential to reduce the scale and cost of damage caused by floods, which have a clear correlation with both the depth of the flood and the amount of early warning given (ROURE et al., 2006).

This work uses the Flood Monitoring System (FMS) to illustrate the proposed approach. The FMS modeling in this work is a system typically based on a network composed of wireless sensors that measure the water level in flood-prone areas close to a river, with a Gateway station that analyzes the measured data and can trigger alerts when a flood condition is detected. Communication between these elements occurs through wireless network connections using ZigBee(HAQUE; ABDELGAWAD; YELAMARTHI, 2022), a communication protocol for IoT devices.

This appendix details the SysADL modeling of a possible architecture for the FMS. Section A.1 presents an overview of the components of the FMS, while Section A.2 shows the details of FMS SysADL architecture from a structural and behavioral view.

A.1 Overview

Figure 57 shows the main architecture of the system, containing all elements: *Sensor, SensorEnv, ZigBee, Gateway, Observer and Env*. The elements *Sensor*, *SensorEnv* and *ZigBee* can be added and removed during system execution, according to the need for river monitoring coverage. In the Figure 57, there are two instances of each (*Sensor*, *SensorEnv* and *ZigBee*).

The *sensors* compoents communicate via *ZigBee* connectors and a *gateway* component receives all measurements to assess the current risk. Each measurement from a sensor is propagated to its neighbors via ZigBee connectors until it reaches the gateway. The environment is modeled through the *Env* component, and the *SensorEnv* connector performs the communication between the sensor components and the environment. *Env* is responsible for synchronizing the model by defining cycles corresponding to the frequency with which the sensors take measurements. A cycle consists of: (i) signaling to

Gateway that a new cycle has started; (ii) updating the state of the river; (iii) flaging each *SensorEnv* connector to deliver a new measurement; and (iv) waiting for each connector *SensorEnv* to confirm that a new measurement has been delivered.

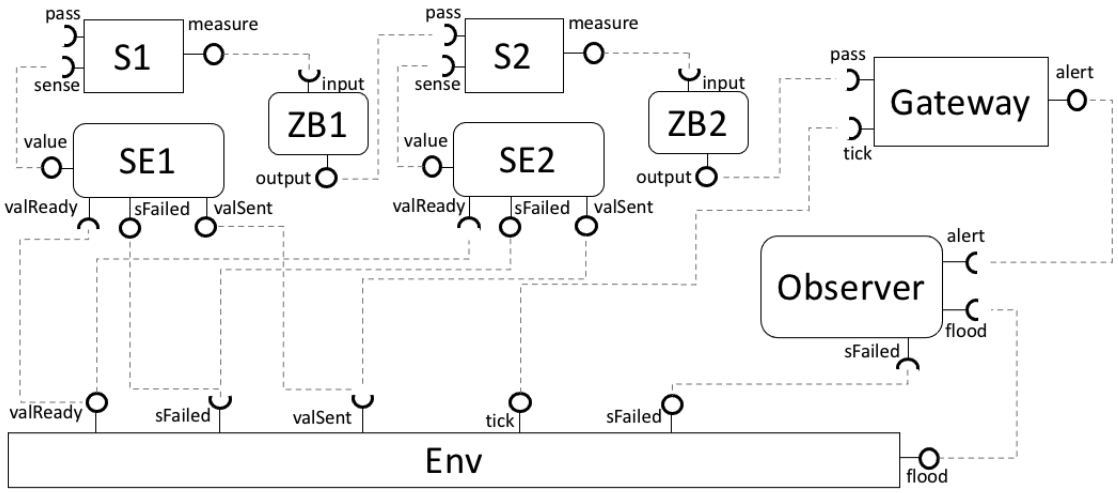


Figure 57 – Flood Monitoring System with two Sensors

A.2 FMS architecture in SysADL

The FMS architecture was modeled using the SysADL Studio tool. It provides an engineering environment and workbench for architectural design based on the SysADL software architecture modeling language. SysADL Studio[1] (LEITE et al., 2018) is a plugin for Eclipse[2], developed using Sirius[3] and Xtext[4], which are also Eclipse plugins for creating graphical modeling workbenches and for the development of programming languages and domain-specific languages, respectively. The diagrams are graphical representations of the well-founded SysADL metamodel and allow you to define models in light of Structural, Behavioral, and Executable viewpoints.

A.2.1 Structural view

The FMS structural view was organized in SysADL Studio in such a way as to separate the definitions of types, ports, connectors, and components in different packages, as shown in Figure 58. Imports were defined so that the elements inside the packages could use the elements described in another package.

[1] SysADL Studio. Available at: <https://sysadl.imd.ufrn.br>
[2] Eclipse. Available at:<https://www.eclipse.org>
[3] Sirius for Eclipse. Available at:<https://www.eclipse.org/sirius/>
[4] Xtext for Eclipse. Available at:<https://www.eclipse.org/Xtext/>

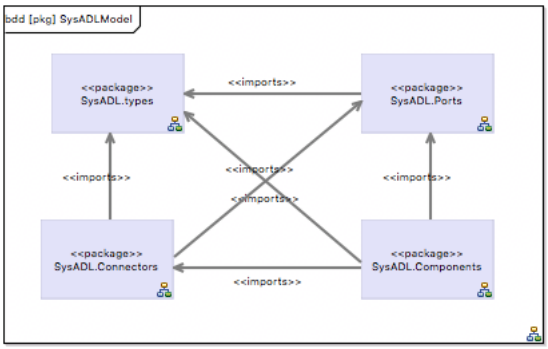


Figure 58 – FMS Model Package Organization in SysADL

A.2.1.1 Types

The *SysADL.types* package defines the various data types that were used in the FMS modeling: the five primitive value types in SysADL (*Int*, *Boolean*, *String*, *Void*, *Real*), two value types (*MV* - to represent the water volume measurement and *IDSensor* - to represent the sensor identifiers), and a data type (*CmH2ODT* - composed of the attributes *idS(IDSensor)* and *mvSensor(MV)*, to represent the measurement information of the measured volume and the sensor identified who performed the measurement). The types defined in *SysADL.types* can be seen in Figure 59.

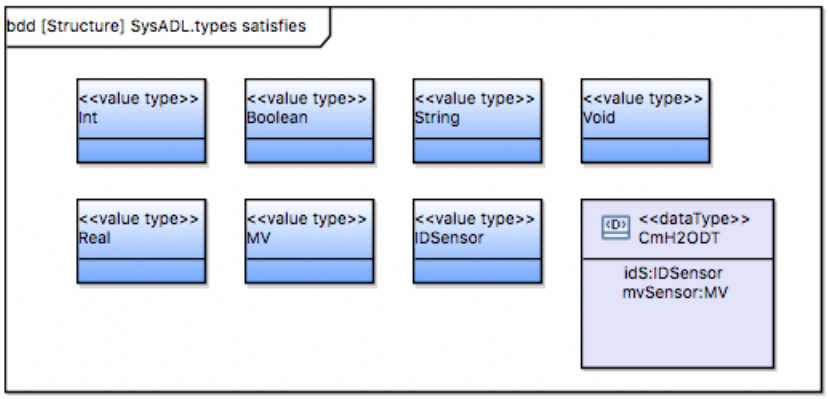


Figure 59 – FMS types defined in the package SysADL.types

A.2.1.2 Ports

In the *SysADL.Ports* package, the ports used in the FMS modeling are defined. Two ports were created for each type defined in *SysADL.type*, the *input* port, and another *output* port. As can be seen in Figure 60: there are eight *input* ports (*intIPT (Int)*, *booleanIPT (Boolean)*, *stringIPT (String)*, *voidIPT (Void)*, *realIPT (Real)*, *mvIPT (MV)*, *sensorIDIPT (sensorID)*, and *CmH2OIPT (CmH20)*), and there are eight *output* ports (*intOPT (Int)*, *booleanOPT (Boolean)*, *stringOPT (String)*, *voidOPT (Void)*, *realOPT (Real)*, *mvOPT (MV)*, *sensorIDOPT (sensorID)*, and *CmH2OOPT (CmH20)*).

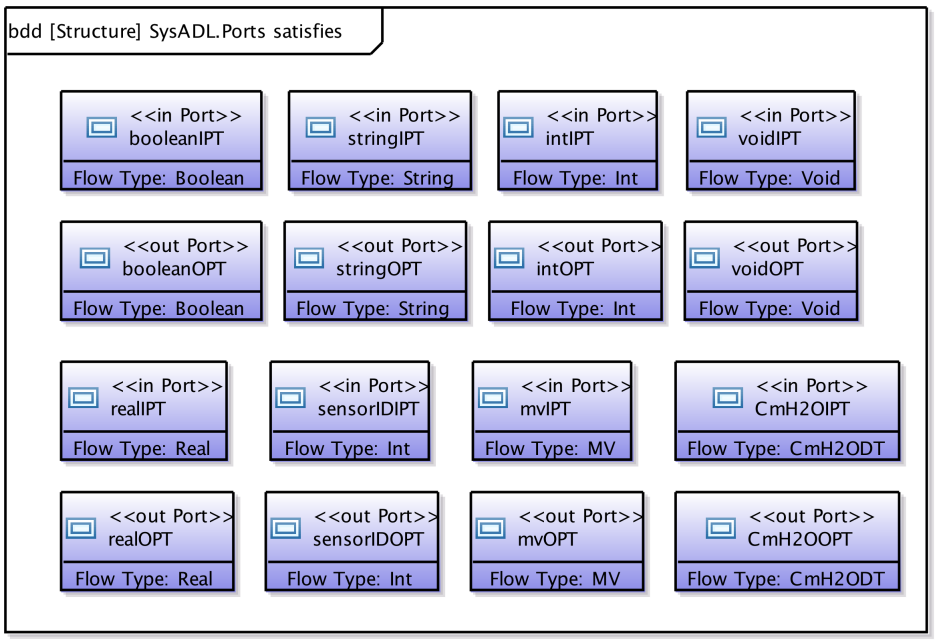


Figure 60 – FMS ports defined in the package SysADL.Ports

A.2.1.3 Connectors

In *SysADL.Connectors* are connector ports used to connect the components of the FMS modeling architecture to provide a path for data traffic between the components. The connectors bond with the compatible ports of their type. In this way, for each port pair (input and output) of the same type, there is a defined connector, as can be seen in Figure 61: *intCN (Int)*, *booleanCN (Boolean)*, *stringCN (String)*, *voidCN (Void)*, *realCN (Real)*, *mvCN (MV)*, *IDSensorCN (IDSensor)*, and *CmH2OCN (CmH2ODT)*. Remember that SysADL connectors use conjugate ports, so the flow of the connectors is always defined from the conjugate output port to the conjugate input port.

A.2.1.4 Components

In the architectural modeling of the FMS in SysADL, the six components shown in Figure 62 were defined: *SensorCP*, *ZigBeeConnectorCP*, *SensorEnvConnectorCP*, *GatewayCP*, *ObserverConnectorCP*, *EnvCP*, in addition to the *FMSa1ARCH* architecture component.

SensorCP. The *SensorCP* component was structurally defined in terms of three ports, two input, and one output:

- *senseIn(MV)*: receives the measurement of the volume of water (in *mv*), performed by the physical sensor;
- *passIn(CmH2ODT)*: receives the measurement of the volume of water (in *CmH2O* (idSensor, mv)) performed by another sensor in the network;

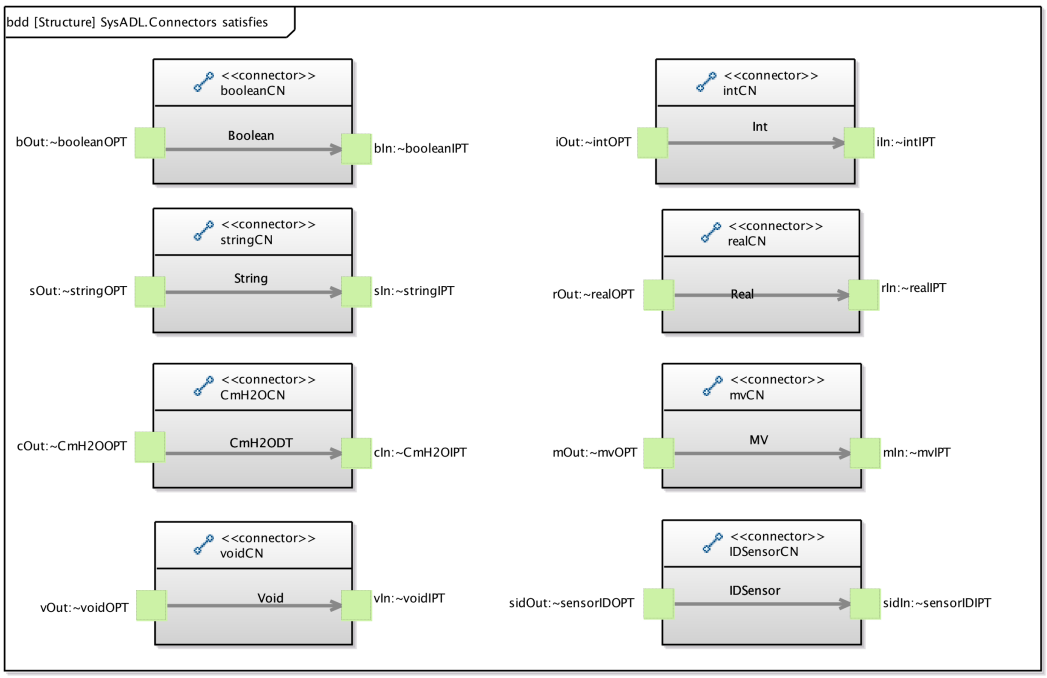


Figure 61 – FMS connectors defined in the package SysADL.Connectors

- *measureOut(CmH2ODT)*: output port through which the water volume measurement (in CmH2O) is sent to the *ZigBee* associated with the sensor;

ZigBeeConnectorCP component. The *ZigBeeConnectorCP* component was structurally defined as a function of two ports, one for input and another for output:

- *passZIn(CmH2ODT)*: receives the measurement of the volume of water (in *CmH2O* (IDSensor, MV)) performed/passed on to *ZigBee* by the previous *sensor* in the network;
- *passZOut(CmH2ODT)*: output port through which the water volume measurement (in CmH2ODT) is sent to the next *Sensor* or *Gateway* in the network.

SensorEnvConnectorCP component. The *SensorEnvConnectorCP* component was structurally defined in terms of four ports,one for input and three for output:

- *valueReadyIn(Boolean)*: receives a signal from *EnvCP* informing that a new volume measurement can be performed;
- *valueSOut(MV)*: sends to *SensorCP* the result of the volume measurement (in *mv*);
- *valueSentOut(Boolean)*: confirms to *EnvCP* that the volume measurement value has already been sent to *SensorCP*;
- *sFailedSOut(IDSensor)*: informs *EnvCP* the identifier (position) of a failed sensor;

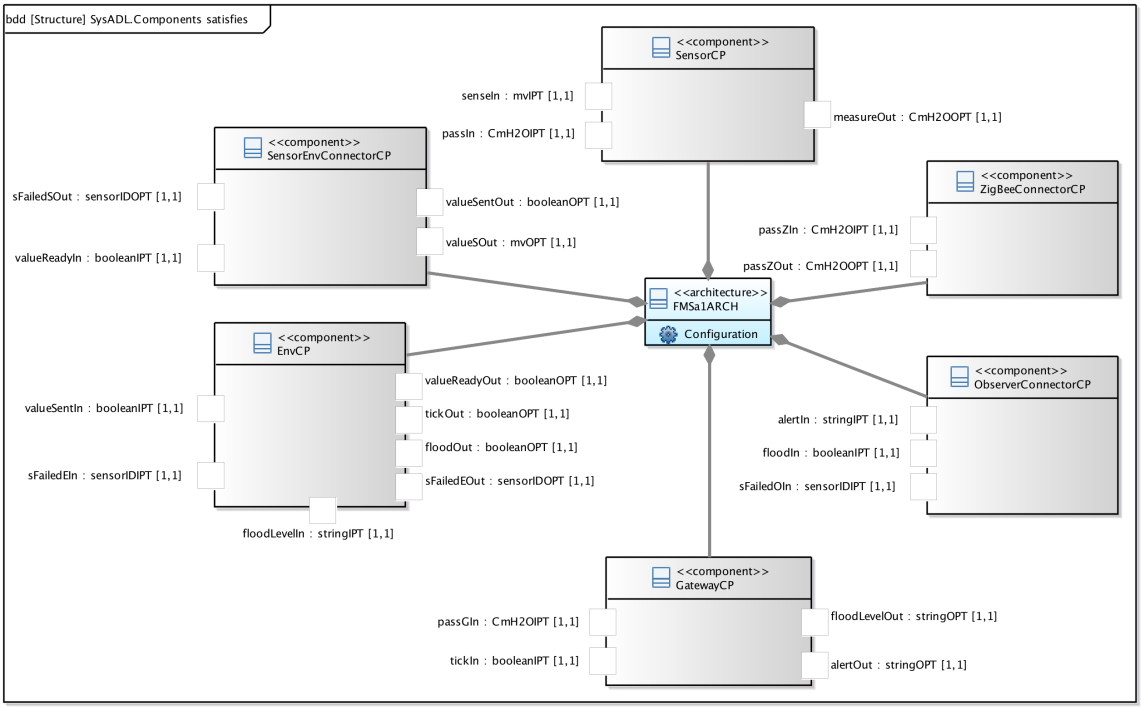


Figure 62 – FMS components defined in the package SysADL.Components

GatewayCP component. The GatewayCP component was defined with four ports, two input and another two output:

- *passGIn(CmH2O)*: receives the measurement of the volume of water passed through the *ZigBee* that precedes the gateway in the network;
- *tickIn(Boolean)*: synchronization signal sent by *EnvCP* to inform that a new cycle has started;
- *floodLevelOut(String)*: informs to *ObserverConnectorCP* if there is flooding in the monitored section or not;
- *alertOut(String)*: sends to *ObserverConnectorCP* an alert message from the volume obtained with river monitoring.

ObserverConnectorCP. Component *EnvCP* has been structurally defined with three input ports:

- *alertIn(String)*: receives from *GatewayCP* an alert message about the volume generated with river monitoring.
- *floodIn(Boolean)*: receives information from *EnvCP* if there is flooding in the monitored section or not;
- *sFailedOIn(IDSensor)*: receives from *EnvCP* an identifier information for a sensor that failed;

EnvCP component. The *EnvCP* component was structurally defined in terms of seven ports, three input, and four output:

- *sFailedEIn(IDSensor)*: receives from *SensorEnvCP* the identifier of a failed sensor to update the network records;
- *valueSentIn(Boolean)*: receives a signal from *SensorEnvCP* informing that a new measurement value has been sent to *SensorCP*;
- *floodLevelIn(String)*: receives informs from GatewayCP if there is flooding in the monitored section or no;
- *tickOut(Boolean)*: sends a signal to *GatewayCP* informing that a new cycle starts;
- *valueReadyOut(Boolean)*: informs *SensorEnv* that there is a new measurement available on the physical sensor;
- *floodOut(boolean)*: informs *ObserverConnectorCP* if there is flooding in the monitored section or not;
- *sFailedEOut(IDSensor)*: informs *ObserverConnectorCP* the sensor fail ID;

A.2.1.5 Architecture

Figure 62 also shows the *FMSa1ARCH* architecture. The definition of a configuration for *FMSa1ARCH* example is seen in Figure 63. The *S1:SensorCP* component is connected to the *EnvS1:SensorEnvConnector* component that informs the measurement of the water volume (*MV*) to *S1* through the connection between the *valueSOut(EnvS1)* and *senseIn(S1)* ports. *S1* also connects to *Zb1:ZigBeeConnectorCP* by sending to *Zb* a data *CmH2ODT (IDSensor, MV)* via the connection between the *measureOut(S1)* and *passZIn(Zb1)* ports. *Zb1* sends the data received from *S1* to the component *Gt:GatewayCP* through the connection between the ports *passZOut(Zb1)* and *passGIn(Gt)*.

Gt has connections to the *Env:EnvCP* and *Obs:ObserverConnectorCP* components. Through the connection between *tickOut(Env)* and *tickIn(Gt)*, *Gt* receives the synchronization signal for a new cycle, and through the connection between *floodLevelOut(Gt)* and *floodLevelIn(Env)*, *Gt* informs *Env* of the calculated flood level. *Gt* also sends an alert message to *Obs* through the connection between the *alertOut(Gt)* and *alertIn(Obs)* ports, considering the previously calculated flood risk.

The *Obs* component still communicates with the *Env* through two connections: between *floodOut(Env)* and *floodIn(Obs)*, and via *sFailedEOut(Env)* and *sFailedOIn(Env)*, which inform the Obs a data telling if there is flooding in the area monitored and an identifier of a sensor that presents itself with failure in the network, respectively.

The *FMSa1ARCH* architecture has three connections between the *Env* and *EnvS1* components. *Env* sends a signal to *EnvS1* via the connection between *valueReadyOut(Env)* and *valueReadyIn(EnvS1)*, informing that a new measurement is available on the physical sensor. Finally, the connections between *sFailedSOut(EnvS1)* and *sFailedEIn(Env)*, and *valueSentOut(EnvS1)* and *valueSentIn(Env)*, respectively notify the identifier of a sensor that fails, and that a new measurement has been sent to component *S1*.

To connect the components ports of *FMSa1ARCH* architecture, eleven connectors named from *c1* to *c11*, of types compatible with the ports to be connected, were instantiated: one *mvCN* connector (*c1*), two *CmH2OCN* connectors (*c2* and *c3*), four *booleanCN* (*c4*, *c6*, *c7*, and *c9*), two *IDsensorCN* connectors (*c5* and *c11*), and finally two *stringCN* connectors (*c8* and *c10*).

Connectors in SysADL can be composed of other connectors and can have *configurations*, but in the approach presented in this thesis, we chose to use only simple connectors with simple ports and whose only function is to connect simple ports.

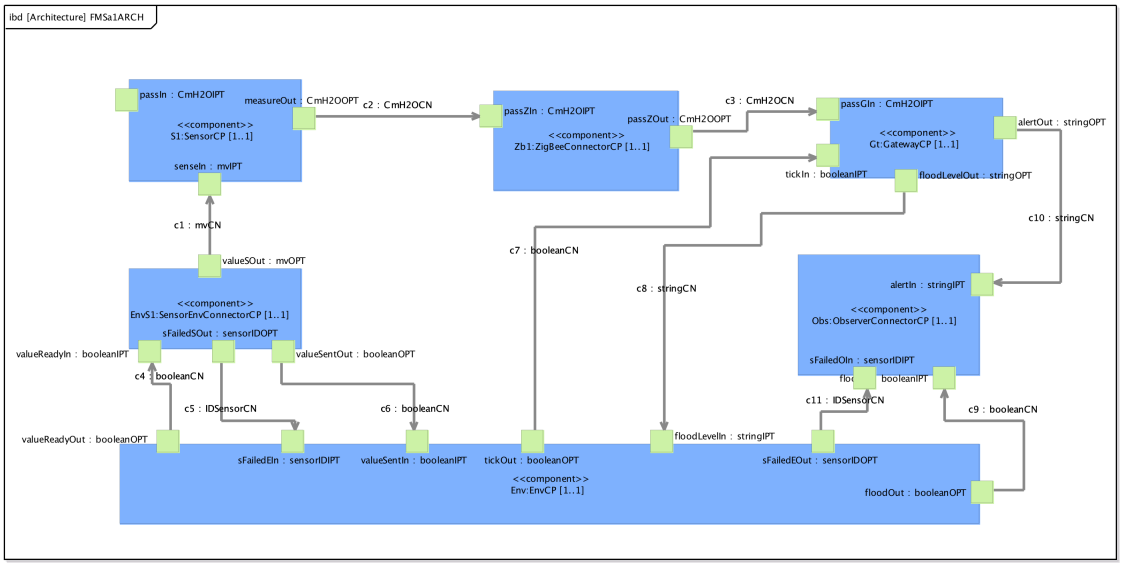


Figure 63 – The software architecture configuration of FMS

A.2.2 Behavioral view

Every simple component that is no boundary or composite must have its behavior described through an *active* and its respective *actions* in the SysADL model. This section describes the behavior of each FMS simple component: *SensorCP*, *ZigBeeConnectorCP*, *SensorEnvConnectorCP*, *GatewayCP*, *ObserverConnectorCP*, and *EnvCP*.

SensorCP. Figure 64a shows a bdd excerpt that describes the definition of activity and actions of the *SensorCP* component through the *SensorCPAC* activity that is composed of two different actions: *PassingMeasureAN* and *LocalMeasuringAN*. Figure 64b exhibits the activity diagram of *SensorCPAC* with its actions and pins.

- LocalMeasuringAN: defines the behavior of *SensorCP* when receiving the volume measurement from the physical sensor through *senseIn*, and forwarding the volume measurement value to *ZigBeeCP* via *measureOut*;

- PassingMeasureAN: describes the action of *SensorCP* receives a volume measurement from a previous *Zigbee* via *passIn* port, and forwarding the volume measurement value to next *ZigBee* via *measureOut*;

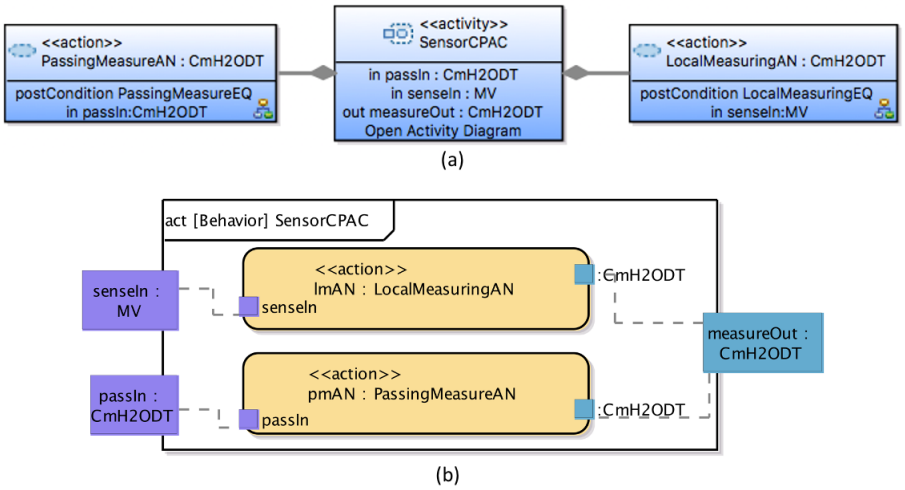


Figure 64 – *SensorCPAC* behavioral BDD and Activity Diagram

ZigBeeConnectorCP. Figure 65a shows a bdd excerpt that describes the definition of activity and actions of the *ZigBeeConnectorCP* component through the *ZigBeeConnectorCPAC* activity that has one single action: *ZbPassingMeasureAN*. Figure 65b exhibits the activity diagram of *ZigBeeConnectorCPAC* with its action and pins.

- ZbPassingMeasureAN: describes the action of *ZigBeeConnectorCP* receives a volume measurement from a previous *SensorCP* via *passZIn* port, and forwarding the volume measurement value to next *SensorCP* or *GatewayCP* via *passZOut*;

SensorEnvConnectorCP. Figure 66a shows a bdd excerpt with the definition of activity and actions of the *SensorEnvConnectorCP* component through the *SensorEnvConnectorCPAC* activity that is composed of three different actions: *GetSendingDataAN*, *NotifyingSendValueAN*, and *SendingWarmingSensorFailAN*. Figure 66b exhibits the activity diagram of *SensorEnvConnectorCPAC* with its action and pins.

- GetSendingDataAN: *SensorEnvConnector* receives a signal from *EnvCP* via *valueReadyIn* informing that a new volume measurement can be performed, and the new measure is sent to *SensorCP* through *valueSOut*

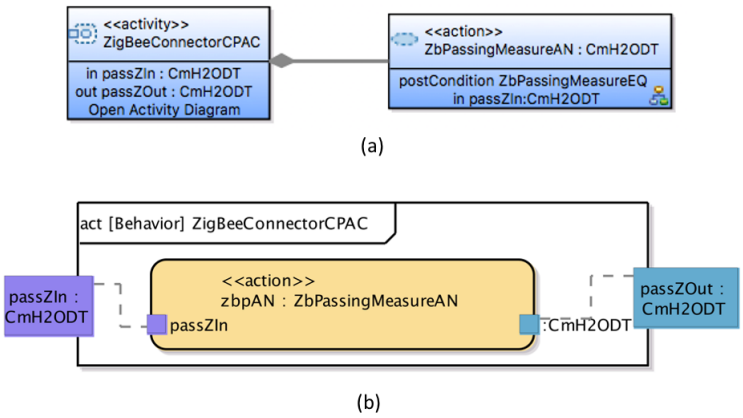


Figure 65 – *ZigBeeConnectorCPAC* behavioral BDD and Activity Diagram

- NotifyingSendValueAN: after confirming that a new measure was sent via *valueSOut*, this behavior signals to *EnvCP* that a value has been sent through *valueSentOut*;
- SendingWarmingSensorFailAN: this behavior checks if the volume measurement sent via *valueSOut* is outside the expected parameters; it means that the sensor is failing, and a failure signal will be sent via *sFailedSOut*;

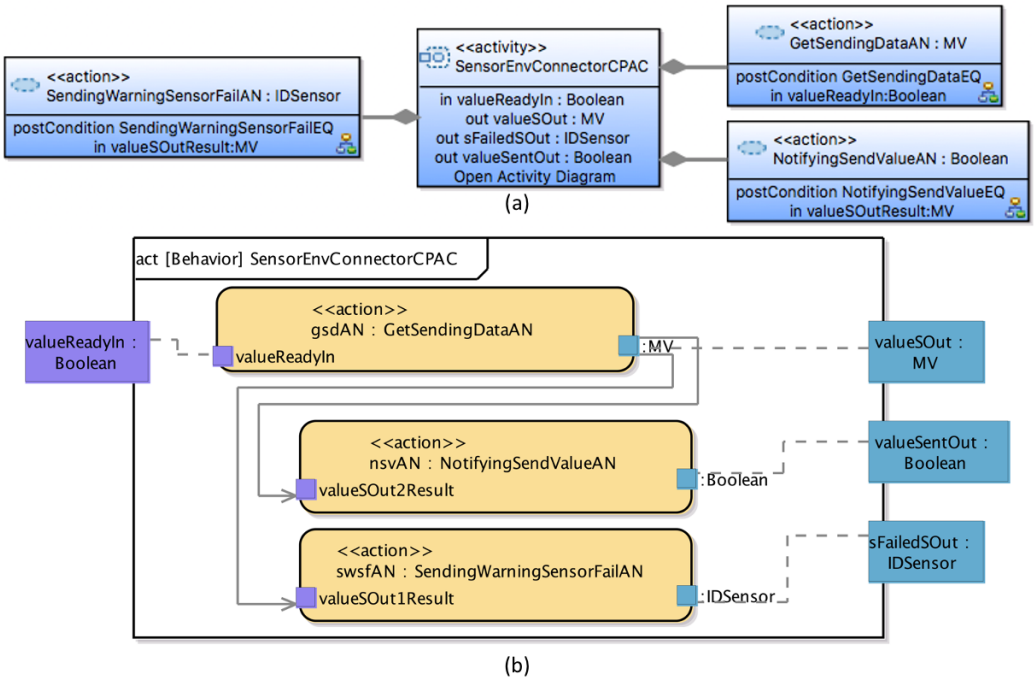


Figure 66 – *SensorEnvConnectorCPAC* behavioral BDD and Activity Diagram

GatewayCP. Figure 67a shows a bdd excerpt with the definition of activity and actions of the *GatewayCP* component through the *GatewayCPAC* activity that is composed of three different actions: *StoreINfoSensorsAN*, *CalculatingFloodLevelAN*, and

SendingAlertLevelAN. Figure 67b exhibits the activity diagram of *GatewayCPAC* with its action and pins.

- StoreINfoSensorsAN: this action models the behavior of the *GatewayCP* when the *ZigBeeConnectorCP* connected to it passes the measurement of one of the network sensors via *passIn*. The *GatewayCP* must update the information about the sensor's measurements, storing the data in the *InfoSensorsDB(CmH2ODT)* datastore;
- CalculatingFloodLevelAN: after receiving and storing the measurement from a sensor, the *GatewayCP* must calculate the flood level, which will be stored in the *AlertsLogDB(String)* datastore and send to the *EnvCP* component through *floodLevelOut*.
- SendingAlertLevelAN: when receiving a new measurement from a sensor, this action will be activated to send an alert message via *alertOut* about the river alert level to the *ObserverConnectorCP* component;

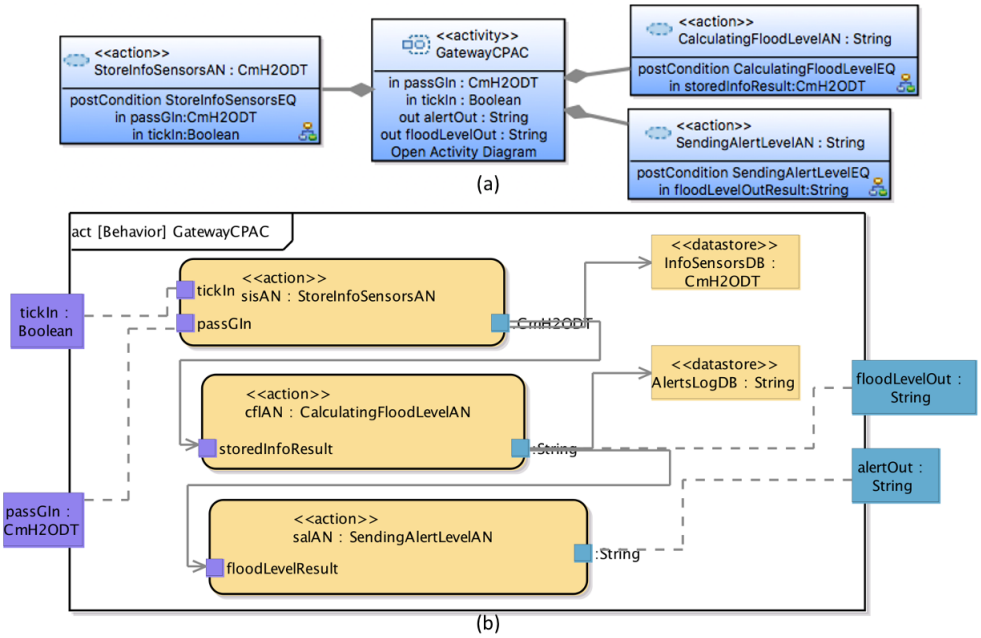


Figure 67 – *GatewayCPAC* behavioral BDD and Activity Diagram

ObserverConnectorCP. Figure 68a shows a bdd excerpt with the definition of activity and actions of the *ObserverConnectorCP* component through the *ObserverConnectorCPAC* activity that is composed of three different actions: *StoreAlertAN*, *StoreFailureSensorAN*, and *StoreFloodStatusAN*. Figure 68b exhibits the activity diagram of *ObserverConnectorCPAC* with its action and pins.

- StoreAlertAN: after receiving via *alertIn* the flood level alert message sent by *GatewayCP*, the *ObserverConnectorCP* stores the message in the *AlertLog(String)* datastore;

- StoreFailureSensorAN: the *ObserverConnectorCP* receives from *EnvCP* via *sFailedOIn* data about a sensor fail id, and it stores the information in the *FailureSensorLog(IDSensor)*;

- StoreFloodStatusAN: after receiving via *floodIn* a sing informing if there is a flood in the monitoring area, sent by *GatewayCP*, *ObserverConnectorCP* stores the message in *FloodStatusLog(Boolean)* datastore ;

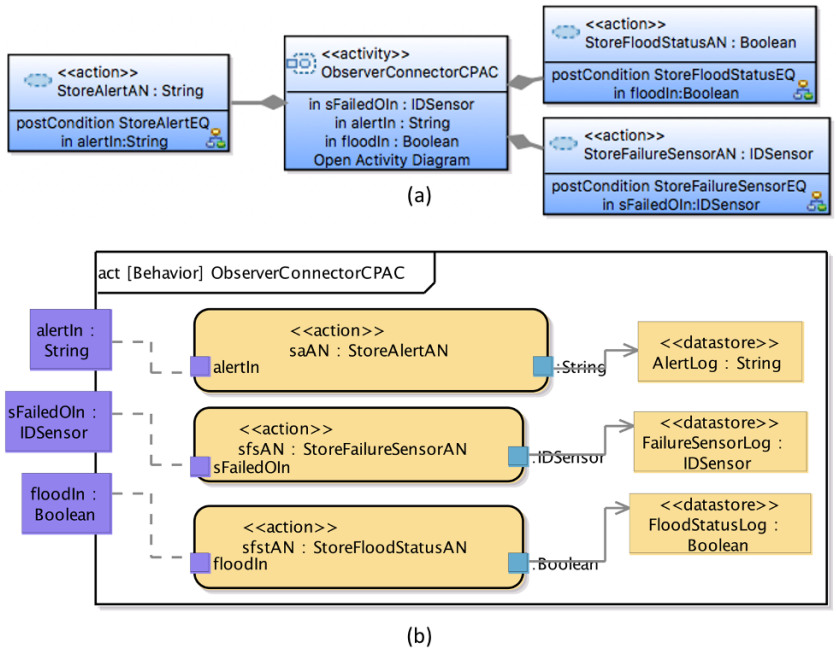


Figure 68 – *ObserverConnectorCPAC* behavioral BDD and Activity Diagram

EnvCP. Figure 69 display a bdd excerpt with the definition of activity and actions of the *EnvCP* component through the *EnvCPAC* activity that is composed of five different actions: *NewCycleAN*, *RequestingNewMeasuresAN*, *isFloodingOccuringAN*, *SendingSensorFailureAlertAN*,and *ValueConfirmationSendAN*. Figure 70 exhibits the activity diagram of *EnvCPAC* with its action and pins.

- NewCycleAN: generates the cycles for system synchronization and informs *GatewayCP* via *tickOut*;

- RequestingNewMeasuresAN: after beginning a new cycle, the *EnvCP* sends a signal requesting a new measurement to the *SensorEnvCP* via *valueReadyOut*;

- isFloodingOccuringAN: receive from *GatewayCP* data informing about flooding during the current cycle. The information is forwarded to *ObserverConnectorCP* via *floodOut*;

- SendingSensorFailureAlertAN: receives a sensor fail identifier from *SensorEnvCP* via *sensorFailedIn*, and forwards this information to *ObserverConnectorCP*, via *seFailedOut*;
- ValueConfirmationSendAN: receives from *SensorEnvCP* a signal ratifying the sending of a new measurement from the physical sensor via *valueSentIn* and stores it in the *valueSentIn(Boolean)* buffer

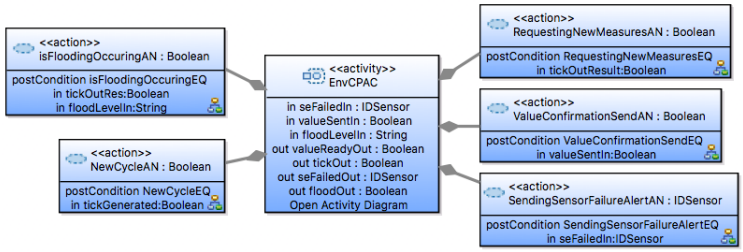


Figure 69 – *EnvCPAC* behavioral BDD

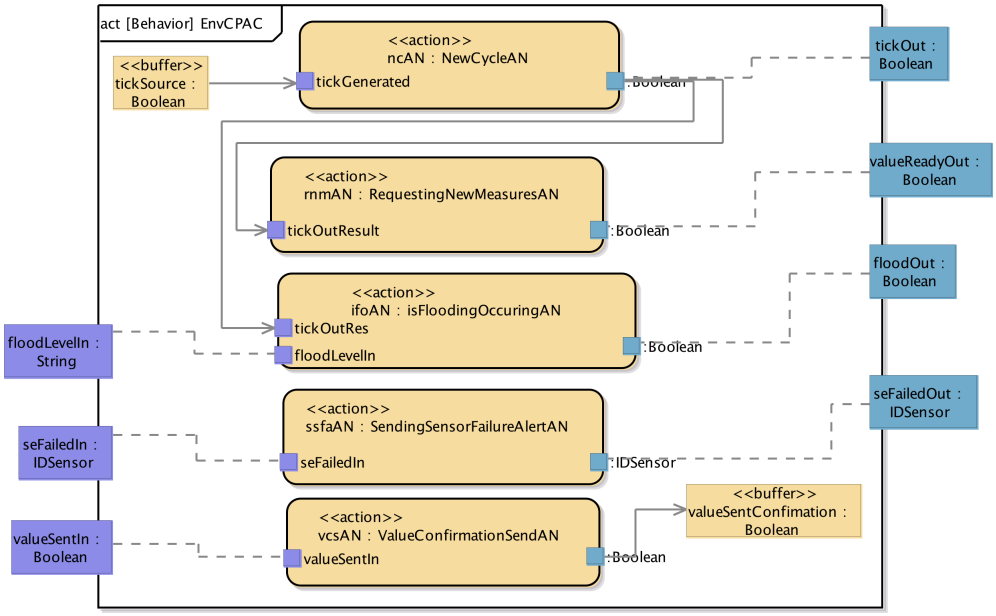


Figure 70 – *EnvCPAC* Activity Diagram BDD

APPENDIX B – Grammars

This appendix describes the concrete textual syntax of the SysADL and π-ADL languages by using the Extended Backus-Naur Form (EBNF) meta-language, a notation for formally describing the context-free grammar of a language, in an adapted version for *xtext*. Section A.1 presents the notation elements used hereinafter; Section A.2 shows the production rules forming the SysADL grammar, whereas Section A.3 gives the production rules forming the π-ADL grammar.

B.1 Grammar notation

The EBNF/Xtext meta-language consists of terminal symbols, a sequence of characters forming an irreducible element of the language, and non-terminal production rules dictating how a syntactic element can be rightfully formed in terms of terminal symbols. Syntactic elements have names used in production rules, and they are distinguished from names and reserved words in the language. Furthermore, the EBNF/Xtext meta-language uses a set of meta-symbols summarized in Table 9

Table 9 – EBNF/Xtext meta-symbols

Meta-symbol	**Usage**
Colon (:)	Definition of production rule: A:B is read as A is defined as B
Pipe symbol (|)	Alternative choice between elements in production rule
Asterisk character (*)	Multiple occurences of element in production rule
Plus character (+)	At least occurence of element in production rule
Question mark character (?)	At most one occurrence of element in production rule
Equal character (=)	Represents a straightforward assignment, and is used for features which take only one element.
Plus equal character (+=)	Expects a multi-valued feature and adds the value on the right hand side to that feature, which is a list feature
Question mark equal character (?=)	Expects a feature of type `EBoolean` and sets it to true if the right hand side was consumed, independently from the concrete value of the right hand side
Brackets ([element])	Optional occurences of element in production rule
Single quotes ('text')	Represents keywords, that is a kind of terminal rule literals.
Parentheses	Element grouping or precedence

B.2 SysADL grammar

```
grammar org.sysadl.SysADL with org.eclipse.xtext.common.Terminals

grammar org.sysadl.SysADL with org.eclipse.xtext.common.Terminals

import "http://org.sysadl"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Model returns Model:
  'Model' name=ID ';'
    packages+=Package
  ;

Package returns Package:
  'package' name=QualifiedName '{'
    ('import' imports+= QualifiedName ';')*
    (definitions+=ElementDef | definitions+=ArchitectureDef)*
  '}';

ElementDef returns ElementDef:
  DataTypeDef | ValueTypeDef | Enumeration | ComponentDef |
   ConnectorDef | CompositePortDef | SimplePortDef;

DataTypeDef returns DataTypeDef:
  'datatype'
  name=ID
  '{'
    ('attributes' ':' attributes+=TypeUse+)?
  '}';

ValueTypeDef returns ValueTypeDef:
  'value' 'type'
  name=ID
  '{'
    ('unit' '=' unit=[UnitDef])?
    ('dimension' '=' dimension=[DimensionDef])?
  '}';

Enumeration returns Enumeration:
  'enum'
  name=ID
```

```
  ('{'
    literals+=EnumLiteralValue (',' literals+=EnumLiteralValue)*
  '}');

ArchitectureDef returns ArchitectureDef:
  'architecture' 'def' name=ID
  '{'
    'ports' ':' ports+=PortUse*
    (composite=Configuration)
  '}'
;

ComponentDef returns ComponentDef:
  (isBoundary?='boundary')? 'component' 'def' name=ID
  '{'
    ('ports' ':' ports+=PortUse*)?
    (composite=Configuration)?
  '}'
  ;

ConnectorDef returns ConnectorDef:
  'connector' 'def' name=ID
  '{'
    ('participants' ':' ports+=PortUse_Reverse*)
    ('flows' ':' flows+=Flow*)?
  '}';

PortDef returns PortDef:
  CompositePortDef | SimplePortDef;

CompositePortDef returns CompositePortDef:
  'port' 'def' name=ID
  '{'
    'ports' ':' ports+=PortUse*
  '}';

SimplePortDef returns SimplePortDef:
  'port' 'def' name=ID
  '{'
    'flow' flowProperties=FlowProperty flowType=[TypeDef]
  '}';
```

```
TypeUse returns TypeUse:
  name=ID ':' definition=[TypeDef]
    ('{' '}' | ';' )
  ;

TypeDef returns TypeDef:
  DataTypeDef | ValueTypeDef | Enumeration;

PortUse returns PortUse:
  name=ID ':' definition=[PortDef]
  ('{' '}' | ';');

PortUse_Reverse returns PortUse:
  '~' name=ID ':' definition=[PortDef]
  ('{' '}' | ';');

Flow returns Flow:
  type=[TypeDef] 'from' source=[PortUse] 'to' destination=[PortUse]
  ;

enum FlowProperty returns FlowProperty:
  in = 'in' | out = 'out' | inout='inout';

Configuration returns Configuration:
  'configuration'
  '{'
    ('components' ':' components+=ComponentUse*)?
    ('connectors' ':' connectors+=ConnectorUse*)?
    ('delegations' ':' delegations+=Delegation*)?
  '}';

ConnectorUse returns ConnectorUse:
  name=ID ':' definition=[QualifiedName] 'bindings'
   bindings=[ConnectorBinding] ';';


ConnectorBinding returns ConnectorBinding:
  source=[QualifiedName] '=' destination=[QualifiedName];

ComponentUse returns ComponentUse:
  name=ID ':' definition=[QualifiedName]
  '{'
```

```
    ('using' 'ports' ':' ports+=PortUse*)?
  '}';

Delegation returns Delegation:
  source=[PortUse] 'to' destination=[PortUse]
;

EnumLiteralValue returns EnumLiteralValue:
  name=ID;

DimensionDef returns DimensionDef:
  'dimension'
  name=ID
  ('{' '}')?;

UnitDef returns UnitDef:
  'unit'
  name=ID
  ('{'
    ('dimension' '=' dimension=[DimensionDef])?
  '}')?;

QualifiedName:
  DotQualifiedName
  | ID
;

DotQualifiedName:
  ID ('.' ID)+
;

ActivityDef returns ActivityDef:
  'activity' 'def' name=ID ('(' inParameters+=Pin (','
   inParameters+=Pin)* ')')*
    (':' '('outParameters+=Pin (',' outParameters+=Pin)* ')')'{'
      ActivityBody
  '}';

ActivityBody returns ActivityBody:
  'body'
  '{'
    ('actions' ':' actions+=ActionUse*)?
```

```
    flows+=ActivityRelation*
    dataObjects+=DataObject*
  '}';

Pin returns Pin:
  {Pin}
  name=ID ':' (isFlow?='flow')? definition=[QualifiedName]
  ;

ActionUse returns ActionUse:
  name=ID ':' definition=[QualifiedName] ('{'
    ('using' 'pins' ':' (pinIn+=Pin ';')* )
  ';');

ActionDef returns ActionDef:
  {ActionDef}
  'action' 'def'
  name=ID '(' inParameters+=Pin ( "," inParameters+=Pin)* ')' ':'
   returnType=[QualifiedName]
  '{'
    'constraint' ':' constraints+=ConstraintUse*
    delegations+=ActivityDelegation*
  '}';


ActivityRelation returns ActivityRelation:
  ActivityDelegation | ActivityFlow;

ActivityDelegation returns ActivityDelegation:
  'delegate' source=[QualifiedName] 'to' (target=[QualifiedName]

ActivityFlow returns ActivityFlow:
  'flow'
  'from' source=[QualifiedName]
  'to' (target=[QualifiedName]
;

ConstraintUse returns ConstraintUse:
  kind=ConstraintKind (definition=[QualifiedName])
  ;

ActivityFlowable returns ActivityFlowable:
```

```
  ActionUse | DataStore | DataBuffer;

enum ConstraintKind returns ConstraintKind:
  preCondition = 'pre-condition' | postCondition = 'post-condition'
   | invariant = 'invariant';


ConstraintDef returns ConstraintDef:
  {ConstraintDef}
  'constraint'
  name=ID ('(' ( inParameters+=Pin (',' inParameters+=Pin)* )? ')')
   (':' '(' outParameters+=Pin (',' outParameters+=Pin)* ')')?
  '{'
    ('equation' '=' equation=Expression)?
  '}';

DataStore returns DataStore:
  'datastore'
  name=ID ':' type=[QualifiedName]
  ('{' '}')?;

DataBuffer returns DataBuffer:
  'databuffer'
  name=ID ':' type=[QualifiedName]
  '{' '}';
```

B.3 π-ADL grammar

```
grammar fr.irisa.archware.PiADL with
    org.eclipse.xtext.common.Terminals

generate piADL "http://www.irisa.fr/archware/PiADL"

ArchitectureDescription: //ok
  archElements+=ArchitecturalElement*
  archs+=Architecture+
  cbehavior=BehaviorDeclaration
;

ArchitecturalElement:
  Component | Connector
```

```
;

Connector://ok
  'connector' name=ID 'is'
  'abstraction()' '{'
    typeDecl+=TypeDeclaration*
    connections+=ConnectionDeclaration*
    protDecl=ProtocolDeclaration?
    behavior=BehaviorDeclaration
  '}'
;

Component://ok
  'component' name=ID 'is'
  'abstraction()' '{'
    typeDecl+=TypeDeclaration*
    connections+=ConnectionDeclaration*
    protDecl=ProtocolDeclaration?
    behavior=BehaviorDeclaration
  '}'
;

Architecture://ok
  'architecture' name=ID 'is'
  'abstraction()' '{'
    'behavior' 'is' '{'
      compose=Composition
    '}'
  '}'
;

TypeDeclaration:
  'type' name=ID 'is' type=ValueType
;

ConnectionDeclaration:
  'connection' name=ID 'is' direction=ConnectionMode '('
   type=ValueType ')'
;

ProtocolDeclaration:
  'protocol' 'is' '{'
```

```
    '('
    protocol+=ProtocolAction*
    ')' '*'
  '}'
;

ProtocolAction:
  '('* 'via' connectionName=ID action=Action type=ValueType ')'*
;


enum Action:
  send='send' | receive='receive'
;


enum ConnectionMode:
  in='in' | out='out'
;

BehaviorDeclaration:
  {BehaviorDeclaration}
  'behavior' 'is' '{'
    body+=BehaviorClause*
  '}'
;

BehaviorClause:
  ConnectionDeclaration | Composition

;

Composition:
  'compose' '{'
    clause+=(BehaviorClause | ElementInstantiation)+
    (('and' | 'and') clause+=(BehaviorClause |
   ElementInstantiation)+)+
  ('}' | '}') uc=UnificationClause?
;

UnificationClause:
  {UnificationClause}
```

```
  'where' '{'
    (unifications+=Unification)*
  '}'
;

ElementInstantiation:
  elementName=ID 'is' elementType=ID
  '(' (parameterName+=ID (',' parameterName+=ID)*)? ')'
;

Unification:
  fromc=ConnectionAccess 'unifies' toc=ConnectionAccess
;

ConnectionAccess:
  elementName=(ID | 'self') '::' connectionName=ID
;

Unobservable:
  'unobservable'
;

FunctionDeclaration:
  functionName=ID 'is' 'function' '('
  (parameters+=Parameter (',' parameters+=Parameter)*)? ')'
  (':' returnType=ValueType)? '{'
    block+=BehaviorClause*
  '}'
;

Parameter:
  name=ID ':' type=ValueType
;

ValueType:
  BaseType | ConstructedType | {TypeRef} idt=ID
;

BaseType:
  NaturalType | IntegerType | RealType | BooleanType | StringType |
   AnyType
;
```

```
NaturalType:
  {NaturalType} type='Natural'
;

IntegerType:
  {IntegerType} type='Integer'
;

RealType:
  {RealType} type='Real'
;

BooleanType:
  {BooleanType} type='Boolean'
;

StringType:
  {StringType} type='String'
;

AnyType:
  {AnyType} type='Any'
;

ConstructedType:
  View
;

View:
  'view''[' labt+=LabeledType (',' labt+=LabeledType)* ']'
;

LabeledType:
  label=ID ':' type=ValueType
;
```

APPENDIX C – Denotational Semantics for SysADL using π-ADL

This appendix presents the necessary elements to define the denotational semantics of SysADL in π-ADL. Section C.1 presents the normal form defined for input file with an architectural description in SysADL. Section C.2 describes the notation used in the definition of denotation semantics, which is finally presented in section C.3.

C.1 Normal Form

To simplify the definition and understanding of the denotational semantics of SysADL in π-ADL, we propose a normal form to place the elements in a textual architectural description in SysADL in order more favorable to the expression of the semantics. This section describes the normal form for architectural descriptions in SysADL.

C.1.1 Architectural elements order

The architectural elements are all enclosed in a unique package ordered as follows:

1. Basics ValueType
2. Created ValueType
3. Enumerations
4. DataTypes: Datatypes are ordered by dependency. A datatype appears only if all types of its attributes have seemed before.
5. PortDef: we consider only simple port because a sequence of simple ports can replace any composite port.
6. ConnectorDef: Similarly to ports, we consider only simple connectors because a sequence of simple connectors can replace any composite connector.
7. ComponentDef: Components are ordered by dependency like DataTypes. A component can only use connectors and other components that were previously defined. Therefore, in the proposed normal form, the composite component only appears after its used connectors and components are declared.
8. ArchitectureDef

C.1.2 Primitives ValueTypes

These types are considered primitive in SysADL. They are the first and appear in the same order generated by the SysADL Studio tool:

1. Int
2. Boolean
3. String
4. Real
5. Void

C.1.3 Created ValueType

Value Types no primitives created without subtypes, dimension, and unit.

C.1.4 Enumerations

There are no Enum changes. Anyway, the initial version of the transformation is not considered the value list of enums.

C.1.5 DataTypes

In the Normal Form, the datatype attributes are ordered according to their types:

1. Basics ValueType
2. Created ValueType
3. Enumerations
4. DataTypes: If the type X of an attribute depends on a type Y of another attribute on the same datatype, then the attribute of type Y appears before the attribute of type X.

C.1.6 PortDef

There is no change in the structure of PortDef. However, it is essential to note that this version of the study does not consider composite ports and that this does not reduce our scope since composite ports can be redefined as a succession of simple ports.

C.1.7 ConnectorDef

Similar to the definition of PortDef, there are no changes in the structure of connectors. This version of the study does not consider composite connectors also.

C.1.8 ComponentDef

In the Normal Form, the ComponentDef are ordered according to the below list:

1. Boundary Components
2. Simple Components
3. Composite Components: If Component X depends on Component Y, component Y is defined before component Y.

C.1.9 ArchitectureDef

There is no change in the ArchitectureDef structure. But, they must appear after all the ComponentDef.

C.1.10 Behavior viewpoint definitions

The activity must have the same name as the associated component and describe the behavior, adding the letters "AC" to the end.

C.1.11 Preprocessing

The transformation defined in our study should happen in its phases:

1. SysADL Model transformation in a SysADL Normal Model (preprocessing)
2. Model in Normal Form SysADL transformation to Model π-ADL.

The preprocessing step serves to adapt the SysADL model to the Normal Form now defined to help transform SysADL into π-ADL.

C.2 Notation

- Semantic Functions are defined with double brackets $[\![SysADL!C]\!]$ and map an element of a syntactic/semantic Class C to a corresponding syntactic/semantic element in π-ADL.
- Semantic functions of the form $[\![es]\!]^*$ represent a mapping of the correspondent semantic function $[\![\ ]\!]$ to all elements of the syntactic element sequence es.

- The expression $env[n]$ signifies a list in the *nth element* position defined in the *env* list defined in section C.3.2.

C.3 Formal Semantic

C.3.1 Function Signature

- $Env : Package \rightarrow \langle e_1, ..., e_{10} \rangle$, where

  $e1$: $seq(SysADL!ValueTypeDef)$

  $e2$: $seq(SysADL!Enumerate)$

  $e3$: $seq(SysADL!DataTypeDef)$

  $e4$: $seq(SysADL!TypeDef)$

  $e5$: $seq(SysADL!SimplePortDef)$

  $e6$: $seq(SysADL!ConnectorDef)$

  $e7$: $seq(SysADL!ComponentDef)$

  $e8$: $seq(SysADL!ActivityDef)$

  $e9$: $seq(SysADL!ActionDef)$

  $e10$: $seq(SysADL!ArchitectureDef)$

  $e11$: $seq(SysADL!ComponentUse)$

- $ValuetypesP : Package \rightarrow seq(SysADL!ValueTypeDef)$

- $Valuetypes : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArquitectureDef) \rightarrow seq(SysADL!ValueTypeDef)$

- $Valuetype : SysADL!ElementDef \rightarrow SysADL!ValueTypeDef$

- $EnumtypesP : Package \rightarrow seq(SysADL!Enumeration)$

- $Enumtypes : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!Enumeration)$

- $Enumtype : SysADL!ElementDef \rightarrow SysADL!Enumeration$

- $DatatypesP : Package \rightarrow seq(SysADL!DataTypeDef)$

- $Datatypes : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!DataTypeDef)$

- $Datatype : SysADL!ElementDef \rightarrow SysADL!DataTypeDef$

- $TypedefsP : Package \rightarrow seq(SysADL!TypeDef)$
- $Typedefs : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!TypeDef)$
- $Typedef : SysADL!ElementDef \rightarrow SysADL!TypeDef$
- $SimplePortsP : Package \rightarrow seq(SysADL!SimplePortDef)$
- $SimplePorts : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!SimplePortdDef)$
- $SimplePort : SysADL!ElementDef \rightarrow SysADL!SimplePortDef$
- $ConnectorsP : Package \rightarrow seq(SysADL!ConnectorDef)$
- $Connectors : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!ConnectorDef)$
- $Connector : SysADL!ElementDef \rightarrow SysADL!ConnectorDef$
- $ComponentsP : Package \rightarrow seq(SysADL!ComponentDef)$
- $Components : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!ComponentDef)$
- $Component : SysADL!ElementDef \rightarrow SysADL!ComponentDef$
- $ActivitiesP : Package \rightarrow seq(SysADL!ActivityDef)$
- $Activities : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!ActivityDef)$
- $Activitie : SysADL!ElementDef \rightarrow SysADL!ActivityDef$
- $ActionsP : Package \rightarrow seq(SysADL!ActionDef)$
- $Actions : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!ActionDef)$
- $Action : SysADL!ElementDef \rightarrow SysADL!ActionDef$
- $ArchsP : Package \rightarrow seq(SysADL!ArchitectureDef)$
- $Archs : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!ArchitectureDef)$
- $Arch : SysADL!ArchitectureDef \rightarrow seq(SysADL!ArchitectureDef)$

- $ComponentsUseP : Package \rightarrow seq(SysADL!ComponentUse)$
- $ComponentsUse : seq(SysADL!ElementDef) \bigcup seq(SysADL!ArchitectureDef) \rightarrow seq(SysADL!ComponentUse)$
- $ComponentUseF : SysADL!ElementDef \rightarrow seq(SysADL!ComponentUse)$
- $ComponentsUseExtractor : SysADL!Configuration \rightarrow seq(SysADL!ComponentUse)$
- $[\![ElementS]\!] : SysADL!ElementS \rightarrow PIADL!ElementP$
  - $[\![SysADL!Model]\!] : SysADL!Model \rightarrow PIADL!ArchitectureDescription$
  - $[\![SysADL!ConnectorDef]\!] : SysADL!ConnectorDef \rightarrow PIADL!Connector$
  - $[\![SysADL!ComponentDef]\!] : SysADL!ComponentDef \rightarrow PIADL!Component$
  - $[\![SysADL!ArchitectureDef]\!] : SysADL!ArchitectureDef \rightarrow PIADL!Architecture$
  - $[\![SysADL!PortUse\_Reverse]\!] : SysADL!PortUse\_Reverse \rightarrow PIADL!ConnectionDeclaration$
  - $[\![SysADL!PortUse]\!] : SysADL!PortUse \rightarrow PIADL!ConnectionDeclaration$
  - $[\![SysADL!TypeDef]\!] : SysADL!TypeDef \rightarrow PIADL!TypeDeclaration$
  - $[\![SysADL!TypeUse]\!] : SysADL!TypeUse \rightarrow PIADL!LabeledType$
  - $[\![SysADL!QualifiedName]\!] : SysADL!QualifiedName \rightarrow String$
- $getNameTypeDefFlows : seq(SysADL!Flow) \rightarrow SysADL!QualifiedName$
- $getLstNameTDefPUse : seq(SysADL!PortUse) \rightarrow seq(SysADL!QualifiedName)$
- $getTypeDef : SysADL!QualifiedName \rightarrow SysADL!TypeDef$
- $getDDependency : SysADL!TypeDef \rightarrow seq(SysADL!QualifiedName)$
- $getTypeDefDep : seq(SysADL!QualifiedName) \times seq(SysADL!TypeDef) \rightarrow seq(SysADL!TypeDef)$
- $makeInjSeq : seq(SysADL!QualifiedName) \rightarrow seq(SysADL!QualifiedName)$
- $distCat : seq(seq(SysADL!QualifiedName)) \rightarrow seq(SysADL!QualifiedName)$
- $getAttTypes : seq(SysADL!TypeUse) \rightarrow seq(SysADL!QualifiedName)$

- $removeQName : SysADL!QualifiedName \times seq(QualifiedName) \rightarrow seq(SysADL!QualifiedName)$
- $getDependencies : SysADL!QualifiedName \times seq(SysADL!TypeDef) \rightarrow seq(SysADL!TypeDef)$
- $getDependLst : seq(SysADL!QualifiedName) \times seq(SysADL!TypeDef) \rightarrow seq(SysADL!TypeDef)$
- $setProtocolDeclaration : SysADL!PortUse\_Reverse \rightarrow PIADL!ProtocolAction$
- $setProtocolDeclarationPU : SysADL!PortUse \rightarrow PIADL!ProtocolAction$
- $setCompLstElementInstantiation : seq(SysADL!ComponentUse) \rightarrow seq(PIADL!ElementInstantiation)$
- $setConLstElementInstantiation : seq(SysADL!ConnectorUse) \rightarrow seq(PIADL!ElementInstantiation)$
- $getPortDef : SysADL!QualifiedName \rightarrow SysADL!SimplePortDef$
- $getDirection : SysADL!SimplePortDef \rightarrow String$
- $getAction : SysADL!SimplePortDef \rightarrow String$
- $setConUnification : seq(SysADL!ConnectorUse) \rightarrow seq(PIADL!Unifications)$
- $setConnectionAccesFrom : SysADL!ConnectorBinding \rightarrow PIADL!ConnectionAccess$
- $setConnectionAccesTo : SysADL!ConnectorBinding \rightarrow PIADL!ConnectionAccess$
- $setCAccessFromCon : SysADL!ConnectorDef \rightarrow PIADL!ConnectionAccess$
- $setCAccessToCon : SysADL!ConnectorDef \rightarrow PIADL!ConnectionAccess$
- $getConnectorDef : SysADL!QualifiedName \times seq(SysADL!ConnectorDef) \rightarrow SysADL!ConnectorDef$
- $getComponentUsePU : SysADL!QualifiedName \times seq(SysADL!ComponentUse) \rightarrow SysADL!QualifiedName$
- $getPortUseReverseOut : SysADL!ConnectorUse \rightarrow SysADL!QualifiedName$
- $getPortUseReverseIn : SysADL!ConnectorUse \rightarrow SysADL!QualifiedName$
- $setDelUnification : SysADL!Delegation \rightarrow PIADL!Unifications$

- $setBehavior : SysADL!ActivityDef \times seq(SysADL!ActionDef) \rightarrow seq(PIADL!FunctionDeclaration)$
- $setFunctions : seq(SysADL!ActionUse) \times seq(SysADL!ActionDef) \rightarrow seq(PIADL!FunctionDeclaration)$
- $getActionDef : SysADL!ActionUse \times seq(SysADL!ActionDef) \rightarrow SysADL!ActionDef$
- $setFunction : SysADL!ActionUse \times SysADL!ActionDef \rightarrow PIADL!FunctionDeclaration$
- $setInParam : seq(SysADL!Pin) \rightarrow seq(PIADL!Parameters)$
- $getActivity : SysADL!QualifiedName \rightarrow SysADL!ActivityDef$

C.3.2 Enviroment

$\underline{\text{Env}}(Package) =$
$\langle \underline{\text{ValuetypesP}}(Package),$
$\underline{\text{EnumtypesP}}(Package),$
$\underline{\text{DatatypesP}}(Package),$
$\underline{\text{TypedefsP}}(Package),$
$\underline{\text{SimplePortsP}}(Package),$
$\underline{\text{ConnectorsP}}(Package),$
$\underline{\text{ComponentsP}}(Package),$
$\underline{\text{ActivitiesP}}(Package),$
$\underline{\text{ActionsP}}(Package),$
$\underline{\text{ArchsP}}(Package),$
$\text{ComponentsUseP}(Package)\rangle$

```
//ValueTypes
ValuetypesP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Valuetypes((ElementDef |
    ArchitectureDef)*)
//-------------------------------------------------------
Valuetypes(ϵ) = ⟨⟩
Valuetypes(ElementDef (ElementDef | ArchitectureDef)*) =
    Valuetype(ElementDef) ⌢Valuetypes((ElementDef | ArchitectureDef)*)
Valuetypes(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Valuetypes((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
Valuetype(ValueTypeDef) = ⟨ValueTypeDef⟩
Valuetype(Enumeration | DataTypeDef | SimplePortDef | ConnectorDef
    |ComponentDef | ActivityDef | ActionDef) = ⟨⟩

//EnumTypes
```

```
EnumtypesP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Enumtypes((ElementDef |
    ArchitectureDef)*)
//-------------------------------------------------------
Enumtypes(ϵ) = ⟨⟩
Enumtypes(ElementDef (ElementDef | ArchitectureDef)*) =
    Enumtype(ElementDef) ⌢Enumtypes((ElementDef | ArchitectureDef)*)
Enumtypes(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Enumtypes((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
Enumtype(Enumeration) = ⟨Enumeration⟩
Enumtype(ValueTypeDef | DataTypeDef | SimplePortDef | ConnectorDef |
    ComponentDef | ActivityDef | ActionDef) = ⟨⟩

//DataTypes
DatatypesP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Datatypes((ElementDef |
    ArchitectureDef)*)
//-------------------------------------------------------
Datatypes(ϵ) = ⟨⟩
Datatypes(ElementDef (ElementDef | ArchitectureDef)*) =
    Datatype(ElementDef) ⌢Datatypes((ElementDef | ArchitectureDef)*)
Datatypes(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Datatypes((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
Datatype(DataTypeDef) = ⟨DataTypeDef⟩
Datatype(ValueTypeDef | Enumeration | SimplePortDef | ConnectorDef |
    ComponentDef | ActivityDef | ActionDef) = ⟨⟩

//TypeDefs
TypedefsP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Typedefs((ElementDef |
    ArchitectureDef)*)
//-------------------------------------------------------
Typedefs(ϵ) = ⟨⟩
Typedefs(ElementDef (ElementDef | ArchitectureDef)*) =
    Typedef(ElementDef) ⌢Typedefs((ElementDef | ArchitectureDef)*)
Typedefs(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Typedefs((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
Typedef(ValueType) = ⟨ValueType⟩
Typedef(Enumeration) = ⟨Enumeration⟩
Typedef(DataTypeDef) = ⟨DataTypeDef⟩
Typedefs(SimplePortDef | ConnectorDef | ComponentDef | ActivityDef |
    ActionDef) = ⟨⟩

//SimplePorts
```

```
SimplePortsP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = SimplePorts((ElementDef |
    ArchitectureDef)*)
//-------------------------------------------------------
SimplePorts(ϵ) = ⟨⟩
SimplePorts(ElementDef (ElementDef | ArchitectureDef)*) =
    SimplePort(ElementDef) ⌢SimplePorts((ElementDef | ArchitectureDef)*)
SimplePorts(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    SimplePorts((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
SimplePort(SimplePortDef) = ⟨SimplePortDef⟩
SimplePort(ValueTypeDef | Enumeration | DataTypeDef | ConnectorDef |
    ComponentDef | ActivityDef | ActionDef) = ⟨⟩

//Connectors
ConnectorsP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Connectors((ElementDef |
    ArchitectureDef)*)
//-------------------------------------------------------
Connectors(ϵ) = ⟨⟩
Connectors(ElementDef (ElementDef | ArchitectureDef)*) =
    Connector(ElementDef) ⌢Connectors((ElementDef | ArchitectureDef)*)
Connectors(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Connectors((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
Connector(ConnectorDef) = ⟨ConnectorDef⟩
Connector(ValueTypeDef | Enumeration | DataTypeDef | SimplePortDef |
    ComponentDef | ActivityDef | ActionDef) = ⟨⟩

//Components
ComponentsP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Components((ElementDef |
    ArchitectureDef)*)
//-------------------------------------------------------
Components(ϵ) = ⟨⟩
Components(ElementDef (ElementDef | ArchitectureDef)*) =
    Component(ElementDef) ⌢Components((ElementDef | ArchitectureDef)*)
Components(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Components((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
Component(ComponentDef) = ⟨ComponentDef⟩
Component(ValueTypeDef | Enumeration | DataTypeDef | SimplePortDef |
    ConnectorDef | ActivityDef | ActionDef) = ⟨⟩

//ActivityDef
ActivitiesP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Activities((ElementDef |
```

```
    ArchitectureDef)*)
//-----------------------------------------------------------
Activities(ϵ) = ⟨⟩
Activities(ElementDef (ElementDef | ArchitectureDef)*) =
    Activity(ElementDef) ⌢Activities((ElementDef | ArchitectureDef)*)
Activities(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Activities((ElementDef | ArchitectureDef)*)
//-----------------------------------------------------------
Activity(ActivityDef) = ⟨ActivityDef⟩
Activity(ValueTypeDef | Enumeration | DataTypeDef | SimplePortDef |
    ConnectorDef | ComponentDef | ActionDef) = ⟨⟩

//ActionDef
ActionsP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Actions((ElementDef |
    ArchitectureDef)*)
//-----------------------------------------------------------
Actions(ϵ) = ⟨⟩
Actions(ElementDef (ElementDef | ArchitectureDef)*) =
    Action(ElementDef) ⌢Actions((ElementDef | ArchitectureDef)*)
Actions(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Actions((ElementDef | ArchitectureDef)*)
//-----------------------------------------------------------
Action(ActionDef) = ⟨ActionDef⟩
Action(ValueTypeDef | Enumeration | DataTypeDef | SimplePortDef |
    ConnectorDef | ComponentDef | ActivityDef) = ⟨⟩

//Archs
ArchsP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = Archs((ElementDef |
    ArchitectureDef)*)
//-----------------------------------------------------------
Archs(ϵ) = ⟨⟩
Archs(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    Arch{ArchitectureDef} ⌢Archs((ElementDef | ArchitectureDef)*)
Archs(ElementDef (ElementDef | ArchitectureDef)*) =
    Archs((ElementDef | ArchitectureDef)*)
//-----------------------------------------------------------
Arch(ArchitectureDef) = ⟨ArchitectureDef⟩

//ComponentsUse
ComponentsUseP('package' QualifiedName '{'('import' QualifiedName ';')*
    (ElementDef | ArchitectureDef)*'}') = ComponentsUse((ElementDef |
    ArchitectureDef)*)
//-----------------------------------------------------------
ComponentsUse(ϵ) = ⟨⟩
ComponentsUse(ElementDef (ElementDef | ArchitectureDef)*) =
```

```
    ComponentUseF(ElementDef) ⌢ComponentsUse((ElementDef |
    ArchitectureDef)*)
ComponentsUse(ArchitectureDef (ElementDef | ArchitectureDef)*) =
    ComponentsUse((ElementDef | ArchitectureDef)*)
//-------------------------------------------------------
ComponentUseF(ValueTypeDef | Enumeration | DataTypeDef | SimplePortDef |
    ConnectorDef | ActivityDef | ActionDef) = ⟨⟩
ComponentUseF('boundary component def' QualifiedName '{'PortUse*'}') = ⟨ \;
    ⟩
ComponentUseF('component def' QualifiedName '{'PortUse*'}') = ⟨ \; ⟩
ComponentUseF('component def' QualifiedName '{'PortUse* Configuration'}')
    = ComponentsUseExtractor(Configuration)
//-------------------------------------------------------
ComponentsUseExtractor('configuration { }') = ⟨⟩
ComponentsUseExtractor('configuration {components :' ComponentUse* '}') =
    ⟨ComponentUse*⟩
ComponentsUseExtractor('configuration {connectors :' ConnectorUse* '}') = ⟨⟩
ComponentsUseExtractor('configuration {delegations :' Delegation* '}') = ⟨⟩
ComponentsUseExtractor('configuration {components :' ComponentUse*
    'connetors :' ConnectorUse* '}') = ⟨ComponentUse*⟩
ComponentsUseExtractor('configuration {components :' ComponentUse*
    'delegations :' Delegation* '}') = ⟨ComponentUse*⟩
ComponentsUseExtractor('configuration {connectors :' ConnectorUse*
    'delegations :' Delegation* '}') = ⟨⟩
ComponentsUseExtractor('configuration {components :' ComponentUse*
    'connectors :' ConnectorUse* 'delegations :' Delegation* '}') =
    ⟨ComponentUse*⟩
```

C.3.3 Structural Elements

```
//Model and Package
⟦'Model' ID ';' Package⟧ ⟨⟩ =
    let env = Env(Package)
    in
        ⟦env[6]⟧* env
        ⟦env[7]⟧* env
        ⟦env[10]⟧* env
        'behavior is {unobservable}'
    end

//SYSADL!ConnectorDef to PIADL!Connector
⟦'connector def' ID '{' ('participants :' PortUse_Reverse*)
('flows :' Flow*)'}'⟧ env =
```

```
    'connector' ID 'is abstraction() {'
        ⟦
   getTypeDefDep(getDependencies(getNameTypeDefFlows(Flow*),
   env[4]), env[4])⟧* env
        ⟦PortUse_Reverse⟧*env
        'protocol is {
   ('setProtocolDeclaration(PortUse_Reverse*)')*}'
        'behavior is {unobservable}'
    '}'

//SYSADL!ComponentDef to PIADL!Component
⟦'boundary component def' ID '{' ('ports :' PortUse*)'}'⟧ env =
    'component boundary'ID'is abstraction() {'
        ⟦
   getTypeDefDep(getDependLst(getLstNameTDefPUse(PortUse*),
   env[4]), env[4])⟧* env
        ⟦PortUse⟧*env
        'protocol is { ('setProtocolDeclarationPU(PortUse*)')*}'
        'behavior is {unobservable}'
    '}'

⟦'component def' ID '{' ('ports :' PortUse*)'}'⟧ env =
    'component' ID 'is abstraction() {'
        ⟦
   getTypeDefDep(getDependLst(getLstNameTDefPUse(PortUse*),
   env[4]), env[4])⟧* env
        ⟦PortUse⟧*env
        'protocol is { ('setProtocolDeclarationPU(PortUse*)')*}'
        'behavior is {'setBehavior(getActivity(ID), env[8])'}'
    '}'

⟦'component def'ID'{' ('ports :' PortUse*) Configuration'}'⟧ env =
    'component' ID 'is abstraction() {'
        ⟦
   getTypeDefDep(getDependLst(getLstNameTDefPUse(PortUse*),
   env[4]), env[4])⟧* env
        ⟦PortUse⟧*env
        'protocol is { ('setProtocolDeclarationPU(PortUse*)')*}'
        'behavior is {' ⟦Configuration⟧'}'
```

```
    '}'

//SYSADL!ArchitectureDef to PIADL!Architecture
[['architecture def'ID'{' ('ports :' PortUse*) Configuration'}']]
   env =
    'architecture' ID 'is abstraction() {'
        'behavior is {' [[Configuration]]'}'
    '}'

//SYSADL!TypeDef to PIADL!TypeDeclaration
[[ϵ]] = ϵ
[['value type' ID '{ }']]env = 'type' ID 'is Any'
[['enum' ID '{ }']]env = 'type' ID 'is Any'
[['datatype' ID '{'('attributes :' TypeUse*)?'}']]env =
    'type' ID 'is view ['[[TypeUse]]*env ']'

//SYSADL!TypeUse to PIADL!LabeledType}
[[ID':'QualifiedName';']]env = ID':' [[QualifiedName]]

//SYSADL!PortUse_Reverse to PIADL!Connection Declaration}
[['~'ID':' QualifiedName'{''}']]env = 'connection' ID 'is'
   getDirection(getPortDef(QualifiedName, env[5])) '(' [[
   QualifiedName]]')'

//SYSADL!PortUse to PIADL!Connection Declaration}
[[ID':' QualifiedName'{''}']]env = 'connection' ID 'is'
   getDirection(getPortDef(QualifiedName, env[5])) '(' [[
   QualifiedName]]')'

//SYSADL!QualifiedName to ECORE!String
[[ID]]env = ID

//SYSADL!Configuration to PIADL!BehaviorClause(Composition)
[['configuration{components:' ComponentUse+ 'connectors:'
   ConnectorUse+ 'delegations:'Delegation*'}']]env =
'compose {'
setCompLstElementInstantiation(ComponentUse+)
setConLstElementInstantiation(ConnectorUse+)'} where {'
  setConUnification(ConnectorUse+, env[6], env[11])
```

```
  setDelUnification(Delegation*)
'}'

⟦'configuration{components:' ComponentUse+ 'connectors:'
   ConnectorUse+}'⟧env =
'compose {'
setCompLstElementInstantiation(ComponentUse+)
setConLstElementInstantiation(ConnectorUse+)'} where {'
  setConUnification(ConnectorUse+, env[6], env[11])
'}'
```

C.3.4 Auxiliary Functions

```
//-------------------------------------------------------
getPortDef(QualifiedName, {'port def' ID '{'flow FlowProperty
   QualifiedName'}'}) = 'port def' ID '{'flow FlowProperty
   QualifiedName'}'
getPortDef(QualifiedName, {'port def' ID '{'flow FlowProperty
   QualifiedName2'}'} ∪ pdefSet) = getPortDef(QualifiedName,
   pdefSet)
getPortDef(QualifiedName, {'port def' ID '{'flow FlowProperty
   QualifiedName'}'} ∪ pdefSet) = 'port def' ID '{'flow
   FlowProperty QualifiedName'}'

//-------------------------------------------------------
getTypeDef(ID: QualifiedName, ⟨⟩) = ⟨⟩
getTypeDef(ID: QualifiedName, ⟨'value' 'type' ID2 '{' '}'⟩) = ⟨⟩
getTypeDef(ID: QualifiedName, ⟨'enum' ID2 '{' '}'⟩) = ⟨⟩
getTypeDef(ID: QualifiedName, ⟨'datatype' ID2 '{'('attributes'
   ':' TypeUse*)?'}'⟩) = ⟨⟩
getTypeDef(ID: QualifiedName, ⟨'value' 'type' ID '{' '}'⟩) = ⟨
   'value' 'type' ID '{' '}'⟩
getTypeDef(ID: QualifiedName, ⟨'enum' ID '{' '}'⟩) = ⟨'enum' ID
   '{' '}'⟩
getTypeDef(ID: QualifiedName, ⟨'datatype' ID '{'('attributes'
   ':' TypeUse*)?'}'⟩) = ⟨'datatype' ID '{'('attributes' ':'
   TypeUse*)?'}'⟩
```

```
getTypeDef(ID: QualifiedName, ⟨'value' 'type' ID2 '{'
    '}'⟩⌢typeDefsList) = getTypeDef(ID: QualifiedName,
    typeDefsList)
getTypeDef(ID: QualifiedName, ⟨'enum' ID2 '{'
    '}'⟩⌢typeDefsList) = getTypeDef(ID: QualifiedName,
    typeDefsList)
getTypeDef(ID: QualifiedName, ⟨'datatype' ID2 '{'('attributes'
    ':' TypeUse*)?'}'⟩⌢typeDefsList) = getTypeDef(ID:
    QualifiedName, typeDefsList)
getTypeDef(ID: QualifiedName, ⟨'value' 'type' ID '{'
    '}'⟩⌢typeDefsList) = ⟨'value' 'type' ID '{' '}'⟩
getTypeDef(ID: QualifiedName, ⟨'enum' ID '{' '}'⟩⌢typeDefsList)
    = ⟨'enum' ID '{' '}'⟩
getTypeDef(ID: QualifiedName, ⟨'datatype' ID '{'('attributes'
    ':' TypeUse*)?'}'⟩⌢typeDefsList) = ⟨'datatype' ID
    '{'('attributes' ':' TypeUse*)?'}'⟩

//-------------------------------------------------------------
getDDependency (⟨'value' 'type' ID2 '{' '}'⟩) = ⟨⟩
getDDependency (⟨'enum' ID2 '{' '}'⟩) = ⟨⟩
getDDependency (⟨'datatype' ID2 '{'('attributes' ':'
    TypeUse*)?'}'⟩) = getAttTypes (TypeUse*)

//-------------------------------------------------------------
getAttTypes (⟨ID ':' QualifiedName';'⟩ ) = ⟨QualifiedName⟩
getAttTypes (⟨ID ':' QualifiedName';'⟩⌢typeUsesList ) = ⟨
    QualifiedName⟩ ⌢getAttTypes(typeUsesList)

//-------------------------------------------------------------
distCat(⟨⟩) = ⟨⟩
distCat(⟨qNameList⟩⌢qNamesList_List) = qNameList ⌢
    distCat(qNamesList_List)

//-------------------------------------------------------------
makeInjSeq(⟨⟩) = ⟨⟩
makeInjSeq(⟨QualifiedName⟩ ⌢qNamesList) = ⟨QualifiedName⟩ ⌢
    remove( QualifiedName ,qNamesList)

//-------------------------------------------------------------
```

```
removeQName(qd: QualifiedName,⟨⟩) = ⟨⟩
removeQName(qd: QualifiedName,⟨qd: QualifiedName⟩⌢qNamesList) =
    remove(qd, qNamesList)
removeQName(qd: QualifiedName,⟨qd2: QualifiedName⟩⌢qNamesList) = ⟨
    qd2⟩ ⌢remove(qd, qNamesList)

//-----------------------------------------------------------
getDependencies(QualifiedName, typeDefsList) =
     makeInjSeq(distCat(⟨getDependencies(td,typeDefsList) | td ⟨-
    getDDependency(QualifiedName)⟩))⌢⟨QualifiedName⟩)

//-----------------------------------------------------------
getDependLst(⟨QualifiedName⟩, typeDefsList) =
     makeInjSeq(distCat(⟨getDependencies(td,typeDefsList) | td ⟨-
    getDDependency(QualifiedName)⟩))⌢⟨QualifiedName⟩)

getDependLst(⟨QualifiedName⟩⌢qnList, typeDefsList) =
     makeInjSeq(distCat(⟨getDependencies(td,typeDefsList) | td ⟨-
    getDDependency(QualifiedName)⟩))⌢⟨QualifiedName⟩) ⌢
    getDependLst(qnList, typeDefsList)

//-----------------------------------------------------------
getTypeDefDep(⟨QualifiedName⟩, typeDefsList) = ⟨
    getTypeDef(QualifiedName, typeDefsList)⟩
getTypeDefDep(⟨QualifiedName⟩ ⌢qNameList, typeDefsList) = ⟨
    getTypeDef(QualifiedName, typeDefsList)⟩ ⌢
    getTypeDef(typeDefsList, typeDefsList)⟩

//-----------------------------------------------------------
getNameTypeDefFlows(⟨qf: QualifiedName 'from' qf1:QualifiedName
    'to' qf2:QualifiedName⟩ ⌢flowsList) = qf

//-----------------------------------------------------------
getLstNameTDefPUse(⟨ID ':' QualifiedName';'⟩) = ⟨QualifiedName⟩
getLstNameTDefPUse(⟨ID ':' QualifiedName';'⟩⌢puList) =
⟨QualifiedName⟩⌢getNameTypeDefPortUse(puList)

//-----------------------------------------------------------
setProtocolDeclaration('~'ID':' QualifiedName'{''}') =
```

```
    'via' ID getAction(getPortDef(QualifiedName,
  env[5]))[[QualifiedName]]

//-------------------------------------------------------
setProtocolDeclarationPU(ID':' QualifiedName'{''}') =
    'via' ID getAction(getPortDef(QualifiedName,
  env[5]))[[QualifiedName]]

//-------------------------------------------------------
getDirection('port def' ID '{'flow 'in' QualifiedName'}') = 'in'
getDirection('port def' ID '{'flow 'out' QualifiedName'}') =
  'out'

//-------------------------------------------------------
getAction('port def' ID '{'flow 'in' QualifiedName'}') =
  'receive'
getAction('port def' ID '{'flow 'out' QualifiedName'}') = 'send'

//-------------------------------------------------------
setBehavior('activity def' ID '('Pin*'):('Pin*'){body {actions
  :' ActionUse* ActivityRelation* DataObject*'}}', env[9]) =
  setFunctions(ActionUse*, env[9])

//-------------------------------------------------------
setFunctions(⟨⟩, env[9]) = ϵ
setFunctions(⟨ID':'QualifiedName('{using
  pins:'Pin';'*'}'⟩⌢listActions, env[9]) =
  setFunction(ID':'QualifiedName('{using pins:'Pin';'*'}',
  getActionDef(QualifiedName, env[9]))
  setFunctions(listActions,env[9])

//-------------------------------------------------------
getActionDef(ID:QualifiedName, ⟨⟩) = ϵ
getActionDef(ID:QualifiedName, ⟨'action def'ID2'('Pin*'):'
  QualifiedName'{constraint:'ConstraintUse*
  ActivityDelegation*'}' ⟩) = ϵ
getActionDef(ID:QualifiedName, ⟨'action def'ID'('Pin*'):'
  QualifiedName'{constraint:'ConstraintUse*
  ActivityDelegation*'}' ⟩⌢listAction) = 'action
```

```
    def 'ID' ( 'Pin*' ) : '  QualifiedName '{constraint : 'ConstraintUse*
    ActivityDelegation*
getActionDef(ID: QualifiedName , ⟨'action def'ID2'('Pin*'):'
    QualifiedName '{constraint : 'ConstraintUse*
    ActivityDelegation*'}' ⟩⌢listAction) =
    getActionDef(ID: QualifiedName , listAction)

//-------------------------------------------------------
setFunction(ID':'QualifiedName1('{using
    pins:'Pin';'*'}','action def'ID'('Pin*'):'
    QualifiedName2'{constraint:'ConstraintUse*
    ActivityDelegation*) = ID 'is function ('setInParam(Pin*)')
    :' QualifiedName2'{return}'

//-------------------------------------------------------
 setInParam(⟨⟩) = ϵ
 setInParam(⟨ ID ':' QualifiedName ⟩) = ⟨ID ':' QualifiedName⟩
 setInParam(⟨ ID ':' QualifiedName ⟩⌢ listParam) = ⟨ID ':'
    QualifiedName⟩ setInParam(listParam)

//-------------------------------------------------------
(ID: QualifiedName , ⟨⟩) = ϵ
(ID: QualifiedName , ⟨'activity def' ID2 '('Pin*'):('Pin*')
    {'ActivityBody'}' ⟩) = ϵ
getActivity(ID: QualifiedName , ⟨'activity def' ID '('
    Pin*'):('Pin*') {'ActivityBody'}'⟩) = 'activity def' ID
    '('Pin*'):('Pin*') {'ActivityBody'}'
getActivity(ID: QualifiedName , ⟨'activity def' ID
    '('Pin*'):('Pin*') {'ActivityBody'}'⟩⌢listActivity) =
    'activity def' ID '('Pin*'): ('Pin*') {'ActivityBody'}'
getActivity(ID: QualifiedName , ⟨'activity def' ID2
    '('Pin*'):('Pin*') {'ActivityBody'}'⟩⌢listActivity) =
    getActivity(ID: QualifiedName , listActivity)

//-------------------------------------------------------
setCompLstElementInstantiation(⟨ID':' QualifiedName'{using
    ports:'PortUse*)'}'⟩)= ID'is'QualifiedName'()'
setCompLstElementInstantiation(⟨ID':' QualifiedName'{using
    ports:'PortUse*)'}'⟩⌢CompULst)= ID'is'QualifiedName'() and'
```

```
    setCompLstElementInstantiation(CompULst)

//-------------------------------------------------------
setConLstElementInstantiation(⟨ID ':' QualifiedName' bindings
    'ConnectorBinding*';'⟩) = 'and' ID 'is' QualifiedName'()'
setConLstElementInstantiation(⟨ID ':' QualifiedName' bindings
    'ConnectorBinding*';'⟩ ⌢ConULst) = 'and'
    ID 'is' QualifiedName() setConLstElementInstantiation(ConUList)

//-------------------------------------------------------
setConUnification(⟨ID ':' QualifiedName 'bindings'
    ConnectorBinding ';'⟩, env[6], env[11])=
setConnectionAccessFrom(ConnectorBinding, env[11]) 'unifies'
    setCAccessFromCon(getConnectorDef(QualifiedName, env[6]))
setCAccessToCon(getConnectorDef(QualifiedName, env[6]))
    'unifies' setConnectionAccessTo(ConnectorBinding, env[11])

setConUnification(⟨ID':'QualifiedName'bindings'ConnectorBinding';'⟩ ⌢ConU
    env[6], env[11]) =
setConnectionAccessFrom(ConnectorBinding, env[11]) 'unifies'
    setCAccessFromCon(getConnectorDef(QualifiedName, env[6]))
setCAccessToCon(getConnectorDef(QualifiedName, env[6]))
    'unifies' setConnectionAccessTo(ConnectorBinding, env[11])
setConUnification(ConUList)

//-------------------------------------------------------
setConnectionAccessFrom(qnSource:QualifiedName '='
    qnDest:QualifiedName, env[11]) =
    qnSource'::'getComponentUsePU(qnSource, env[11])

//-------------------------------------------------------
setConnectionAccessTo(qnSource:QualifiedName '='
    qnDest:QualifiedName, env[11]) =
    qnDest'::'getComponentUsePU(qnDest, env[11])

//-------------------------------------------------------
setCAccessFromCon('connector def' ID '{' ('participants :'
    PortUse_Reverse*)
```

```
('flows :' Flow*)'}') =
    ID'::'getPorUseReverseOut(PortUse_Reverse*)

//-------------------------------------------------------
setCAccessToCon('connector def' ID '{' ('participants :'
    PortUse_Reverse*)
('flows :' Flow*)'}') =
    ID'::'getPorUseReverseIn(PortUse_Reverse*)

//-------------------------------------------------------
getConnectorDef(ID:QualifiedName, ⟨ 'connector def' ID '{'
    ('participants :' PortUse_Reverse*)('flows :' Flow*)'}'⟩) =
    'connector def' ID '{' ('participants :'
    PortUse_Reverse*)('flows :' Flow*)'}'
getConnectorDef(ID:QualifiedName, ⟨ 'connector def' ID2 '{'
    ('participants :' PortUse_Reverse*)('flows :'
    Flow*)'}'⟩⌢ConDefList) = getConnectorDef(ID:QualifiedName,
    ConDefList)

//-------------------------------------------------------
getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU1:QualifiedName ':'
    QualifiedName'{ }'⟩'}'⟩) = nameComp

getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU1:QualifiedName ':'
    QualifiedName'{ }'⟩⌢puList'}'⟩) = nameComp

getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU2:QualifiedName ':'
    QualifiedName'{ }'⟩⌢puList'}'⟩) =
getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'puList)⟩

getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU1:QualifiedName ':'
    QualifiedName'{ }'⟩'}'⟩⌢compUlist) = nameComp
```

```
getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU1: QualifiedName ':'
    QualifiedName'{ }'⟩ ⌢puList'}'⟩ ⌢compUlist) = nameComp

getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU2: QualifiedName ':'
    QualifiedName'{ }'⟩'}'⟩ ⌢compUlist) =
    getComponentUsePU(namePU1: QualifiedName, compUlist)

getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU1: QualifiedName ':'
    QualifiedName'{ }'⟩ ⌢puList'}'⟩ ⌢CompUlist) = nameComp

getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'⟨namePU2: QualifiedName ':'
    QualifiedName'{ }'⟩ ⌢ puList'}'⟩ ⌢compUlist) =
getComponentUsePU(namePU1: QualifiedName, ⟨ID ':' nameComp:
    QualifiedName '{ using ports :'puList)⟩ ⌢compUlist)

//-------------------------------------------------------
getPorUseReverseOut(⟨'~' ID ':' 'port' 'def' ID2 '{flow' 'out'
    TypeDef'}' '{ }'⟩) = '~' ID

getPorUseReverseOut(⟨'~' ID ':' 'port' 'def' ID2 '{flow' 'out'
    TypeDef'}' '{ }'⟩ ⌢puRevList) = '~' ID

getPorUseReverseOut(⟨'~' ID ':' 'port' 'def' ID2 '{flow' 'in'
    TypeDef'}' '{ }'⟩ ⌢puRevList) =
    getPorUseReverseOut(puRevList)

//-------------------------------------------------------
getPorUseReverseIn(⟨'~' ID ':' 'port' 'def' ID2 '{flow' 'in'
    TypeDef'}' '{ }'⟩) = '~' ID

getPorUseReverseIn(⟨'~' ID ':' 'port' 'def' ID2 '{flow' 'in'
    TypeDef'}' '{ }'⟩ ⌢puRevList) = '~' ID

getPorUseReverseIn(⟨'~' ID ':' 'port' 'def' ID2 '{flow' 'out'
    TypeDef'}' '{ }'⟩ ⌢puRevList) =
```

```
    getPorUseReverseOut(puRevList)

//-------------------------------------------------------------
setDelUnification(⟨[ID1 ':' QualifiedName1] 'to' [ID2 ':'
    QualifiedName2]⟩) =
QualifiedName1 '::' ID1 'unifies' QualifiedName2 '::' ID2
```

APPENDIX D – Challenges and Limitations of the Implementation

During the development of this work, challenges arose that caused some limitations in our approach, as listed below.

- The systematic review was performed in December 2018, so it is possible that new relevant studies were published after this period. We even carried out a complementary survey in December 2022, however, without the rigor of a systematic mapping, and we have already included some results in the related works section.
- The SysADL language support tool, SysADL Studio, still needs to be more stable when supporting some structures provided in the language, such as implementing the executable point of view.
- SysADL Studio supports the implementation of composite ports; however, in practice, the use of these ports has become impractical, as it is impossible to differentiate the instances of these ports when used in connectors or components. Due to this tool limitation, our approach restricted the use of composite ports.
- There needs to be more support material for using SysADL, which aligns with the features implemented in the SysADL Studio tool. A published book presents the language; however, some elements implemented in the tool differ from the version shown in the book. We understand that these are minor changes related to the tool's implementation; however, they make it difficult to use
- Due to time restrictions, it was not possible to formally verify the correctness of the denotational semantics of SysADL as a function of π-ADL.
- It was a challenge to find support for using π-ADL and its tools and an active community in using the language and tools. There is no supporting material and varied examples beyond the published articles.
- We encountered difficulties using the tool to π-ADL de Cavalcante (CAVALCANTE, 2016). Initially, it is necessary to have more details about the versions of compilers and plugins necessary for the tool to work, a tutorial or material describing how it worked, and the directory structure necessary to use it.
- Another challenge related to the use of the tool by Cavalcante (CAVALCANTE, 2016), was realizing that the tool's syntactic analyzer was not recognizing some

elements of the language grammar and compiler, for example, the reserved word `self`, preventing the use of composite components in our test examples.

- A limitation in validating our approach was integrating a Model Checking tool into our process. We attempted to integrate the Plasma Lab (PLASMA, 2022) tool to verify properties expressed in DynBLBT on π-ADL architectures formally. However, we needed support to solve the problems with the tool, which had an anomalous behavior in the tests we carried out, working in some cases and not in others. We even tried to contact members of the development team, but we have yet to be successful.

ANNEX A – FMS - architecture execution

```
init
0 create_node _S1 SensorCP senseIn,type,in passIn,type,in
measureOut,type,out
0 create_node _S2 SensorCP senseIn,type,in passIn,type,in
measureOut,type,out
0 create_node _S3 SensorCP senseIn,type,in passIn,type,in
measureOut,type,out
0 create_node _S4 SensorCP senseIn,type,in passIn,type,in
measureOut,type,out
0 create_node _S5 SensorCP senseIn,type,in passIn,type,in
measureOut,type,out
0 create_node _SE1 SensorEnvConnectorCP valueReadyIn,type,in
valueSentOut,type,out valueSOut,type,out sFailedSOut,type,out
0 create_node _SE2 SensorEnvConnectorCP valueReadyIn,type,in
valueSentOut,type,out valueSOut,type,out sFailedSOut,type,out
0 create_node _SE3 SensorEnvConnectorCP valueReadyIn,type,in
valueSentOut,type,out valueSOut,type,out sFailedSOut,type,out
0 create_node _SE4 SensorEnvConnectorCP valueReadyIn,type,in
valueSentOut,type,out valueSOut,type,out sFailedSOut,type,out
0 create_node _SE5 SensorEnvConnectorCP valueReadyIn,type,in
valueSentOut,type,out valueSOut,type,out sFailedSOut,type,out
0 create_node _ZB1 ZigBeeConnectorCP passZIn,type,in
passZOut,type,out
0 create_node _ZB2 ZigBeeConnectorCP passZIn,type,in
passZOut,type,out
0 create_node _ZB3 ZigBeeConnectorCP passZIn,type,in
passZOut,type,out
0 create_node _ZB4 ZigBeeConnectorCP passZIn,type,in
passZOut,type,out
0 create_node _ZB5 ZigBeeConnectorCP passZIn,type,in
passZOut,type,out
0 create_node _EnvCP EnvCP sFailedEIn,type,in valueSentIn,type,in
valueReadyOut,type,out tickOut,type,out sFailedEOut,type,out
floodOut,type,out floodLevelIn,type,in
0 create_node _gw GatewayCP passGIn,type,in tickIn,type,in
alertOut,type,out floodLevelOut,type,out
0 create_node _obs ObserverConnectorCP alertIn,type,in
floodIn,type,in sFailedOIn,type,in
0 create_node _CMP1 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CMP2 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CMP3 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CMP4 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CMP5 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CPP1 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CPP2 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CPP3 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CPP4 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CPP5 CmH2OCN cIn,type,out cOut,type,in
0 create_node _CVS1 mvCN mOut,type,in mIn,type,out
0 create_node _CVS2 mvCN mOut,type,in mIn,type,out
0 create_node _CVS3 mvCN mOut,type,in mIn,type,out
0 create_node _CVS4 mvCN mOut,type,in mIn,type,out
0 create_node _CVS5 mvCN mOut,type,in mIn,type,out
0 create_node _CVR1 booleanCN bIn,type,out bOut,type,in
```

```
0 create_node _CVR2 booleanCN bIn,type,out bOut,type,in
0 create_node _CVR3 booleanCN bIn,type,out bOut,type,in
0 create_node _CVR4 booleanCN bIn,type,out bOut,type,in
0 create_node _CVR5 booleanCN bIn,type,out bOut,type,in
0 create_node _CFS1 IDSensorCN sidOut,type,in sidIn,type,out
0 create_node _CFS2 IDSensorCN sidOut,type,in sidIn,type,out
0 create_node _CFS3 IDSensorCN sidOut,type,in sidIn,type,out
0 create_node _CFS4 IDSensorCN sidOut,type,in sidIn,type,out
0 create_node _CFS5 IDSensorCN sidOut,type,in sidIn,type,out
0 create_node _CSV1 booleanCN bIn,type,out bOut,type,in
0 create_node _CSV2 booleanCN bIn,type,out bOut,type,in
0 create_node _CSV3 booleanCN bIn,type,out bOut,type,in
0 create_node _CSV4 booleanCN bIn,type,out bOut,type,in
0 create_node _CSV5 booleanCN bIn,type,out bOut,type,in
0 create_node _CT booleanCN bIn,type,out bOut,type,in
0 create_node _CSF IDSensorCN sidOut,type,in sidIn,type,out
0 create_node _CF1 booleanCN bOut,type,in bIn,type,out
0 create_node _CF2 booleanCN bIn,type,out bOut,type,in
0 create_node _CAL stringCN sOut,type,in sIn,type,out
0 create_link _EnvCP.tickOut _CT.bOut
0 create_link _CT.bIn _gw.tickIn
0 create_link _S1.measureOut _CMP1.cOut
0 create_link _CMP1.cIn _ZB1.passZIn
0 create_link _S2.measureOut _CMP2.cOut
0 create_link _CMP2.cIn _ZB2.passZIn
0 create_link _S3.measureOut _CMP3.cOut
0 create_link _CMP3.cIn _ZB3.passZIn
0 create_link _S4.measureOut _CMP4.cOut
0 create_link _CMP4.cIn _ZB4.passZIn
0 create_link _S5.measureOut _CMP5.cOut
0 create_link _CMP5.cIn _ZB5.passZIn
0 create_link _ZB1.passZOut _CPP1.cOut
0 create_link _CPP1.cIn _S2.passIn
0 create_link _ZB2.passZOut _CPP2.cOut
0 create_link _CPP2.cIn _S3.passIn
0 create_link _ZB3.passZOut _CPP3.cOut
0 create_link _CPP3.cIn _S4.passIn
0 create_link _ZB4.passZOut _CPP4.cOut
0 create_link _CPP4.cIn _S5.passIn
0 create_link _ZB5.passZOut _CPP5.cOut
0 create_link _CPP5.cIn _gw.passGIn
0 create_link _SE1.valueSOut _CVS1.mOut
0 create_link _CVS1.mIn _S1.senseIn
0 create_link _SE2.valueSOut _CVS2.mOut
0 create_link _CVS2.mIn _S2.senseIn
0 create_link _SE3.valueSOut _CVS3.mOut
0 create_link _CVS3.mIn _S3.senseIn
0 create_link _SE4.valueSOut _CVS4.mOut
0 create_link _CVS4.mIn _S4.senseIn
0 create_link _SE5.valueSOut _CVS5.mOut
0 create_link _CVS5.mIn _S5.senseIn
0 create_link _EnvCP.valueReadyOut _CVR1.bOut
0 create_link _CVR1.bIn _SE1.valueReadyIn
0 create_link _EnvCP.valueReadyOut _CVR2.bOut
```

```
0 create_link _CVR2.bIn _SE2.valueReadyIn
0 create_link _EnvCP.valueReadyOut _CVR3.bOut
0 create_link _CVR3.bIn _SE3.valueReadyIn
0 create_link _EnvCP.valueReadyOut _CVR4.bOut
0 create_link _CVR4.bIn _SE4.valueReadyIn
0 create_link _EnvCP.valueReadyOut _CVR5.bOut
0 create_link _CVR5.bIn _SE5.valueReadyIn
0 create_link _SE1.sFailedSOut _CFS1.sidOut
0 create_link _CFS1.sidIn _EnvCP.sFailedEIn
0 create_link _SE2.sFailedSOut _CFS2.sidOut
0 create_link _CFS2.sidIn _EnvCP.sFailedEIn
0 create_link _SE3.sFailedSOut _CFS3.sidOut
0 create_link _CFS3.sidIn _EnvCP.sFailedEIn
0 create_link _SE4.sFailedSOut _CFS4.sidOut
0 create_link _CFS4.sidIn _EnvCP.sFailedEIn
0 create_link _SE5.sFailedSOut _CFS5.sidOut
0 create_link _CFS5.sidIn _EnvCP.sFailedEIn
0 create_link _SE1.valueSentOut _CSV1.bOut
0 create_link _CSV1.bIn _EnvCP.valueSentIn
0 create_link _SE2.valueSentOut _CSV2.bOut
0 create_link _CSV2.bIn _EnvCP.valueSentIn
0 create_link _SE3.valueSentOut _CSV3.bOut
0 create_link _CSV3.bIn _EnvCP.valueSentIn
0 create_link _SE4.valueSentOut _CSV4.bOut
0 create_link _CSV4.bIn _EnvCP.valueSentIn
0 create_link _SE5.valueSentOut _CSV5.bOut
0 create_link _CSV5.bIn _EnvCP.valueSentIn
0 create_link _EnvCP.sFailedEOut _CSF.sidOut
0 create_link _CSF.sidIn _obs.sFailedOIn
0 create_link _gw.floodLevelOut _CF1.bOut
0 create_link _CF1.bIn _EnvCP.floodLevelIn
0 create_link _EnvCP.floodOut _CF2.bOut
0 create_link _CF2.bIn _obs.floodIn
0 create_link _gw.alertOut _CAL.sOut
0 create_link _CAL.sIn _obs.alertIn
0 done
0 rdv _EnvCP.tickOut _CT.bOut true
1 rdv _EnvCP.valueReadyOut _CVR3.bOut true
1 rdv _CT.bIn _gw.tickIn true
1 rdv _EnvCP.valueReadyOut _CVR4.bOut true
1 rdv _EnvCP.valueReadyOut _CVR1.bOut true
1 rdv _CVR1.bIn _SE1.valueReadyIn true
1 rdv _EnvCP.valueReadyOut _CVR5.bOut true
1 rdv _CVR3.bIn _SE3.valueReadyIn true
1 rdv _SE1.valueSOut _CVS1.mOut 3.7205106025E+00
1 rdv _SE3.valueSOut _CVS3.mOut -1.0000000000E+00
1 rdv _EnvCP.valueReadyOut _CVR1.bOut true
1 rdv _SE1.sFailedSOut _CFS1.sidOut 439
1 rdv _CVR4.bIn _SE4.valueReadyIn true
1 rdv _CVS3.mIn _S3.senseIn
1 rdv _SE3.sFailedSOut _CFS3.sidOut 0
1 rdv _S3.measureOut _CMP3.cOut tuple<node<_S3>, -1.0000000000E+00>
1 rdv _CVR5.bIn _SE5.valueReadyIn true
1 rdv _SE5.valueSOut _CVS5.mOut 3.8061550959E+00
```

```
1 rdv _CVS5.mIn _S5.senseIn
1 rdv _SE4.valueSOut _CVS4.mOut -1.0000000000E+00
1 rdv _CMP3.cIn _ZB3.passZIn
1 rdv _CVS1.mIn _S1.senseIn
1 rdv _S1.measureOut _CMP1.cOut tuple<node<_S1>, 3.7205106025E+00>
1 rdv _CMP1.cIn _ZB1.passZIn
1 rdv _ZB1.passZOut _CPP1.cOut
1 rdv _CVS4.mIn _S4.senseIn
1 rdv _S5.measureOut _CMP5.cOut tuple<node<_S5>, 3.8061550959E+00>
1 rdv _SE3.valueSentOut _CSV3.bOut true
1 rdv _SE5.sFailedSOut _CFS5.sidOut 439
1 rdv _CSV3.bIn _EnvCP.valueSentIn true
1 rdv _SE1.valueSentOut _CSV1.bOut true
1 rdv _S4.measureOut _CMP4.cOut tuple<node<_S4>, -1.0000000000E+00>
1 rdv _ZB3.passZOut _CPP3.cOut
1 rdv _CPP3.cIn _S4.passIn
1 rdv _CVR1.bIn _SE1.valueReadyIn true
1 rdv _SE1.valueSOut _CVS1.mOut -1.0000000000E+00
1 rdv _CFS1.sidIn _EnvCP.sFailedEIn 439
1 rdv _SE4.sFailedSOut _CFS4.sidOut 0
1 rdv _CMP5.cIn _ZB5.passZIn
1 rdv _SE5.valueSentOut _CSV5.bOut true
1 rdv _CMP4.cIn _ZB4.passZIn
1 rdv _SE1.sFailedSOut _CFS1.sidOut 0
1 rdv _S4.measureOut _CMP4.cOut tuple<node<_S3>, -1.0000000000E+00>
1 rdv _SE4.valueSentOut _CSV4.bOut true
1 rdv _ZB5.passZOut _CPP5.cOut
1 rdv _CPP5.cIn _gw.passGIn
1 rdv _gw.alertOut _CAL.sOut "low"
1 rdv _CSV1.bIn _EnvCP.valueSentIn true
1 rdv _CFS3.sidIn _EnvCP.sFailedEIn 0
1 rdv _CSV5.bIn _EnvCP.valueSentIn true
1 rdv _CFS1.sidIn _EnvCP.sFailedEIn 0
1 rdv _CSV4.bIn _EnvCP.valueSentIn true
1 rdv _CFS5.sidIn _EnvCP.sFailedEIn 439
1 rdv _CVS1.mIn _S1.senseIn
1 rdv _CAL.sIn _obs.alertIn "low"
1 rdv _SE1.valueSentOut _CSV1.bOut true
1 rdv _ZB4.passZOut _CPP4.cOut
1 rdv _CSV1.bIn _EnvCP.valueSentIn true
1 rdv _CPP1.cIn _S2.passIn
1 rdv _CFS4.sidIn _EnvCP.sFailedEIn 0
1 rdv _S1.measureOut _CMP1.cOut tuple<node<_S1>, -1.0000000000E+00>
1 rdv _S2.measureOut _CMP2.cOut tuple<node<_S1>, 3.7205106025E+00>
1 rdv _CMP4.cIn _ZB4.passZIn
1 rdv _CMP2.cIn _ZB2.passZIn
1 rdv _CPP4.cIn _S5.passIn
1 rdv _S5.measureOut _CMP5.cOut tuple<node<_S4>, -1.0000000000E+00>
1 rdv _ZB4.passZOut _CPP4.cOut
1 rdv _CPP4.cIn _S5.passIn
1 rdv _CMP5.cIn _ZB5.passZIn
1 rdv _ZB5.passZOut _CPP5.cOut
1 rdv _S5.measureOut _CMP5.cOut tuple<node<_S3>, -1.0000000000E+00>
1 rdv _CMP5.cIn _ZB5.passZIn
```

```
1 rdv _ZB2.passZOut _CPP2.cOut
1 rdv _CMP1.cIn _ZB1.passZIn
1 rdv _CPP2.cIn _S3.passIn
1 rdv _S3.measureOut _CMP3.cOut tuple<node<_S1>, 3.7205106025E+00>
1 rdv _CPP5.cIn _gw.passGIn
1 rdv _ZB1.passZOut _CPP1.cOut
1 rdv _CPP1.cIn _S2.passIn
1 rdv _ZB5.passZOut _CPP5.cOut
1 rdv _S2.measureOut _CMP2.cOut tuple<node<_S1>, -1.0000000000E+00>
1 rdv _CMP2.cIn _ZB2.passZIn
1 rdv _CMP3.cIn _ZB3.passZIn
1 rdv _ZB3.passZOut _CPP3.cOut
1 rdv _CPP3.cIn _S4.passIn
1 rdv _gw.alertOut _CAL.sOut "low"
1 rdv _ZB2.passZOut _CPP2.cOut
1 rdv _CPP2.cIn _S3.passIn
1 rdv _S4.measureOut _CMP4.cOut tuple<node<_S1>, 3.7205106025E+00>
1 rdv _CAL.sIn _obs.alertIn "low"
1 rdv _CMP4.cIn _ZB4.passZIn
1 rdv _ZB4.passZOut _CPP4.cOut
1 rdv _CPP5.cIn _gw.passGIn
1 rdv _CPP4.cIn _S5.passIn
1 rdv _S3.measureOut _CMP3.cOut tuple<node<_S1>, -1.0000000000E+00>
1 rdv _CMP3.cIn _ZB3.passZIn
1 rdv _S5.measureOut _CMP5.cOut tuple<node<_S1>, 3.7205106025E+00>
1 rdv _gw.alertOut _CAL.sOut "low"
1 rdv _CMP5.cIn _ZB5.passZIn
1 rdv _ZB5.passZOut _CPP5.cOut
1 rdv _CAL.sIn _obs.alertIn "low"
1 rdv _CPP5.cIn _gw.passGIn
1 rdv _gw.alertOut _CAL.sOut "low"
1 rdv _ZB3.passZOut _CPP3.cOut
1 rdv _CAL.sIn _obs.alertIn "low"
1 rdv _CPP3.cIn _S4.passIn
1 rdv _S4.measureOut _CMP4.cOut tuple<node<_S1>, -1.0000000000E+00>
1 rdv _CMP4.cIn _ZB4.passZIn
1 rdv _ZB4.passZOut _CPP4.cOut
1 rdv _CPP4.cIn _S5.passIn
1 rdv _S5.measureOut _CMP5.cOut tuple<node<_S1>, -1.0000000000E+00>
1 rdv _CMP5.cIn _ZB5.passZIn
1 rdv _ZB5.passZOut _CPP5.cOut
1 rdv _CPP5.cIn _gw.passGIn
1 rdv _gw.alertOut _CAL.sOut "low"
1 rdv _CAL.sIn _obs.alertIn "low"
1 rdv _gw.floodLevelOut _CF1.bOut false
1 rdv _CF1.bIn _EnvCP.floodLevelIn false
1 rdv _EnvCP.sFailedEOut _CSF.sidOut 0
1 rdv _EnvCP.floodOut _CF2.bOut false
1 rdv _CF2.bIn _obs.floodIn false
1 rdv _EnvCP.tickOut _CT.bOut true
2 rdv _EnvCP.valueReadyOut _CVR1.bOut true
2 rdv _CVR1.bIn _SE1.valueReadyIn true
2 rdv _EnvCP.valueReadyOut _CVR1.bOut true
2 rdv _EnvCP.valueReadyOut _CVR4.bOut true
```