A Formally Founded Framework for Dynamic Software Architectures

Everton Ranielly de Sousa Cavalcante

To cite this version:

Everton Ranielly de Sousa Cavalcante. A Formally Founded Framework for Dynamic Software Architectures. Software Engineering [cs.SE]. Université de Bretagne Sud; Universidade federal do Rio Grande do Norte (Natal, Brésil), 2016. English. NNT : 2016LORIS403 . tel-01426029

HAL Id: tel-01426029
https://theses.hal.science/tel-01426029

Submitted on 4 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE
BRETAGNE
LOIRE

THESE / UNIVERSITE BRETAGNE SUD
sous le sceau de l'Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITE BRETAGNE SUD

Mention: STIC
Ecole doctorale: SICMA

Présentée par

Everton Ranielly DE SOUSA CAVALCANTE

Préparée à l'unité mixte de recherche 6074
Institut de Recherche en Informatique et Systèmes Aléatoires
Université Bretagne Sud

Un framework formel pour les architectures logicielles dynamiques

Thèse soutenue le 10 juin 2016, devant le jury composé de :

M. Paulo Roberto FREIRE CUNHA
Professeur, Université Fédérale de Pernambuco, Recife, Brésil / Président

M. Khalil DRIRA
Directeur de Recherches CNRS, LAAS-CNRS, Toulouse, France / Rapporteur

Mme. Elisa Yumi NAKAGAWA
Maître de Conféréndes, HDR, Université de São Paulo, São Carlos, Brésil / Rapporteur

M. Gibeon Soares AQUINO JUNIOR
Maître de Conféréndes, Université Fédérale du Rio Grande do Norte, Natal, Brésil / Examinateur

M. Jair CAVALCANTI LEITE
Professeur, Université Fédérale du Rio Grande do Norte, Natal, Brésil / Examinateur

M. Carlos Enrique CUESTA QUINTERO
Maître de Conféréndes, HDR, Université Rey Juan Carlos, Madrid, Espagne / Examinateur

Mme. Thais VASCONCELOS BATISTA
Maître de Conféréndes, HDR, Université Fédérale du Rio Grande do Norte, Natal, Brésil / Directrice de thèse

M. Flavio OQUENDO
Professeur des Universités, IRISA - Université Bretagne Sud, Vannes, France / Directeur de thèse

A Formally Founded Framework for Dynamic Software Architectures

Everton Ranielly de Sousa Cavalcante

Doctoral dissertation submitted in partial fulfillment of the requirements for the degrees of *Doutor em Ciência da Computação* and *Docteur en Informatique* under the joint supervision agreement between the Universidade Federal do Rio Grande do Norte (UFRN), Brazil, and Université Bretagne Sud (UBS), France.

Supervisors

Prof. Dr. Thais Vasconcelos Batista
Universidade Federal do Rio Grande do Norte, Natal, Brazil

Prof. Dr. Flavio Oquendo
IRISA-UMR CNRS/Université Bretagne Sud, Vannes, France

June 2016

Acknowledgments

Saying "thanks" is to admit that there was a moment at which someone was needed, is to acknowledge that a man can never achieve the gift of being self-sufficient. None and nothing grows alone: a supportive look, a word of encouragement, a gesture of understanding, an attitude of love is always necessary. It may hard to say this in words, but I would like to say "thanks" to several people who have contributed in some way, directly or not, to allow me to write these words. It is still being possible that there are other people who could be cited here, but unfortunately they were not for malfunction of my memory. To those ones, my sincere and humble apologies.

I praise God, the Author of life, for one more opportunity given and for the achievements in the midst of adversity. I am grateful to him for giving me perseverance, strength, and courage to dream and keep fighting for my goals, for making me who I am today, and for concretizing what seemed far away or even impossible to my limited eyes.

I thank my parents, Maria Gorete and José Cavalcante (in memory), for the effort under difficulties that had to be overcome to provide me with a good education, for the principles and values that do not pass and undoubtedly are elements that I will carry with me for the rest of my existence. They have taught me how to live life with dignity and honesty and made me see that the road goes beyond what you can see.

From the bottom of my heart, I thank to my supervisors, Professors Thais Vasconcelos Batista (UFRN, Brazil) and Flavio Oquendo (IRISA/Université Bretagne Sud, France), for everything. Life was very generous in allowing me to work not only with two professionals of excellence, but with unique human beings. In general, the route during PhD is significantly hard, but I can say that this time incredibly was a happy journey for me. This is undoubtedly resulted from dedication, attention, and partnership in the trio that we formed, directly reflecting into the quality of this work. They have always allowed me to be free and provided means of maturing my ideas, always pointing out the correct direction. In every step given, I did it with more confidence because I knew that I had their support. Namely:

> To Thais: *Our greatest desire in life is to find someone to make us the best that we can. I have no doubt that she was the most important person of my academic life so far, to the point that I do not know what else can be expected from a supervisor. I absolutely had more than I expected or even deserved, because I had her friendship, availability, advices, patience, trust, and many opportunities offered. I am very grateful to Thais for her professional example that inspires me every day and mainly for being a human being with a huge heart. Anything that I say here will be still few to thank her. For these and other many reasons is that this PhD thesis is dedicated to her.*
>
> To Flavio: *I can proudly say today that I was supervised by one of the most world-*

renowned researchers in Software Architecture. I am very grateful to Flavio since his positive answer to the invitation to be my supervisor, for closely following the conduction of this work, and mainly for receiving me with open arms in Vannes, France, during the almost two years spent there during PhD, more than six thousand kilometers from my homeland. I have grown very much as a scientist and this owes to his example of wisdom, deep knowledge, commitment, and vision, a standard that I will always seek to follow.

I thank Professors Jair Cavalcanti Leite, Gibeon Aquino Junior (UFRN, Brazil), Paulo Cunha (UFPE, Brazil), Elisa Yumi Nakagawa (USP, Brazil), Khalil Drira (Université Fédérale de Toulouse-Midi-Pyrénées, France), and Carlos Cuesta (Universidad Rey Juan Carlos, Spain) who have kindly volunteered their time and accepted the invitation to participate in the examination committee for evaluating this PhD thesis. It was a great honor having a number of researchers of excellence willing to give their contribution to this work.

I thank professors Flavia Coimbra Delicato and Paulo de Figueiredo Pires (UFRJ, Brazil), who have participated in the examination committee of my PhD qualification. The final result of this work has important contributions from them and for which I am very grateful.

I thank the friends who are either close or far, the ones who I see almost every day, the ones who I "see" only through Internet. Unfortunately, I will not be able to cite the names of all of them, for both available space and the risk or injustice of forgetting someone, but please receive here my words of gratitude. Nonetheless, I would like to mention Gustavo Alves and Everton Lima, who have always been close, could share several moments with me, and offered their precious friendship and support, fundamental elements to this journey. Thank you very much.

I thank the researchers from the ArchWare group – Equipe de Recherche sur les Architectures Logicielles from IRISA/Université Bretagne Sud for receiving me when I was in Vannes, France, and with whom I could deepen my knowledge on software architectures. In particular, special thanks to Jean Quilbeuf, who has directly participated and contributed to this work, for his constant availability and patience to solve my doubts regarding statistical model checking in software architectures. "Merci beaucoup".

I thank the colleagues and friends from ICMC-USP (São Carlos, Brazil), who which I had the pleasure of meeting when I was in Vannes and establish fruitful partnership, specially Lucas Oliveira, Milena Guessi, and Professor Elisa Yumi Nakagawa. From this group, I still need to express my words of gratitude to Marcelo Gonçalves, not only for the partnership in the several works that we have conducted together, but mainly for the special person that he is.

Finally, I thank the institutions that have encouraged the execution of this project and provided the essential financial support to the development of this work: to the Brazilian National Agency of Petroleum, Natural Gas and Biofuels (ANP), through the Program for Human Resources in Geology, Geophysics and Information Technology in Oil and Gas (PRH-22); to the Brazilian Funding Authority for Studies and Projects (FINEP); to the Brazilian

Ministry of Science, Technology, Innovations and Communications (MCTIC); to the Brazilian National Council for Scientific and Technological Development (CNPq); and to the Brazilian Coordination for the Improvement of Higher Education Personnel (CAPES).

Agradecimentos

Dizer "obrigado" é admitir que houve um momento em que se precisou de alguém, é reconhecer que o homem jamais poderá lograr para si o dom de ser autossuficiente. Ninguém e nada cresce sozinho: sempre é preciso um olhar de apoio, uma palavra de incentivo, um gesto de compreensão, uma atitude de amor. Talvez seja difícil dizer isto em forma de palavras, mas gostaria de dizer "obrigado" a diversas pessoas que, de alguma forma, contribuíram diretamente ou não para que eu pudesse estar escrevendo estas palavras. É possível ainda que haja outras pessoas que poderiam ser citadas aqui, mas, por disfunção da minha memória, infelizmente não o foram. A estes desde já peço sinceras e humildes desculpas.

Agradeço a Deus, o Autor da Vida, por mais esta oportunidade concedida e pelas conquistas alcançadas em meio às adversidades. A Ele sou grato por me dar perseverança, força e coragem para sonhar e seguir lutando pelos meus objetivos, por me fazer quem sou hoje, e por concretizar aquilo que parecia longínquo ou até mesmo impossível aos meus olhos limitados.

Agradeço aos meus pais, Maria Gorete e José Cavalcante (in memoriam), pelo esforço sob dificuldades que tiveram de ser ultrapassadas para proporcionar uma boa formação, pelos ensinamentos e valores que não passam e que, sem dúvida, são elementos que levarei comigo pelo resto da minha existência. Eles me ensinaram como viver a vida com dignidade e honestidade e me fizeram ver que a estrada vai além do que se vê.

Agradeço do fundo do coração aos meus orientadores, os Professores Thais Vasconcelos Batista (UFRN, Brasil) e Flavio Oquendo (IRISA/Université Bretagne Sud, França), por absolutamente tudo. A vida foi muito generosa ao me permitir trabalhar não apenas com dois profissionais de excelência, mas sobretudo com seres humanos ímpares. Em geral, o percurso durante o Doutorado é significantemente árduo, porém posso dizer que esse período para mim foi uma feliz jornada, incrivelmente. Isso sem dúvida é resultado da dedicação, atenção e trabalho em parceria no trio que formamos, o que se refletiu diretamente na qualidade deste trabalho. Eles sempre me permitiram ter liberdade e proporcionaram meios para amadurecer minhas ideias, indicando-me sempre o direcionamento correto. Cada passo que pude dar, o fiz com mais confiança por saber que tinha o apoio deles. Nominalmente:

> À Thais. *Nosso maior desejo na vida é encontrar alguém que nos faça fazer o melhor que pudermos. Não tenho dúvidas que ela foi a pessoa mais importante da minha vida acadêmica até aqui, a ponto que eu não sei o que mais se pode esperar de um orientador. Eu sem dúvida tive muito mais do que eu esperava ou mesmo merecia, pois pude ter sua amizade, disponibilidade, conselhos, paciência, confiança e diversas oportunidades proporcionadas. À Thais sou muito grato pelo exemplo de profissional que me inspira a cada dia e principalmente pelo ser humano de coração enorme que ela é. Qualquer coisa que eu disser aqui ainda será muito pouco para agradecê-la. Por essas e outras diversas razões que esta tese de Doutorado é dedicada a ela.*

Ao Flavio: *Posso hoje dizer com muito orgulho que pude ser orientando de um dos pesquisadores mais renomados da área de Arquitetura de Software no mundo. Sou muito grato ao Flavio desde sua resposta positiva ao convite para ser meu orientador, por acompanhar de perto a condução deste trabalho e principalmente por me receber de braços abertos em Vannes, França, durante os quase dois anos que lá passei durante o Doutorado, a mais de seis mil quilômetros de distância da minha terra natal. Cresci bastante como cientista e isso se deve muito ao seu exemplo de sabedoria, profundo conhecimento, comprometimento e visão, um padrão que sempre procurarei seguir.*

Agradeço aos Professores Jair Cavalcanti Leite, Gibeon Aquino Junior (UFRN, Brasil), Paulo Cunha (UFPE, Brasil), Elisa Yumi Nakagawa (USP, Brasil), Khalil Drira (Université Fédérale de Toulouse-Midi-Pyrénées, França) e Carlos Cuesta (Universidad Rey Juan Carlos, Espanha) que gentilmente dispuseram do seu tempo e aceitaram o convite para participarem do comitê examinador para avaliação desta tese de Doutorado. Foi uma grande honra para mim ter tido um conjunto de pesquisadores de excelência dispostos a dar sua contribuição a este trabalho.

Agradeço aos professores Flavia Coimbra Delicato e Paulo de Figueiredo Pires (UFRJ, Brasil), que participaram do comitê examinador da minha Qualificação de Doutorado. O resultado final deste trabalho possui importantes contribuições deles e pelas quais sou muito grato.

Agradeço aos amigos de perto e de longe, aos que vejo quase todos os dias, aos que vejo eventualmente, e aos que "vejo" apenas pela Internet. Infelizmente não vou poder citar o nome de todos, tanto pelo espaço disponível quanto pelo risco ou injustiça de esquecer de alguém, mas recebam aqui as minhas palavras de gratidão. Ainda assim, gostaria de mencionar os nomes de Gustavo Alves e Everton Lima, que sempre estiveram por perto, puderam compartilhar comigo diversos momentos e me ofereceram de sua preciosa amizade e apoio, fundamentais nesta trajetória. Muito obrigado.

Agradeço aos pesquisadores do grupo ArchWare – Equipe de Recherche sur les Architectures Logicielles do IRISA/Université Bretagne Sud por me receberem no período em que estive em Vannes, França, e com os quais pude aprofundar meus conhecimentos sobre arquiteturas de software. Em particular, agradecimentos especiais ao Jean Quilbeuf, que teve participação e contribuição direta neste trabalho, pela sua constante disponibilidade e paciência em sanar minhas dúvidas com relação a statistical model checking em arquiteturas de software. "Merci beaucoup".

Agradeço aos colegas e amigos do ICMC-USP (São Carlos, Brasil), os quais tive o prazer de conhecer no período em que estive em Vannes e estabelecer profícuas parcerias, especialmente Lucas Oliveira, Milena Guessi e Professora Elisa Yumi Nakagawa. Desse grupo, ainda necessito expressar minhas palavras de gratidão ao Marcelo Gonçalves, não só pela parceria em diversos trabalhos que conduzimos juntos, mas principalmente pela pessoa especial que é.

Por fim, agradeço às instituições que incentivaram a execução deste projeto e forneceram o apoio financeiro fundamental para o desenvolvimento deste trabalho: à Agência Nacional do Petróleo, Gás Natural e Biocombustíveis (ANP), por meio do Programa de Formação em Recursos Humanos em Geologia, Geofísica e Informática no Setor de Petróleo e Gás (PRH-22); à Financiadora de Estudos e Projetos (FINEP); ao Ministério da Ciência, Tecnologia, Inovações e Comunicações (MCTIC) da República Federativa do Brasil; ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), e; à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).

This work was developed with financial support from the following institutions:

- Brazilian National Agency of Petroleum, Natural Gas and Biofuels (ANP) through the Program for Human Resources in Geology, Geophysics and Information Technology in Oil & Gas (PRH-22)
- Brazilian Funding Authority for Studies and Projects (FINEP)
- Brazilian Ministry of Science, Technology, Innovations and Communications (MCTIC)
- Brazilian National Council for Scientific and Technological Development (CNPq)
- Brazilian Coordination for the Improvement of Higher Education Personnel (CAPES)

Success is born from the will, determination, and persistence to
accomplish a goal. Even though not reaching the target,
the one who seeks and overcomes obstacles
will do at least outstanding things.

José de Alencar, Brazilian writer (1829-1877)

O sucesso nasce do querer, da determinação e da
persistência em se chegar a um objetivo.
Mesmo não atingindo o alvo, quem busca e vence obstáculos,
no mínimo fará coisas admiráveis.

José de Alencar, escritor brasileiro (1829-1877)

A Formally Founded Framework for Dynamic Software Architectures

Author: Everton Ranielly de Sousa Cavalcante

Supervisors: Prof. Dr. Thais Vasconcelos Batista and Prof. Dr. Flavio Oquendo

ABSTRACT

Software architectures play a significant role in the development of software-intensive systems in order to allow satisfying both functional and non-functional requirements. In particular, dynamic software architectures have emerged to address characteristics of the contemporary systems that operate on dynamic environments and consequently subjected to changes at runtime. Architecture description languages (ADLs) are used to represent software architectures, producing models that can be used at design time and/or runtime. However, most existing ADLs have limitations in several facets: (i) they are focused on structural, topological aspects of the architecture; (ii) they do not provide an adequate support for representing behavioral aspects of the architecture; (iii) they do not allow describing advanced aspects regarding the dynamics of the architecture; (iv) they are limited with respect to the automated verification of architectural properties and constraints; and (v) they are disconnected from the implementation level, thus entailing inconsistencies between architecture and implementation. In order to tackle these problems, this thesis proposes formally founded framework for dynamic software architectures. Such a framework comprises: (i) π-ADL, a formal language for describing software architectures under both structural and behavioral viewpoints; (ii) the specification of programmed dynamic reconfiguration operations; (iii) the automated generation of source code from architecture descriptions; and (iv) an approach based on statistical model checking (SMC) to formally express and verify properties in dynamic software architectures. The main contributions brought by the proposed framework are fourfold. First, the π-ADL language was endowed with architectural-level primitives for describing programmed dynamic reconfigurations. Second, architecture descriptions in π-ADL are translated towards implementation source code in the Go programming language, thereby contributing to minimize architectural drifts. Third, a novel logic, called DynBLTL, is used to formally express properties in dynamic software architectures. Fourth, a toolchain relying on SMC was built to automate the verification of architectural properties while striving to reduce effort, computational resources, and time for performing such a task. In this work, two wireless sensor network-based systems are used to validate the framework elements.

Keywords: Software architectures, Architecture description languages, Dynamic reconfiguration, Formal verification, Temporal logic, Statistical model checking.

Um Framework Formal para Arquiteturas de Software Dinâmicas

Autor: Everton Ranielly de Sousa Cavalcante

Orientadores: Prof.ª Dra. Thais Vasconcelos Batista e Prof. Dr. Flavio Oquendo

RESUMO

Arquiteturas de *software* exercem um papel significativo no desenvolvimento de sistemas intensivos de *software* a fim de permitir satisfazer tanto requisitos funcionais quanto não-funcionais. Em particular, arquiteturas de *software* dinâmicas têm surgido para endereçar características dos sistemas contemporâneos que operam em ambientes dinâmicos e consequentemente sujeitos a mudanças em tempo de execução. Linguagens de descrição arquitetural (ADLs) são utilizadas para representar arquiteturas de *software*, produzindo modelos que podem ser utilizados tanto em tempo de projeto quanto em tempo de execução. Contudo, a maioria das ADLs existentes possui limitações em diversos aspectos: (i) possui enfoque em aspectos estruturais, topológicos da arquitetura; (ii) não provê um suporte adequado à representação de aspectos comportamentais da arquitetura; (iii) não permite descrever aspectos avançados relativos à dinâmica da arquitetura; (iv) é limitada com relação à verificação de propriedades arquiteturais e restrições, e; (v) é desconectada do nível de implementação, resultando em inconsistências entre arquitetura e implementação. No intuito de endereçar esses problemas, esta tese propõe um *framework* formal para arquiteturas de *software* dinâmicas. Tal *framework* envolve: (i) π-ADL, uma linguagem formal para descrever arquiteturas de *software* sob as perspectivas estrutural e comportamental; (ii) a especificação de operações de reconfiguração dinâmica programada; (iii) a geração automática de código fonte a partir de descrições arquiteturais, e; (iv) uma abordagem baseada em verificação estatística (SMC) para expressar e verificar formalmente propriedades em arquiteturas de *software* dinâmicas. As principais contribuições trazidas pelo *framework* proposto são quatro. Primeiro, a linguagem π-ADL passou a ser dotada de primitivas de nível arquitetural para descrever reconfigurações dinâmicas programadas. Segundo, descrições arquiteturais em π-ADL são traduzidas para código fonte de implementação na linguagem de programação Go, contribuindo assim para minimizar desvios arquiteturais. Terceiro, uma nova lógica chamada DynBLTL é utilizada para expressar formalmente propriedades em arquiteturas de *software* dinâmicas. Quarto, um ferramental baseado em SMC foi construído para automatizar verificação de propriedades arquiteturais enquanto busca reduzir esforço, recursos computacionais e tempo para realizar essa tarefa. Neste trabalho, dois sistemas baseados em redes de sensores sem fio são utilizados para validar os elementos do *framework*.

Palavras-chave: Arquiteturas de software, Linguagens de descrição arquitetural, Reconfiguração dinâmica, Verificação formal, Lógica temporal, Verificação estatística.

Un Framework Formel pour les Architectures Logicielles Dynamiques

Auteur : Everton Ranielly de Sousa Cavalcante

Directeurs de thèse : Dr Thais Vasconcelos Batista (MCF-HDR) and Pr Flavio Oquendo

RESUME

Les architectures logicielles ont un rôle important dans le développement de systèmes à logiciel prépondérant afin de permettre la satisfaction tant des exigences fonctionnelles que des exigences extra-fonctionnelles. En particulier, les architectures logicielles dynamiques ont émergé pour faire face aux caractéristiques des systèmes contemporains qui opèrent dans des environnements dynamiques et par conséquent susceptibles de changer en temps d'exécution. Les langages de description architecturale (ADLs) sont utilisés pour représenter les architectures logicielles en produisant des modèles qui peuvent être utilisés pendant la conception ainsi que l'exécution. Cependant, la plupart des ADLs existants sont limités sur plusieurs facettes : (i) ils ne décrivent que les aspects structurels, topologiques de l'architecture ; (ii) ils ne fournissent pas un support adéquat pour représenter les aspects comportementaux de l'architecture ; (iii) ils ne permettent pas de décrire des aspects avancés de la dynamique de l'architecture ; (iv) ils sont limités en ce qui concerne la vérification automatisée des propriétés et des contraintes architecturales ; et (v) ils sont déconnectés du niveau d'implémentation et entraînent souvent des incohérences entre l'architecture et l'implémentation. Pour faire face à ces problèmes, cette thèse propose un framework formel pour les architectures logicielles dynamiques. Ce framework comprend : (i) π-ADL, un langage formel pour décrire des architectures logicielles dynamiques sous les perspectives structurelles et comportementales ; (ii) la spécification des opérations de reconfiguration dynamique programmée ; (iii) la génération automatique de code source à partir des descriptions architecturales ; et (iv) une approche basée sur la vérification statistique pour exprimer et vérifier formellement des propriétés des architectures logicielles dynamiques. Les contributions principales apportées par le framework proposé sont quatre. Premièrement, le langage π-ADL a été doté de primitives de niveau architectural pour décrire des reconfigurations dynamiques programmées. Deuxièmement, les descriptions architecturales dans π-ADL sont transformées vers le code source d'implémentation dans le langage de programmation Go, en contribuant à minimiser les dérives architecturales. Troisièmement, une nouvelle logique appelée DynBLTL est utilisée pour exprimer formellement des propriétés dans les architectures logicielles dynamiques. Quatrièmement, un outil basé sur SMC a été développé pour automatiser la vérification des propriétés architecturales en cherchant à réduire l'effort, les ressources computationnelles, et le temps pour réaliser cette tâche. Dans ce travail, deux systèmes

basés sur réseaux de capteurs sans fil sont utilisés pour valider les éléments du framework.

Mots-clés : Architectures logicielles, Langages de description architecturale, Reconfiguration dynamique, Vérification formelle, Logique temporelle, Vérification statistique.

List of Figures

Figure 1 – Taxonomic dimensions to characterize dynamic reconfiguration approaches for software architectures 36

Figure 2 – Working schema of the SMC technique 46

Figure 3 – Deployment layout of the flood monitoring system in a flood-prone area in downtown São Carlos, Brazil (adapted from [70]) 50

Figure 4 – Flood monitoring system scenario 51

Figure 5 – Deployment of wireless sensors in a pipeline network 53

Figure 6 – Detection of leaks in pipelines by measuring flow debits 54

Figure 7 – Generic architecture for WSN-based monitoring 54

Figure 8 – Main architectural concepts of the π-ADL language 59

Figure 9 – Layered type system of π-ADL 60

Figure 10 – Life cycle for active software architectures (adapted from [88]) 70

Figure 11 – Illustration of the exogenous approach for the dynamic reconfiguration of a simple client-server architecture. A primary server is replaced by a secondary one in case of unavailability of the former 72

Figure 12 – Result of the endogenous approach for the dynamic reconfiguration of a simple client-server architecture. An auxiliary virtual machine is created and attached to the primary server in case of overloading. 73

Figure 13 – π-ADL description of the *Sensor* component 74

Figure 14 – π-ADL description of the *Gateway* component 75

Figure 15 – π-ADL description of the *ZigBee* connector 76

Figure 16 – π-ADL description of the *WSNFloodMonitoring* architecture 76

Figure 17 – Partial π-ADL description of the *WSNFloodMonitoringEvol* evolved architecture for realizing a reconfiguration aimed to replace a sensor mote due to low battery level by means of an exogenous approach. 77

Figure 18 – Partial π-ADL description for realizing a reconfiguration aimed to increase accuracy and avoid false positives by means of an endogenous approach. 79

Figure 19 – Screenshot of the Eclipse-based π-ADL textual editor 91

Figure 20 – Elements for generating Go source code from π-ADL architecture descriptions 92

Figure 21 – Description of the *Sensor* component in π-ADL (left) and corresponding implementation in Go (right) 94

Figure 22 – Description of the *Gateway* component in π-ADL (left) and corresponding implementation in Go (right) 95

Figure 23 – Description of the *ZigBee* connector in π-ADL (left) and corresponding implementation in Go (right) 96

Figure 24 – Description of the *WSNFloodMonitoring* architecture in π-ADL (left) and corresponding implementation in Go (right) 97

Figure 25 – Excerpt of Go source code generated from the π-ADL description of the *WSNFloodMonitoringEvol* evolved architecture, following an exogenous approach... 98

Figure 26 – Excerpt of Go source code generated from the π-ADL description of the *Gateway* component, following an endogenous approach 99

Figure 27 – Illustration of an execution trace for a simple client-server system 106

Figure 28 – Working schema of the scheduler to support the stochastic execution of a π-ADL architectural model 116

Figure 29 – Overview of the developed SMC-based toolchain for verifying properties expressed in DynBLTL regarding dynamic software architectures described in π-ADL 118

Figure 30 – Effect of the precision variation in the analysis of three properties upon analysis time (measured in seconds) 121

Figure 31 – Effect of the precision variation in the analysis of three properties upon RAM consumption (in megabytes) 121

Figure 32 – Main constituents of the proposed framework and correlation to goals and associated research questions 124

List of Tables

Table 1 – Base types defined in π-ADL$_{FO-BV}$ 60

Table 2 – Constructed types defined in π-ADL$_{FO-CV}$ 61

Table 3 – Collection types defined in π-ADL$_{FO-CT}$ 61

Table 4 – Process constructs defined in π-calculus 63

Table 5 – Behavior constructs defined in π-ADL 63

Table 6 – Comparative analysis of existing ADLs considering some taxonomic dimensions for characterizing dynamic reconfiguration approaches in software architectures 83

Table 7 – Summary of correspondences between the main architecture-level elements of π-ADL and implementation-level elements of Go 85

Table 8 – Summary of the mappings from data types defined in π-ADL to types in Go 88

Table 9 – Summary of the mappings from behavior constructs defined in π-ADL to Go 89

Table 10 – Summary of actions on a state graph $g = (V, E)$ 107

Table 11 – Descriptive statistics for execution time of the analysis (in seconds) 119

Table 12 – Descriptive statistics for RAM consumption (in megabytes) 120

List of Acronyms

AADL	Architecture Analysis and Design Language
ADL	Architecture description language
ANP	Agência Nacional do Petróleo, Gás Natural e Biocombustíveis
AST	Abstract syntax tree
BLTL	Bounded linear temporal logic
C2SADEL	C2 Software Architecture Description and Evolution Language
CDN	Cognitive Dimensions of Notations
CCS	Calculus of Communicating Systems
CIL	Common Intermediate Language
CNRS	Centre National de la Recherche Scientifique
CSP	Communicating Sequential Processes
DSL	Domain-specific language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
FSP	Finite State Processes
IEC	International Electronical Commission
IEEE	Institute of Electrical and Electronic Engineers
INRIA	Institut National de Recherche en Informatique et en Automatique
IRISA	Institut de Recherche en Informatique et Systèmes Aléatoires
ISO	International Organization for Standardization
LTL	Linear temporal logics
M2T	Model-to-Text
MWE	Modeling Workflow Engine
PoN	Physics of Notations
SMC	Statistical model checking
UBS	Université Bretagne Sud
UFRN	Universidade Federal do Rio Grande do Norte

UML	Unified Modeling Language
UMR	Unité Mixte de Recherche
WSN	Wireless sensor network

List of Symbols

General symbols

π	Lowercased *pi*, sixteenth letter of the Greek alphabet
$\vee$	Disjunction logical connective (*or*)
$\wedge$	Conjunction logical connective (*and*)
$\neg$	Negation logical connective (*not*)
$\exists$	Existential quantifier (*exists*)
$\nexists$	Negation of the existential quantifier (*not exists*)
$\equiv$	Equivalence (*is equivalent to*)
$\stackrel{\text{def}}{=}$	Definition (*defined as*)

Statistical model checking

M	Executable model of a system
φ, ψ	Property (formula)
ω	Simulation
σ	Execution trace
p	Probability value
θ	Threshold
$\vDash$	Satisfaction (e.g., A $\vDash$ B is read as A *satisfies* B)
$\nvDash$	Non-satisfaction (e.g., A $\nvDash$ B is read as A *does not satisfy* B)
B	Random variable following a Bernoulli probability distribution
Pr	Probability function
H, K	Hypotheses
α, β	Confidence bounds for hypotheses
ε	Precision
p'	Estimation for probability
δ	Probability value for estimation

Process algebras

P, Q	Processes
$a.P$	Sequential composition (read as *executing an action* a *and sequentially behaving as* P)
$c(x).P$	Input prefixing action (read as *reading a value* x *from a channel* c *and sequentially behaving as* P)
$\bar{c}(x).P$	Output prefixing action (read as *writing a value* x *in a channel* c *and sequentially behaving as* P)
τ	Unobservable action
$P \mid Q$	Parallel composition of processes *P* and *Q*
$P + Q$	Non-deterministic choice between processes *P* and *Q*
0	Inert process

State graphs for execution traces

g	State graph
V	Finite set of nodes
E	Finite set of edges
Val	Set of possible values to be exchanged between two nodes
v	Node in a state graph
e	Edge in a state graph
$v::c$	Connection of a node
C	Finite set of connections of a node
id	Unique identifier of a node
T	Type of a node

Temporal logics

t	Time instant
$\mathcal{F}$	*Finally* (or *eventually*) temporal operator
$\mathcal{G}$	*Globally* (or *always*) temporal operator
$\mathcal{U}$	*Until* temporal operator
$\mathcal{X}$	*Next* temporal operator

EBNF meta-symbols

$A \rightarrow B$	Definition of production rule (read as *A is defined as B*)
$A \mid B$	Alternative choice between elements *A* and *B*
[*X*]	Element *X* is optional in the production rule
*X**	Zero or multiple occurrences of element *X* in the production rule
X+	One or multiple occurrences of element in the production rule
A&*B*	Any occurrence order of elements *A* and *B* in the production rule
m…*n*	Range between characters *m* and *n*

Table of Contents

1 INTRODUCTION **26**

1.1 Problem statement **27**

1.1.1 Enhancing the representation of dynamic software architectures 27

1.1.2 Software architectures and their implementation: Two disconnected worlds 29

1.1.3 Analyzing software architectures: A challenging activity 31

1.2 Goals and research questions **32**

1.3 Contributions **33**

1.4 Outline **35**

2 BACKGROUND **36**

2.1 Characterizing dynamic software architectures **36**

2.2 The Go programming language **37**

2.2.1 Fundamentals 38

2.2.2 Control statements 40

2.2.3 Arrays and slices 41

2.2.4 Interfaces 41

2.2.5 Concurrency support 42

2.3 Statistical model checking **44**

3 RUNNING APPLICATIONS: WIRELESS SENSOR NETWORK-BASED SYSTEMS **48**

3.1 A flood monitoring system **48**

3.2 Monitoring oil and gas pipelines **51**

3.3 An architecture for WSN-based monitoring systems **54**

3.4 The dynamic scenario of WSNs **55**

4 THE π-ADL ARCHITECTURE DESCRIPTION LANGUAGE **57**

4.1 Architectural abstractions **58**

4.2 Type system **59**

4.2.1 Base types 60

4.2.2 Constructed types 60

4.2.3 Collection types 61

4.3 Behavior constructs **62**

4.4 Specifying architectural elements in π-ADL **65**

4.4.1 Specifying behavior of components and connectors 65

4.4.2 Statements 67

4.4.3 Specifying architectural configurations 68

4.5 Dynamic software architectures in π-ADL 69
4.6 Describing the flood monitoring system in π-ADL 73
4.6.1 Architectural elements 73
4.6.2 Exogenous reconfiguration: Low battery of a sensor node 77
4.6.3 Endogenous reconfiguration: Avoiding false positives 78
4.7 Related work: Languages for describing dynamic software architectures 80

5 ARCHITECTURE-BASED CODE GENERATION 84
5.1 Correspondences between π-ADL and Go 85
5.2 Code generation procedure 89
5.2.1 π-ADL textual editor 90
5.2.2 Code generation procedure 92
5.3 Generating code for the flood monitoring system 93
5.3.1 Architectural elements 94
5.3.2 Exogenous reconfiguration 97
5.3.3 Endogenous reconfiguration 99
5.4 Related work: Supporting the implementation of software architectures 100

6 VERIFYING DYNAMIC SOFTWARE ARCHITECTURES 103
6.1 Representing traces of dynamic software architectures 104
6.2 Expressing properties about dynamic software architectures 107
6.2.1 Underlying formalisms for expressing properties 107
6.2.2 A novel logic and notation for expressing properties in dynamic software architectures 109
6.2.2.1 DynBLTL elements 110
6.2.2.2 Example 113
6.3 Statistical model checking of π-ADL architectural models 115
6.3.1 Stochastic execution of π-ADL architecture descriptions 115
6.3.2 An SMC-based toolchain to simulate and verify dynamic software architectures 117
6.3.3 Quantitative evaluation 118
6.4 Related work: Formal specification and verification of architectural properties in dynamic systems 121

7 CONCLUSION 123
7.1 Revisiting the proposal and its contributions 124
7.2 Future work 126
7.2.1 Short-term work 126
7.2.2 Long-term work 126

REFERENCES 128

APPENDIX A – π-ADL GRAMMAR 142
A.1 Grammar notation 142
A.2 π-ADL production rules 144
A.3 References 149

APPENDIX B – THE π-ADL TEXTUAL EDITOR 150
B.1 Preliminaries 150
B.2 The Xtext-based π-ADL textual editor 151
B.2.1 The π-ADL grammar 151
B.2.2 Automatic generation of the π-ADL infrastructure 153
B.2.3 Validations 154
B.2.4 Interpreting expressions 156
B.2.5 Features of the π-ADL textual editor 157
B.3 References 158

APPENDIX C – DYNBLTL NOTATION 159
C.1 Grammar notation 159
C.2 DynBLTL production rules 160
C.3 References 161

APPENDIX D – LIST OF PUBLICATIONS 162
D.1 Publications resulted from this work 162
D.2 Correlated publications 162

1 Introduction

The increasing complexity of software development and the demand for quality have called for systematic approaches to engineer software systems. Due to its inherently laborious nature, software development has been a quest for more powerful design abstractions to help engineers and developers to build even larger, more complex systems and to reason about their structure, behavior, and properties [1]. Nowadays, software systems with these characteristics are the norm rather than the exception and hence they need coarser-grained abstractions to tame their complexity while guiding the construction effort.

In this context, *software architectures* have emerged as the backbone of any successful software-intensive system, thereby contributing to the achievement of both business goals and quality requirements as well as being manageable, meaningful abstractions of the system under development [2, 3, 4]. The ISO/IEC/IEEE 42010 International Standard [5] defines 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 guiding the system design and evolution. Applied throughout the software life cycle, good architectural practice has the potential of (i) increasing the understandability of the system and the development process used to create it, (ii) ensuring the satisfaction of requirements, and (iii) reducing the overall cost of the software development process [6]. Therefore, a software architecture can be used as a relevant artifact in activities such as requirements specification, system design and analysis, successive model refinements towards implementation, reuse, maintenance, and runtime adaptation [7]. It also captures and preserves designers' intentions about system structure and behavior thereby providing a defense against design decay as a system ages [8].

The literature traditionally distinguishes two main types of software architectures according to their evolution upon changes in their environment, namely *static* and *dynamic* software architectures [9, 10, 11]. The architecture of a software system is said to be static if it is not subjected to changes during runtime. On the other side of the spectrum, dynamic software architectures are those that encompass evolution rules for a software system and its elements during runtime. The latter case is of particular importance for this work as dynamism is an important concern for contemporary systems, which often operate on environments that are dynamic, subjected to changes. Moreover, support for dynamism is important mainly in the case of certain safety- and mission-critical systems, such as air traffic control, energy, disaster management, environmental monitoring, and health systems. Systems in these scenarios are required to maintain a high level of availability, so that stopping

them is not an option due to financial costs, physical damages, or even threats to life and safety of people.

Considering dynamicity and dependability concerns while conceiving the architecture of a software-intensive system has grown in importance due to the complexity of emerging applications, mainly in critical domains such as the aforementioned ones. In this context, software architectures can document and allow reasoning about changes that might occur during the system execution and provide a basis for the evolution of the system [12, 13]. Software architectures should also allow for a flexible, extensible creation, interconnection, and/or removal of constituent elements and connections or even a whole rearrangement of such elements with minimal or no disruption. Nevertheless, these concerns are often handled late in the development process, thus making a system without an adaptable architecture degenerate sooner than a system based on an architecture that takes changes into account.

1.1 Problem statement

1.1.1 Enhancing the representation of dynamic software architectures

The representation of software architectures is one of the main activities of an architecture-driven software development process as it allows anticipating important decisions regarding the system design. This activity results in *architecture descriptions*, the set of artifacts expressing a software architecture and making it tangible [5], even though such a representation lies at a high abstraction level. Architecture descriptions play an essential role as the main means of communication among stakeholders, e.g., architects, developers, etc. At the same time, the precise communication of this artifact is one of the most complex and expensive tasks in software architecture design. As stated by Lago et al. [14], a badly specified architecture design causes design and implementation flaws in a software system and can create misunderstanding.

Several studies in the literature rely on architecture descriptions to support the documentation, maintenance, evaluation, and evolution of software architectures. In this context, *architecture description languages* (ADLs) have become well-accepted means for systematically representing and analyzing software architectures, thereby producing models that can be used at design time and/or runtime [15, 16]. According to the ISO/IEC/IEEE 42010 International Standard [5], an ADL is any form of expression used to support the representation of a software architecture. ADLs emerged since the 1990s mainly resulting from the research devoted to the problem of providing more precise ways to characterize the structure and behavior of software architectures as well as to derive properties on these structures [7]. However, the last twenty years of research on ADLs have witnessed a proliferation of languages due to

a series of reasons, in particular (i) the lack of a common agreement on which architectural aspects shall be documented by these languages, (ii) the misunderstanding of merits and limitations of existing notations, (iii) the variety of stakeholder concerns to be considered, and (iv) the trade-off between formality and understandability [17, 18]. Indeed, Malavolta et al. [18] state that an ideal, general-purpose ADL is not likely to exist and hence ADLs must be able to focus on what is needed by the stakeholders involved in the architecting process.

Classically, ADLs have been classified into three broad categories: (i) *formal*, i.e., typically textual notations with precise (often mathematically-based) syntax and semantics that support automated architectural analysis; (ii) *semi-formal*, i.e., notations with well-defined syntax, but lack of complete semantics; and (iii) *informal*, i.e., ad-hoc box-and-lines diagrams that cannot be formally analyzed and limit the usefulness of the architecture description [2, 16]. The selection of an appropriate formalism level is one of the most important tasks when creating architecture descriptions as this decision should be aligned with their expected uses, such as documentation or evaluation. In a recent survey about the use of ADLs in industry, Malavolta et al. [18] observed that, from practitioners' point of view, these languages should support the definition of functional and non-functional properties, formal semantics for improving precision and allowing automated analysis, and both graphical and textual representations for easing the communication among stakeholders and users of architecture descriptions. Additionally, Medvidovic and Taylor [17] highlight that an important source of discord is the level of support that an ADL should provide to developers. On the one hand, it can be argued that the primary role of architecture descriptions is to support understanding and communication about a software system, thus requiring ADLs to have a simple, understandable syntax and well-understood (but not necessarily formally defined) semantics. On the other hand, the tendency has been to endow ADLs with a more formal syntax and semantics, powerful analysis tools, code synthesis mechanisms, etc. Even though researchers have generally adopted one of these extremes, it is acknowledged that both are important and should be reflected in an ADL [14].

Regardless the diversity of existing ADLs, the description of software architectures is commonly characterized by two viewpoints, namely the *structural* and *behavioral* viewpoints. The structural viewpoint is concerned with the structure of the system in terms of three main building blocks: (i) *components*, units of computation representing functional elements of the system; (ii) *connectors*, which represent the interconnections among components supporting their interactions; and (iii) architectural *configurations* determining the way in which components and connectors can be interwoven forming the software architecture itself [17]. Relevant features characterizing components and connectors typically encompass interfaces defining the

interaction points between these elements and the environment, high-level models of their behavior, and constraints limiting their usage. In turn, the behavioral viewpoint is related to the internal operation (actions) of components and connections as well as the interaction among these elements.

A proper description of dynamic software architectures using ADLs faces two main problems. First, most existing ADLs focus only on the structural viewpoint by providing means of describing topological aspects of the software architecture, as it is the case of well-known languages such as xADL [19] and Acme [20]. On the other hand, few ADLs (such as Wright [21] and Darwin [22]) support the representation of behavioral aspects of the system. The second issue is that ADLs for describing dynamic software architectures must provide elements and mechanisms for specifying the changes to be performed over the architecture as well as realizing these changes at runtime [17]. However, the literature reports few ADLs supporting an expressive description of dynamic aspects of software architectures. Examples of these languages are Darwin [22], Dynamic Wright [23], and RAPIDE [24].

Therefore, the first problem addressed in this work can be stated as follows:

Problem I:
Most existing ADLs are not able to properly describe dynamic software architectures at both structural and behavioral viewpoints as well as the changes that can be performed over the architecture and its constituent elements.

1.1.2 Software architectures and their implementation: Two disconnected worlds

In their well-known book about the Software Architecture discipline, Taylor et al. [11] introduce the notions of *prescriptive* and *descriptive* architectures. A *prescriptive architecture* is an as-intended, as-conceived architecture for a software system comprising the design decisions made by the architects in order to reflect their intent. Such a prescriptive architecture does not need to exist in any tangible form, but it can be captured by a notation such as an ADL. Nonetheless, documenting the prescriptive architecture of a software system is not enough. It needs to be realized with a set of artifacts that may include further refinements of architectural design decisions towards their implementations in a programming language. This set of artifacts embodying and realizing design decisions is referred to as a *descriptive architecture*, the as-implemented architecture showing how the system has actually been built.

During the lifespan of a software system, a large number of prescriptive and descriptive architectures can be created. Each corresponding pair of such architectures represents the system's software architecture at a given time in terms of design decisions and the artifacts realizing them. As explained by Taylor et al. [11], these

architectures would always be identical in an ideal scenario, i.e., the descriptive architecture would be a perfect realization of the prescriptive architecture. However, this rarely happens in practice. When evolving a software system, its descriptive architecture is often directly modified while its prescriptive architecture should have been ideally modified first. Such a problematic divergence between the prescriptive and descriptive architectures of a software system has been referred to as *architectural drift*, as introduced by Perry and Wolf [25] in the early 1990s. An architectural drift is a form of architectural degradation characterized by the introduction of design decisions orthogonally to a system's prescriptive architecture, that is, they are not included in, encompassed by, or implied by the prescriptive architecture, even though not violating it [11]. Such a drift results from direct changes to the artifacts realizing the software architecture, e.g., changing the implementation of the architecture without accounting for the impact relative to its original conception. Possible reasons for this problem are developers' sloppiness (i.e., they merely do not want to modify the prescriptive architecture after modifying the descriptive architecture), perception of short deadlines, need or desire for code optimizations, etc.

Considering ADLs as notations able to describe prescriptive architectures, architectural drifts may arise due to the gap often observed between architecture descriptions and their respective implementations. As software architectures are typically defined independently from implementation, the decoupling between these levels leads to inconsistencies between the architecture and its corresponding implementation, mainly as the architecture evolves. Consequently, even if a system is built in conformance to the previously defined architecture, its implementation may become inconsistent with respect to such an original architecture along the time. Not providing a way of avoiding architectural drifts and inconsistencies between architecture description and their implementations will ultimately lose all advantages of designing an appropriate architecture. These inconsistencies may also lead to increased maintenance time and cost as the original design aims have been lost [26].

The discrepancies between architecture descriptions and implementation may become worse due to the emergence of new-generation programming languages, which aim at making use of concurrency, distribution, and multicore computer architectures. Existing mainstream programming languages such as C++ and Java do not well support these features and hence increase the required complexity for constructing large-scale and dynamic software systems, which are becoming typical in several application domains. In addition, the main problem in this new context is that most ADLs available in the literature do not properly capture these new features of modern programming languages, thereby making the bridge between architecture description and implementation harder to build.

The second problem addressed in this work can be summarized as:

Problem II:
There is a significant gap between architecture descriptions and their corresponding implementation, mainly in the new context of evolving software architectures and contemporary programming languages.

1.1.3 Analyzing software architectures: A challenging activity

One of the major challenges in the design of software-intensive systems consists in verifying the correctness of their software architectures, i.e., if the envisioned architecture is able to fully realize the established requirements. Ensuring correctness and other relevant system properties becomes more important mainly when evolving these systems since such a verification needs to be performed before, during, and after evolution. In this context, software architectures play an essential role since they represent an early blueprint for the system construction, deployment, execution, and evolution, thereby fostering an early analysis of a system and contributing to reduce the cost of software maintenance. Such an *architectural analysis* refers to the activity of discovering important system properties using architectural models [11].

Architecture descriptions should not cover only structure and behavior of a software architecture, but also the required and desired architectural properties, in particular the ones related to consistency and correctness [27]. For instance, after describing a software architecture, a software architect might want to verify if it is complete, consistent, and correct with respect to architectural properties. The requirements to be verified are typically concerned with the relationship between the system behavior (e.g., receiving or sending a particular value) and an architectural property, such as checking if a component is connected to or disconnected from another component. For illustrative purposes, consider a sensor-based system in which sensors measure values from the environment and transmit it to a base station, possibly via other sensors. A requirement of interest in this context would be that a sensor signaling a low battery level (a behavioral property) gets disconnected from the other sensors (an architectural property).

Due to the critical nature of many complex software systems, rigorous architectural models (such as formal architecture descriptions) are quite desirable as means of better supporting automated architectural analysis. Despite the inherent difficulty of pursuing formal methods [28], the main advantage of a formal verification is to precisely determine if a software system can satisfy properties related to user requirements. Additionally, automated verification provides an efficient method to check the correctness of architectural design. As reported by Zhang et al. [29], one of the most used techniques for analyzing software architectures is *model checking*, an exhaustive, automatic verification technique whose general goal is to verify if an architectural specification satisfies architectural properties [30]. It takes as inputs a

representation of the system (e.g., an architecture description) and a set of property specifications expressed in some notation. The model checker returns true, if the properties are satisfied, or false with the case in which a given property is violated. The input model of the system is usually expressed as a finite state machine, i.e., a directed graph consisting of vertices and edges. A vertex represents a state with a set of held atomic propositions (properties) whereas an edge represents a possible execution that changes the system's state. If the input model is finite, the model checking problem is reduced to a graph search [11].

Despite its wide and successful use, model checking faces a critical challenge with respect to scalability. Holzmann [31] remarks that no currently available traditional model checking approach is exempted from the state space explosion problem, that is, the exponential growth of the state space. This problem is exacerbated in the contemporary dynamic software systems for two main reasons, namely (i) the non-determinism of their behavior caused by concurrency and (ii) the unpredictable environmental conditions in which they operate. In spite of the existence of a number of techniques aimed at reducing the state space, such a problem remains intractable for some software systems, thereby making the use of traditional model checking techniques a prohibitive choice in terms of execution time and computational resources. As a consequence, software architects have to trade-off the risks of possibly undiscovered problems related to the violation of architectural properties against the practical limitations of applying a model checking technique on a very large architectural model [11].

Finally, the third problem addressed in this work can be described as:

Problem III:
Traditional techniques for formally verifying properties in dynamic software architectures are not scalable, computationally efficient.

1.2 Goals and research questions

Aiming to tackle the problems elicited in Section 1.1, the main general goal of this work is to propose a formally founded framework[1] to support dynamic software architectures. This general goal will be achieved through the satisfaction of the

[1] In this work, the definition for *framework* follows the statements found in the ISO/IEC/IEEE 42010 International Standard [5]: *An architecture framework establishes a common practice for creating, interpreting, analyzing and using architecture descriptions within a particular domain of application or stakeholder community. Uses of architecture frameworks include, but are not limited to: create architecture descriptions; developing architecture modeling tools and architecting methods; and establishing processes to facilitate communication, commitments and interoperation across multiple projects and/or organizations.*

following sub-goals:

G1: To use a formal ADL for describing dynamic software architectures under both structural and behavioral viewpoints as well as specifying dynamic reconfiguration operations.

G2: To fill the existing gap between the description of dynamic software architectures using ADLs and their respective implementation.

G3: To propose an efficient approach for verifying properties in dynamic software architectures.

In order to achieve the general goal and its sub-goals, the following research questions (RQs) need to be answered:

RQ1. How to describe dynamic software architectures while fostering their rigorous analysis with respect to both functional and non-functional properties? The aim of this RQ is to investigate the use of a formal ADL and how it can be expressive for describing both structure and behavior dynamic software architectures while paving the way for formally verifying architecture descriptions.

RQ2. How it is possible to minimize the risk of architectural drifts in dynamic software architectures? This RQ aims at investigating how architecture-based code generation can contribute to fill the existing gap between architecture descriptions and their respective implementation, thereby minimizing the risk of architectural drifts. The main premise here is to automatically generate implementation source code from architecture description expressed in an ADL as means of tackling the gap between these levels and maintaining them consistent with each other. Moreover, executing the code resulting from such an architecture-to-implementation mapping can be a useful way of fostering the validation of the specified architecture.

RQ3. How to reduce the required effort for verifying properties in dynamic software architectures? Finally, this RQ aims to investigate how it is possible to reduce effort, computational resources, and time for formally verifying properties in dynamic software architectures. Being apart from traditional exhaustive approaches available in the literature, the main concern is to achieve an efficient approach able to promote better scalability and less consumption of computational resources, important concerns to be considered when analyzing software architectures of complex critical systems.

1.3 Contributions

The proposed framework comes up with four main contributions, each one summarized in the following. In this work, each of these elements was validated using two wireless sensor network-based systems.

An ADL able to formally describe dynamic software architectures. The first contribution brought by this work is the endowment of π-ADL [9], a formal language to describe software architectures under both structural and behavioral viewpoints, with architecture-level primitives for specifying *programmed reconfiguration* operations, i.e., foreseen, pre-planned changes described at design time and triggered at runtime by the system itself under a given condition or event [32]. In addition, two common approaches for enacting programmed dynamic reconfiguration [33, 34] were incorporated into π-ADL. The first approach is *exogenous*, in which it is possible to control all elements of the software architecture and apply the changes on the whole structure. In turn, the second approach is *endogenous*, in which the architectural elements themselves are able to manage dynamic reconfiguration actions. This culminates into an expressive language able to describe both structure and behavior of a dynamic software architecture, as well as the reconfiguration operations that can be applied over it at runtime [35].

An automated process to generate source code from architecture descriptions. The second main contribution of the proposed framework is the mapping of architecture descriptions in the π-ADL language to implementation source code in Go [36], an easy, modern general-purpose programming language designed to address the construction of scalable distributed systems as well as to handle multicore and networked computer architectures, as required by contemporary systems. Go was chosen to serve as implementation language because it relies on the same underlying formalism of π-ADL, thus fostering a straightforward relationship between elements of these languages. Such a mapping process resulted in an automated process for generating source code from an architecture description [35, 37], thereby contributing to minimize the risk of architectural drifts and allowing for the validation of the architecture itself.

An architecturally-driven, computationally efficient approach and toolchain for verifying properties in dynamic software architectures. The third contribution regards the use of *statistical model checking* (SMC) to support the formal analysis of dynamic architectures expressed in the π-ADL language [38]. SMC is a probabilistic, simulation-based technique intended to verify, at a given confidence level, if a certain property is satisfied during the execution of a system [39]. Unlike conventional formal verification techniques (such as model checking), SMC does not suffer from the state space explosion problem as it does not analyze the internal logic of the target system [40]. In a nutshell, SMC executes a stochastic model of the system under verification multiple times, so that the validity of the properties is probabilistically verified in each of these simulations. To support the verification process, a toolchain was developed upon a flexible, modular statistical model checker while striving to reduce effort, computational resources, and time for performing such a task. As far as it is concerned,

this is the first work on the application of SMC to verify properties in dynamic software architectures.

A novel logic and notation for formally expressing properties in dynamic software architectures. SMC has been applied to verify *bounded properties*, i.e., properties that can be defined in terms of finite executions of the system under verification. Besides a system model whose execution is probabilistic, stochastic, SMC requires a language for expressing the properties to be verified. Expressing properties regarding a dynamic software architecture needs to take into account architectural elements that are created or removed at runtime, i.e., they may exist at a given instant in time and no longer exist at another. As the existing notations available in the literature are not able to cope with such a characteristic, the fourth contribution is DynBLTL, a novel logic and notation aimed to express properties in dynamic software architectures [41]. In particular, DynBLTL was designed to handle the absence of an architectural element in a given bounded formula expressing a property.

1.4 Outline

The remainder of this doctoral dissertation is organized as follows. Chapter 2 establishes the background underlying this work. Chapter 3 briefly describes the two wireless sensor network-based systems that will be used hereinafter, more specifically in the presented examples. Chapter 4 introduces the π-ADL language, its formal underpinnings and main elements, and how it can be used for describing dynamic software architectures. Chapter 5 presents the approach for automatically generating source code from architecture descriptions expressed in the π-ADL language as well as the implementation intended to support such a process. Chapter 6 describes the SMC-based approach to support the formal specification and automated verification of properties in dynamic software architectures. Chapter 7 revisits the achieved contributions and presents perspectives for future work.

Four appendices are also provided as additional material. Appendix A presents the grammar specification for the π-ADL language. Appendix B details the tool developed to support architectural description and architecture-based code generation. Appendix C presents the grammar specification for the DynBLTL notation. Finally, Appendix D contains a list of publications resulted from this work and other correlated publications.

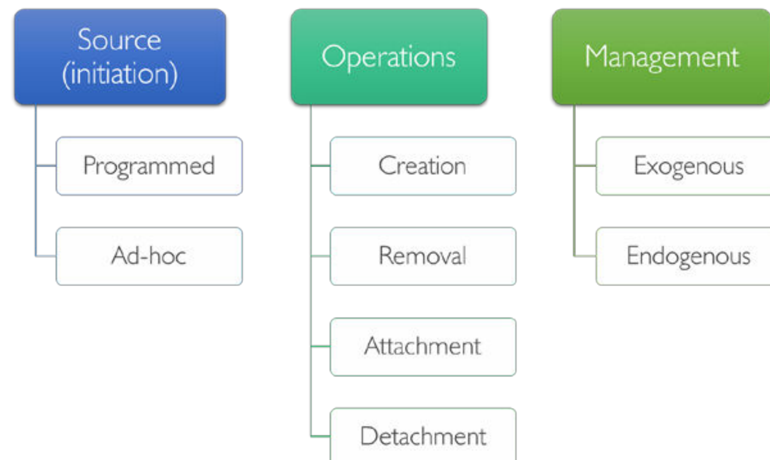
2 Background

This chapter presents the conceptual foundations underlying this work. Section 2.1 provides a characterization about dynamic reconfiguration approaches for software architectures often found in the literature. Section 2.2 gives an overview of Go, the programming language used as target of the aforementioned architecture-based code generation process. Finally, Section 2.3 presents some important concepts related to statistical model checking, the technique used in this work for verifying properties in dynamic software architectures.

2.1 Characterizing dynamic software architectures

Wermelinger [33] and Bradbury [42] provide relevant characterizations of dynamic reconfiguration approaches for software architectures. Figure 1 illustrates the main taxonomic dimensions presented by these authors and considered in this work.

Figure 1 – Taxonomic dimensions to characterize dynamic reconfiguration approaches for software architectures



Source (or initiation). Dynamic reconfiguration of software architectures can be either *programmed* or *ad-hoc*. In programmed (a.k.a. foreseen or proactive) reconfigurations, changes are pre-planned, foreseen at design time and applied at runtime under a given condition or event [32]. Therefore, the software architect specifies *when* such changes will be realized and *which* operations must be realized. In turn, ad-hoc (a.k.a. unforeseen or reactive) reconfiguration stands for changes that occur at runtime without being previously planned. These changes typically come from agents that are external to the architecture (e.g., the user) and they are applied through an interface of the system with the environment in which it is deployed. It is noteworthy that most of the works about dynamic software architectures addresses programmed reconfiguration [43]. Both programmed and ad-hoc reconfigurations

have their advantages and drawbacks, thus making them complementary approaches that shall be supported by architectural approaches. While programmed reconfigurations can enable a system to be autonomically reconfigured in response to certain conditions, ad-hoc reconfigurations can allow applying changes/updates without necessarily foreseeing them in advance. Nevertheless, it is practically impossible to predict all possible operations that might be required by a system to be reconfigured. On the other hand, ad-hoc reconfigurations must be somewhat constrained, carefully applied in order to avoid architectural erosion.

Operations. In spite of the several nomenclatures adopted in different works in the literature, reconfiguration operations to be applied on the architectural elements of a software system are essentially four [23, 32, 44]: (i) *creation* of instances of architectural elements; (ii) *removal* of instances of architectural elements; (iii) *attachment* of architectural elements; and (iv) *detachment* of architectural elements.

Management. The management of the reconfiguration process can be either centralized with a special entity or distributed across architectural elements. The so-called *exogenous dynamism* refers to the existence of a central entity (e.g., a configuration manager) that has control over all architectural elements and it is responsible for applying the reconfiguration actions on the architecture. In turn, the *endogenous dynamism* stands for the decentralization of the dynamic reconfiguration process, in which the architectural elements themselves are able to perform the reconfiguration actions. The main disadvantage of the exogenous dynamism is the centralization of the reconfiguration process, so that the entity responsible for it might become a bottleneck at the implementation level and reduce the performance of the architecture at runtime. In addition, reconfiguration actions associated to different architectural elements may be tangled, each one requiring a specific set of operations that must be described and performed independently from each other [34]. However, in the endogenous approach, reconfiguration actions may also be tangled with the functionality/behavior of architectural elements, thus hampering reuse and maintainability. Moreover, the endogenous dynamism requires architectural elements to have knowledge about each other, so that one could regard this as a violation of the basic features of a component or a connector. Nevertheless, it is necessary to establish some sort of trade-off between these dynamism approaches according to each specific scenario and architecture.

2.2 The Go programming language

Go [36] is a new, evolving general-purpose programming language launched as an internal project at Google, Inc. in 2007 and became open-source in November 2009. In 2012, Go was stably released as Go 1 by including a language specification [45], standard libraries, and custom tools. At the time of writing, the language is

currently in version 1.6.2, released in April 2016. In the last years, Go has been used by Google and a variety of commercial and noncommercial organizations and it is also integrated into the Google Cloud Platform [46].

Go was designed to address the construction of new-generation large-scale software systems, which are to be efficient, dynamic, and deployed on multicore and networked computer architectures. In order to achieve these purposes, Go aims at combining the lightweight, ease of use, and expressiveness of interpreted and dynamically typed languages (e.g., JavaScript and Python) with the efficiency and safety of traditional statically typed, compiled languages such as C and Java. Moreover, it is possible to directly compile even a large Go program to native code in few seconds, thereby fostering the development of large software systems. Go maintains a resemblance with the C-syntax so as to be immediately familiar with a majority of developers, but its syntax is greatly simplified and made more concise and clean in comparison to C/C++.

For the sake of space, this section briefly reviews Go by presenting the essential constructs of the language, in particular the elements used in the automated generation of source code in Go from architecture descriptions expressed in π-ADL. More details about the main elements of the language and its syntax can be found in the official language specification [45] and recent books about it [47, 48, 49].

2.2.1 Fundamentals

A Go source file basically consists of three parts:

- `package` *statement.* Go code is arranged in packages, similar to both libraries and header files in C. Every Go program must contain a `main` package containing a `main` function, the entry point from which the program always starts.
- `import` *statements.* This part specifies the packages that the current source file uses and how they should be imported. Imported packages contain types, variables, constants and functions.
- *Declarations.* The remainder of a Go code contains declarations of types, variables, and functions.

Consider the following Go code:

```
package main
import “fmt”

func main() {
    fmt.Println(“Hello, world!”)
}
```

This program simply prints “`Hello, world!`” on the console. The import instruction tells Go that the program needs (functions, or other elements, from) the `fmt` package, which implements functionalities for formatted input/output. In the `main` function, the `Println` function available at the `fmt` package is called to print the message (received as argument) on the console. The program exits immediately and successfully when the `main` function ends.

A Go code can also contain *functions*, which typically appear out of the main function[2]. Using the `func` keyword, a function can be declared taking zero or more arguments and returning any number of values. The simplest function declaration has the format:

```
func functionName() {
    // function body
}
```

with its body enclosed by braces. The execution of a function is stopped when the closing brace is reached or when a return statement is encountered. Afterwards, the execution of the program continues with the line following the call of the function. Although `main` is a function, it is important to highlight that it must have no arguments and no return values.

Variables contain data and can be of different types. In Go, variables are declaring using the `var` keyword along with the variable name and type, at package or function level. For example, the instruction

```
var a int
```

declares an integer variable named `a` and initialized with the default null value (in this case, zero). It is important to note is that the type is written after the identifier of the variable, contrary to almost any other programming language. A variable declaration can include initializing values (one per variable), but the type can be omitted if the initializer is present since the variable will take the type of the initializer. For instance, the instruction

```
var a = 1
```

declares the variable as integer (type of the initializer) and with value `1`. Inside a function, the short assignment statement (`:=`) can be used instead of a `var` declaration with explicit type. When declaring a variable without specifying an explicit type (either by using a short assignment or a typical variable declaration), the variable’s type is inferred from the value on the right hand side. As an example, the instruction

```
a := 1
```

[2] Go also allows declaring *closures*, i.e., functions inside other functions (including the `main` function).

declares a variable named `a` and with value `1`. The type of `a` is `int` (integer), inferred from the value on the right hand side of the short assignment statement.

2.2.2 Control statements

Control statements in the Go language are mostly similar to those in C/C++ and Java, but they are slightly different. For example, control statements in Go do not require parentheses and braces are mandatory even there is only one statement in the body. As an example, the *isGreater* function below compares the values of two integers received as arguments:

```
func isGreater(x, y int) bool {
    if x > y {
        return true
    } else {
        return false
    }
}
```

Testing different conditions and executing different statements in each case can be done with the selection statement, which is useful mainly to avoid several if-else statements. Compared to C/C++ or Java, this statement is more flexible in Go since the cases to be tested do not need to be constant values, but they must be of the same type or expressions evaluating to that type. In addition, more than one value can be simultaneously tested in a case. Consider the following excerpt of a Go program:

```
num := 7
switch {
    case num < 0:
        fmt.Println("Number is negative")
    case num >= 0 && num <= 10:
        fmt.Println("Number is between 0 and 10")
    default:
        fmt.Println("Number is greater than 10")
    }
}
```

The `switch` statement executes different tests based on the value of the variable `num`. The cases are evaluated from top to bottom, automatically breaking when a case succeeds. The `default` branch is optional and it is executed if the previous cases have not succeeded.

Although having conditional (if-else) and selection (switch) statements, Go has only one iterative construct, the `for` loop. The basic `for` loop has three elements, separated by semicolons: (i) an *init* statement, which is executed before the first iteration; (ii) a *condition* expression (Boolean), which is evaluated before each iteration and determines when the loop should terminate; and (iii) a *post* statement, which is executed after each iteration. Unlike other languages, there are no parentheses surrounding these elements and braces are always required. Moreover, the init and

post statements are optional. In a `for` loop, the `break` and `continue` instructions can alter the behavior of the loop. For example, consider the following Go program:

```
package main
import "fmt"

func main() {
    for i := 1; i <= 5; i++ {
        fmt.Printf("This is the %d iteration\n", i)
    }
}
```

The body of the loop is repeated for a known number of times as counted by the variable `i`. The loop starts with an *initialization* for `i` as a short assignment statement (`i := 1`), followed by a *conditional check* on the value of `i` (`i <= 5`) performed before each iteration: when it is true, the iteration is done, otherwise the loop is stopped. Next, a *modification* of `i` (`i++`) is performed after each iteration, at which point the condition is checked again.

2.2.3 Arrays and slices

The *array* type notation is well-known in almost every programming language as the basic workhorse in applications. The Go array is similar to those found elsewhere, but it has a few peculiarities. Arrays in Go are not dynamic and are somewhat inflexible in that they are a fixed-length sequence of elements of the same type. The expression

```
var a [10]int
```

declares a variable `a` as an array of ten integers. The items contained into the array can be accessed (and changed) through their index (the position), which starts from zero.

As an array's length is part of its type, arrays cannot be resized. However, Go provides a convenient way of handling dynamicity with arrays, the so-called *slices*. By definition, a slice is a reference to a contiguous segment of an underlying array, thus not requiring additional memory and being more efficient to use than arrays. Unlike an array, the length of a slice can dynamically change during the execution of a program. The expression

```
var a []int
```

declares a variable `a` as a slice to hold integer values.

2.2.4 Interfaces

Despite borrowing some concepts from the object-oriented programming paradigm, Go is not a classic object-oriented language in the same sense of C++ and Java because it does not have the notions of classes and inheritance. However, Go has a concept of *interfaces*, which allow for polymorphism.

Interfaces in Go are abstract representations of behavior and define sets of method signatures, i.e., they are abstract and do not have a body. Therefore, any object *implements* (or *satisfies*) an interface if it implements the methods of such an interface, but no explicit annotation is required. This implicit relationship allows decoupling the definition of an interface from its implementation. For illustrative purposes, consider the following excerpt of a Go program:

```
type Shape interface {
    area() int
}

type Rectangle struct {
    length, width int
}

type Square struct {
    side int
}

func (r Rectangle) area() int {
    return r.length * r.width
}

func (s Square) area() int {
    return s.side * s.side
}
```

In this example, the `Shape` interface represents a geometric shape and it is defined with one method (`area`), which returns the area of such a geometric shape. Different geometric shapes have different ways to calculate area, so that an implementation for this method needs to be provided for each of them. In the example, the `Rectangle` and `Square` struct types respectively represent a rectangle and a square: these types *implement* the `Shape` interface as there are implementations for the `area` method defined in such an interface. This is the Go's version the well-known polymorphism concept in the object-oriented programming paradigm.

A special type of interface is an *empty interface*, which has no methods. It is declared as

```
type Any interface{}
```

so that any types implement it. In other words, such an interface may hold values of any type. Empty interfaces are typically used to handle values whose type is not known a priori, thereby being an analogy to a generic data type.

2.2.5 Concurrency support

One of the main features of Go is the lightweight support for concurrent communication and execution through high-level operations, in contrast to the considerable effort required to develop, maintain, and debug concurrent programs in

mainstream languages such as C++ and Java. In this perspective, the solution provided by Go is threefold. First, the high-level support for concurrency enables programmers to easily develop concurrent programs. Second, concurrent processing is performed by means of *goroutines*, lightweight processes (similar to threads, but lighter) that can be created and automatically load-balanced across the available processors and cores, thereby making Go a language suited to the contemporary multicore computer architectures. Third, an automatic, efficient garbage collection relieves programmers of the memory management typically required by concurrent programs.

Goroutines are the basic primitive for concurrency in Go. In essence, goroutines are functions that can run simultaneously, i.e., concurrently. A goroutine is created by prefixing any function call with the `go` keyword. In Go, goroutines communicate by using typed *channels*. Channels are first-class objects used for sending and receiving values of any type, including other channels. When a channel communication takes place, the channels and their respective goroutines are synchronized at the moment of the communication. Therefore, explicit locking and other low-level details are abstracted away from programmers, thus simplifying the development of concurrent programs[3]. In Go, channels are defined by using the `chan` keyword along with the type of data that they can hold. Values can be sent to and received from channels using the channel operator `<-`.

For illustrative purposes, imagine a cake making and packing factory implemented in the following Go program in which `makeCakeAndSend` is a function representing the cake making whereas `receiveCakeAndPack` is a function representing the cake packing. Once launched simultaneously, these goroutines synchronize their operations through the channel `cs`, received as argument in both functions. As `cs` is an unbuffered channel (the default in Go), the `makeCakeAndSend` goroutine is blocked until the `receiveCakeAndPack` goroutine has completely consumed the value sent through cs.

[3] Go also provides low-level synchronization primitives (e.g., mutexes and condition variables) in the sync package, which are similar to the ones found in other programming languages. However, the use of channels is considered the most idiomatic (and recommended) way of synchronizing goroutines.

```
package main
import (
    “fmt”
    “strconv”
)

var i int

func makeCakeAndSend(cs chan string) {
    i = i+1
    cakeName := “Strawberry cake #” + strconv.Itoa(i)
    fmt.Println(“Making a cake and sending...”, cakeName)
    cs <- cakeName                // sends through channel
}

func receiveCakeAndPack(cs chan string) {
    s := <-cs                     // receives through channel
    fmt.Println(“Packing received cake:”, s)
}

func main() {
    cs := make(chan string)        // creates a channel to strings
    for i := 0; i < 3; i++ {       // make and pack three cakes
        go makeCakeAndSend(cs)     // launch cake making goroutine
        go receiveCakeAndPack(cs) // launch cake packing goroutine
    }
}
```

2.3 Statistical model checking

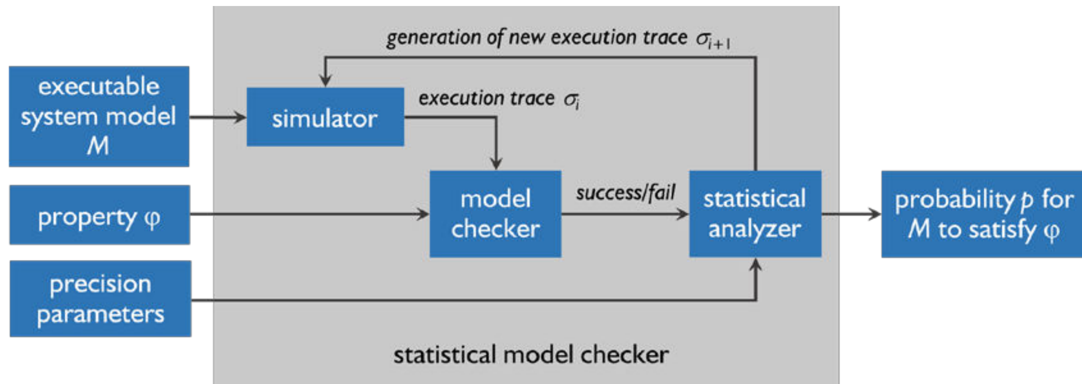
As previously discussed in Section 1.1.3, the main drawback of traditional model checking techniques for formally analyzing software architectures is the exponential growth of the state space, the so-called *state space explosion problem*. Aiming at overcoming this limitation, alternative techniques have been proposed in the last years envisioning the reduction of the state space or even avoiding an entire representation of it. One of these techniques is *statistical model checking* (SMC), 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 [39].

SMC provides a number of advantages in comparison to traditional model checking techniques. First (and perhaps the most important one), this technique does not suffer from the state space explosion problem since it does not analyze the internal logic of the system under verification, neither requires the entire representation of the state space, thus making it a promising approach for verifying complex large-scale and critical software systems [40]. Second, SMC requires only the system be able to be simulated, so that it can be applied to larger classes of systems including black-box and infinite-state systems. Third, the proliferation of parallel computer architectures makes the production of multiple independent simulation runs relatively easier, thus helping the verification of large-scale systems even though it is still necessary to make

the simulation procedure as efficient as possible [39]. Fourth, despite SMC provides approximate results (as opposed to the exact results provided by traditional model checking), it is compensated by less consumption of computational resources and a better scalability. In some cases, knowing the result with less than 100% of confidence is quite acceptable or even the unique available option [51]. Therefore, SMC allows trading-off between verification accuracy and computational time by selecting appropriate precision parameter values. For example, if the project time is limited, it may be more valuable obtaining less precise verification in short time than more precise verification results in much longer time.

Figure 2 illustrates a general schema on how the SMC technique works. A statistical model checker basically consists of a simulator for running the system under verification, a model checker for verifying properties, and a statistical analyzer responsible for calculating probabilities and performing statistical tests. It receives three inputs: (i) an *executable stochastic model* of the target system M; (ii) a formula φ expressing a *bounded property* to be verified, i.e., a property that can be defined in terms of finite executions of M; and (iii) user-defined *precision parameters* determining the accuracy of the probability calculations. M is stochastic in the sense that the next state is probabilistically chosen among the states that are reachable from the current one. As a consequence, some executions of M might satisfy φ and others might not. The simulator executes M and generates an *execution trace* σ_i, composed of a sequence of *states*. Next, the model checker determines if σ_i satisfies φ and sends the result (either success or failure) to the statistical analyzer, which in turn estimates the probability p for M to satisfy φ. The simulator repeatedly generates other execution traces σ_{i+1} until the statistical analyzer determines that enough traces have been analyzed to produce an estimation of p satisfying the precision parameters. It is important to highlight that a higher accuracy of the answer provided by the model checker requires generating more execution traces through simulations.

Figure 2 – Working schema of the SMC technique



In essence, SMC answers two questions. The first one is qualitative: *Is the probability p for M to satisfy φ greater or equal than a certain threshold θ?* The second question is quantitative: *What is the probability p for M to satisfy φ?* [52]. In both cases, producing an execution trace σ_i and checking if it satisfies φ (i.e., $\sigma_i \vDash \varphi$) is modeled as a random variable B_i following a Bernoulli distribution[4] of parameter p [53]. The possible values of B_i are either 0 (if $\sigma_i \nvDash \varphi$) or 1 (if $\sigma_i \vDash \varphi$), with probability functions $Pr[B_i = 1] = p$ and $Pr[B_i = 0] = 1 - p$. Each variable B_i is associated with one simulation of M and the main goal here is to evaluate p.

Qualitative approach. The main existing SMC approaches proposed to answer the qualitative question [54, 55] rely on *hypothesis testing* as means of inferring if the simulated execution traces provide statistical evidence on the satisfaction or violation of a property [56]. In order to determine if $p \geq \theta$, two hypotheses can be considered, namely (i) H: $p \geq \theta$ and (ii) K: $p < \theta$. The test is parameterized by two bounds, α and β. The probability of accepting the hypothesis K when the hypothesis H holds is bounded by α and the probability of accepting H when K holds is bounded by β. Such algorithms sequentially perform simulations until either H or K can be returned with confidence of α or β. Other sequential hypothesis testing algorithms are based on the Bayesian approach [57].

Quantitative approach. In order to compute the probability p for M to satisfy φ, Hérault et al. [58] and Laplante et al. [59] propose an estimation procedure based on the Chernoff-Hoeffding bound [60], which provides a priori the minimum number of simulations required to ensure the desired confidence level. Therefore, given a precision ε, such a procedure computes an estimate p' of p with confidence δ, thereby

[4] In Statistics, the *Bernoulli distribution* with parameter p is the probability distribution of a random variable B that takes value 1 with *success probability* of p (i.e., $Pr[B = 1] = p$) and the value 0 with *failure probability* of $1 - p$ (i.e., $Pr[B = 0] = 1 - p$), $0 \leq p \leq 1$ [53].

ensuring $Pr(|p' - p|) \geq \delta$ [51]. As defined by Hoeffding [60], δ is related to the number of required simulations N by $\delta = 2e^{-2N\varepsilon^2}$, resulting in $N = \lceil (\ln 2 - \ln \delta)/2\varepsilon^2 \rceil$.

The quantitative approach is used when there is no known approximation of the probability to evaluate, i.e., when one wants to obtain a first approximation. This method is used when the goal of the analysis is to have a glimpse on how well the model behavior. On the other hand, the quantitative approach determines whether the probability is above a given threshold with a high confidence and in a minimal number of simulations.

3 Running applications: Wireless sensor network-based systems

Wireless sensor networks (WSNs) are typically made up of multiple spatially-distributed *motes*, tiny low-power hardware/software platforms equipped with an embedded CPU, wireless networking capabilities, and simple sensors able to perform measurements of physical phenomena, e.g., temperature, luminosity, humidity, barometric pressure, vibration, acoustics, magnetic field, etc. [61]. These motes operate in a collaborative way by extracting environmental data, which may undergo some processing. Such sensor-collected data can also be transmitted to other sensors and/or to other computationally-powerful nodes (often called sink nodes), which gather them and make them available to end-users and applications.

The main concern for a WSN-based system is to observe the physical world to obtain useful information from it, besides processing and making decisions upon the collected information. Starting from such a premise, WSNs have been increasingly used in several application domains, e.g., disaster management, environmental monitoring, structure health, ambient assisted living, traffic control, surveillance, and military intelligence [62]. In general, applying WSNs to these scenarios promote a number of advantages in comparison to traditional, wired networking techniques, especially in terms of (i) reduced cost, (ii) simplified deployment, (iii) capability of monitoring hazardous or remote areas, and (iv) ability to adapt to changing environmental conditions [63].

In this chapter, two WSN-based systems are presented to serve as running applications throughout this work aiming at (i) illustrating how to describe a dynamic software architecture using the π-ADL language, (ii) showing how to automatically generate implementation source code in Go, and (iii) demonstrating how to specify and verify properties of dynamic software architectures. Besides being interesting examples with high real-world relevance, these systems were chosen because they are inserted into the inherently dynamic scenario of WSNs, thus requiring their software architectures to be also dynamic. Sections 3.1 and 3.2 respectively introduce a flood monitoring system and a system to remotely monitor oil and gas pipelines, describing their context, goals. Section 3.3 shows a general picture of a software architecture for these systems. In turn, Section 3.4 discusses some dynamicity concerns to be considered in these WSN-based systems.

3.1 A flood monitoring system

Floods are one of the major problems in many countries around the world. In rainy seasons, this type of event can be quite devastating in both developed and

underdeveloped countries where urban centers are traversed by rivers as they may cause material, human, and economic losses. Regardless of their magnitude, floods represent a risk and must be detected. As an example of the potential damage caused by floods in a large urban agglomeration, a 2002 report by the French Government stated that, in a worst-case scenario, a flooding of the Seine river crossing Paris, France, would cost about 10 billion euros, cut telephone service for a million of Parisians, and leave 200,000 people without electricity and 80,000 people without gas [64]. More recently, heavy rainfalls in October 2015 in Southern Brazil triggered flooding of rivers, affecting more than 200,000 people from 229 municipalities in the states of Santa Catarina and Rio Grande do Sul and forcing more than 12,000 people to be displaced or be homeless [65].

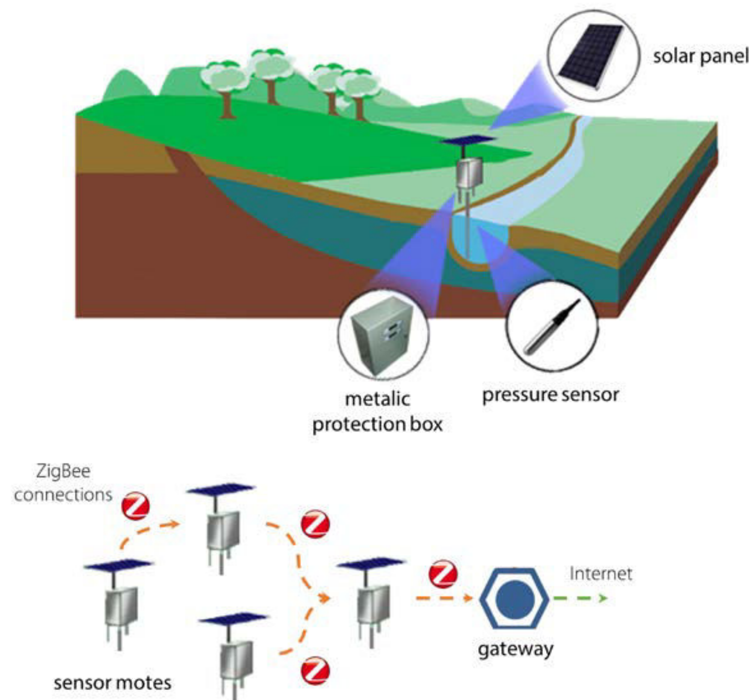
Although it is possible to forecast rainfall or track the path of a storm with the support of meteorological systems and satellite images, it is still necessary to monitor the river water flow and level in order to allow for up-to-date and reasonable decisions on the required actions according to the current conditions. It has been noticed that the cost of damage incurred during a flood is correlated with two main factors, the depth of the flooding (i.e., the water level) and the time in advance at which a warning is given [66]. Therefore, a *flood monitoring system* can support monitoring urban rivers and create alert messages to notify authorities and citizens about the risks of an imminent flood. This type of system can also play an important role in terms of obtaining more precise data and fostering effective predictions in a timely manner, as well as it can improve warning times. Such features are important to ensure a better planning of management activities towards reducing the possible damages caused by the flood, e.g., the definition of evacuation plans, rearrangement of traffic in the proximities of the flooded areas, and coordination of rescue actions [67]. With these actions, the impact of a disaster can be alleviated.

A successful example of flood monitoring system is the one deployed to monitor the Monjolinho river in São Carlos, Southeastern Brazil [63, 68], as illustrated in Figure 3. By using a WSN, motes are spread in flood-prone areas near the river and monitor the water level (centimeters of water), which is used as an indicator of floods. In addition, a gateway station analyzes data measured by motes, makes such data available, and can trigger alerts when a flood condition is detected. Indeed, WSNs have been one of the widely used infrastructures to build a flood monitoring and warning system because they make it easier to collect data and have a wide coverage area as a result of communication capabilities of the sensor nodes [69].

Figure 3 – Deployment layout of the flood monitoring system in a flood-prone area in downtown São Carlos, Brazil (adapted from [70])

Motes can use pressure and/or ultrasound sensors to respectively gauge the depth and the average speed of the water flow, raw data that need to undergo some processing in order to provide the height reached by the water level (centimeters of water). As a sensor node is usually a resource-constrained device in terms of power and networking capabilities, the gateway station responsible for receiving data may be far from the sensor site and hence out of its network coverage area. In order to overcome this limitation, sensed data can be transmitted in a multihop communication, i.e., data sensed by some motes in their respective sites are successively sent to neighbor sensors, which in turn forward such data to other neighbor sensors until reaching the gateway station. The communication among these elements can take place by using wireless network connections such as WiFi, ZigBee, GPRS, Bluetooth, etc. Figure 4 illustrates this scenario.

Figure 4 – Flood monitoring system scenario



3.2 Monitoring oil and gas pipelines

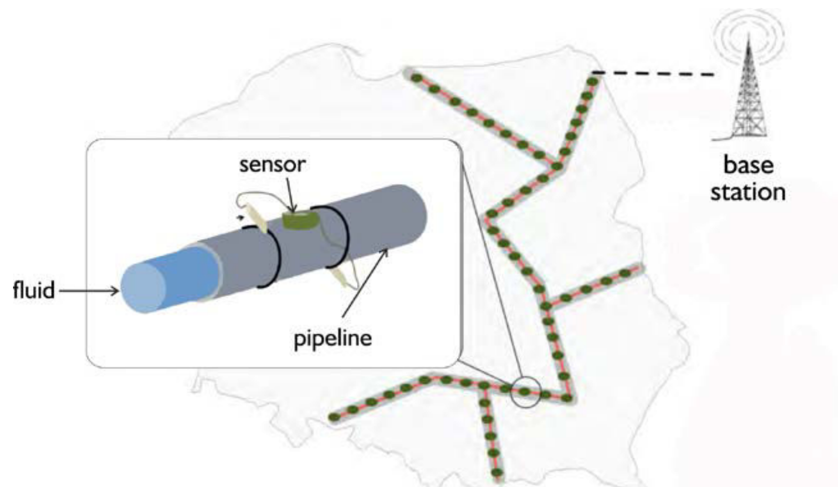
The oil and gas industry includes processes for exploring, extracting, refining, transporting, and marketing oil products. Among these activities, the transportation of oil, gas, and its derivatives is a high-costly sector in the production chain of the oil industry and hence it offers a significant contribution to the success of activities in this context. For this reason, oil and gas companies need to develop and adopt new technologies towards improving operations in order to increase productivity.

Pipelines has been widely used as modal for transporting oil, gas, and its derivatives along the last 40 years since they play a key role in the transportation of oil, gas, and derivatives by linking production areas, harbors, refineries, and consumer centers, which are significantly far from each other. In Brazil, the situation is not different. According to a 2014 report from the Brazilian National Agency of Oil, Gas and Biofuels (ANP), Brazil had 601 pipelines forming a network of 19,700 kilometers long to distribute oil, derivatives, gas, and other products across the country [70]. Despite the high initial investment and inherent complexity, pipelines are the most effective means of transporting large volumes over long distances (thus overcoming the geographical limitations), besides reducing losses and providing a high reliability. This whole conjuncture makes transporting products in pipelines an issue of high importance from the economic point of view as the final price of these products largely depends on the transportation costs.

WSN features such as small size, wireless architecture, ease of deployment, and ubiquitous nature makes such a technology quite attractive for transportation through pipeline networks. WSNs are promising in this context especially due to (i) reduced costs, (ii) hazardous, remote, inaccessible conditions of the areas traversed by the pipelines, (iii) the difficulty and cost regarding the introduction of wired devices near pipelines, and (iii) the difficulty and inefficiency of directly inspecting the state of each pipeline using personnel *in situ* along its extension, which may extend to thousands of kilometers [71, 72]. Recently, the WSN technology has been successfully applied to pipeline monitoring, thereby bringing benefits in terms of cost, flexibility, and control efficiency. For instance, WSNs promote unremittingly monitoring and/or estimation of the pipeline status without manual intervention and work under severe conditions [73, 74].

A *pipeline monitoring system* can make use of a WSN composed of sensor nodes spread along the extension of a pipeline, as illustrated in Figure 5. Each of these sensors is connected to each other via wireless links and they successively transmit the gathered measures until reaching a base station, where such data are processed. As a single type of sensor cannot capture all physical anomalies to which operational pipelines are subjected, a WSN targeted to monitor pipelines can make use of several sensing modalities, e.g., piezoelectric, acoustic, ultrasonic, thermal, optical, chemical, magnetic, etc., each one providing a different information about the pipeline status. Sensors for pipeline monitoring generally use *steady-state* or *transient* detection methods [75]. The first one refers to detection methods that are used when the parameters of operational pipelines (pressure, flow, anomaly, vibration) are expected to remain unchanged over time except when an anomaly such as leakage or third-party damage occurs. In such conditions, the sensor is only required to distinguish between the normal operation characteristics and the occurrence of an anomaly using pre-defined thresholds. On the other hand, the second one is useful for scenarios in which the variables experienced by operational change rapidly and/or frequently over time. When an anomaly (such as leakage or rupture) occurs, the implied change in pressure or vibration propagates through the network and can be detected hydraulically at other locations.

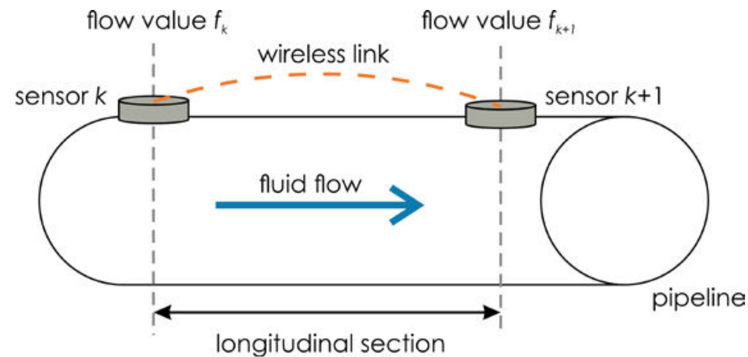
Figure 5 – Deployment of wireless sensors in a pipeline network



One of the main aims of pipeline monitoring regards the detection of leaks in the pipelines, which might have different causes, e.g., deformations caused by earthquakes, collisions, corrosion, structural cracks, third-party intrusion, etc. In the last decades, several disasters caused by leaks in oil pipelines led to devastating environmental hazards, as well as severe material and financial damages. Leakage not only wastes commodities and cuts profits, but it also brings law suits and raises liability issues when it directly affects people's lives and the environment. For this reason, the development of leakage detection systems is critical in pipeline networks.

When a leak occurs, the most important issues are detecting it immediately and executing preventive/corrective actions, e.g., triggering alarms, stop pumps, automatic closing of valves, etc. However, the success of the detection depends on the time at which the leakage is observed: the faster the leakage is detected, the smaller are the losses and the environmental impact. As depicted in Figure 6, a simple, but effective technique to detect leaks in pipelines can be simply is placing sensors to observe the fluid flow at the extremities of some sections of the pipeline, i.e., longitudinal sections [73]. When a leak occurs within a longitudinal section, there is a difference between the flow values observed at each side, while it should be constant. By comparing the two values of flow, an alert can be triggered whenever the difference between the flow measured is greater than a given threshold established for such a longitudinal section. Another conventional way of detecting leaks is by using acoustic sensors, which are able to pick up variations in sound pressure induced by the passing fluid as it escapes from the pipeline under pressure, as well as it can indicate where the leak is.

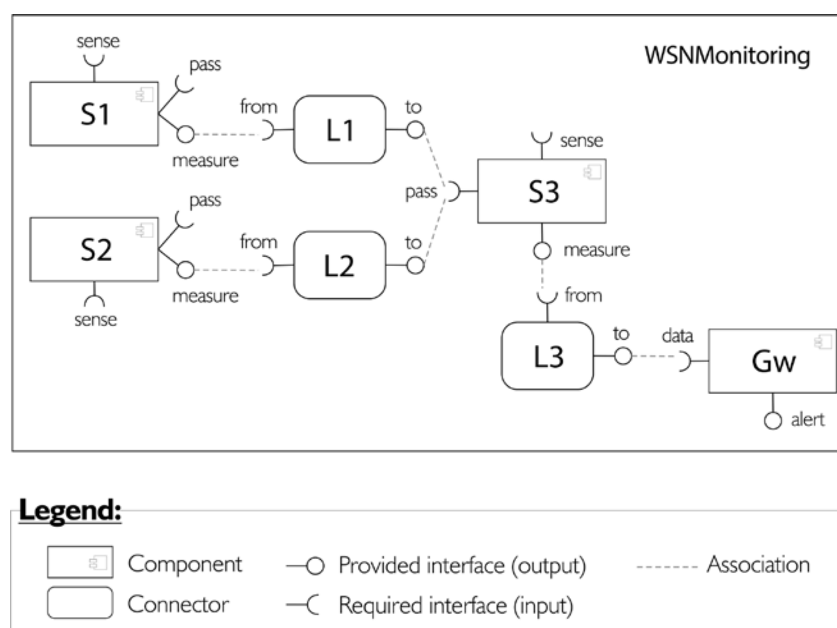
Figure 6 – Detection of leaks in pipelines by measuring flow debits



3.3 An architecture for WSN-based monitoring systems

The WSN-based monitoring systems described in Section 3.1 and 3.2 are equally composed of spread sensor motes that interact with each other via wireless links in a multihop communication and send data to a gateway or base station. Due to these similarities, it is possible to depict a generic architecture that might fit both systems. Figure 7 illustrates a simplified picture of such a generic architecture, composed of three sensor components (*S1*, *S2* and *S3*), one gateway component (*Gw*), and three wireless connectors linking these components (*L1*, *L2* and *L3*). In this architecture, data (water level in the flood monitoring system and fluid flow in the pipeline monitoring system) are measured by sensors *S1* and *S2* and sent to sensor *S3* by using the links *L1* and *L2*. Sensor *S3* receives these data and forwards them to the gateway *Gw* via the link *L3*, with no additional processing. Both structure and behavior of these architectural elements are detailed in the following.

Figure 7 – Generic architecture for WSN-based monitoring



***Sensor* component.** *Sensor* components can (i) either receive raw data measured by the physical sensors (and that might undergo some processing) via the *sense* input or data from a neighbor sensor via the *pass* input and (ii) send data via the *measure* output. The behavior of this component comprises the preprocessing of sensed raw data (e.g., making a unit conversion). Moreover, data received via the *pass* input are directly sent from this input to the *measure* output, not undergoing any processing.

***Link* connector.** *Link* connectors receive data via the *from* input and send them via the *to* output. Such connectors do not make any additional processing, thereby only transmitting data from their input to output.

***Gateway* component.** The *Gateway* component receives data from sensors via the *data* input. The behavior of this component comprises functions to determine the risk of flood or leakage. The calculation of the flood risk relies on the hazard index [76], a measure that indicates the potential of flood based on the water level measures gathered by the sensors. In turn, the leakage detection relies on the debit (difference) between flow measures gathered by two neighbor sensors. When an imminent flood or an anomaly in a pipeline is detected, alert messages are sent via the *message* output.

***WSNMonitoring* configuration.** The *WSNMonitoring* configuration represents the architecture itself and it comprises instances of the *Sensor* and *Gateway* components (*S1*, *S2*, *S3* and *Gw*), as well as instances of the *Link* connector (*L1*, *L2* and *L3*). The associations depicted in Figure 7 show how these architectural elements are connected with each other.

3.4 The dynamic scenario of WSNs

WSNs are typically inserted into highly dynamic, sometimes remote and/or even hostile environments, thereby adaptation strategies to ensure the availability of the network and gathered data. These networks should have an autonomous behavior and be able to tolerate several types of failures, such as faulty nodes, low-level battery, lack of coverage and connectivity, etc. General requirements that lead to a dynamic reconfiguration of WSN-based systems are related to: (i) efficiency in the use of the available resources aiming at extending the network lifetime, especially in terms of power consumption and communication; (ii) resiliency of the system in case of unavailability of motes during operation; (iii) precise, accurate measures; and (iv) autonomous, proactive adaptation upon failures or unpredictable circumstances while minimizing manual intervention and disruption. Interesting properties that a WSN should have are *self-configuration*, i.e., the ability of reconfiguring and adapting the networking and sensing behaviors of sensor nodes by dynamically changing parameters according to the conditions and state of the network, and *self-healing*, in to ensure reliability and correctness of the network especially upon failures [77].

Hughes et al. [78] discuss some interesting scenarios for dynamic adaptation in WSNs, taking into account four important concerns, namely communication latency, resilience, power consumption, and prediction accuracy. These scenarios are briefly outlined in the following.

Adaptation due to power level of motes. In WSNs, power efficiency and communication efficacy are often conflicting strategies. For example, WiFi has a better performance and resilience, but it requires more power capabilities. In turn, ZigBee (IEEE 802.15.4) requires lower power although it has reduced performance and demands for short distances between sensor nodes. As these communication infrastructures have quite different properties, it is sometimes desirable having motes with both capabilities and able to switch between them according to the power level of the motes and the current environmental conditions of the river.

Adaptation of the general behavior of the WSN. If an anomaly in the normal operation of the monitored site is detected, then the frequency at which sensors send measures needs to be dynamically increased in order to increase prediction accuracy, even though this implies in a greater consumption of power and networking resources. When the normal operation conditions are reestablished, such a frequency needs to be reconfigured back to the previous operation conditions.

Adaptation to deal with mote failure. In order to minimize manual intervention and foster system dependability, dealing with mote failure is an important concern and requires the system to be capable of reconfiguring itself at runtime by inserting/activating new sensing nodes in the network. When a sensor node fails, its functions should be covered by neighboring nodes, if possible. Moreover, a mote failure may also imply reconfiguring the WSN in terms of communication strategy or even how the remaining motes are connected with each other. Depending on the distances between motes, different networks can be used to communicate, e.g., Bluetooth or WiFi for shorter distances and GPRS for longer distances.

Adaptation to increase accuracy. Ensuring accuracy of predictions is an important concern as a corrective procedure (e.g., evacuating a flood-prone area or triggering personnel to fix leakage in pipelines) might be expense. For this reason, false positives must be avoided as much as possible. A way of improving accuracy is using other information sources and combine them with data gathered by the sensors. In a flood monitoring system, a possible strategy is using image-based flow prediction, i.e., use digital cameras fixed in given sites or endowed in unmanned aerial vehicles (a.k.a. UAVs or drones) in order to capture images of the river and estimate its current flow rate. These images can be also processed and combined with data provided by sensor motes to confirm whether a flood is imminent or not. In the case of pipeline monitoring, multiple sensing modalities can be used to improve accuracy.

4 The π-ADL architecture description language

Process algebras (or *process calculi*) refer to mathematical theories for formally modeling concurrent systems and describing their communications/interactions and synchronizations at a high abstraction level [79]. The basic elements of a process algebra are a collection of *processes*, which represent the behavior of a system, and a set of *operators*, which are used to manipulate and analyze such processes.

Regardless the variety of existing process algebras, all of them have several common features, such as: (i) a *concurrency model* describing processes/systems; (ii) a *communication model* specifying how such independent processes can communicate and be synchronized with each other; and (iii) *algebraic laws* for process operators, which allow manipulating processes and formally reasoning about them. Leading examples of process algebras include Communicating Sequential Processes (CSP) [50] and the Calculus of Communicating Systems (CCS) [80], both introduced about 30 years ago. More recent additions include π-calculus [81], a computationally complete (Turing-complete[5]) process algebra able to provide a universal model of computation [82].

Process algebras have played a relevant role as formal underpinnings for describing software architectures. By making a correspondence with software architectures and their formal specification, the concurrency model serves as a basis for specifying components that coexist to compose the architecture of a software system. In turn, the communication model serves as a basis for specifying connectors representing the interactions between components (processes). Examples of architecture description languages that take advantage of process algebras for specifying behavior of software architectures are: (i) Wright [21, 23], which uses CSP; (ii) Darwin [22], which uses finite state processes (FSP) [83] drawn from CCS; and (iii) LEDA [84], which uses π-calculus and its operators as-are for specifying the behavior of architectural elements.

Despite their expressiveness and maturity, process algebras typically do not provide constructs for easily describing software architectures, thereby hampering their adoption as notations for this activity in software development. Aldini et al. [85] argue that the usability of process algebras for describing software architectures can be enhanced by supporting a user-friendly component-oriented way for modeling

[5] Arthur John **Robin** Gorell **Milner** (1934-2010), British computer scientist and the creator of π-calculus, was awarded with the 1991 ACM A. M. Turing Award, one of the highest worldwide distinction prizes in Computer Science: http://amturing.acm.org/award_winners/milner_1569367.cfm.

software systems with process algebras. Software architects can thereby reason in terms of composable software units without worrying about technicalities of the underlying formalism. In this perspective, adapting process algebras to the architectural level can increase both the degree of usability of this type of formalism and the degree of formality and analyzability of software architectures.

Aiming at providing a simple, but expressive notation for describing software architectures while being well-founded theoretically, π-ADL [9] was proposed one decade ago as a formal language intended to describe software architectures under both structural and behavioral viewpoints, unlike most existing architecture description languages. This language extends π-calculus (hence the name) by providing formally founded constructs for architecture description while achieving computational completeness and high expressiveness. As one of the contributions resulting from this work, π-ADL was endowed with architectural-level primitives for specifying *programmed reconfiguration* operations [32]. In addition, two common approaches for enacting programmed dynamic reconfiguration [33, 34] were incorporated into π-ADL. The first approach is *exogenous*, in which it is possible to control all elements of the software architecture and apply the changes on the whole structure. In turn, the second approach is *endogenous*, in which the architectural elements themselves are able to manage dynamic reconfiguration actions. This culminates in an expressive language able to describe both structure and behavior of a dynamic software architecture, as well as the reconfiguration operations that can be applied over it at runtime [35].

This chapter presents the π-ADL language and how it can be used to describe dynamic software architectures. Section 4.1 introduces the main architectural abstractions of π-ADL. Section 4.2 presents the type system defined in the language. Section 4.3 presents the formally founded constructs used to represent the behavior of architectural elements. Section 4.4 describes the approach proposed in this work for specifying programmed reconfigurations of dynamic software architectures expressed in π-ADL. In Section 4.5, the WSN-based architecture presented in Chapter 3 is used as an illustrative example of how to describe dynamic software architectures in π-ADL. Finally, Section 4.6 discusses some related work on languages for describing dynamic software architectures.

4.1 Architectural abstractions

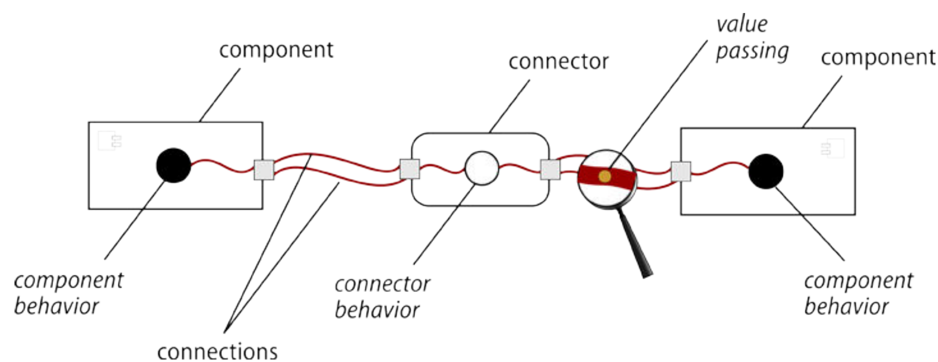
From the structural viewpoint, a software architecture is described in π-ADL in terms of *components*, *connectors*, and their composition to form the system, i.e., the *architecture* itself as a configuration of components and connectors. Components represent the functional elements of the system whereas connectors manage interactions among components. Components and connectors can be also composed

to construct composite elements, which may be themselves components or connectors. From the behavioral viewpoint, both components and connectors comprise a *behavior*, which expresses the interaction of an architectural element and its internal computation and uses *connections* to connect and transmit values. In π-ADL, architectures, components, and connectors are formally specified in terms of *abstractions* over behaviors.

In π-calculus, interactions among concurrent processes take place through communication *channels* for synchronizing such processes by sending and receiving messages (values or names)[6]. Analogously, π-ADL provides *connections*, which are abstractions representing communication channels between architectural elements. By using typed connections, components and connectors can send (*output connections*) and receive (*input connections*) any value of the existing types as well as connections themselves. In order to attach a component to a connector, at least a connection of the former must be attached to a connection of the latter. Such an attachment takes place by means of a *unification*, so that attached connections can transport values, connections or even architectural elements.

Figure 8 depicts 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 8 – Main architectural concepts of the π-ADL language



4.2 Type system

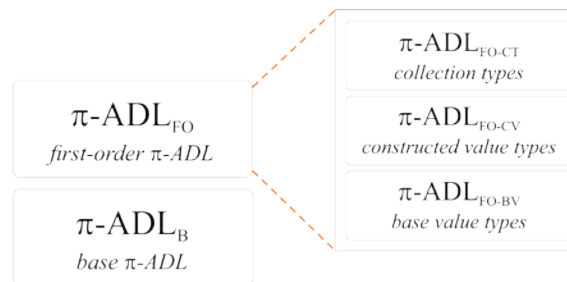
π-ADL is formally defined by a transition and type system [9, 86]. As depicted

[6] In spite of the same notions of communicating processes, the main feature that differentiates π-calculus from previous process algebras such as CCS and CSP is *name mobility*, i.e., the ability of passing names over channels. These names may refer to processes or even channels, so that it is possible to send channels over other channels, for example.

in Figure 9, the formal type system of π-ADL is structured upon two main layers:

- π-ADL_B (*base* π-ADL), which provides the connection and behavior constructs; and
- π-ADL_{FO} (*first-order* π-ADL), which extends π-ADL_B with base, constructed, and collection data types, respectively represented by the π-ADL_{FO-BV}, π-ADL_{FO-CV}, and π-ADL_{FO-CT} sub-layers.

Figure 9 – Layered type system of π-ADL



Sections 4.2.1 to 4.2.3 present the atomic and composite data types defined in π-ADL_{FO}. The behavior constructs defined in π-ADL_B are presented in Section 4.3.

4.2.1 Base types

The base value types are used to express atomic values. Table 1 shows the base value types defined in the π-ADL_{FO-BV} layer (see Figure 9).

Table 1 – Base types defined in π-ADL_{FO-BV}

Type	Syntactic representation	Definition
Natural	`Natural`	Natural numbers (non-negative integers)
Integer	`Integer`	Integer numbers (signed)
Real	`Real`	Real numbers (floating-point)
Boolean	`Boolean`	Boolean logical values
String	`String`	Character strings

In addition to these atomic types, π-ADL defines a type called `Any` that works as a generic type in the language. This type admits values of any type (base or constructed ones), so that it can be seen as a union type with no constraint on the type of value that it can hold.

4.2.2 Constructed types

The π-ADL_{FO-CV} layer (see Figure 9) provides constructors for defining composite types by using the base types from π-ADL_{FO-BV}. Table 2 summarizes these constructed value types, each one described as follows.

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

Type	Syntactic 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_i and is of type T_i

Tuple. Values of a tuple type `tuple[`T_1, T_2, …, T_n`]` are n-uples (v_1, v_2, …, v_n) ($n \geq 2$) in which each value v_i is of type T_i. For example, the declaration

```
t is tuple[Integer, String]
```

refers to the declaration of a tuple t associated to pairs in which the first value is an integer value and the second one is a string value. The individual values within a tuple can be projected into other variables by using an *explicit projection*. Therefore, for the tuple t exemplified above, the instruction

```
project t as a : Integer, b : String
```

sequentially assigns values within t to the variables a and b, which respectively receive an integer value and a string value.

View. A view can be understood as a labeled form of a tuple. Values of a view type `view[`l_1`:` T_1, l_2`:` T_2, …, l_n`:` T_n`]` are views ($l_1 : v_1$, $l_2 : v_2$, …, $l_n : v_n$) ($n \geq 2$) in which each value v_i is labeled as l_i and is of type T_i. For example, the declaration

```
v is view[x : Integer, y : String]
```

refers to the declaration of a tuple v associated to pairs in which the first value (x) is an integer value and the second one (y) is a string value. The individual values within a view can be also projected into other variables by using the same explicit projection instruction used for tuples.

4.2.3 Collection types

The π-ADL$_{FO-CT}$ layer (see Figure 9) provides constructors for defining collection types by using base and constructed types from π-ADL$_{FO-BV}$ and π-ADL$_{FO-CV}$. Table 3 summarizes these collection value types, each one described as follows.

Table 3 – Collection types defined in π-ADL$_{FO-CT}$

Type	Syntactic representation	Definition
Set	`set[`T`]`	Unordered collection of elements of type T
Sequence	`sequence[`T`]`	Ordered collection of elements of type T

Set. In π-ADL, a set is an unordered collection of elements of the same type T. Values within a set type `set[`T`]` are values v_1, v_2, …, v_n in which each value v_i pertains

to such a set and is of type T. For example, the declaration

```
s is set[Integer]
```

refers to the declaration of a set s of integer values.

Sequence. In π-ADL, a sequence is an ordered collection of elements of the same type T. Values within a sequence type `sequence[T]` are values v_1, v_2, …, v_n in which each value v_i pertains to such a sequence and is of type T. For example, the declaration

```
q is sequence[Integer]
```

refers to the declaration of a sequence q of integer values.

Elements can be added to both set and sequence collections by using the `add` operator. For example, the instruction

```
q add 12
```

adds the value 12 to the sequence q of integer values. In case of adding elements to a sequence, they are added at the end of the sequence to ensure element ordering. This is not guaranteed for sets as they represent unordered collections of elements.

π-ADL also provides the `iterate` construct for iterating over the elements of sets and sequences. The `iterate` construct is of the form

```
iterate collection by iterator
from accumulator initially initial_value
accumulate {
    // statements
}
```

in which *collection* refers to the name of the collection (set or sequence) to be iterated, *iterator* is the variable used to iterate over the collection, *accumulator* is a variable returned as result of the iteration, and *initial_value* is the initial value set to such an accumulator. As an example, consider the following instructions:

```
iterate s by i
from count initially 0
accumulate {
    count = count + i
}
```

This `iterate` construct counts the number of elements of the s and stores the result in the accumulator variable *count*, initially set to zero. The variable i is used to iterate over the elements of s.

4.3 Behavior constructs

Besides the connection abstraction, the π-ADL$_B$ layer provides behavior

constructs to represent the internal behavior of architectural elements, which make use of connections to send and receive values (see Section 4.1). 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.

Let P and Q independent processes, c a channel, and x a value or name. π-calculus formally defines the set of process constructs summarized in Table 4.

Table 4 – Process constructs defined in π-calculus

Notation	Process construct	Definition
$c(x).P$	Input prefixing action	A process that waits to read x from c before sequentially proceeding as P
$\bar{c}(x).P$	Output prefixing action	A process that waits to write x (previously accepted in an input prefixing action) in c before sequentially proceeding as P
$\tau.P$	Unobservable action	A process executing an action that is not observable from an external viewpoint
$P \mid Q$	Parallel composition	A process that runs P and Q concurrently
$P + Q$	Non-deterministic choice	A process that behaves as either P or Q depending on whether an action within P or an action within Q is executed
0	Inert (nil) process	A process that does nothing, i.e., it has no remaining computation

Similarly to these basic π-calculus constructs, π-ADL provides the behavior constructs defined in Table 5, each one described in the following.

Table 5 – Behavior constructs defined in π-ADL

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

Prefixing actions. Behaviors run by performing *prefixing actions*. As shown in Table 5, three types of prefixing actions are defined in π-ADL:

- *input prefixing action*, which expresses the ability of receiving a value v via a connection c;
- *output prefixing action*, which expresses the ability of sending a value v via a connection c;
- *unobservable action* (a.k.a. *silent prefixing action*), which expresses an internal action unobservable from an external viewpoint.

Composition behavior. A *composition behavior* expresses the ability of a behavior to compose sub-behaviors in parallel, each one independently proceeding from the others in a separate execution thread. These sub-behaviors can interact among each other via shared connections, e.g., when a behavior sends a value to another executing behavior via an attached connection. As an example, consider the following instructions:

```
compose {
   via input receive v : Integer
   via output send (v+1)
   and
   via signal send true
}
```

In this composition behavior, an integer value v is received via the *input* connection and its increment is sent via the *output* connection, simultaneously to the sending of the Boolean value true via the *signal* output connection. As another example, consider the following instructions:

```
compose {
   via x send v

   and
   via x receive y : Any
}
```

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

Choice behavior. In π-ADL, a *choice behavior* expresses the ability of a behavior to execute alternative sub-behaviors. When a given sub-behavior is enacted, the others are no longer available, i.e., only one of the choice sub-behaviors is executed. As an example, consider the following instructions:

```
choose {
   via input receive v : Integer
   via output send (v+1)

   or
   via alt receive s : Integer
   db = s
}
```

This choice behavior is expressed by two alternative options, (i) receiving an integer value *v* via the *input* connection and then sending the increment of *v* via the *output* connection, or (ii) receiving an integer value *s* via the *alt* connection and then assigning it to a variable called *db*. It is worth mentioning that when more than one sub-behavior is eligible to be executed at the same time, the selection criterion for choosing the block to be executed is non-deterministic.

Inert (nil) process. The behavior of an *inert process* (expressed by the `done` keyword) represents the end of execution of a behavior, i.e., it does not execute any further actions.

4.4 Specifying architectural elements in π-ADL

A software architecture is described in π-ADL in terms of *abstractions* representing components, connectors, and their composition in order to form the architecture of the system. This section briefly introduces the elements provided by π-ADL for specifying behavior of these architectural abstractions. More details about the syntax of architecture descriptions in π-ADL and its main elements can be found in [9, 87].

4.4.1 Specifying behavior of components and connectors

The specification of components and connectors is quite similar, except for the keyword used to determine the type of the architectural element (`component` or `connector`). These elements have an identifier as a unique name and they can optionally take a list of parameters in the form *name* : *type* as input. Furthermore, exactly one behavior must be mandatorily declared in order to specify the behavior of an architectural element. Besides the behavior constructs introduced in Section 4.3, the behavior of both components and connectors can comprise the following instructions.

Type declarations. In π-ADL, it is possible to declare data types within the scope of a given architectural declaration. This type declaration is specified in the form

`type` *s* `is` *T*, in which *s* is an identifier[7] defining an alias for the type and *T* is an existing type (base, constructed or declared one). As an example, consider the following instructions:

```
type Latitude is Real
type Longitude is Real
type GeoCoordinate is tuple[Latitude, Longitude]
```

The two first instructions create the *Latitude* and *Longitude* types, both having a real value as underlying type. In turn, the third instruction creates the *GeoCoordinate* type as a tuple composed of values of the *Latitude* and *Longitude* types for representing a geographic coordinate.

Connection declarations. π-ADL allows declaring typed connections that represent the communication channels used to transmit (send/receive) values. The declaration of a connection within the scope of an architectural element is specified in the form `connection` *c* `is` *d*(*T*), in which *c* is an identifier, *d* is the direction of the connection, and *T* is an existing type (base, constructed or declared one). Two directions are available for declaring a connection, namely `in` for input connections and `out` for output connections. For example, the instructions

```
connection x is in(String)
connection y is out(Boolean)
```

declare an input connection *x* for receiving string values and output connection *y* for sending Boolean values.

Variable declarations and assignments. Local variables can be declared in the form *s* is *T*, in which *s* is an identifier and *T* is an existing type (base, constructed or declared one). For example, the instruction

```
a is Real
```

declares a real variable named *a*.

Protocol declarations. In π-ADL, protocols are used within the specification of architectural elements in order to enforce the value types that must be transmitted via connections (complying with their respective declarations) and the order in which the sending/receiving operations must be performed. A protocol is declared as a set of connection actions specifying the action to be performed by the connection (`send` or `receive`) and the type of the values that will be transmitted via the connection. Furthermore, protocol declarations can make use of multiplicity symbols to specify

[7] Conventionally, alias names for types shall start with capital letters in order to differentiate them from variable and function names.

how many times the connection actions can be performed: an asterisk character (*) is used to specify that a given action (or set of actions) is performed zero or more times whereas the plus character (+) is used to specify that a given action (or set of action) is performed one or more times, i.e., at least one. It is also possible to specify alternative options by using the pipe character (|), so that *A* | *B* indicates that either action (or group of actions) *A* or action (or group of actions) *B* can be performed. As an example, consider the following protocol declaration:

```
protocol is {
   ((via a receive Integer | via b receive Integer)
     via c send Integer)*
}
```

This protocol enforces receiving an integer value via connection *a* or connection *b* and then sending an integer value via connection *c*. The asterisk character after the outermost parentheses specifies that all of these actions (receiving via *a* or *b* then sending via *c*) can be performed multiple times. When declaring a protocol, specifying the multiplicity of the actions (by using the asterisk or the plus characters) is mandatory.

4.4.2 Statements

The specification of both components and connectors can also comprise a set of statements, which are mainly used in behavior and function declarations. Such instructions are briefly described in the following.

Variable assignments. Assigning values to declared variables is done by using the equal sign (=) after the identifier naming the variable.

Function declarations and calls. Local functions within behaviors have an identifier and can optionally take a list of parameters in the form *name* : *type* as input. In addition, they can also specify a return type. As an example, the following instruction

```
increment is function(v : Integer) : Integer {
   // statements
}
```

declares a function named *increment*, which receives an integer value as parameter and returns an integer value. Declared functions can be called by making reference to its respective identifier and providing the required parameters.

Conditional statements. The *if-then-else* conditional statement provided by π-ADL can be used within behaviors and function declarations. Upon the evaluation of a specified Boolean expression, if the condition is true then a sequence of statements is executed, otherwise the execution proceeds in the following branch, specified by

optional *else-if* or *else* blocks. The basic structure of a conditional statement in π-ADL is as follows:

```
if expression then {
   // statements
} else if expression then {
   // statements
} else {
   // statements
}
```

Condition-controlled loops. π-ADL provides a *while* statement as condition-controlled loop that can be used within behavior and function declarations. Upon the evaluation of a specified Boolean expression, a sequence of statements is repeatedly executed until such a condition does not hold. The basic structure of a condition-controlled loop in π-ADL is as follows:

```
while expression do {
   // statements
}
```

Iteration loops. π-ADL provides a *for* statement as iteration loop that can be used within behavior and function declarations. In this type of loop, a variable identifier is used as iterator in conjunction with a Boolean stop condition and an expression specifying how to advance to the next iteration. At each iteration, a set of statements is repeatedly executed until the stop condition is achieved. The basic structure of an iteration loop in π-ADL is as follows:

```
for (start; expression; step) do {
   // statements
}
```

4.4.3 Specifying architectural configurations

The specification of the system architecture is declared with its respective keyword (`architecture`), a unique identifier, and an optional list of parameters to be taken as input, similarly to the declaration of components and connectors. The main difference resides in the fact that it comprises two basic elements, namely a set of *element instances* and a set of *unifications*. Element instances are contained within a composition behavior (see Section 4.3) and are in the form *i* `is` *E*, in which *i* is the instance identifier and *E* is the identifier of an architectural element (component or connector)[8]. In turn, unifications are the means of passing values from an output

[8] If an architectural element requires input parameters (according to its declaration as an abstraction), the respective parameters must be provided when declaring an instance of such an element.

connection of an element to an input connection of another. These unifications are in the form c_o`::`E_1 `unifies` c_i`::`E_2, in which c_o is an output connection of the element E_1 and c_i is an input connection of the element E_2. Therefore, values can be transmitted from E_1 to E_2 (their behaviors) through such connections.

As an example, consider the following specification of a basic architecture for the well-known pipe-filter architectural style, which comprises *filter* components for transforming data and *pipe* connectors for transmitting such data from a filter to another [11]:

```
architecture PipeFilter is abstraction() {
   behavior is {
      compose {
            F1 is Filter()
         and P1 is Pipe()
         and F2 is Filter()
      } where {
         F1::outFilter unifies P1::inPipe
         P1::outPipe   unifies F2::inFilter
      }
   }
}
```

This architecture is named *PipeFilter* and it is composed of three instances of architectural elements, namely two instances of the *Filter* component (*F1* and *F2*) and one instance of the *Pipe* connector (*P1*), all of them not requiring input parameters. In order to attach these architectural instances, two unifications take place:

(i) the *outFilter* output connection of the filter component *F1* is bound to the *inPipe* input connection of the pipe connector *P1*, thus representing a message flow from *F1* to *P1*; and

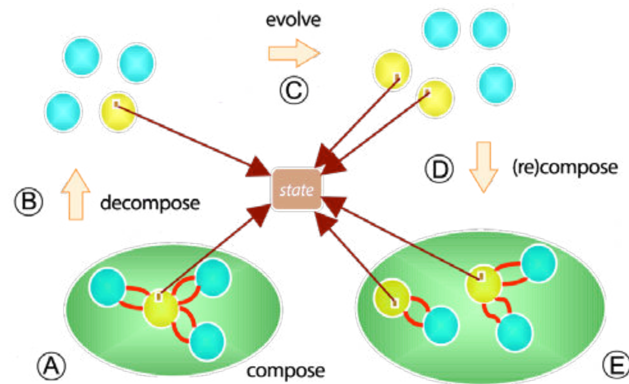
(ii) the *outFilter* output connection of the pipe connector *P1* is bound to the *inFilter* input connection of the filter component *F2*, thus representing a message flow from *P1* to *F2*.

4.5 Dynamic software architectures in π-ADL

The specification of dynamic software architectures in π-ADL is based on the concept of *active architectures* [88]. Active software architectures are: (i) *dynamic* in the sense that the structure and cardinality of the components and interactions are changeable during execution; (ii) *updatable* in the sense that architectural elements can be dynamically replaced; (iii) *decomposable* in the sense that an executing system can be dismantled into its architectural elements; and (iv) *evolvable* in the sense that the specification of components and interactions can be evolved at runtime. Figure 10 depicts the basic life cycle for an active architecture as envisioned in π-ADL. At the initial stage Ⓐ, the architectural abstractions of the system are *composed* in order to form its software architecture. At stage Ⓑ, the system is *decomposed* to yield its

individual architectural abstractions disconnected from each other. Next, such individual architectural abstractions can be *evolved* at stage Ⓒ, e.g., in terms of creating and/or removing elements. At last, the system architecture is *(re)composed* at stage Ⓓ by composing a new configuration of architectural elements aiming to form the new version of the architecture (stage Ⓔ). It is important to mention that the state of each architectural abstraction (and respective data) must be *conserved* at each evolution stage, thus maintaining consistency along the reconfiguration process.

Figure 10 – Life cycle for active software architectures (adapted from [88])



In the light of active architectures, dynamic software architectures can be specified in π-ADL by using two main operations, namely *composition* (`compose` operator) and *decomposition* (`decompose` operator). The former is used to compose the architecture by (i) instantiating abstractions corresponding to architectural elements and (ii) unifying connections to attach such elements and allow their synchronization and communication. In turn, the latter is used to dismantle an architecture into a set of behaviors corresponding to the previously composed architectural elements, now detached from each other. Through decomposition, the executing system is broken into its constituent elements, which can be changed and further recomposed to form an evolved system. Even though the composition and decomposition operations provided by π-ADL seem to be divergent from the usual reconfiguration actions adopted in most works in literature [23, 32, 44], they allow specifying dynamic software architectures in a comprehensive way while preserving the formal foundations of the language upon the π-calculus process algebra. Therefore, supporting these operations with a suitable underlying formalism fosters formal verification of the specified software architectures as well as the enforcement of structural, behavioral, and quality properties before, during, and after the reconfiguration process itself.

As previously mentioned, this work is concerned with supporting programmed, foreseen reconfiguration operations under both *exogenous* and *endogenous* approaches for describing dynamism in software architectures. The

exogenous approach concerns having two coexisting architectures, namely the *initial architecture* and the *evolved architecture*, the latter resulted from a reconfiguration applied over the former upon a stimulus [89]. The evolved architecture describes the reconfiguration actions to be performed over the initial architecture in order to produce it. Both architectures are executed by a coordinating behavior, the external entity that has control over all architectural abstractions. In turn, the endogenous approach concerns specifying the reconfiguration actions within the behavior of the architectural element(s) responsible for applying them. Reconfiguration actions in both approaches may comprise instantiating new architectural elements, modifying architectural elements or decomposing architecture configurations.

In order to illustrate the exogenous and endogenous approaches considered in π-ADL[9], consider a simple client-server architecture initially composed of one client and one server interacting through a link connector. Moreover, consider the two following possible reconfigurations of such an architecture at runtime:

R_1: The architecture can have two available servers, namely (i) a primary server that is more desirable to use, but which may go down unexpectedly, and (ii) a secondary server, which can replace the former in case of unavailability. Therefore, the client can use the secondary server when the primary one is unavailable.

R_2: When the server is overloaded, it can allocate an auxiliary virtual machine for load balancing, thus maximizing throughput.

To illustrate the exogenous reconfiguration process, consider two coexisting architectures for the client-server architecture, namely (i) an initial architecture called *ClientServer* and (ii) a new, evolved architecture called *ClientServerEvol* resulting from reconfiguration operations applied on *ClientServer*. As depicted in Figure 11, the *ClientServer* initial architecture is composed of one instance of the client component (*Cl*) and one instance of the server component representing the primary server (*PrSv*), which interact with each other through one instance of the link connector (*lnk*) (stage Ⓐ). In order to perform reconfiguration R_1, *ClientServerEvol* must describe the decomposition of the architectural elements of *ClientServer*, detached from each other (stage Ⓑ). Finally, *ClientServerEvol* must describe (i) the creation a new instance of the server component to represent the secondary server, (ii) the composition of the previously instantiated client component and link connector with the new server component, and (iii) the unification of their connections in order to attach them. Upon

[9] Examples textually describing both exogenous and endogenous reconfiguration approaches in π-ADL are provided in Sections 4.6.2 and 4.6.3 with the WSN-based flood monitoring system.

the execution of these operations, the *ClientServerEvol* evolved architecture is launched by the controlling behavior (stage Ⓒ).

Figure 11 – Illustration of the exogenous approach for the dynamic reconfiguration of a simple client-server architecture. A primary server is replaced by a secondary one in case of unavailability of the former.

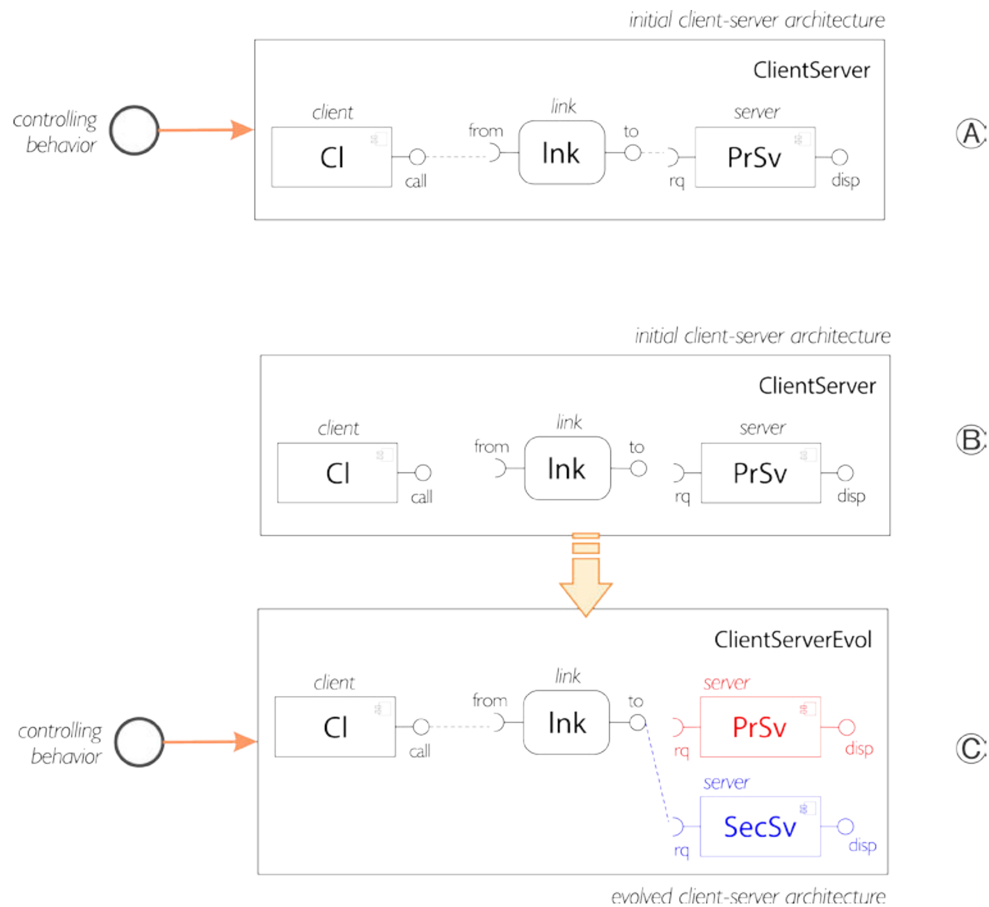
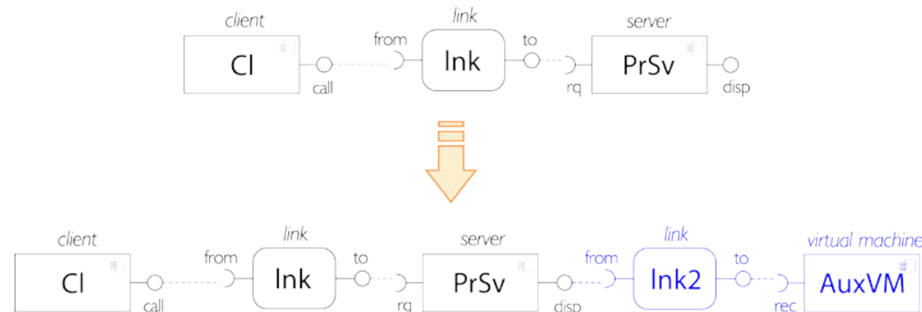


Figure 12 illustrates the result of reconfiguration R_2 on the *ClientServer* architecture. In this case, the server component (*PrSv*) representing the primary server is responsible for performing the reconfiguration, so that the required operations must be specified in its behavior. When the primary server is overloaded, it should create a new instance of the link connector (*lnk2*) and an instance of the component representing the auxiliary virtual machine for load balancing. After creating these instances, they must be attached by unifying their respective connections. It is important to mention that this case does not require decomposing the *ClientServer* architecture since reconfiguration R_2 does not comprise detaching architectural elements, but only creating and attaching new ones.

Figure 12 – Result of the endogenous approach for the dynamic reconfiguration of a simple client-server architecture. An auxiliary virtual machine is created and attached to the primary server in case of overloading.



4.6 Describing the flood monitoring system in π-ADL

In this section, the WSN-based flood monitoring system presented in Chapter 3 is used to exemplify how to describe dynamic software architectures in π-ADL. Section 4.6.1 presents partial descriptions of the architecture for this system, based on the one previously depicted in Figure 7 (see Section 3.3). In turn, Sections 4.6.2 and 4.6.3 respectively present programmed reconfigurations on such an architecture considering both exogenous and endogenous approaches (see Section 4.5).

4.6.1 Architectural elements

***Sensor* component.** Figure 13 shows the specification of the *Sensor* component in π-ADL. Two user-defined types are declared within this component, both relying on the *Real* base type: (i) *MV*, which represents a raw value in millivolts measured by the pressure sensor (line 2), and (ii) *CmH2O*, which represents a value in centimeters of water (line 3). This component comprises three connections, namely: (i) *sense*, an input connection used for receiving raw data measured by the pressure sensor (line 4); (ii) *pass*, an input connection used for receiving data from a neighbor sensor (line 5); and (iii) *measure*, an output connection used for sending data (line 6). The protocol of this component (lines 7 to 10) enforces receiving either a value of the *MV* type via the *sense* connection or a value of the *CmH2O* type via the *pass* connection, and then sending a value of the *CmH2O* type via the *measure* connection, actions that are performed sequentially and at multiple times. Next, the behavior of this component encompasses the definition of the *convertRawData* function (line 12), which receives a value of the *MV* type and converts it to a value of the *CmH2O* type. As the implementation of the *convertRawData* function may change according to the sensor specifications provided by the respective manufacturers, it is set as unobservable (line 13). Furthermore, such a behavior can proceed through two alternative, non-deterministic options: (i) data received via the *sense* input connection are processed by the *convertRawData* function and then sent via the *measure* output connection (lines 16

and 17); or (ii) data received via the *pass* input connection are directly sent via the *measure* output connection (lines 21 and 22). After executing their respective instructions, each sub-behavior self-recurses (i.e., it continues being executed), as expressed by the `behavior()` call in lines 18 and 23.

Figure 13 – π-ADL description of the *Sensor* component

```
1:   component Sensor is abstraction() {
2:      type MV is Real
3:      type CmH2O is Real
4:      connection sense is in(MV)
5:      connection pass is in(CmH2O)
6:      connection measure is out(CmH2O)
7:      protocol is {
8:         ((via sense receive MV | via pass receive CmH2O)
9:           via measure send CmH2O)*
10:     }
11:     behavior is {
12:        convertRawData is function(measure : MV) : CmH2O {
13:           unobservable
14:        }
15:        choose {
16:           via sense receive d : MV
17:           via measure send convertRawData(d)
18:           behavior()
19:
20:           or
21:           via pass receive m : CmH2O
22:           via measure send m
23:           behavior()
24:        }
25:     }
26:  }
```

***Gateway* component.** Figure 14 shows the specification of the *Gateway* component in π-ADL. This component comprises two connections, namely (i) *data*, an input connection used for receiving data collected by sensor nodes (line 3), and (ii) *alert*, an output connection used for sending alert messages in case of risk of flood. The protocol of this component (lines 5 to 8) enforces receiving a value of the *CmH2O* type (declared in line 2) and sequentially sending a string value, actions performed at multiple times. Next, the behavior of this component encompasses the definition of two functions. The *calculateHI* function (line 10) calculates the hazard index based on the water level measures given as input. In turn, the *triggerAlert* function (line 14) uses the previously calculated hazard index to determine the severity of the flood risk [63] and send a message accordingly. Therefore, data received via the *data* input connection are provided as input to the *triggerAlert* function (line 28), which will analyze the potential of flood risk according to the calculated hazard index (lines 17 to 25) and then

send the corresponding message via the *alert* output connection (line 29). After executing these instructions, the behavior continues being executed (line 30).

Figure 14 – π-ADL description of the *Gateway* component

```
1:   component Gateway is abstraction() {
2:      type CmH2O is Real
3:      connection data is in(CmH2O)
4:      connection alert is out(String)
5:      protocol is {
6:         (via data receive CmH2O
7:          via alert send String)*
8:      }
9:      behavior is {
10:        calculateHI is function(data : CmH2O) : Real {
11:           unobservable
12:        }
13:
14:        triggerAlert is function(measure : CmH2O) : String {
15:           hi is Real
16:           hi = calculateHI(measure)
17:           if (hi > 0.0 && hi < 0.5) then {
18:              return "Low risk"
19:           } else if (hi >= 0.5 && hi < 1.0) then {
20:              return "Medium risk"
21:           } else if (hi >= 1.0 && hi < 1.4) then {
22:              return "High risk"
23:           } else {
24:              return "Very high risk"
25:           }
26:        }
27:
28:        via data receive d : CmH2O
29:        via alert send triggerAlert(d)
30:        behavior()
31:     }
32:  }
```

***ZigBee* connector.** Figure 15 shows the specification of the *ZigBee* connector in π-ADL, which represents a ZigBee wireless connection. As shown in lines 3 and 4, this connector comprises an input connection (*from*) for receiving data and an output connection for sending data (*to*). The protocol of this connector specifies receiving data of the *CmH2O* type via the *from* connection and then sending string data via the *to* connection at multiple times (lines 5 to 8). Furthermore, the behavior of this connector encompasses an input prefixing action (line 10) for receiving data via the *from* connection and an output prefixing action (line 11) for sending data via the *output* connection. After executing these instructions, the behavior recurses (line 12).

Figure 15 – π-ADL description of the *ZigBee* connector

```
1:  connector ZigBee is abstraction() {
2:     type CmH2O is Real
3:     connection from is in(CmH2O)
4:     connection to is out(CmH2O)
5:     protocol is {
6:        (via from receive CmH2O)
7:         via to send CmH2O)*
8:     }
9:     behavior is {
10:       via from receive m : CmH2O
11:       via to send m
12:       behavior()
13:    }
14: }
```

***WSNFloodMonitoring* architecture.** Figure 16 shows the π-ADL specification of the *WSNFloodMonitoring* architecture, based on the one previously depicted in Figure 7 (see Section 3.3). The behavior of this abstraction comprises the composition of three instances of the sensor component (lines 4 to 6), three instances of the ZigBee connector (lines 7 to 9), and one instance of the gateway component (line 10). The attachments of these architectural elements take place through six unifications (lines 12 to 17) that represent data flows from an architectural element to another. Therefore, such unifications of connections allow transmitting data from sensors until reaching the gateway by using the created links.

Figure 16 – π-ADL description of the *WSNFloodMonitoring* architecture

```
1:  architecture WSNFloodMonitoring is abstraction() {
2:     behavior is {
3:        compose {
4:               S1 is Sensor()
5:           and S2 is Sensor()
6:           and S3 is Sensor()
7:           and L1 is ZigBee()
8:           and L2 is ZigBee()
9:           and L3 is ZigBee()
10:          and Gw is Gateway()
11:       } where {
12:          S1::measure unifies L1::from
13:          S2::measure unifies L2::from
14:          L1::to      unifies S3::pass
15:          L2::to      unifies S3::pass
16:          S3::measure unifies L3::from
17:          L3::to      unifies Gw::data
18:       }
19:    }
20: }
```

4.6.2 Exogenous reconfiguration: Low battery of a sensor node

For the sake of simplicity, consider the architecture of the flood monitoring system shown in Figure 7 (see Section 3.3). In addition, consider a situation in which the battery of sensor *S3* is low, thus requiring the replacement of this mote by another one. Despite this new mote is near to the other two ones, it is far from the gateway station, so that it is not possible to use ZigBee as wireless connection. For this reason, a GPRS wireless connection needs to be used as it is suitable for communications over long distances.

To realize this reconfiguration by means of an exogenous approach, consider the *WSNFloodMonitoring* architecture described in Figure 16 as the initial architecture. Figure 17 shows a partial π-ADL description of the *WSNFloodMonitoringEvol* evolved architecture, resulted from reconfiguration actions to be applied on the initial architecture. First, *WSNFloodMonitoring* is decomposed into a sequence of seven detached behaviors (*abs*), each one associated to the architectural elements previously instantiated (line 5)[10]. In lines 6 to 16, the previous instances of the sensor and gateway components and of the ZigBee connectors are composed with a new instance of the sensor component (*S4*) and a new instance of the GPRS connector (*Gprs1*), which is similar to the ZigBee connector described in Figure 15. Next, the connections of these elements are unified to attach them (lines 14 to 21). The reconfiguration is triggered when a value is sent via the connection *lowb* in the initial architecture (*iarch*), thereby indicating that the battery level of the sensor is low (line 28). In this case, the coordinating behavior performs an application to run *WSNFloodMonitoringEvol* (line 30), i.e., the evolved version of the initial architecture to realize the reconfiguration itself.

Figure 17 – Partial π-ADL description of the *WSNFloodMonitoringEvol* evolved architecture for realizing a reconfiguration aimed to replace a sensor mote due to low battery level by means of an exogenous approach.

```
1:  architecture WSNFloodMonitoringEvol is
2:     abstraction(lowb : connection[Boolean], iarch : Any) {
3:     behavior is {
4:        abs is sequence[Any]
5:        abs = decompose iarch     // decomposing the initial architecture (iarch)
6:        compose {
```

[10] The elements representing the architectural abstractions (behaviors) are assigned to the *abs* sequence in the order in which they were previously declared. These elements can be directly accessed by using integer indexes or the `iterate` construct.

```
7:            abs[0] is Sensor()    // previous Sensor component instance (S1)
8:        and abs[1] is Sensor()    // previous Sensor component instance (S2)
9:        and S4     is Sensor()    // new Sensor component instance
10:       and abs[3] is ZigBee()    // previous ZigBee connector instance (Zb1)
11:       and abs[4] is ZigBee()    // previous ZigBee connector instance (Zb2)
12:       and Gprs1  is GPRS()      // new GPRS connector instance
13:       and abs[6] is Gateway()   // previous Gateway component instance (Gw)
14:     } where {
15:       abs[0]::measure unifies abs[3]::from
16:       abs[1]::measure unifies abs[4]::from
17:       abs[3]::to      unifies S4::pass
18:       abs[4]::to      unifies S4::pass
19:       S4::measure     unifies Gprs1::from
20:       Gprs1::to       unifies abs[6]::data
21:     }
22:   }
23: }
24:
25: behavior is {
26:   connection lowb is in(Boolean)
27:   iarch = WSNFloodMonitoring(lowb)
28:   via lowb receive v : Boolean
29:   if (v == true) then {
30:     WSNFloodMonitoringEvol(iarch)
31:   }
32: }
```

4.6.3 Endogenous reconfiguration: Avoiding false positives

Consider another situation in which a flood was detected based on data collected by the motes. As an evacuation procedure might be expensive, it is necessary to improve the accuracy of measures aiming at avoiding false positives. For this purpose, drones endowed with digital cameras and WiFi networking capabilities can be used to capture images from the river in order to estimate its flow rate. Captured images can be sent to the gateway station, which will effectively process and combine them with data provided by the motes spread along the river, thereby confirming whether a flood is imminent or not.

Figure 18 shows a partial description in π-ADL realizing this reconfiguration through an endogenous approach. Lines 1 from 8 show the specification of the *UAV* component, which comprises the *camera* input connection for receiving images captured by the embedded digital camera (line 3), and the *output* output connection for sending the captured images (line 4). In turn, lines 10 to 17 show the specification of the *WiFi* connector, similar to the *ZigBee* connector described in Figure 15. Lines 19 to 62 show a new specification for the *Gateway* component (previously described in Figure 14), which now comprises an additional input connection for receiving images (*image*, line 23) and a function for processing them (*processImage*, lines 33 to 35). To

apply the required changes for adding a drone component connected to the gateway via WiFi, the gateway component first verifies if data provided by the sensor motes indicate imminent risk of flood (line 44). If such a risk is classified as high or very high, then the gateway component performs a composition behavior that creates an instance of the *UAV* component (*dr*, line 46), creates an instance of the *WiFi* connector (*wf*, line 47), and attaches these new instances by unifying their respective connections (lines 49 and 50). It is worth mentioning that the initial architecture *WSNFloodMonitoring* presented in Figure 7 does not need to be decomposed as the reconfiguration does not require detaching architectural elements, but creating and attaching new ones.

Figure 18 – Partial π-ADL description for realizing a reconfiguration aimed to increase accuracy and avoid false positives by means of an endogenous approach.

```
1:  component UAV is abstraction() {
2:     type Image is Any
3:     connection camera is in(Image)
4:     connection output is out(Image)
5:     behavior is {
6:        unobservable
7:     }
8:  }
9:
10: connector WiFi is abstraction() {
11:    type Image is Any
12:    connection input is in(Image)
13:    connection output is out(Image)
14:    behavior is {
15:       unobservable
16:    }
17: }
18:
19: component Gateway is abstraction() {
20:    type CmH2O is Real
21:    type Image is Any
22:    connection data is in(CmH2O)
23:    connection image is in(Image)
24:    connection alert is out(String)
25:    protocol is {
26:       ((via data receive CmH2O | via image receive Image)
27:         via alert send String)*
28:    }
29:    behavior is {
30:       calculateHI is function(data : CmH2O) : Real {
31:          unobservable
32:       }
33:
34:       processImage is function(i : Image) : Boolean {
35:          unobservable
36:       }
37:
38:       triggerAlert is function(measure : CmH2O) : String {
39:          unobservable
40:       }
```

```
41:         via data receive d : CmH2O
42:         risk is String
43:         risk = triggerAlert(d)
44:         if (risk == "High" || risk == "Very high") then {
45:           compose {
46:                  dr is UAV()
47:              and wf is WiFi()
48:           } where {
49:              dr::output unifies wf::input
50:              wf::output unifies self::image
51:           }
52:         }
53:         via image receive i : Image
54:         if (processImage(i) == true) then {
55:           via alert send "Flood risk confirmed"
56:         } else {
57:           via alert send risk
58:         }
59:         behavior()
60:      }
61:   }
```

4.7 Related work: Languages for describing dynamic software architectures

Bradbury et al. [43] report that some ADLs have been proposed in the last 15 years for specifying dynamic software architectures aiming at allowing for their automated, rigorous analysis. In particular, most works address programmed dynamic reconfiguration by providing specific reconfiguration primitives at the architectural level to describe when and how the system architecture shall be reconfigured. This section briefly discusses some of these approaches.

One of the earliest ADLs addressing dynamic software architectures is Darwin [22], a formal declarative language with an operational semantics based on FSP and that allows hierarchically specifying distributed systems. Dynamic behavior is defined in Darwin by means of *lazy* and *direct* instantiations: in the former, each component is not instantiated until one of its services is requested; in the latter, components are directly instantiated. However, Darwin only allows instantiating components, but not removing them neither creating/destructing links.

Dynamic Wright [23] is a formal ADL that allows describing the behavior and reconfiguration of a system by using a variant of CSP. However, CSP is able to specify only static configurations, i.e., dynamic reconfigurations are not supported and have to be simulated. By adopting an exogenous approach, Dynamic Wright provides a

configuror, a special component in the architecture responsible for centralizing all reconfiguration operations since only it can modify the architecture. The supported reconfiguration operations are *new*, *del*, *attach*, and *detach*, respectively used to create and remove instances, and to link and unlink them.

LEDA [84] is another formal ADL based on π-calculus. It is structured upon: (i) *components*, which represent system modules and can be either functional elements or connectors; (ii) *roles*, which describe the observable behavior of components; and (iii) *attachments*, which define connections among component instances. The approach adopted in LEDA for dynamism is endogenous, decentralized, so that the reconfiguration operations are described along with the behavior specification of components. The main operations are (i) the instantiation of components (processes) and (ii) dynamic attachments between them by using channel mobility capabilities provided by π-calculus. However, there are no means of detaching and removing components. In addition, the behavior of architectural elements is specified by directly using the operators defined in π-calculus as-are, i.e., unlike π-ADL, LEDA does not provide architectural abstractions over these constructs, thus making architecture descriptions more difficult.

The Architecture Analysis and Design Language (AADL) [90] is a language intended to describe both software and hardware architectures of distributed real-time embedded systems. This language allows specifying reconfigurable systems by using state machines that describe *modes* and *mode transitions*: modes represent particular (state) configurations whereas transitions specify events that enable the system to be reconfigured, i.e., changed from the current mode to another, so that modes and transitions in AADL are programmed, statically defined. As AADL is used to specify embedded systems at a low level (e.g., in terms of processors, platforms, etc.), modeling reconfigurations in such a language is constrained to a specific system and platform. Moreover, AADL lacks of a native formal semantics, thereby severely limiting both unambiguous communication and the use of formal analysis techniques.

Plastik [91] is a framework for reconfiguration of component-based systems that extends the well-known Acme/Armani ADLs [20, 92] to allow describing dynamic reconfiguration operations. In terms of programmed reconfiguration, Plastik introduces the *on-do* construct, a predicate-action element to specify reconfiguration actions when the specified predicate is true. In addition to the typical operators for creating, removing, and attaching architectural elements, two new explicit operators are introduced, namely *detach* and *remove*, for respectively detaching and destroying architectural elements. Finally, there is a construct for specifying dependencies between architectural elements. Despite of its ease of use for specifying programmed dynamic reconfiguration, the Acme/Armani/Plastik ensemble lacks of formal underpinnings to allow further automated architectural analysis.

To sum up, Table 6 shows a comparative analysis among the aforementioned ADLs. Considering the taxonomic dimensions for characterizing dynamic reconfiguration approaches discussed in Section 2.1, it is clearly possible to notice that π-ADL is able to provide a comprehensive notation for describing dynamic software architectures while paving the way for their formal verification.

Table 6 – Comparative analysis of existing ADLs considering some taxonomic dimensions for characterizing dynamic reconfiguration approaches in software architectures

ADL	**Underlying formalism**	**Description viewpoint**		**Reconfiguration operations**				**Reconfiguration management**	
		Structural	**Behavioral**	**Create**	**Remove**	**Attach**	**Detach**	**Exogenous**	**Endogenous**
Darwin [22]	FSP	✓	✓	✓	○	×	×	×	✓
Dynamic Wright [23]	CSP-based	✓	○	✓	✓	✓	✓	×	✓
LEDA [84]	π-calculus	✓	✓	✓	×	✓	×	✓	×
AADL [90]	N/A	✓	✓	○	○	○	○	✓	×
Plastik [91]	N/A	✓	×	✓	✓	✓	✓	✓	×
π-ADL	**π-calculus**	✓	✓	✓	○	✓	○	✓	✓

Key:
✓ = supported
○ = partially supported
× = not supported
N/A = not applicable
? = lack of evidences

5 Architecture-based code generation

Besides capturing important design decisions about a system at a high abstraction level, software architectures can be used to derive the implementation of such a system. Taylor et al. [17] point out that relating software architectures to implementation is a *mapping* problem. A good, well-analyzed architectural design has a limited value unless there is a clear, direct relationship between the architectural level and artifacts at the implementation level [6]. In this context, choosing how to create and maintain this mapping is critical in architecture-based software development since the less complete or automated it is, the more opportunity for an architectural drift exists. For this reason, it is highly desirable providing some sort of mechanism that allows generating source code from architecture descriptions. Manually converting an architectural model into a running application may result in many problems in terms of consistency and traceability between the architecture and its implementation [17]. Furthermore, maintaining traceability between architecture and implementation helps system developers to easily understand architecture designs and provides support for software quality control and maintenance [93].

Supporting code generation by translating architecture descriptions to a target programming language is a research subject addressed in the literature since the dawn of Software Architecture as a discipline in the 1990s [94, 95]. This concern has remained relevant along the years mainly due to the concern of maintaining conceptual integrity between the representation of a software architecture and its corresponding implementation. Nonetheless, the existing gap between these two levels has been sometimes neglected and became a more severe problem considering the inherent dynamicity of many contemporary software systems. As previously highlighted, even if a system is built in conformance to the defined prescriptive architecture, its implementation has a significant probability of becoming inconsistent with respect to the original architecture over time.

Aiming at tackling such an existing gap between architectural and implementation levels while contributing to minimize the risk of architectural drifts and allowing for the validation of the architecture itself, this chapter presents a process to generate source code in the Go programming language [36] from architecture descriptions expressed in the π-ADL, another contribution brought by this work [35, 37]. Section 5.1 defines the correspondences between the elements of π-ADL and Go. Section 5.2 presents the automated process for generating source code from architecture descriptions. Section 5.3 shows the generation of source code in Go from the π-ADL architecture description of the flood monitoring system. Finally, Section 5.4 briefly discusses related work on the support for implementing software architectures.

5.1 Correspondences between π-ADL and Go

As previously mentioned, Go was chosen to serve as target language for generating implementation-level artifacts from architecture descriptions in π-ADL because it is suitable for constructing scalable distributed systems and handling multicore and networked computer architectures, as required by many contemporary systems. The integration of π-ADL and Go is fostered mainly by their common basis on the π-calculus process algebra [81] and the straightforward relationship between elements of these languages, such as the use of *connections* in π-ADL and *channels* in Go as means of communication and synchronization between concurrent processes. This section defines the correspondences between the elements of these languages.

Table 7 summarizes the correspondences between the main architecture-level elements of π-ADL and implementation-level elements of Go, each one described in the following. Relationships regarding other elements such as statements and expressions were omitted as they are practically identical in both languages and therefore straightforward.

Table 7 – Summary of correspondences between the main architecture-level elements of π-ADL and implementation-level elements of Go

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

Architectural abstractions and their behavior. Components, connectors, and architectures are defined in π-ADL as abstractions over behaviors. In Go, these elements are represented by functions called as goroutines (see Section 2.2), thereby being equivalent to the notion of communicating processes in π-calculus. The signature of such functions is defined by the respective name of the architectural abstraction and the list of parameters that they require. In turn, the body of these functions comprises the respective element behavior.

Connections. As introduced in Section 4.1, channels are among the main

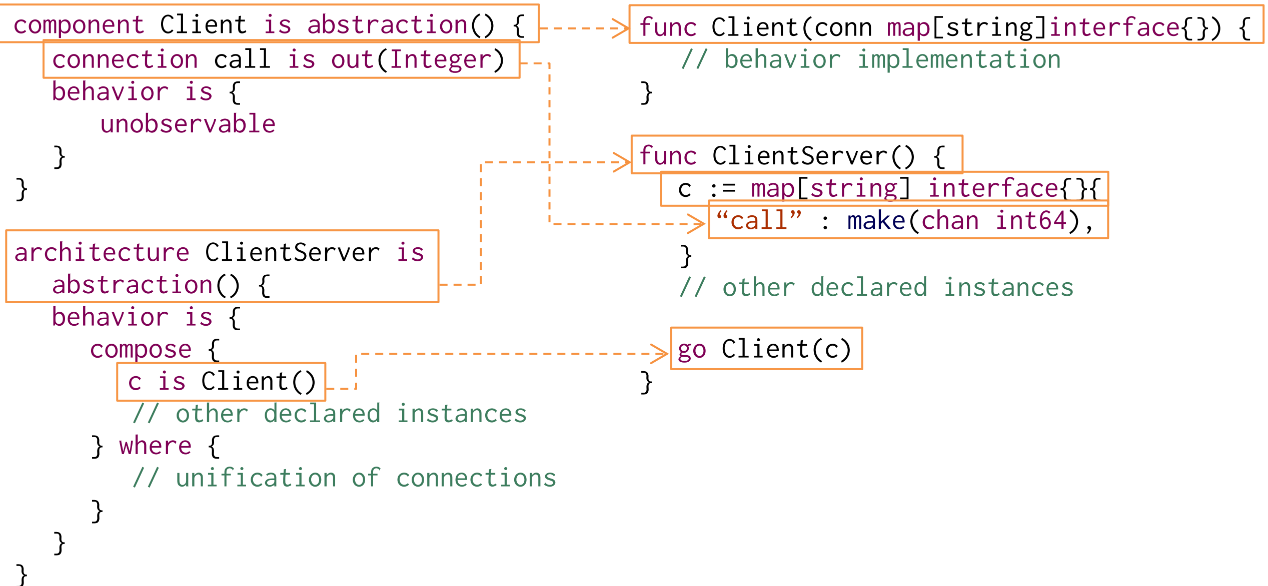
elements of the π-calculus process algebra and they are used as means of communication and synchronization among concurrent processes. In π-ADL, connections are used for sending/receiving values between architectural abstractions and their behaviors. Straightforwardly to π-calculus, typed channels in Go are used 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 the values to be transmitted through a channel is the one specified in the connection declaration.

Connection declarations. The set of connections comprised in the structure of both components and connectors is represented in Go by <*string*, *channel*> maps[11] of channels. These maps use the names of the declared connections as keys and map such keys to channel objects representing the connections themselves.

Instantiation of architectural elements. In Go, the instantiation of components and connectors within an architecture encompasses two steps. The first one refers to the creation of the maps of channels representing the set of connections comprised by such architectural elements. The second one involves to launch the goroutines that respectively represent these architectural elements within the function associated to the architecture. In these goroutine calls, the map of channels representing the created instance is provided as parameter to the goroutine. When performing reconfiguration operations to create new components/connectors, it is necessary to make new calls to the respective goroutines implementing them.

As an example, consider the aforementioned simple client-server architecture composed of one client component and one server component connected through a link connector. The creation of an instance of the client component (*c*) first encompasses creating the map of channels that represents the set of connections declared for this component. Next, this component is run by calling the *Client* goroutine and providing the previously created map of channels as input parameter.

[11] A *map* (a.k.a. *associative array*) is an unordered, non-sequential collection of pairs used to search for a value through a key, which works as an index that allows accessing the value associated to it.



Unification of connections. In π-ADL, a unification of connections allows attaching an output connection of a component/connector to an input connection of another component/connector, thereby enabling these elements to communicate and exchange data. This process is implemented in Go by means of a special-purpose function called *unifies*, which receives the connections to be unified as parameters. Such a function basically reads the contents of the sending channel (output connection) and writes them to the receiving channel (input connection).

```
func unifies(sender, receiver interface{}) {
  if (sender != nil && receiver != nil) {
    v, _ := reflect.ValueOf(sender).Recv() // reading from output connection
    reflect.ValueOf(receiver).Send(v)      // writing to input connection
  }
}
```

Connection detachment. As described in Section 4.5, the execution of a decomposition operation automatically removes the existing unifications between the architectural elements previously composed. When mapping from π-ADL to Go, removing these attachments is equivalent to closing the communication channels used to synchronize the goroutines that represent architectural elements. In Go, closing an unbuffered communication channel[12] indicates that no more values will be transmitted through it and leads to an immediate blockage of the goroutines that use such a channel.

Coordinating behavior. To enable a system architecture to execute, a

[12] In Go, communication is *synchronous* and *unbuffered*: a sending operation does not complete until there is a receiver to accept the value. Therefore, send/receive operations block until the other side is ready.

coordinating behavior performs an *application* (similar to a call) of the abstraction corresponding to such an architecture. In Go, this behavior represented by the main function, which is the first function called when a Go program is executed. Therefore, the main function calls the goroutine that represents the architecture itself, which in turn calls the goroutines associated to the instances of architectural elements (components and connectors).

Base, constructed, and collection types. As presented in Section 4.2, π-ADL provides three data types: (i) *base types*, which are used to express atomic values; (ii) *constructed types*, composite types constructed upon base types; and (iii) *collection types*, which are types representing collections based on base and/or constructed value types. Table 8 summarizes the mappings from these types defined in π-ADL to data types in Go.

Table 8 – Summary of the mappings from data types defined in π-ADL to types in Go

Type	**π-ADL**	**Go**	
		Syntactic representation	**Semantics**
Basic types	Natural	`uint64`	Unsigned integer numbers
	Integer	`int64`	Signed integer numbers
	Real	`float64`	Floating-point numbers
	Boolean	`bool`	Boolean logical values
	String	`string`	Character strings
Constructed types	Tuple	`[`*n*`]interface{}`	Empty interface array of size *n* (*n* is the number of composing types)
	View	`map[string]interface{}`	Map whose keys are the labels for the view values
Collection types	Set	`map[`*T*`]bool`	Map with keys of type *T* of the set
	Sequence	`[]`*T*	Slice (dynamic array) with elements of type *T*

π-ADL also provides a special base type named `Any`, which works as a generic type in the language and admits values of any type (see Section 4.2.1). For similar purposes, this type is mapped to empty interfaces (`interface{}`), which represent means of generic typing in Go. As empty interfaces do not have defined methods, any type is able to satisfy them.

Behavior constructs. Table 9 summarizes the mappings from behavior types defined in π-ADL (see Section 4.3) to Go.

Table 9 – Summary of the mappings from behavior constructs defined in π-ADL to Go

π-ADL behavior construct	Go	
	Syntax representation	Semantics
Output prefixing action	`conn["c"].(chan T) <- v`	Send value v of type T via channel (connection) c
Input prefixing action	`v := <- conn["c"].(chan T)`	Receive value s of type T from channel (connection) c
Unobservable action	`// Empty block`	Empty block[13]

Non-deterministic choice	`select {`
	`case p1:`
	`B1`
	`case p2:`
	`B2`
	`case pn:`
	`Bn`
	`}`
		Selection of a block (corresponding to a sub-behavior B_i) to execute based on receiving/sending operations over channels p_i (prefixing actions)

Parallel composition	`go func() {`
	`B:`
	`}()`
		Creation and invocation of a goroutine for each sub-behavior B_i

Inaction	`return`	Empty return

5.2 Code generation procedure

Taylor et al. [17] distinguish two main generative approaches in the relationship between architectural models and implementation artifacts. *One-way* approaches allow a software artifact to be generated from another, e.g., generating source code from architectural models. In turn, *round-trip* approaches allow changes in the target artifact to be automatically reflected-back in the source artifact. For example, in a one-way approach, a component in an architectural model might result in the creation of a new Java package containing class files. In a round-trip approach, the creation of a new Java package might result in a new component in the architectural model as well. Although round-trip approaches are preferable to one-way ones, they are generally tricky to be correctly implemented, especially when architectural modeling and code development tools are not well integrated (as it is often the case).

This section describes the process for producing source code in Go from architecture descriptions in π-ADL by following a one-way generative approach.

[13] The generated empty block can be filled with any instruction at the developer's choice since a silent prefixing action refers to details that are not observable at the architectural level.

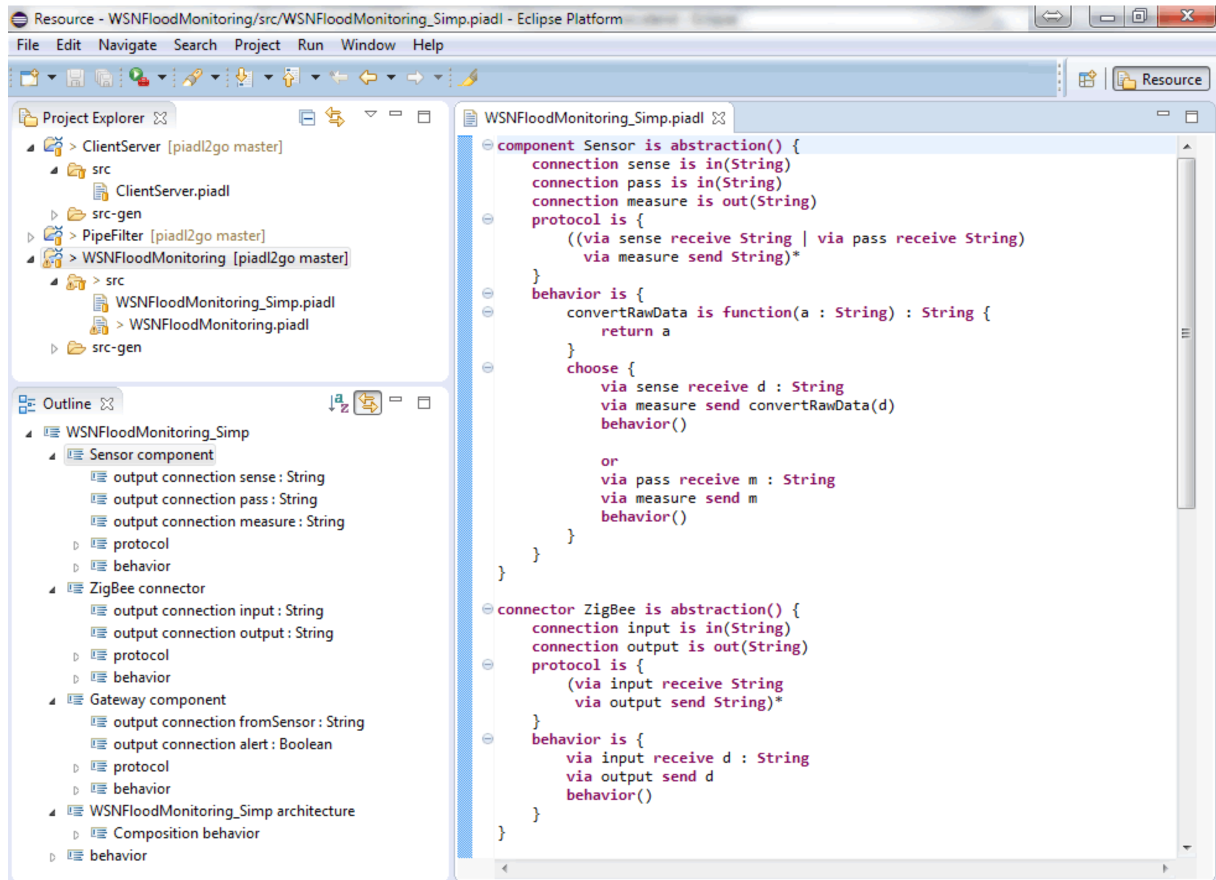
Section 5.2.1 first gives an overview of the tool developed to assist software architects in architecture description using the π-ADL language. Next, Section 5.2.2 presents the technical elements used to realize the mapping between π-ADL and Go defined in Section 5.1 towards automatically generating source code.

5.2.1 π-ADL textual editor

One of the concerns regarding the usefulness of an ADL is directly related to the tools that it provides for supporting the activities encompassed by an architecture-driven software development approach, i.e., architectural description, analysis, code generation, and evolution [17]. Tool support is especially vital for the successful use of any ADL since architecture descriptions persist throughout the development lifecycle and evolve along with the described software system, so that ADL tools play an important role for creating, analyzing, and maintaining these documents over time. Furthermore, a recent survey about ADLs in the industry context has revealed the importance of ADL tools in this scenario, despite few description languages for software architectures are supported by satisfactory tools [18]. Such a survey identified some requirements for these tools, such as simplicity, intuitiveness, high degree of usability, provisioning of comprehensive textual and graphical notations, meaningful communication and documentation, and alignment of software architecture descriptions with their respective implementation.

In the context of the ArchWare European Project in which the π-ADL language was originally conceived, an open-source toolset was developed in order to support formal description, analysis, refinement, code generation, and evolution of software architectures [96, 97]. Altogether, these tools allow compiling architectural models into their executable representations as well as the formal analysis and evolution of such models. However, considering the evolutions in the π-ADL language along the years and the requirements of new generation software systems (e.g., distribution, large-scale, concurrency, and dynamicity), this work contributed with the development of a completely new tool support for π-ADL. As a first step, a textual editor based on the Eclipse platform [98] was developed for assisting architects to make architecture descriptions using the π-ADL language and further automatically generate implementation code in Go. A screenshot of the tool is shown in Figure 19.

Figure 19 – Screenshot of the Eclipse-based π-ADL textual editor



The π-ADL textual editor was developed upon Xtext [99], a well-known open-source, highly customizable framework for developing domain-specific languages (DSLs). Xtext covers all aspects of a complete language infrastructure by parsing textual models written in such a language and allows generating code from them in another language. Furthermore, this infrastructure is fully integrated with the Eclipse development environment and provides the π-ADL textual editor with useful features, such as:

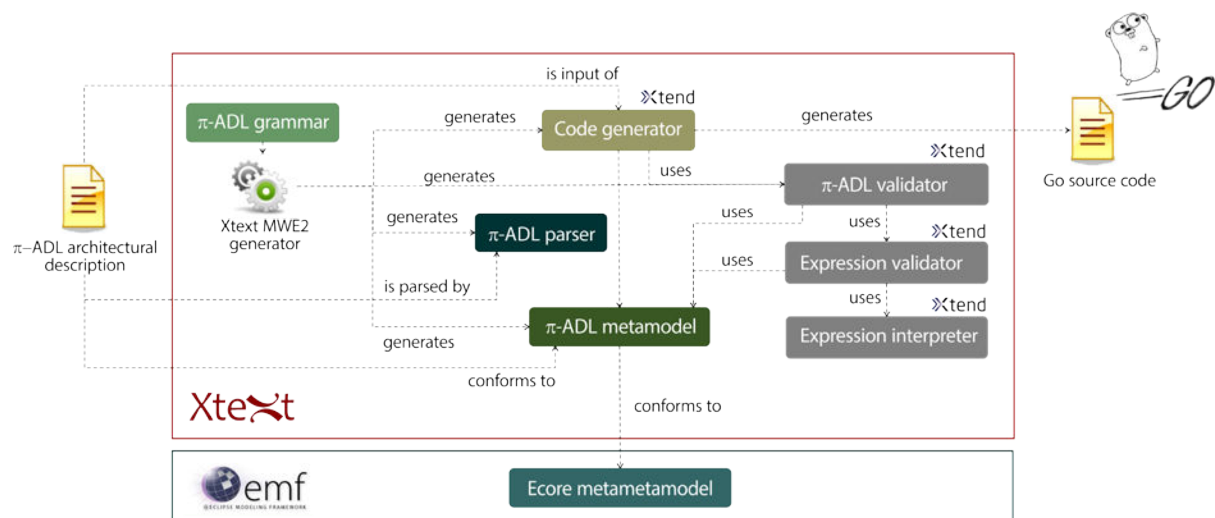
- *syntax highlighting*, a useful feature for distinguishing keywords from identifiers;
- *syntactic and semantic validation* of architecture descriptions by showing error and warning alerts, thereby enabling architects to early detect and fix errors and potential problems;
- *interpretation of expressions*, which is useful for type checking purposes;
- *auto formatting*;
- *content assist*, which provides suggestions on how to complete a given statement/expression based on the syntactic rules; and
- *automatic build on save* for automatically generating Go source code from the architecture description in π-ADL (if they are correct according to the

syntactic and validation rules of the language) when it is saved in the language editor.

5.2.2 Code generation procedure

Figure 20 depicts the technical elements related to the generation source code in Go from architecture descriptions in π-ADL. From the π-ADL grammar specification in the Extended Backus-Naur Form (EBNF) [100], Xtext automatically generates the π-ADL infrastructure by running a script written in the Modeling Workflow (MWE2) DSL. More specifically, this script (also automatically created by Xtext) is used to derive a specification from the π-ADL grammar compatible with the ANTRL parser generator [101], which is used for generating the parser of the language. When running the script, Xtext generates a set of artifacts: (i) a Java implementation of a *parser*, which is automatically generated by ANTRL and is responsible for the syntactic analysis of the architecture textual description; (ii) a *metamodel* defining the abstract entities of the language and the relationships among them; (iii) the entry point for a *code generator*, which is used to generate code in Go from the architecture description in π-ADL; and (iv) an Eclipse-based *code editor* for assisting the textual description of a software architecture in π-ADL. In addition, Xtext creates an *abstract syntax tree* (AST) from the input textual model for representing the structure of the parsed model as well as the respective Java classes to persist such an AST.

Figure 20 – Elements for generating Go source code from π-ADL architecture descriptions



The code generator within the π-ADL textual editor is implemented by using facilities provided by the Xtend programming language [102], a fully Java-interoperable programming language featuring a more compact, easier to use syntax, as well as advanced features such as type inference and lambda expressions. As the AST model needs to be continuously traversed, Xtend provides useful mechanisms to

straightforwardly doing this while being easy to use and allowing for a better readable code. Moreover, as Xtend programs are compiled to plain Java code, they can access all of the libraries available in Java, thus allowing these languages to coexist seamlessly.

Once the architecture description is checked as correct by the parser, it might still have errors since its overall correctness cannot always be determined during the parsing procedure. Besides performing such a syntactic analysis of the textual architecture description against the rules defined in the π-ADL grammar, the code generator makes use of some *validators* to semantically analyze a π-ADL architecture description. In Xtext, validators are classes that contain methods (validation rules) implementing additional constraint checks over the abstract elements of the current model. These validation rules (also implemented in Xtext) are detailed in Appendix B.

In order to describe the behavior of components and connectors, a software architect can make use of *expressions* that are similar to the ones used in programming languages, e.g., logical, relational, equality, and arithmetic expressions. Determining the data types handled by such expressions is important for ensuring that an expression value sent via a connection is the one expected, i.e., its type is equal to the type specified when declaring such a connection. However, type checking cannot be performed during parsing and expressions are resolved at runtime, i.e., their value is calculated while they are described. As Xtext is mainly concerned with syntactic analysis and it does not support expression resolution, an *interpreter* and a *validator* were developed in Xtend for handling expressions in π-ADL architecture descriptions.

Finally, the code generator itself uses *extension methods* and *template expressions*. Extension methods implement how a given abstract element of the input π-ADL architecture textual description (defined in the AST) can be translated to its representation in the source code to be generated. These extension methods make use of template expressions, which allow for more readable string concatenation and proper formatting to write characters into the output Go source code. These mechanisms provided by Xtend are used to translate the abstract elements defined in the π-ADL grammar to their respective implementation in Go based on the correspondences defined in Section 5.1.

5.3 Generating code for the flood monitoring system

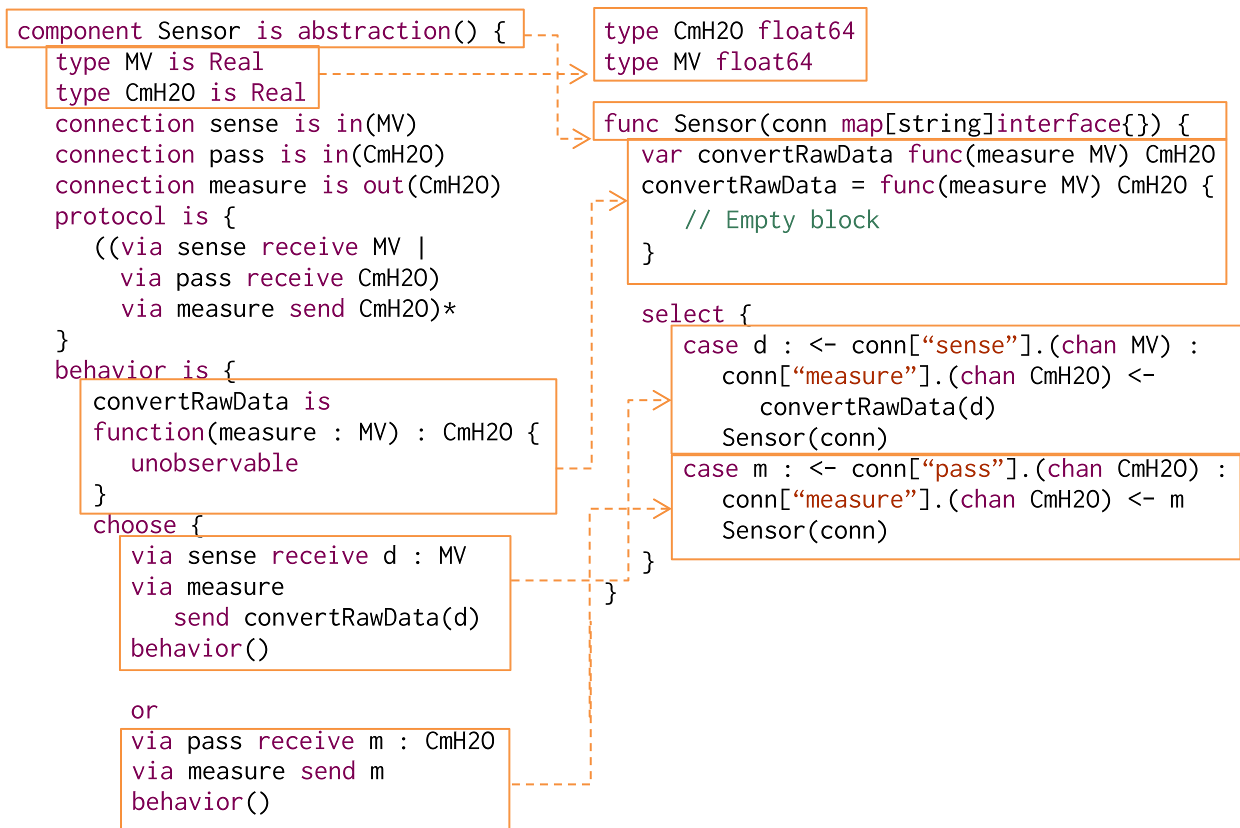
This section illustrates how the π-ADL architecture description of the flood monitoring system presented in Section 4.6 can be automatically translated to its respective implementation in Go, including the programmed reconfiguration operations performed at runtime. It is worth mentioning that Go has recently started to be used as a support platform to some frameworks targeting sensors, such as Patchwork [103], EMDB [104], and Gobot [105]. These frameworks have chosen Go for

implementation aiming at leveraging the benefits promoted by the language to the device realm, in particular cross-platform building directly to native code, performance, productivity, and easy application development. Nevertheless, Go programs consume more memory than C/C++ code and compilation artifacts are larger when compared to the ones produced by other languages, issues that may be problematic in a very limited physical device.

5.3.1 Architectural elements

***Sensor* component.** Figure 21 shows an excerpt of the Go code generated from the π-ADL description of the *Sensor* component (see Figure 13). This component is implemented in Go by the *Sensor* function, which receives a map of channels (*conn*) representing its declared connections. The *Sensor* function also comprises the declaration of a local function (closure) corresponding to the *convertRawData* function specified in the component behavior. The `select` instruction is used for representing the non-deterministic choice behavior in terms of selecting the sub-behavior to be executed according to the order in which values are received through the channels (connections). Therefore, the value to be written to the *measure* output connection can be either the one received via the *sense* input channel (sensed data) or the one received via the *pass* input channel (data from another sensor). The *MV* and *CmH2O* data types declared within the component behavior are respectively translated to global type declarations in the form `type` *T U*, in which *T* is an alias for the underlying type *U*.

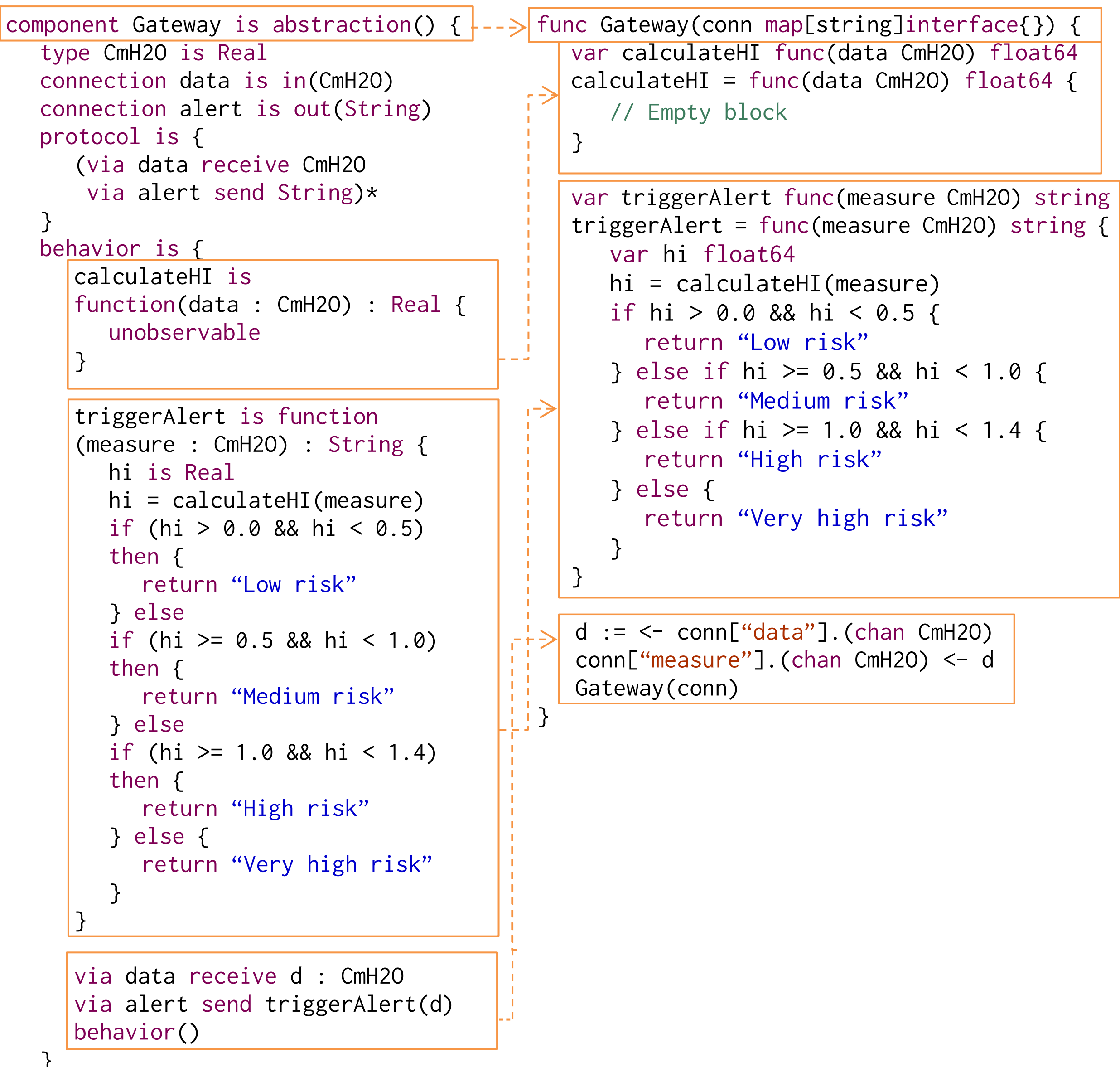
Figure 21 – Description of the *Sensor* component in π-ADL (left) and corresponding implementation in Go (right)



```
      }
    }
  }
```

***Gateway* component.** Figure 22 shows an excerpt of the Go code generated from the π-ADL description of the *Gateway* component (see Figure 14). This component is implemented in Go by the *Gateway* function, which receives as parameter a map of channels (*conn*) representing its declared connections. The *Gateway* function also comprises the declaration of two local functions (closures) corresponding to the *calculateHI* and *triggerAlert* functions specified in the component behavior. Note that the statements implementing these functions are practically identical to the ones used when describing this component in π-ADL.

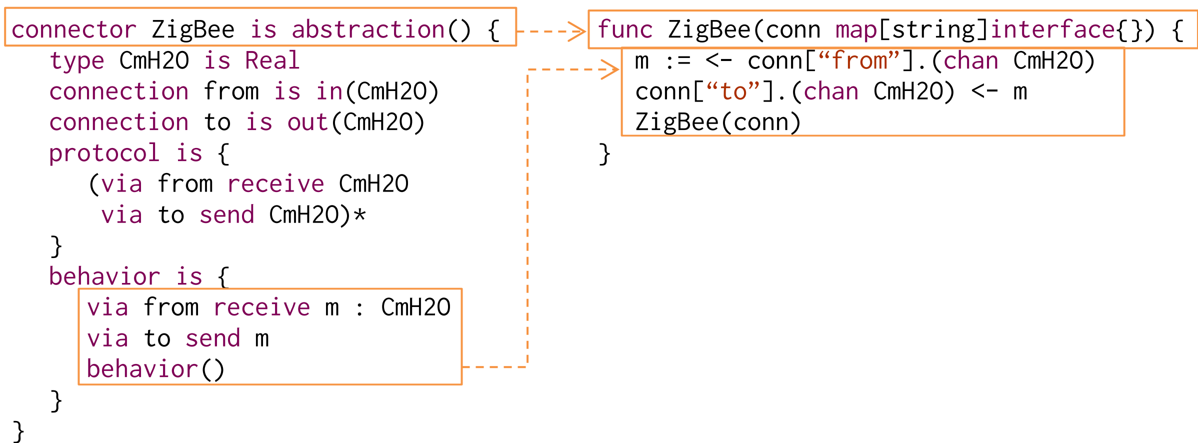
Figure 22 – Description of the *Gateway* component in π-ADL (left) and corresponding implementation in Go (right)



}

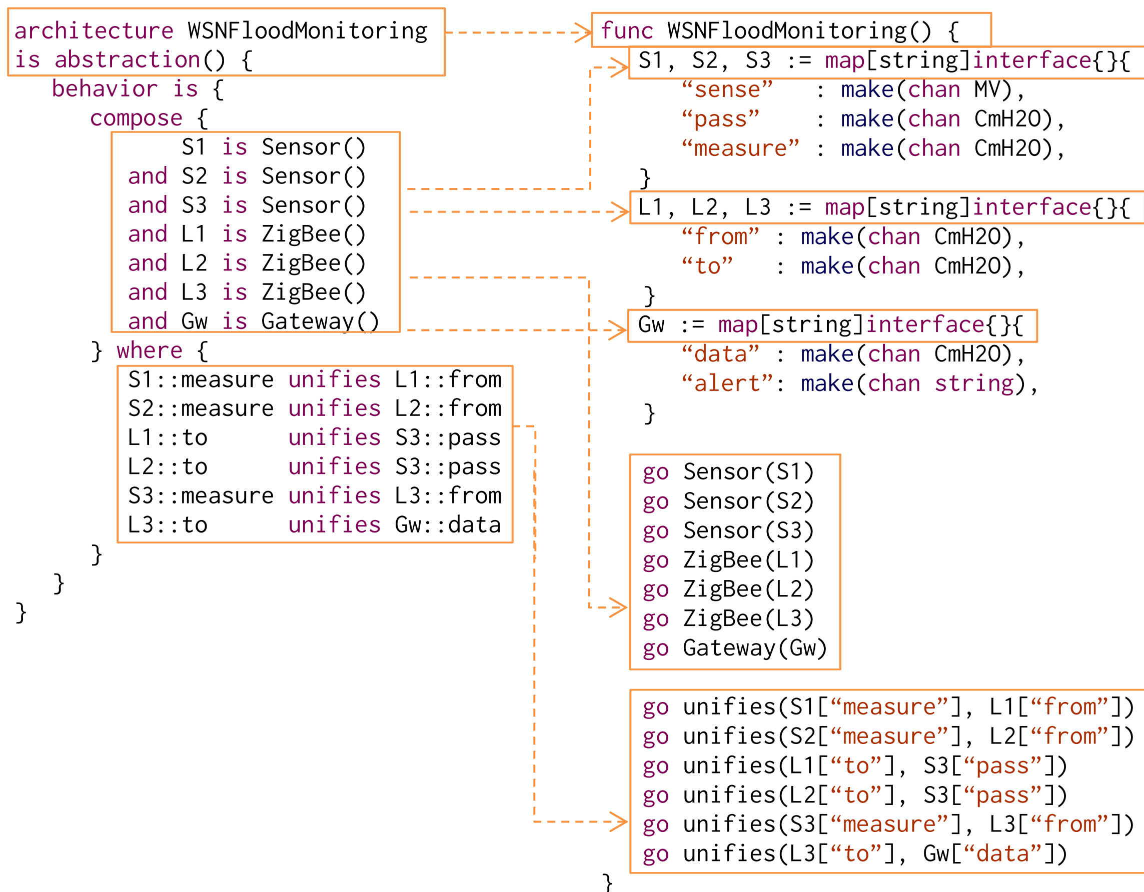
***ZigBee* connector.** Figure 23 shows an excerpt of the Go code generated from the π-ADL description of the *ZigBee* connector (see Figure 15). This connector is implemented in Go by the *ZigBee* function, which receives as parameter a map of channels (*conn*) representing its declared connections. In the *ZigBee* function, the value received via the *from* input channel is assigned to a variable (*m*) to be sent via the *to* output channel.

Figure 23 – Description of the *ZigBee* connector in π-ADL (left) and corresponding implementation in Go (right)



***WSNFloodMonitoring* architecture.** Figure 24 shows an excerpt of the Go code generated from the π-ADL description of the *WSNFloodMonitoring* architecture (see Figure 16), implemented by the *WSNFloodMonitoring* function. In this function, the *Sensor* and *Gateway* components and the *ZigBee* connector are instantiated by (i) creating the maps of channels corresponding to the instances of these architectural elements and their declared connections and (ii) calling the goroutines that represent such elements and their behavior. Next, these elements are attached by calling the *unifies* goroutine with the channel objects representing the connections to be unified. For example, the first call to the *unifies* goroutine unifies the *measure* output connection of sensor *S1* to the *from* input connection of the ZigBee connector *L1*

Figure 24 – Description of the *WSNFloodMonitoring* architecture in π-ADL (left) and corresponding implementation in Go (right)



5.3.2 Exogenous reconfiguration

Consider the situation described in Section 4.6.2 in which the flood monitoring system needs to be reconfigured due to the low battery level of a sensor. In this case, the reconfiguration encompasses replacing such a node by another one and connecting the latter by using a wireless connection. Realizing this reconfiguration by means of an exogenous approach comprises *WSNFloodMonitoringEvol*, an evolved architecture resulting from reconfiguration actions applied on the *WSNFloodMonitoring* initial architecture (see Figure 16 and Figure 17).

Figure 25 shows an excerpt of the Go source code generated from the π-ADL description of the evolved architecture, implemented by the *WSNFloodMonitoringEvol* goroutine that receives the references to the elements instantiated within the *WSNFloodMonitoring* initial architecture as parameters. These elements are returned by the *WSNFloodMonitoring* goroutine through a slice containing the maps of channels that represent the declared instances (*iarch*), which are copied to another slice named *abs* (lines 2 to 4). As the sensor *S3* and the ZigBee connector *Zb3* instances are unused in the evolved architecture, their respective channels are closed by calling the `close`

built-in function[14] (lines 5 to 16). Next, the new instances of the sensor component and the GPRS connector are created (lines 18 to 26) and provided as parameters to the goroutines that implement them (lines 29 to 35). At last, the unifications attaching the architectural elements take place as before by calling the *unifies* goroutine (lines 38 to 43).

Figure 25 – Excerpt of Go source code generated from the π-ADL description of the *WSNFloodMonitoringEvol* evolved architecture, following an exogenous approach

```
1:  func WSNFloodMonitoringEvol(iarch []map[string]interface{}) {
2:     var abs []map[string]interface{}      // decomposing initial architecture
3:     abs = iarch
4:
5:     select {                              // closing channels (connections)
6:         case <-abs[2][“sense”].(chan MV):
7:             close(abs[2][“sense”].(chan CmH2O))
8:         case <-abs[2][“pass”].(chan CmH2O):
9:             close(abs[2][“pass”].(chan CmH2O))
10:        case <-abs[2][“measure”].(chan CmH2O):
11:            close(abs[2][“measure”].(chan CmH2O))
12:        case <-abs[5][“from”].(chan CmH2O):
13:            close(abs[5][“from”].(chan CmH2O))
14:        case <-abs[5][“from”].(chan CmH2O):
15:            close(abs[5][“to”].(chan CmH2O))
16:    }
17:
18:    S4 := map[string]interface{}{       // instantiating new sensor component
19:        “sense”   : make(chan MV),
20:        “pass”    : make(chan CmH2O),
21:        “measure” : make(chan CmH2O),
22:    }
23:    Gprs1 := map[string]interface{}{    // instantiating new GPRS connector
24:        “from” : make(chan CmH2O),
25:        “to”   : make(chan CmH2O),
26:    }
27:
28:    // relaunching architectural elements (goroutine calls)
29:    go Sensor(abs[0])     // previous sensor component (S1)
30:    go Sensor(abs[1])     // previous sensor component (S2)
31:    go Sensor(S4)         // new sensor component
32:    go ZigBee(abs[3])     // previous ZigBee connector (Zb1)
33:    go ZigBee(abs[4])     // previous ZigBee connector (Zb2)
34:    go GPRS(Gprs1)        // new GPRS connector
35:    go Gateway(Gw)        // previous gateway component (Gw)
36:
37:    // reunifying connections
38:    go unifies(abs[0][“measure”], abs[3][“from”])  // from S1 to Zb1
39:    go unifies(abs[1][“measure”], abs[4][“from”])  // from S2 to Zb2
40:    go unifies(abs[3][“to”], S4[“pass”])           // from Zb1 to S4
41:    go unifies(abs[4][“to”], S4[“pass”])           // from Zb2 to S4
```

[14] For a channel c, the built-in function `close(c)` explicitly records that no more values will be sent on c.

```
42:     go unifies(S4["measure"], Gprs1["from"])        // from S4 to Gprs1
43:     go unifies(Gprs1["to"], abs[6]["data"])         // from Gprs1 to Gw
44:  }
```

5.3.3 Endogenous reconfiguration

Consider the situation described in Section 4.6.3 in which the flood monitoring system needs to be reconfigured in order to increase the accuracy of measures provided by motes aiming at avoiding false positives. In this case, the reconfiguration encompasses (i) allocating drones endowed with digital cameras and WiFi networking capabilities to capture images from the river and (ii) sending them to the gateway station for further processing. To realize this reconfiguration by means of an endogenous approach, the gateway component can create an instance of a drone component and link it by using a WiFi connector (see Figure 18).

Figure 26 shows an excerpt of the Go source code generated from the π-ADL description of the *Gateway* component, the one responsible for performing the reconfiguration actions. In case of high or very high risk given by the *triggerAlert* function (line 19), the *Gateway* function creates *dr* and *wf*, the maps of channels that respectively represent instances of the *UAV* component and the *GPRS* connector (lines 21 to 28). Next, these elements are run by respectively calling the *UAV* and *GPRS* goroutines, which receive the created maps as parameters (lines 30 and 31). At last, these new instances are attached to the existing elements by calling the *unifies* goroutine (lines 33 and 34).

Figure 26 – Excerpt of Go source code generated from the π-ADL description of the *Gateway* component, following an endogenous approach

```
1:   func Gateway(conn map[string]interface{}) {
2:      var calculateHI func(measure CmH2O) float64
3:      calculateHI = func(measure CmH2O) float64 {
4:          // implementation of the calculateHI function
5:      }
6:
7:      var processImage func(i Image) bool
8:      processImage = func(measure Image) bool {
9:          // implementation of the processImage function
10:     }
11:
12:     var triggerAlert func(measure CmH2O) string
13:     triggerAlert = func(measure CmH2O) string {
14:         // implementation of the triggerAlert function
15:     }
16:
17:     d := <- conn["data"].(chan CmH2O)
```

```
18:     var risk string
19:     risk = triggerAlert(d)
20:     if risk == “High” || risk == “Very high” {
21:         dr := map[string]interface{}{          // instantiating UAV component
22:             “camera” : make(chan Image),
23:             “output” : make(chan Image),
24:         }
25:         wf := map[string]interface{}{          // instantiating WiFi connector
26:             “input”  : make(chan Image),
27:             “output” : make(chan Image),
28:         }
29: 
30:         go UAV(dr)    // running UAV component
31:         go WiFi(wf)   // running WiFi connector
32: 
33:         go unifies(dr[“output”], wf[“input”])
34:         go unifies(wf[“output”], conn[“image”])
35:     }
36: 
37:     i := <- conn[“image”].(chan Image)
38:     if processImage(i) == true {
39:         conn[“alert”].(chan string) <- “Flood risk confirmed”
40:     } else {
41:         conn[“alert”].(chan string) <- risk
42:     }
43: }
```

5.4 Related work: Supporting the implementation of software architectures

In spite of the existence of ADLs for describing dynamic software architectures, these languages lack of a proper integration between the architectural and implementation levels. Furthermore, there is still no work on the integration of ADLs with new generation programming languages towards tackling the gap between these levels in the context of the construction of large-scale software systems with concurrent and distributed elements. This section briefly discusses some existing work on the integration of architecture descriptions with implementation and its limitations.

The Medvidovic et al.'s work [95] is one of the first works on the relationship between architecture descriptions and implementation languages. The proposed approach encompasses Dradel, an environment for modeling, analyzing, evolving, and implementing architectures described in the C2 Software Architecture Description and Evolution Language (C2SADEL), an extension of the C2 language designed to support architecture-based evolution. A Java class is generated for each component specified in C2SADEL and a method is generated for each component service, commented with preconditions and postconditions. In addition, developers need to

provide an implementation for these application-specific methods.

ArchJava [106] is an extension to Java that tangles software architecture specifications to implementation code in order to ensure traceability between architecture and code (i.e., the conformation of the implementation with the specified architecture) and to support the co-evolution of both architecture and implementation. ArchJava adds new language constructs to Java for specifying components and connections among them, while their behavior is implemented together the services that they provide. In terms of dynamicity support, components can be dynamically instantiated in a similar way to ordinary objects and connected at runtime. Despite the freshness of the approach as an architectural solution, ArchJava is limited in that it is more concrete than traditional ADLs due to its strong implementation basis. Architectures specified in ArchJava cannot be subjected to formal reasoning because it basically relies on an informal Java foundation, even though a formally well-founded type system can be found for ArchJava component extensions [107]. Furthermore, the generated Java implementations are to be executed on a single Java Virtual Machine, a condition that hinders the use of multicore and networked computer architectures for constructing large-scale, dynamic systems.

The MontiArcAutomaton language [108, 109] extends the MontiArc ADL [110] targeting cyber-physical systems modeling. MontiArcAutomaton encompasses a component/connector view to model the architectural structure and it uses automata to describe behavior of components, comprising the representation of possible states and transitions. To generate code from architecture descriptions in the Java and Python languages, MontiArcAutomaton makes use of complex templates that directly access the AST resulted from a compilation of the architecture description. Moreover, there is no support for dynamicity.

π-ADL.NET Project [111] was proposed some years ago as the result of the integration of π-ADL with the Microsoft® .NET Framework [112]. In π-ADL.NET, formal architecture descriptions in π-ADL are compiled to the low-level Common Intermediate Language (CIL) [113], resulting in a code able to access the existing resources provided by the .NET platform. π-ADL.NET also supports runtime analysis of the concrete software architecture by executing the architecture description, thus seeking to preserve architectural integrity of the system at the implementation level. Despite its intention of bringing a formally founded ADL to an implementation platform, the main limitation of π-ADL.NET that makes it not suited for contemporary software systems regards the lack of counterparts when performing mappings from π-ADL to CIL or to the .NET platform. As an example, behaviors and abstractions in π-ADL communicate through connections, but these elements have no corresponding entities in CIL. For this reason, a .NET class was developed by hand to emulate π-ADL connections, with requisite threading and synchronization functionality. In turn, π-

ADL connections are straightforwardly mapped to channels in Go, which are first-class elements of the language and can be easily managed mainly when synchronizing processes. Furthermore, π-ADL.NET also lacks of support for distribution, a feature that is easily and natively supported by Go and typically required for contemporary software.

In summary, it is possible to observe that the ensemble proposed in this work as the result of the integration of the π-ADL and Go stands out due to a series of important aspects. π-ADL and Go are not only formally founded languages, but they also have the same underlying formalism, the π-calculus process algebra. Such a common foundation allows for a straightforward mapping between elements of the languages, thereby fostering the understanding of these relationships and easing their implementation at the technological side. Finally, both π-ADL and Go languages are suitable candidates for coping with requirements posed by contemporary software systems at both architectural and implementation levels. On the one hand, π-ADL is a formal, theoretically well-founded language for comprehensively describing dynamic software architectures by encompassing both structural and behavioral viewpoints. On the other hand, Go is an easy general-purpose language designed to address the construction of scalable distributed systems and handle multicore and networked computer architectures. Therefore, integrating π-ADL and Go can contribute to minimize architectural drifts while bringing benefits to important activities such as automatic generation of implementation code and automated analysis of software architectures.

6 Verifying dynamic software architectures

As previously discussed, the critical nature of many complex software systems calls for formal, rigorous architecture descriptions as means of supporting automated verification and enforcement of architectural properties and constraints. Model checking has been one of most used techniques to automatically analyzing software architectures with respect to the satisfactions of architectural properties [29, 30], even though it leads to an explosion of the state space due to the exhaustive exploration of all possible states of the system under verification. Such a state explosion problem becomes more severe for verifying the contemporary software systems due to their typical non-deterministic runtime behavior and the unpredictable conditions in which they operate.

Formal verification techniques such as model checking require not only significant execution time and computational resources, but also an unneglectable effort from architects. This is one of the major reasons that often hinders the adoption of formal-based techniques in software industry, as revealed in a recent survey in this context [18]. Therefore, providing affordable, computationally efficient approaches for rigorously verifying properties in dynamic software architectures is a major concern.

In order to cope with the aforementioned issues, this chapter presents the work[15] on the use of *statistical model checking* (SMC) to support the formal analysis of dynamic software architectures described in the π-ADL language while striving to reduce effort, computational resources, and time to perform this task [38]. Such an approach requires a stochastic execution model of the system, in which the choice of the next action to execute is done according to a probabilistic distribution. With a stochastic system, a property to be verified might be satisfied by some executions and not be satisfied by some others. In the proposed approach, a number of stochastic simulations of that system is executed and the approximated probability of the system to meet the property under verification is evaluated. It is worth mentioning that requiring the system execution to be probabilistic is not a limitation because dynamic software systems can be reasonably described by assigning probabilities for new components to appear or for the existing components to fail and be disconnected, for example. Moreover, probability distributions can be used to model input values.

Besides an executable probabilistic model of the system, the proposed SMC-based approach requires a language for expressing properties to be verified and a

[15] This work was conducted in collaboration with researchers from IRISA-UBS (Vannes, France) and INRIA Rennes Bretagne Atlantique (Rennes, France).

monitor for deciding them on finite traces. The particular nature of dynamic software systems is hat architectural elements (components or connectors) may appear, disappear, be connected or be disconnected at runtime. Therefore, expressing behavioral and structural properties regarding a dynamic software architecture needs to take into account architectural elements that are dynamically created and removed, i.e., they may exist at a given instant in time and no longer exist at another. To cope with these characteristics, this work also introduces DynBLTL, a novel logic and notation aimed to express properties in dynamic software architectures [41]. DynBLTL was designed to handle the absence of an architectural element in a given formula expressing a property by means of the *undefined* value ($\mathfrak{U}$), which is returned when reading values from components that are no longer in the system. DynBLTL and the SMC-based approach itself were implemented as a plug-in for PLASMA [114, 115], a flexible, modular statistical model checker.

The remainder of this chapter is structured as follows. Section 6.1 specifies how to formalize execution traces of dynamic software architectures. Section 6.2 presents how to formally express properties to be verified. Section 6.3 describes the toolchain developed to support the verification of dynamic software architectures by using the SMC approach. Finally, Section 6.4 briefly discusses related works.

6.1 Representing traces of dynamic software architectures

As discussed in Section 2.1, typical operations performed on dynamic software architectures comprise creating, removing, attaching, and detaching components and connectors. Exchanging values among such architectural elements can be considered as one of the main indicators of behavior in a software architecture at runtime, so that a value exchange can lead the system from a given state to another.

In the SMC-based approach proposed in this work, a *state* of a dynamic software architecture is represented by a directed graph $g = (V, E)$ comprising a set of finite nodes V and a finite set of edges E. Each node $v \in V$ represents an architectural element (component or connector) of the system. In turn, each edge $e \in E$ represents a communication channel between two architectural elements and is labeled y the values exchanged between the nodes. The set of all possible values is represented by Val, which contains the *undefined* value $\mathfrak{U}$ to represent the absence of a value. More specifically:

- Each node $v \in V$ is defined by a tuple (id, T, C) in which id is a globally unique identifier for the architectural element, T is the declared type of the architectural element, and C is a finite set representing its connections. $id(v)$ returns the identifier for node v, $C(v)$ denotes the set of connections of the node v, and $v.c$ denotes a connection $c \in C(v)$.
- Each edge $e \in E$ connecting two nodes in the graph is labeled by the

values exchanged between them. These values are contained into *Val*, the set of all possible values that can be exchanged between two nodes. Formally, $E \subset \{(v_1.c_1, x, v_2.c_2) \mid x \in Val \wedge \bigwedge_{i=1}^{2} v_i \in V \wedge c_i \in C(v_i)\}$.

Given a state graph g, $V(g)$ and $E(g)$ respectively denote its sets of nodes and of edges.

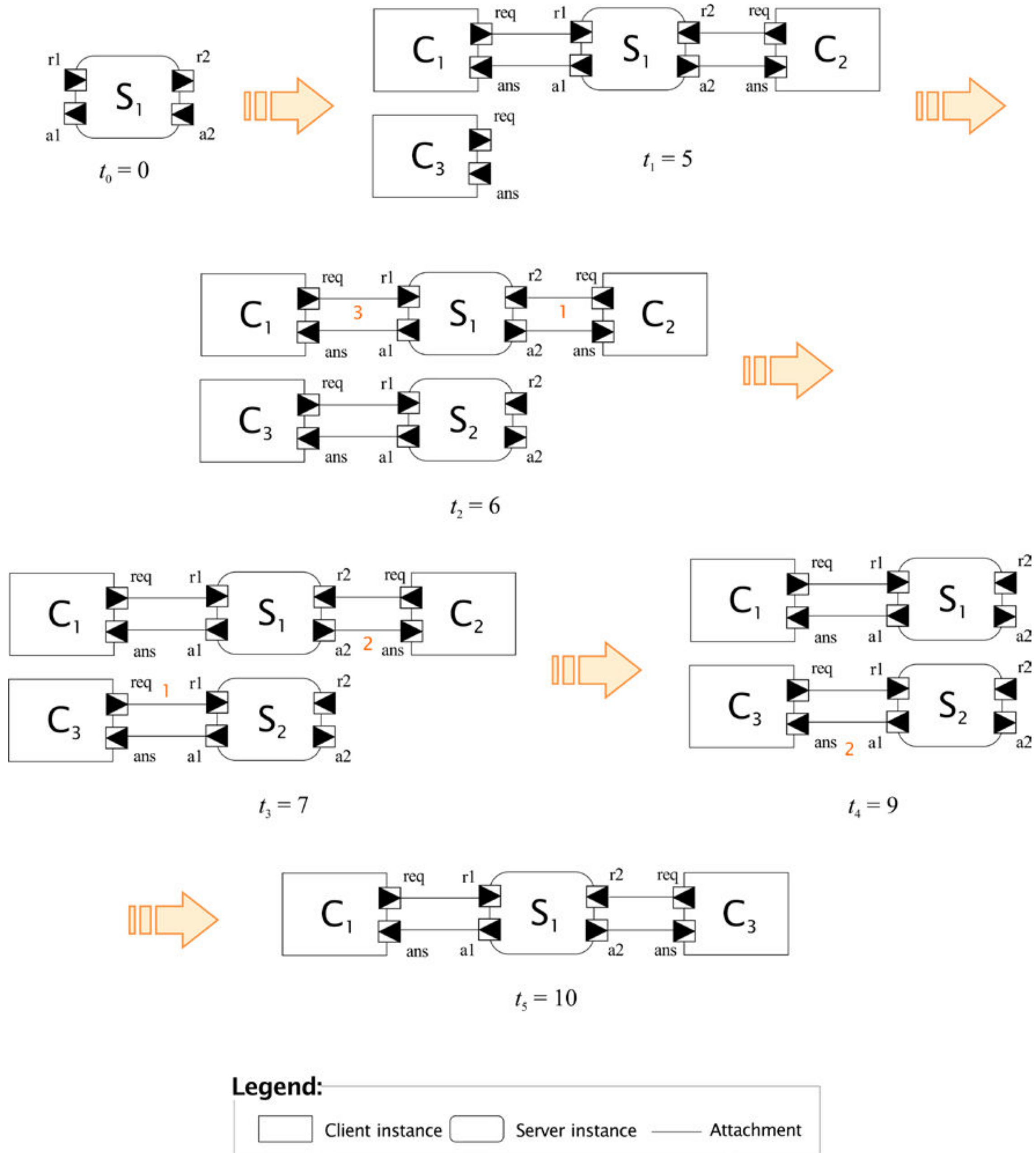
The SMC technique relies on checking multiple *execution traces* resulted from simulations of the system under verification against the specified properties. Therefore, as a simulation ω results in a trace σ composed of a finite sequence of states, ω can be defined as a sequence of state graphs g_i ($i \in \mathbb{N}$). In the proposed approach, it important to distinguish between *untimed traces* and *timed traces*. An untimed trace σ^{ut} is a simple sequence $g_0, g_1, \ldots, g_n$ of states. In turn, a timed trace σ is a sequence $((t_0, g_0), (t_1, g_1), \ldots, (t_n, g_n))$ of states with timestamps t, such that $\forall i\ t_i \in \mathbb{R} \wedge ti \leq t_{i+1}$.

SMC allows verifying systems that are *stochastic processes*, in the mathematical sense. Therefore, we require that traces are produced by a stochastic process, i.e., each state in the trace is the restriction of a complete system state and the choice of next one is governed by a probability distribution at each state. For verifying timed systems, the system will eventually produce a state (t_i, g_i) with $t_i > M$ for any value $M \in \mathbb{R}$. In other words, the time converges towards $+\infty$ during the execution of the system.

As an example, consider a simple client-server architecture that dynamically adapts to the demand. In such a system, clients may appear and interact with a server by sending requests and receiving answers. It is assumed that each server can handle a limited number of clients (two in the example). If all servers have reached that limit and a new client appears, the systems spawns a new server to handle the new client. Whenever the client has completed its interaction with the server, it disconnects and disappears from the system. If a server has no client left, it is shut down and disappears from the system. At last, if the overall utilization of the servers is low, one tries to shut down some servers in order to save energy. This is done by reallocating clients so that some severs become unused.

Figure 27 presents an illustration of an execution trace for a simple client-server system, made up client and server instances. Initially, only one server is present in the system, and a server has four connections (r_1, r_2, a_1 and a_2). At $t = 5$, three new clients appear and two of them are directly connected to the server. At $t = 6$, a new server spawns and connects to the third client, while the two first clients send their requests (requests and answers are represented as numbers). At $t = 7$, the client C_2 receives the answer to its request while the client C_3 sends a request to server S_2. At $t = 9$, the client C_3 receives the answer to its request and the client C_2 has disappeared. At $t = 10$, the client C_3 is relocated to server S_1 and the server S_2 is removed.

Figure 27 – Illustration of an execution trace for a simple client-server system



Aiming at obtaining an execution trace from an architecture description in π-ADL, the simulation emits explicit messages recording a set of actions on the state graph, namely: (i) *add*, which refers to the creation of an instance of an architectural element; (ii) *link*, which represents an unification of an output connection of a component/connector with an input connection of another component/connector; (iii) *rdv*, i.e., the sequential execution of a output prefixing action followed by an input prefixing action; and (iv) *decompose*, which stands for the dismantling of the architectural configuration. These actions and the respective conditions for their validity are summarized in Table 10. Enacting an action leads to a transition from a

given state graph $g = (V, E)$ to a state graph $g' = (V', E')$.

Table 10 – Summary of actions on a state graph $g = (V, E)$

Action	Correspondence to architectural level	Description	Condition
`add(`v`)`	Create an instance v of an architectural element	Add a node v to the state graph	
`link(`$v_1.c_1$, $v_2.c_2$`)`	Unify output connection $v_1::c_1$ to input connection $v_2::c_1$	Link connection c_1 of node v_1 to connection c_2 of node v_2	$v_i \in V \wedge c_i \in v_i$ $(i = 1, 2)$
`rdv(`$v_1.c_1$, x, $v_2.c_2$`)`	Send value x via connection $v_1::c_1$ and receive x via connection $v_2::c_2$	Exchange value x between connection c_1 of node v_1 and connection c_2 of node v_2	$v_i \in V \wedge c_i \in v_i \wedge \exists x \in Val.$ $(v_1.c_1, x, v_2.c_2) \in E$ $(i = 1, 2)$
`decompose()`	Dismantle an architecture into composing architectural elements	Decompose the architecture	—

6.2 Expressing properties about dynamic software architectures

Mateescu and Oquendo [27] highlight that architecture descriptions using ADLs should cover not only structure and behavior of a software architecture, but also the required and desired architectural properties, in particular the ones related to consistency and correctness. The notation for expressing such properties must hence complement (or be part of) an ADL to specify and support verification of architecture-related semantic properties. These properties can be (i) *structural*, e.g., cardinality of architectural elements, interconnection topology, etc., or (ii) *behavioral*, e.g., safety, liveness or fairness defined on actions of the system. As reported by Zhang et al. [29], most existing notations allow specifying only properties about components and their interfaces, types, and instances while few ones are expressive enough to specify properties regarding all architectural elements and configurations [21, 23, 27].

6.2.1 Underlying formalisms for expressing properties

Most architectural properties to be verified by using model checking techniques are *temporal* [29], i.e., they are qualified and can be reasoned upon a sequence of system states along the time. Temporal properties typically verified in the context of software architectures are *safety* and *liveness*: safety properties usually state that *something (bad) never happens* while liveness properties state that *something (good) will eventually happen* or *keeps happening*.

Zhang et al. [29] report that linear temporal logic (LTL) [116] has been often

used in the literature as underlying formalism for specifying temporal architectural properties and verifying them through model checking. LTL extends classical Boolean logic with *temporal operators* (a.k.a. *modalities*) that allow reasoning on the temporal dimension of the execution of the system. In this perspective, LTL can be used to express properties about the future of the execution (sequence of states), e.g., a condition that will eventually true, a condition that will be true until another fact becomes true, etc. LTL has been well studied along the years and it is known to be useful for verifying and specifying concurrent systems [117].

As SMC relies on simulation, it verifies *bounded properties*, i.e., properties that can be defined in terms of finite executions of the system under verification. While LTL-based formulas aim at specifying the infinite behavior of the system, a time-bounded form of LTL called BLTL considers finite sequences of execution states of the system during a *relative time interval* [0, *t*]. The bounds are specified on temporal operators, e.g., the *always* operator. In LTL, this operator states that a property must be verified at each step of a (potentially infinite) trace, while in BLTL it has a bound and a state that the property must hold until the bound is reached.

The basic syntax of BLTL is defined as follows:

$$\varphi = \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathcal{F}^t\varphi \mid \mathcal{G}^t\varphi \mid \psi\mathcal{U}^t\varphi \mid \mathcal{X}\varphi \mid \kappa$$

where $\vee$, $\wedge$ and $\neg$ are standard propositional logic operators and κ is a Boolean constant or an atomic proposition constructed from numerical constants, state variables, and relational operators. Four temporal operators are also defined:

(i) $\mathcal{X}$ is the *next* operator. $\mathcal{X}\varphi$ means that the formula φ will be true in the next step.

(ii) $\mathcal{F}$ is the *finally* or *eventually* operator, which is bounded by a relative time interval [0, *t*]. $\mathcal{F}^t\varphi$ means that the formula φ will be true at least once in the time interval [0, *t*].

(iii) $\mathcal{G}$ is the *globally* or *always* operator, which is bounded by a relative time interval [0, *t*]. $\mathcal{G}^t\varphi$ means that the formula φ will be true at all times in the time interval [0, *t*].

(iv) $\mathcal{U}$ is the *until* operator, which is bounded by a relative time interval [0, *t*]. In the time interval [0, *t*], $\psi\mathcal{U}^t\varphi$ means that either the formula φ is initially true or ψ will be true until φ is true at the current or a future time.

Combining these temporal operators creates complex properties with interleaved notions of *eventually* ($\mathcal{F}$), *always* ($\mathcal{G}$), and *one thing after another* ($\mathcal{U}$).

6.2.2 A novel logic and notation for expressing properties in dynamic software architectures

A key characteristic of dynamic software systems is the impossibility of foreseeing the exact set of architectural elements deployed at a given point of execution. Furthermore, it is of particular interest verifying that new components respect a particular behavior. Both LTL and BLTL are unable to handle this characteristic as they would require knowing the set of components that will appear prior to the execution and writing a dedicated formula for each of them. To tackle such a limitation, this work comes up with DynBLTL as a novel logic and notation for expressing linear temporal properties over traces of dynamic software architectures. DynBLTL can express the required behavior of new components by having quantifiers over the set of existing components. Aiming at specifying a behavior for the quantified components, DynBLTL allows interleaving quantifiers and temporal operators. In DynBLTL, all temporal operators are *bounded*, thereby making properties decidable on finite traces.

DynBLTL is designed to handle the absence of an architectural element in a given formula expressing a property. In practice, this means that a Boolean expression can take three values, namely *true*, *false* or *undefined* ($\mathfrak{U}$). The undefined additional value refers to the fact that an expression may not be evaluated at a given execution state depending on the current runtime configuration of the system. This is necessary for situations in which it is not possible to evaluate an expression at the considered state, e.g., a statement about an architectural element that does not exist at that moment. As an example, the expression `c1.req > 3.2` cannot be evaluated if the component `c1` does not exist (as at t_0 in Figure 27) or the connection `c1.req` is not involved in a communication at that state (as at t_1 in Figure 27).

DynBLTL is not typed, so that a property can be evaluated to any type, i.e., Boolean, integer, string or undefined. As SMC requires a Boolean value as the result of the evaluation of a property on a trace, a syntactical constraint on properties is added to enforce that the returned value is Boolean. For example, the `until` or `isTrue` operators always return a Boolean value. Consequently, the root operator of a property must be either `until`, `isTrue` or a Boolean combination of them.

The semantics of a property φ is a function $[[\varphi]]$ that takes a trace σ as argument and returns a value in *Val*. The semantics for a timed trace is defined as $\sigma = ((t_0, g_0), (t_1, g_1), \ldots, (t_n, g_n))$. If the system is untimed, only temporal operators whose bound is expressed in steps can be evaluated. Assume that φ is a property in which all temporal operators' bounds are expressed in steps. Evaluating an untimed trace $\sigma^{ut} = (g_0, g_1, \ldots, g_n)$ falls back to evaluating a timed trace with the same states and arbitrary timestamps. Indeed, timestamps are only relevant for temporal operators whose bound is

expressed in time units.

A property can be specified by a formula containing *literals*, *identifiers* referring to nodes and connections in the state graph, *operations* and *comparisons*, predefined *functions*, *quantified expressions*, and *temporal operators*. These elements are briefly described in Section 6.2.2.1 along with some examples in Section 6.2.2.2 using the WSN-based system depicted in Section 3.3. Appendix C shows the concrete syntax of DynBLTL in the Extended Backus-Naur Form (EBNF).

6.2.2.1 DynBLTL elements

Literals and identifiers. As basic elements, a formula expressing a property can contain (i) a literal, which can be a Boolean value, numerical value or a string, (ii) an identifier representing a node of the state graph, or (iii) a connection of a node of the state graph. The evaluation of these literals are as follows:

- if φ is a literal l, then $[[\varphi]](\sigma) = [[l]](\sigma) = l$, i.e., the formula is evaluated to the respective value of l;
- if φ is an identifier idt representing a node, then $[[\varphi]](\sigma) = [[idt]]((t_0, g_0), (t_1, g_1), \ldots, (t_n, g_n))$ = `true` if there exists a node with that name at the current state, i.e., if $\exists v \in V(g_i) \, . \, id(v) = idt$; otherwise, the formula is evaluated to $\mathfrak{U}$;
- if φ refers to a connection c of a node v of a state graph ($v.c$), then $[[\varphi]](\sigma) = [[v.c]]((t_0, g_0), (t_1, g_1), \ldots, (t_n, g_n))$ is evaluated to the only non-undefined value labeling any edge of the graph state g_i attached to the connection $v.c$; otherwise, the formula is evaluated to $\mathfrak{U}$.

Operations and comparisons. Arithmetic operations as well as inequalities and equalities are evaluated as usual or set as $\mathfrak{U}$ if at least one argument is out of their definition domain. DynBLTL supports the usual arithmetic operators (`+`, `−`, `*`, `/`) and comparisons (<, <=, >, >=, =, `!=`). Note that both $\mathfrak{U}$ `!=` $\mathfrak{U}$ and $\mathfrak{U} = \mathfrak{U}$ are evaluated to $\mathfrak{U}$.

Usual Boolean operators are also supported. The `not` operator works as usual on Boolean values and returns $\mathfrak{U}$ with other values. The `or` operator returns `true` if at least one of the operands is evaluated to `true`, `false` if both operands are evaluated to false, and $\mathfrak{U}$ otherwise. Note that it may return `true` even if one of the operands is $\mathfrak{U}$. Other usual Boolean operators are obtained as follows:

$$\varphi_1 \text{ and } \varphi_2 \stackrel{\text{def}}{=} \texttt{not (not } \varphi_1 \texttt{ or not } \varphi_2)$$

$$\varphi_1 \texttt{ implies } \varphi_2 \stackrel{\text{def}}{=} \texttt{not } \varphi_1 \texttt{ or } \varphi_2$$

Functions. DynBLTL provides four predefined functions that can be used to explore the architectural configuration, i.e., the nodes of a state graph:

- `allOfType(`T`)` returns a collection with all nodes of type T;

- `areConnected(`v_1`,` v_2`)` returns `true` if nodes v_1 and v_2 are connected by an edge in the state graph `false` if v_1 and v_2 exist in the state graph, but they are not connected by an edge, or $\mathfrak{U}$ otherwise;
- `areLinked(`$v_1.c_1$`,` $v_2.c_2$`)` returns `true` if the connection c_1 of node v_1 and the connection c_2 of node v_2 are connected by an edge in the state graph, `false` if both $v_1.c_1$ and $v_2.c_2$ exist in the state graph, but they are not connected by an edge, or $\mathfrak{U}$ otherwise; and
- `lastValue(`$v.c$`)` returns the last non-undefined value of the connection c of node v or $\mathfrak{U}$ if its value was always undefined.

Quantified expressions. In DynBLTL, three types of quantified expressions can be used to specify formulas expressing properties, namely the existential and universal quantified expressions traditionally used in predicate logic, as well as an additional quantifier for counting elements upon the satisfaction of a predicate. These quantified expressions comprise an identifier r, a function f that returns a collection of elements, and a formula φ with free occurrences of r. In the following, $[[f]](\sigma) = e = \{e_1, \ldots, e_n\}$ and $\varphi[r \leftarrow e_i]$ refers to the formula φ where each free occurrence of r is replaced by the element e_i. Quantifiers are defined as follows:

- `exists` r `:` $f\,\varphi$ returns `true` if $\varphi[r \leftarrow e_i]$ is evaluated to `true` for at least one element e_i ($1 \leq i \leq n$) or to `false` if $\varphi[r \leftarrow e_i]$ is evaluated to false for all elements e_i, or to $\mathfrak{U}$ otherwise;
- `forall` r `:` $f\,\varphi$ returns `true` if $\varphi[r \leftarrow e_i]$ is evaluated to `true` for all elements e_i ($1 \leq i \leq n$) or to `false` if $\varphi[r \leftarrow e_i]$ is evaluated to false for at least one element e_i, or to $\mathfrak{U}$ otherwise;
- `count` r `:` $f\,\varphi$ returns how many elements $e_i \in e$ evaluate $\varphi[r \leftarrow e_i]$ to `true`.

Temporal operators. Similarly to traditional BLTL (see Section 6.2.1), DynBLTL provides four temporal operators, namely `in`, `until`, `eventually before`, and `always during`. These operators are bounded either by means of *steps* or *time units*. They are defined as follows:

- The `in` operator evaluates its argument at a later point specified by the bound. If the bound is expressed in b steps ($b \in \mathbb{N}$), the trace is translated by that number of steps:

$$[[\texttt{in}\ b\ \texttt{steps}\ \varphi]]((t_0, g_0), \ldots, (t_n, g_n)) = [[\varphi]]((t_b, g_b), \ldots, (t_n, g_n))$$

  If the bound is expressed in terms of b time units ($b \in \mathbb{N}$), the trace is translated by the amount of time units provided as argument:

$$[[\texttt{in}\ b\ \texttt{time units}\ \varphi]]((t_0, g_0), \ldots, (t_n, g_n)) = [[\varphi]]((t_k, g_k), \ldots, (t_n, g_n))$$

  where $k = \min(\{0 \leq i \leq n \mid t_i - t_0 > b\})$.

- The `until` operator returns a Boolean value. An `until` expression is

evaluated to `true` if its right argument is evaluated to `true` within the bound and if the left argument is evaluated to `true` or to $\mathfrak{U}$ until the right argument becomes `true`. Consider the following standard notations:

$$\sigma \vDash \varphi \equiv [[\sigma]](\sigma) = \texttt{true}$$
$$\sigma \nvDash \varphi \equiv [[\sigma]](\sigma) = \texttt{false}$$[16]

If the bound is expressed in steps:

$$((t_0, g_0), \ldots, (t_n, g_n)) \vDash \varphi_1 \texttt{ until } b \texttt{ steps } \varphi_2 \text{ iff}$$
$$\exists 0 \leq i \leq b \;.\; ((t_i, g_i), \ldots, (t_n, g_n)) \vDash \varphi_2 \wedge \forall 0 \leq j \leq i \;.\; \neg((t_j, g_j), \ldots, (t_n, g_n)) \nvDash \varphi_1$$

If the bound is express in time units:

$$((t_0, g_0), \ldots, (t_n, g_n)) \vDash \varphi_1 \texttt{ until } b \texttt{ time units } \varphi_2 \text{ iff}$$
$$\exists 0 \leq i \leq n \;.\; (t_i - t_0 \leq b) \wedge ((t_i, g_i), \ldots, (t_n, g_n)) \vDash \varphi_2 \wedge$$
$$\forall 0 \leq j \leq i \;.\; \neg((t_j, g_j), \ldots, (t_n, g_n)) \nvDash \varphi_1$$

- The `eventually before` operator can be defined by reusing the previous definition of the `until` operator as:

$$\texttt{eventually before } b\ \varphi \stackrel{\text{def}}{=} \texttt{true until } b\ \varphi$$

- The `always during` operator can be defined by reusing the previous definition of the `eventually before` operator as:

$$\texttt{always during } b\ \varphi \stackrel{\text{def}}{=} \texttt{not eventually before } b\ \varphi$$

Note that the value $\mathfrak{U}$ is treated in a particular way when defining the `until` operator. Indeed, when $\mathfrak{U}$ appears on the left side of `until`, it is treated as `true`. However, when it appears on the right side, it is treated as `false`. This choice was made for the sake of intuitiveness. For example, the property

```
c1.req < 2 until 10 steps c2.req = 5
```

can return `true`, even if `c1.req < 2` evaluates to $\mathfrak{U}$ during the ten steps. Therefore, evaluating to $\mathfrak{U}$ on the left side of until does not invalidate the formula. However, if `c1.req < 2` evaluates to `false` before `c2.req = 5`, then the whole expression is evaluated to `false`.

The `isTrue` operator enforces the evaluation of a property to a Boolean value. Formally:

$$[[\texttt{isTrue } \varphi]](\sigma) = \sigma \vDash \varphi$$

This operator can be used to modify the behavior of the until operator. For example:

---

[16] Note that if φ is not evaluated to a Boolean, then neither $\sigma \vDash \varphi$ nor $\sigma \nvDash \varphi$ holds.

```
(isTrue c2.req < 2) until 10 steps c2.req = 5
```

will evaluate to false if `c2.req < 2` evaluates to $\mathfrak{U}$ before `c2.req` evaluates to 5. Its dual operator is defined as:

$$\texttt{isNotFalse}\ \varphi \overset{\text{def}}{=} \texttt{not isTrue not}\ \varphi$$

6.2.2.2 Example

Consider the WSN-based system architecture depicted in Section 3.3. It is possible to express some interesting properties about such an architecture. For instance, one can ensure that all sensors must send data in less than three time units. Assuming that a sensor successfully holds measured data if they are positive values, such a statement can be expressed by the `SendData(X)` property, defined as:

```
always during X time units {
   forall s:allOfType(Sensor) {
      isTrue (s.sense > 0 or s.pass > 0) implies
      eventually before 3 time units s.measure > 0
   }
}
```

In this property, receiving data either from the *sense* input connection (data gathered by the sensor itself) or via the *pass* input connection (data received from a neighbor sensor) implies that data need to be sent via the *measure* output connection in less than three time units. As previously mentioned, the bound of X time units on the `always during` operator is need to ensure that the property can be decided on a finite trace. Therefore, the property checks only the first X time units of the execution trace.

An important property to be verified is the correctness of the architectural model with respect to its main goal, i.e., warning upon the risk of flood (in the flood monitoring system) or detection of leakage (in the pipeline monitoring system). In this context, a false negative occurs when the system fails to make the expected prediction. This can be expressed by the `FalseNeg(X,Y,Z)` property, defined as:

```
eventually before X time units {
   (Gw.alert = “low risk”) and (eventually before Y time units Gw.data > Z)
}
```

This property characterizes a false negative: the gateway component (*Gw*) predicts that there is a low risk for an anomalous event (imminent flood or leakage), but data received from sensors via the input connection *data* characterizes that such an event is actually happening. The parameters of this formula are X, the time during which the system is monitored, Y, the time during which the prediction of the gateway should hold, and Z, the maximum threshold considered for normal conditions.

Similarly, a false positive occurs when the system predicts an anomalous event that does not actually occur. This can be expressed by the `FalsePos(X,Y,Z)` property, defined as:

```
eventually before X time units {
   (Gw.alert = “anomaly detected”) and
   (always during Y time units not Gw.data > Z)
}
```

The system is said to be correct if there are no false negatives nor false positives for the expected prediction anticipation (parameter Y).

Note that these three formulas are actually BLTL formulas as they involve simple predicates on the state. However, DynBLTL allows expressing properties about the dynamic architecture of the system. For example, suppose that one wants to check if a sensor is available, i.e., at least one sensor is connected to the gateway. More precisely, it is required that there is a wireless link connecting a sensor to a gateway, otherwise such a sensor needs to appear in less than Y time units. This can be expressed by the `SensorAvailable(X,Y)` property, defined as:

```
always during X time units {
   (not (exists l:allOfType(Link) areLinked(l.output, gw.data)
    and (exists s:allOfType(Sensor) areLinked(s.measure, l.from))))
   implies eventually before Y time units {
      exists l:allOfType(Link) areLinked(l.to, gw.data)
      and (exists s:allOfType(Sensor) areLinked(s.measure, l.from))
   }
}
```

The parameters of this formula are X, the time during which the system is monitored, and Y, the maximum time at which the sensor must appear.

Finally, suppose that one wants to check if a sensor is failing (i.e., its gathered measures are negative values), then it should be removed from the system in a reasonable amount of time. This disconnection is needed because a faulty sensor will not pass incoming measures neither will gather correct values. The removal of a given sensor component is characterized by the fact that it is not attached to a link connector. As sensors may appear or disappear during execution, the temporal pattern needs to be dynamically instantiated at each step for each existing sensor. This can be expressed by the `RemoveSensor(X,Y)` property, defined as:

```
always during X time units {
   forall s:allOfType(Sensor) {
      (isTrue s.measure < 0) implies
      eventually before Y time units {
         not exists l:allOfType(Link) areLinked(s.measure, l.from)
      }
   }
}
```

The parameters of this formula are X, the time during which the system is monitored,

and Y, the maximum time at which the faulty sensor must appear as disconnected within the system. The `RemoveSensor(X, Y)` property cannot be stated in BLTL since it does not have a construct (such as the `forall` universal quantifier) for instantiating a variable number of temporal sub-formulas, where the number depends on the current state.

6.3 Statistical model checking of π-ADL architectural models

This section describes how to perform statistical model checking of π-ADL architecture descriptions. As SMC is a stochastic technique, the executable model representing the system needs to be stochastic, a feature that the π-ADL language does not possess. For this reason, it was necessary to provide a way of producing a stochastic executable model from π-ADL architecture descriptions, thus allowing for property verification using SMC.

Section 6.3.1 describes how to make π-ADL architectural models stochastic whereas Section 6.3.2 presents the developed SMC-based toolchain to verify properties expressed in DynBLTL regarding dynamic software architectures described in π-ADL. Finally, Section 6.3.3 reports the results of some experiments on the computational effort for verifying properties regarding the WSN-based monitoring system described in Chapter 3.

6.3.1 Stochastic execution of π-ADL architecture descriptions

In π-ADL, non-determinism occurs in two different ways. First, whenever several actions are possible, any one of them can be executed as the next action, i.e., the choice of the next action to execute is non-deterministic. Second, some functions can be declared as *unobservable* (see Section 4.3), thus meaning that its internal operations are concealed at the architectural level. In this case, the value returned by the function is also non-deterministic because it is not defined in the model. As a stochastic process is required for performing SMC[17] [118], the non-determinism of π-ADL models are resolved by using probabilities. The following describes how to proceed in the aforementioned cases.

Resolving non-determinism in the choice of the next action. The Go code from a π-ADL architecture description encodes architectural element (component or connector) as a concurrent goroutine. The communication between architectural elements takes place via a channel. If several communications are possible, the Go runtime chooses one of them to execute according to a FIFO (first-in, first out) policy.

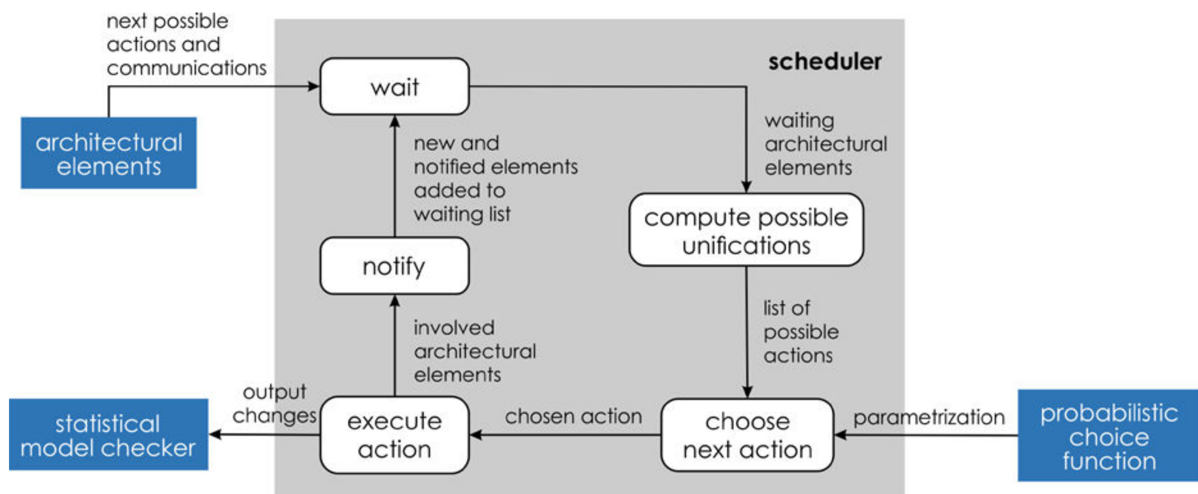
[17] It is worth mentioning that SMC also applies to non-stochastic systems [119].

Such a policy is not suitable for SMC since it is necessary to specify how the next action is chosen.

To support the stochastic scheduling of actions, a scheduler was implemented as a goroutine that controls all non-local actions, i.e., composition, decomposition or communication. Whenever an architectural element needs to perform a non-local action, it informs the scheduler and blocks until the scheduler responds. The scheduler responds with the action executed (if the component submitted a choice between several actions) and a return value, corresponding either to the reception side of a communication or a decomposed architecture.

Figure 28 depicts the behavior of the scheduler. The scheduler waits until all architectural components (components and connectors) have indicated their possible actions. At this step, the scheduler builds a list of possible rendezvous by checking which declared unifications have both sender and receiver ready to communicate. For this purpose, the scheduler maintains a list of the active architectures and the corresponding unifications. The possible communications are added to the list of possible actions and the scheduler chooses one of them according to a probabilistic choice function. The scheduler then executes the action and outputs its effect to the statistical model checker. At last, the scheduler notifies the components and connectors involved in the action.

Figure 28 – Working schema of the scheduler to support the stochastic execution of a π-ADL architectural model



Resolving non-determinism in unobservable functions. Functions declared as unobservable require an implementation to allow simulating the model. In practice, this implementation is provided in form of a Go function whose return value can be determined by a probability distribution. Such an implementation relies on the Go libraries that implement usual probability distributions. In particular, such functions

can model inputs of the systems that have a known probabilistic value, i.e., input to a component, time to the next failure of a component, etc.

6.3.2 An SMC-based toolchain to simulate and verify dynamic software architectures

SMC techniques rely on the simulation of an executable model of the system under verification against a set of formulas expressing bounded properties to be verified (see Section 2.3). These elements are provided as inputs to a statistical model checker, which basically consists of (i) a simulator for running the executable model of the system under verification, (ii) a model checker for verifying properties, and (iii) a statistical analyzer responsible for calculating probabilities and performing statistical tests.

Among the SMC tools available in the literature, PLASMA [114, 115] is a compact, flexible platform that enables users to create custom SMC plug-ins atop it. PLASMA incorporates in-built compilers to create bytecode for execution on its own stack-based machine, thereby contributing to increase the efficiency of the SMC procedure. PLASMA also offers three alternative modes for SMC, namely a simple Monte Carlo probabilistic algorithm [120], a Monte Carlo algorithm with Chernoff confidence bounds [60], and sequential hypothesis testing [54].

One of the outstanding features of PLASMA is the ability of developing plug-ins atop the platform, so that users can take advantage of the PLASMA environment to create custom statistical model checkers. For instance, users who have developed their own model description language can use it with PLASMA by providing a simulator plug-in. Similarly, users can add custom languages for specifying properties and use the available SMC algorithms through a checker plug-in. Besides its efficiency and good performance results [115, 121], such a flexibility was one of the main reasons motivating the choice of PLASMA to serve as basis to develop the toolchain for specifying and verifying properties of dynamic software architectures described in π-ADL. In addition, PLASMA has been applied to problems from different application domains and it is also used by several European projects [122].

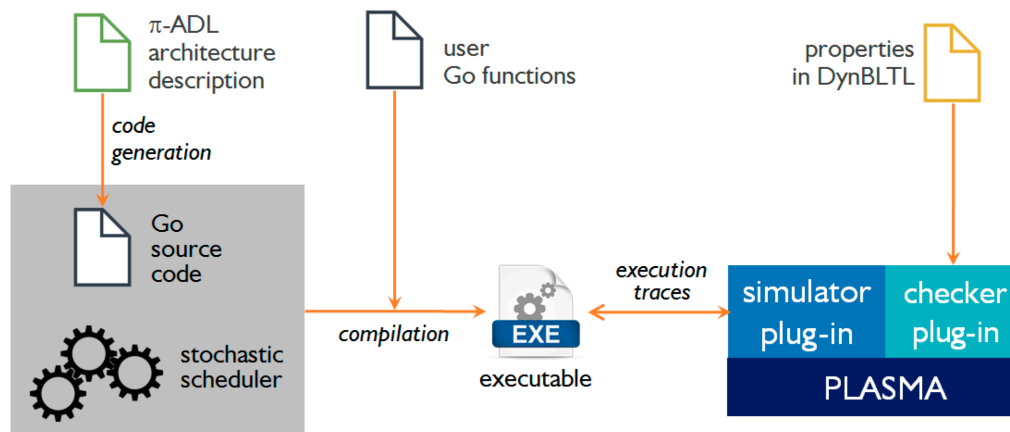
Figure 29 provides an overview of the developed SMC-based toolchain[18] for verifying properties of dynamic software architectures. The inputs for the process are (i) an architecture description in π-ADL and (ii) a set of properties specified in DynBLTL. By following the process described in Chapter 5, the architecture description in the π-ADL language is translated towards generating source code in the

---

[18] The developed toolchain is publicly available at http://plasma4pi-adl.gforge.inria.fr.

Go programming language, but augmented with the state transition actions introduced in Section 6.1. Instrumenting the Go source code with these actions is necessary in order to allow generating execution traces upon the compilation and execution of such a code. These execution traces are provided as input to the simulator plug-in, which makes use of the SMC algorithms originally implemented in PLASMA.

Figure 29 – Overview of the developed SMC-based toolchain for verifying properties expressed in DynBLTL regarding dynamic software architectures described in π-ADL



As π-ADL architectural models do not have a stochastic execution, they are linked to a stochastic scheduler parameterized by a probability distribution for drawing the next action, as described in Section 6.3.1. Furthermore, existing probability distribution Go libraries are used to model inputs of system models as user functions. The program resulting from the compilation of the generated Go source code emits messages referring to transitions from a given state to another in case of addition, attachment, detachment, and value exchanges of architectural elements.

Two plug-ins were developed atop the PLASMA platform, namely (i) a simulator plug-in that interprets execution traces produced by the generated Go program and (ii) a checker plug-in that implements DynBLTL. With this toolchain, a software architect is able to evaluate the probability of a π-ADL architectural model to satisfy a given property specified in DynBLTL.

6.3.3 Quantitative evaluation

In this section, some experiments aiming to quantitatively evaluate the efficiency of the proposed SMC-based approach to support the architectural analysis activity are reported. Considering that the literature already reports that PLASMA and its SMC algorithms outperform other existing approaches (c.f. [115, 121, 122]), the experiments concerned assessing *how* efficient is such an approach and toolchain to verify properties in dynamic software architectures. In the experiments, computational effort in terms of execution time and RAM consumption were chosen

as metrics, which were used to observe the performance of the toolchain when varying the precision of the verification. As PLASMA is executed upon a Java Virtual Machine, 20 runs were performed for each precision value in order to ensure a proper statistical significance for the results. The experiments[19] were conducted under GNU/Linux on a computer equipped with a quad-core 3 GHz processor and 16 GB of RAM. Time and RAM consumption measures were obtained by using the *time* utility from Linux.

The toolchain was evaluated with the `FalsePositive`, `SensorAvailable`, and `RemoveSensor` properties described in Section 6.2.2.2 with predefined time units for the temporal operators. These properties were evaluated using the Chernoff algorithm [60] from PLASMA, which requires a precision and a confidence degree as parameters and returns an approximation of the probability with an error below the precision parameter, with the given confidence. A confidence of 95% and a precision ranging from 0.1 to 0.02 were chosen. Using descriptive statistics [53], Table 11 and Table 12 show the minimum, maximum, average, and standard deviation values for the execution time (in seconds) and RAM consumption (in megabytes).

Table 11 – Descriptive statistics for execution time of the analysis (in seconds)

Property	**Precision**	**Minimum**	**Maximum**	**Average**	**Standard deviation**
	0.10	49.69	59.49	54.49	2.2730
	0.05	182.19	199.80	192.50	4.8063
`FalsePositive(100, 3)`	0.04	293.65	335.97	301.32	9.7782
	0.03	512.89	563.13	528.36	13.8758
	0.02	1138.11	1233.96	1175.72	25.7600
	0.10	16.68	18.14	17.68	0.4253
	0.05	58.30	62.67	59.58	1.0564
`SensorAvailable(50, 2)`	0.04	88.59	96.15	90.42	1.6064
	0.03	153.80	159.89	156.16	1.5584
	0.02	340.90	363.49	350.06	7.6525
	0.10	38.29	42.85	41.40	1.1708
	0.05	144.27	160.55	150.37	4.3215
`RemoveSensor(100, 4)`	0.04	222.78	235.86	229.54	3.3158
	0.03	398.23	421.50	406.23	6.0654
	0.02	883.93	968.54	910.49	22.1756

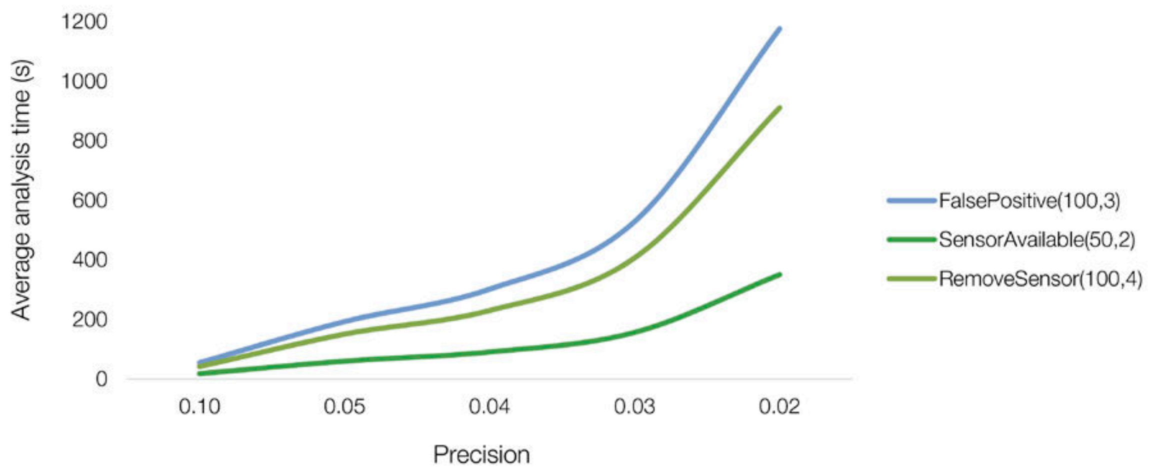
[19] The complete instructions on how to reproduce the performed experiments are publicly available at http://plasma4pi-adl.gforge.inria.fr/

Table 12 – Descriptive statistics for RAM consumption (in megabytes)

Property	Precision	Minimum	Maximum	Average	Standard deviation
FalsePositive(100, 3)	0.10	1747	1885	1791	38.0714
	0.05	1721	1950	1812	61.4808
	0.04	1724	1973	1842	73.5887
	0.03	1706	2012	1858	85.3014
	0.02	1721	2023	1870	71.1738
SensorAvailable(50, 2)	0.10	1367	1585	1493	60.6001
	0.05	1531	1625	1583	34.7258
	0.04	1514	1627	1574	33.2827
	0.03	1511	1643	1579	35,7215
	0.02	1546	1634	1573	25.8882
RemoveSensor(100, 4)	0.10	1670	1821	1740	37.4755
	0.05	1749	1871	1799	33.1524
	0.04	1747	1856	1801	31.9481
	0.03	1726	1870	1815	41.5488
	0.02	1732	1974	1842	78.1475

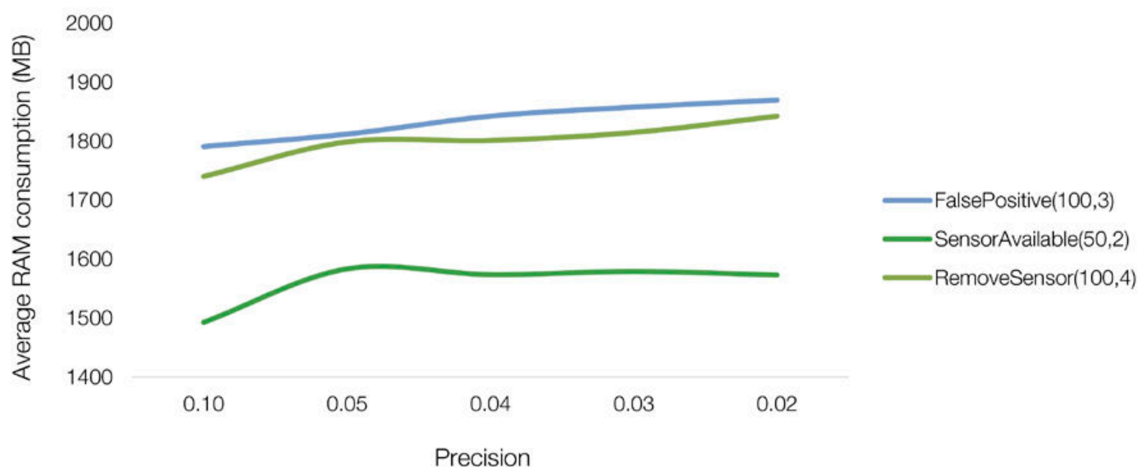
Figure 30 shows how the analysis time increases when the precision increases (i.e., the error decreases). As highlighted in Section 2.3, a higher accuracy of the answer provided by the statistical model checker requires generating more execution traces through simulations, thereby increasing the analysis time. In case of rare events, i.e., properties that have a very low probability to happen, a better convergence can be obtained by using dedicated methods [115]. The property regarding the sensor availability evaluated over a window of 50 time units requires less time than the other properties evaluated over a window of 100 time units because the analysis of each trace is faster.

Figure 30 – Effect of the precision variation in the analysis of three properties upon analysis time (measured in seconds)



In Figure 31, it is possible to observe that the amount of RAM required to perform the analyses are nearly constant, thus meaning that the precision has no strong influence on the RAM consumption. This can be explained by the fact that SMC only analyzes one trace at a time. Therefore, it is possible to conclude that the proposed SMC-based approach and toolchain can be regarded as efficient with respect to both execution time and RAM consumption.

Figure 31 – Effect of the precision variation in the analysis of three properties upon RAM consumption (in megabytes)



6.4 Related work: Formal specification and verification of architectural properties in dynamic systems

As far as it is concerned, this is the first work on the application of SMC to verify properties in dynamic software architectures. Therefore, the brief discussion presented in this section mainly concerns existing approaches on the specification of properties

in dynamic software systems, in contrast to the features exhibited by DynBLTL.

Traditional versions of temporal logics such as LTL and BLTL are expressed over atomic predicates that evaluate properties to a Boolean value at every point execution. However, a key characteristic of dynamic software systems is the impossibility of foreseeing the exact set of architectural elements deployed at a given point of execution. Such traditional formalisms do not allow reasoning about elements that may appear, disappear, be connected or be disconnected during the execution of the system for two main reasons. First, specifying a predicate for each property of each element is not possible as the set of architectural elements may be unknown a priori. Second, there is no canonical way of assigning a truth value to a property about an element that does not exist at the considered point of execution. In addition, existing approaches to tackle such issues typically focus on behavioral properties, but they do not address structural properties [123]. On the other hand, some approaches assume that architectures are static [124]. DynBLTL overcomes these limitations by being an extension of BLTL able to cope with both structural and behavioral properties in dynamic software architectures, as well as to handle the absence of architectural elements by means of the undefined truth value (see Section 6.2.2).

The Bandera specification language allows model checking multithreaded Java programs [125]. The dynamicity is handled by bounding the number of classes that can be dynamically created to be able to statically build a representation of the state space, but such an approach requires the user to annotate the Java code. Cho et al. [123] also proposed a logic for dealing with dynamic systems based on freeze quantifiers. In both cases, the logic cannot express architectural properties. The π-AAL language [27] was developed to express properties about π-ADL models, but its semantics is not suitable for performing SMC since properties are evaluated on each trace, not on the computation tree.

An important part of the verification of dynamic systems deals with validation of reconfiguration operations and hence several works have provided ways to specify what a correct reconfiguration means. In the Mazzara and Bhattacharyya's work [126], several frameworks for describing and analyzing dynamic reconfiguration are studied, but they do not handle logics similar to DynBLTL. The idea of interleaving quantifiers and temporal logics is not new and has been used in LTL(MSO) [127], for example, in which the number of constituents is constant throughout the execution and therefore such a logic is not applicable to dynamic systems. Basso et al. [128] express architectural properties with additional predicates encoding the state of the architecture, but this logic does not allow interleaving quantifiers (over sets) and temporal operators. Finally, Dormoy et al. [129] propose a logic where architectural properties are used as predicate and expressed through quantifiers, but quantifiers and temporal operators cannot be interleaved as DynBLTL allows for.

7 Conclusion

Software systems have grown in size and complexity and are now an integrated part of every aspect of the society, including finance, transportation, communication, and health care. One of the most prominent ways of taming such a complexity is by means of a *software architecture*, which provides manageable, meaningful system abstractions and play a significant role in the achievement of both functional and quality requirements. Indeed, software architectures are quite useful in system development as they can be a cornerstone at both (i) design time, for verification and validation purposes, and (ii) runtime, for guiding the system implementation and contributing to avoid architectural erosion along the time.

Dynamicity is increasingly becoming an intrinsic property of the contemporary systems, which operate on environments that are highly dynamic, subjected to a number of changes. Therefore, software architectures for these systems need to be dynamic to accommodate such changes, as well as to encompass evolution rules for a software system and its elements during runtime. In a dynamic software architecture, constituent elements may be created, interconnected or removed, or even may undergo a whole rearrangement at runtime, ideally with minimum or no disruption. For this reason, supporting dynamism is important mainly in the case of certain safety- and mission-critical systems, such as traffic control, energy, disaster management, environmental monitoring, and health systems.

Due to its importance, dynamism shall be taken into account at all activities of the software architecture lifecycle. With respect to the architectural representation activity, most of the existing ADLs are not able to properly describe dynamic software architectures, either because they do not cover both structural or behavioral viewpoints or because they do not allow specifying the changes that can be performed over the architecture and its constituent elements. Another important issue in this context is the significant gap between the description and the implementation of a software architecture, a problem that becomes worse mainly as the architecture needs to evolve.

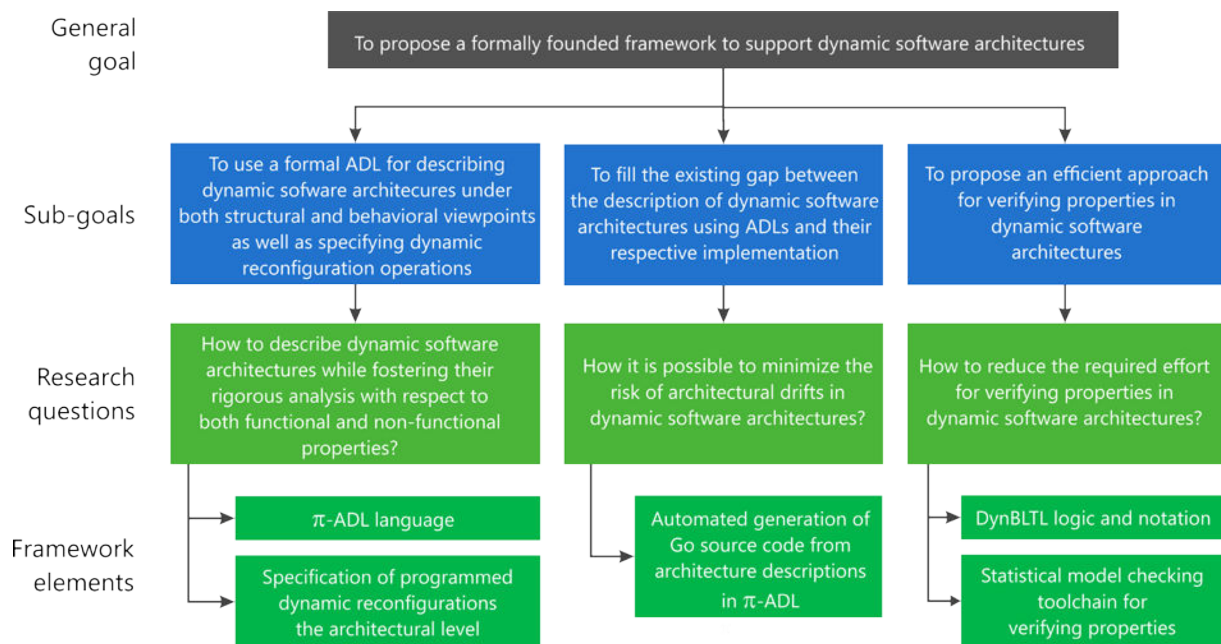
Finally, a cumbersome issue in software architectures is the verification of architectural properties and constraints. Formal techniques such as model checking, albeit being able to precisely determine if a software system can satisfy properties related to user and quality requirements, suffer from limitations regarding scalability and required effort to perform such a task. As previously mentioned, this problem is exacerbated in the contemporary dynamic software systems for two main reasons, namely the non-determinism of their behavior caused by concurrency and the unpredictable environmental conditions in which they operate. This makes the traditional techniques used to verify properties in dynamic software architectures

unfeasible in terms of execution time and computational resources.

7.1 Revisiting the proposal and its contributions

In order to tackle the aforementioned problems, this work has proposed a formally founded framework intended to support dynamic software architectures and their representation, verification, and validation. These concerns are typically not addressed together at the state of the art. Figure 32 depicts the main constituents of the proposed framework (described in the previous chapters) and how they are related to the goals and research questions established in this work. In essence, the framework encompasses: (i) π-ADL, a formal language for describing software architectures under both structural and behavioral viewpoints; (ii) the specification of programmed dynamic reconfiguration operations; (iii) the automated generation of source code from architecture descriptions; and (iv) an SMC-based approach to formally express and verify properties in dynamic software architectures.

Figure 32 – Main constituents of the proposed framework and correlation to goals and associated research questions



The main contributions of this work are fourfold: (i) an ADL able to describe dynamic software architectures; (ii) an automated process to generate source code from architecture descriptions; (iii) an architecturally-driven, computationally efficient approach and toolchain to verify properties in dynamic software architectures; and (iv) a novel logic and notation to formally express properties in dynamic software architectures. Each one of these contributions is described in the following.

An ADL able to formally describe dynamic software architectures. As described in Chapter 4, the π-ADL language was endowed with architecture-level primitives for specifying *programmed reconfiguration* operations. In addition, two common approaches for enacting programmed dynamic reconfiguration were incorporated into π-ADL. The first approach is *exogenous*, in which it is possible to control all elements of the software architecture and apply the changes on the whole structure. In turn, the second approach is *endogenous*, in which the architectural elements themselves are able to manage dynamic reconfiguration actions. This has culminated in an expressive language able to describe both structure and behavior of a dynamic software architecture, as well as the reconfiguration operations that can be applied over it at runtime [35].

An automated process to generate source code from architecture descriptions. Chapter 5 presented the second main contribution of this work, the mapping of architecture descriptions in the π-ADL to implementation source code in Go. Such a mapping process resulted in an automated process for generating source code from an architecture description [35, 37], thereby tackling the existing gap between architectural and implementation levels, contributing to minimize the risk of architectural drifts, and allowing for the validation of the architecture itself.

An architecturally-driven, computationally efficient approach and toolchain for verifying properties in dynamic software architectures. The third contribution, described in Chapter 6, regards the use of SMC to support the formal analysis of dynamic architectures expressed in π-ADL. To support the verification process, a toolchain was developed upon PLASMA, a flexible, modular statistical model checker, striving to reduce effort, computational resources, and time for performing such a task. As far as it is concerned, this is the first work on the application of SMC to verify properties in dynamic software architectures.

A novel logic and notation for formally expressing properties in dynamic software architecture. At last, the fourth contribution of this work, also introduced in Chapter 6, is DynBLTL, a novel logic and notation intended to express properties in dynamic software architectures [41]. Aiming at overcoming the inability of existing notations available at the literature to deal with dynamicity concerns, DynBLTL was designed to take into account architectural elements that are created or removed at runtime, i.e., they may be present in the architecture at a given instant of time and be absent at another. This is achieved by means of the *undefined* truth value ($\mathfrak{U}$), to which a given formula expressing a property is evaluated when considering architectural elements that do not exist in the system at the current state.

7.2 Future work

In spite of the contributions of this work described in Section 7.1, there are many other directions for ongoing and future work. Some of these directions described in Sections 7.2.1 and 7.2.2 respectively as short-term and long-term work.

7.2.1 Short-term work

Improvements on the developed tools. In the scope of this work, two main tools were developed, namely (i) the π-ADL textual editor (c.f. Section 5.2), intended to assist software architects in architectural representation and code generation, and (ii) the SMC-based toolchain to specify and verify properties in dynamic software architectures (c.f. Section 6.3). These tools will be constantly updated and improved, culminating in a development environment to assist software architects in activities such as description of dynamic software architectures, automated generation of source code, validation of software architectures by means of simulation, and verification of architectural properties. Furthermore, it is highly desirable providing π-ADL with a graphical notation that must be consistent and synchronized with the textual notation.

Use of model-transformation metrics within the process for generating Go source code from π-ADL architecture descriptions. As the mapping from π-ADL to Go can be viewed as a model-to-text (M2T) transformation (i.e., the generation of textual artifacts from abstract models) [136], model transformation metrics available in the literature can be used to perform a quantitative evaluation of the process for generating Go source code from π-ADL architecture descriptions. A potential candidate to be used in this evaluation is the set of metrics introduced in the Nguyen's work [137], which comprises Xtend-based M2T transformations.

7.2.2 Long-term work

Expansion of the SMC-based approach. An interesting investigation refers to the expansion of the SMC-based approach proposed in this work towards the specification and verification of properties in systems-of-systems (SoS), a class of systems resulted from the interaction among other distributed, heterogeneous independent systems (the so-called constituent systems) that cooperate to form a larger and more complex system towards the accomplishment of global goals [138]. Each constituent system accomplishes its individual goals and can contribute to the accomplishment of the global goals of the overarching SoS. However, the result of such an interaction is said to be more than the sum of the constituents as it enables the SoS to offer new functionalities that cannot be provided by any of these constituent systems working alone. SoS software architectures are inherently dynamic, i.e., they can be composed and reconfigured at runtime, as well as its concrete constituent systems may be partially known or even unknown at design time [139]. Therefore, the

DynBLTL logic and notation seems to be promising in this context as it can be able to cope with such a lack of prior knowledge about the constituent systems that compose an SoS software architecture, besides the dynamic appearance/disappearance of constituent systems within an SoS at runtime. In this work, π-ADL will be investigated as the ADL for describing SoS architectures, targeting mainly acknowledged SoS.

Investigation on the use of models@runtime to support dynamic reconfiguration of software architectures. Models@runtime can be defined as the abstract representation of a system (including its structure and behavior) that exists in tandem with such a system during its execution [140]. Such an approach has been recently advocated as promising to support dynamic evolution of software systems mainly in cases of unanticipated changes unforeseen at design time [141]. Therefore, the intention is to investigate the applicability of models@runtime as means of supporting the ad-hoc, unforeseen dynamic reconfiguration of a software architecture. Models@runtime can also allow having traceable, manageable models representing the elements of a software architecture at both design time and runtime. Furthermore, the causal connection between the architectural and execution levels allows addressing how reconfiguration actions specified at the former can be reflected into the latter (and vice-versa) while maintaining consistency between them. In this work, models@runtime will be investigated as a technology for implementing the formal definition of π-ADL as a front-end for the translation from π-ADL to Go.

References

[1] Anthony J. Lattanze. Architecting software intensive systems: A practitioner's guide. Boca Raton, FL, USA: Auerbach Publications/Taylor & Francis Group, Inc., 2009.

[2] Len Bass, Paul Clements, Rick Kazman. Software architecture in practice – 3rd edition. USA: Addison-Wesley, 2013.

[3] Mary Shaw, Paul Clements. The Golden Age of Software Architecture. IEEE Software, vol. 23, no. 2, pp. 31–39, Mar./Apr. 2006.

[4] Anthony I. Wasserman. Toward a discipline of Software Engineering. IEEE Software, vol. 13, no. 6, pp. 23–31, Nov. 1996.

[5] ISO/IEC/IEEE 42010:2011(E). ISO/IEC/IEEE International Standard for Systems and Software Engineering – Architectural Description. Geneva, Switzerland: ISO, 2011.

[6] John C. Georgas, Eric M. Dashofy, Richard N. Taylor. Architecture-centric development: A different approach to Software Engineering. Crossroads Magazine, vol. 12, no. 4, Aug. 2006.

[7] David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In: Marco Bernardo, Paola Inverardi (eds.) Formal methods for Software Architecture. Lecture Notes in Computer Science, vol. 2804. Germany: Springer Berlin Heidelberg, 2003, pp. 1–24.

[8] Philippe Kruchten, Henk Obbink, Judith Stafford. The past, present, and future for Software Architecture. IEEE Software, vol. 23, no. 2, pp. 22–30, Mar./Apr. 2006.

[9] Flavio Oquendo. 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, vol. 29, no. 3, pp. 1–14, May 2004.

[10] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'96), San Francisco, CA, USA. New York, NY, USA: ACM, 1996, pp. 3–14.

[11] Richard N. Taylor, Nenad Medvidovic, Eric M. Dashofy. Software Architecture: Foundations, theory, and practice. USA: John Wiley & Sons, Inc., 2010.

[12] Olivier Barais, Anne Françoise Le Meur, Laurence Duchien, Julia Lawall. Software architecture evolution. In: Tom Mens, Serge Demeyer (eds.) Software evolution. Germany: Springer Berlin Heidelberg, 2008, pp. 233–262.

[13] Hongyu Pei Breivold, Ivica Crnkovic, Magnus Larsson. A systematic review of software architecture evolution research. Information and Software Technology, vol. 54, no. 1, pp. 16–40, Jan. 2012.

[14] Patricia Lago, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, Anthony Tang. The road ahead for architectural languages. IEEE Software, vol. 32, no. 1, pp. 98–105, Jan./Feb. 2015.

[15] Paul Clements. A survey of architecture description languages. Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD'96), Schloss Velen, Germany. USA: IEEE Computer Society, 1996, pp. 16–25.

[16] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, Judith Stafford. Document software architectures: Views and beyond – 2nd edition. USA: Addison-Wesley, 2011.

[17] Nenad Medvidovic, Richard N. Taylor. A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70–93, Jan. 2000.

[18] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, Anthony Tang. What industry needs from architectural languages: A survey. IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 869–891, Jun. 2013.

[19] Eric M. Dashofy, André van der Hoek, Richard N. Taylor. A highly-extensible, XML-based architecture description language. Proceedings of the 2001 Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), Amsterdam, The Netherlands. Washington, DC, USA: IEEE Computer Society, 2001, pp. 103–112.

[20] David Garlan, Robert Monroe, David Wile. Acme: An architecture description interchange language. Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97), Toronto, ON, Canada. USA: IBM Press, 1997, pp. 169–189.

[21] Robert J. Allen. A formal approach to software architecture. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

[22] Jeff Magee, Naranker Dulay, Susan Eisenbach, Jeff Kramer. Specifying distributed software architectures. In: Wilhelm Schäfer, Pere Botella (eds.) Proceedings of the 5th European Software Engineering Conference (ESEC'95),

Sitges, Spain. Lecture Notes in Computer Science, vol. 989. United Kingdom: Springer-Verlag London, 1995, pp. 137–153.

[23] Robert Allen, Rémi Douence, David Garlan. Specifying and analyzing dynamic software architectures. In: Egidio Astesiano (ed.) Proceedings of the First International Conference on Fundamental Approaches to Software Engineering (FASE'98), Lisbon, Portugal. Lecture Notes in Computer Science, vol. 1382. Germany: Springer Berlin Heidelberg, 1998, pp. 21–37.

[24] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of software architecture using RAPIDE. IEEE Transactions on Software Engineering, vol. 21, no. 4, pp. 336–355, Apr. 1995.

[25] Dewayne E. Perry, Alexander L. Wolf. Foundations for the study of Software Architecture. ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[26] Jacek Rosik, Andrew Le Gear, Jim Buckley, Muhammad Ali Babar, Dave Connolly. Assessing architectural drift in commercial software development: A case study. Software: Practice and Experience, vol. 41, no. 1, pp. 63–86, Jan. 2011.

[27] Radu Mateescu, Flavio Oquendo. π-AAL: An architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures. ACM SIGSOFT Software Engineering Notes, vol. 31, no. 2, pp. 1–19, Mar. 2006.

[28] Dino Mandrioli. On the heroism of really pursuing formal methods. Proceedings of the 3rd FME Workshop on Formal Methods in Software Engineering (FormaliSE 2015), Florence, Italy. Washington, DC, USA: IEEE, 2015, pp. 1–5.

[29] Pengcheng Zhang, Henry Muccini, Bixin Li. A classification and comparison of model checking software architecture techniques. Journal of Systems and Software, vol. 83, no. 5, pp. 723–744, May 2010.

[30] Edmund M. Clarke, Jr., Orna Grumberg, Doron A. Peled. Model checking. Cambridge, MA, USA: The MIT Press, 1999.

[31] Gerard J. Holzmann. The logic of bugs. Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'02/FSE-10), Charleston, SC, USA. New York, NY, USA: ACM, 2002, pp. 81–87.

[32] Markus Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. Proceedings of the 1992 International Workshop on Configurable Distributed Systems. United Kingdom: IET, 1992, pp. 68–79.

[33] Miguel Alexandre Wermelinger. Specification of software architecture reconfiguration. PhD thesis, Universidade Nova de Lisboa, Lisbon, Portugal, 1999.

[34] Cristóbal Costa-Soria, Jennifer Pérez, Jose Ángel Carsí. Handling the dynamic reconfiguration of software architectures using aspects. Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09), Kaiserslautern, Germany. Washington, DC, USA: IEEE Computer Society, 2009, pp. 263–266.

[35] Everton Cavalcante, Thais Batista, Flavio Oquendo. Supporting dynamic software architectures: From architectural description to implementation. Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015), Montreal, QC, Canada. Washington, DC, USA: IEEE Computer Society, 2015, pp. 31–40.

[36] The Go programming language. Available at: https://golang.org/

[37] Everton Cavalcante, Flavio Oquendo, Thais Batista. Architecture-based code generation: From π-ADL descriptions to implementations in the Go language. In: Paris Avgeriou, Uwe Zdun (eds.) Proceedings of the 8th European Conference on Software Architecture (ECSA 2014), Vienna, Austria. Lecture Notes in Computer Science, vol. 8627. Switzerland: Springer International Publishing, 2014, pp. 130–145.

[38] Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, Axel Legay. Statistical model checking of dynamic software architectures. In: Bedir Tekinerdogan, Uwe Zdun (eds.) Proceedings of the 10th European Conference on Software Architecture (ECSA 2016), Copenhagen, Denmark. Lecture Notes in Computer Science. Switzerland: Springer International Publishing, 2016.

[39] Axel Legay, Benoît Delahaye, Saddek Bensalem. Statistical model checking: An overview. In: Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, Nikolai Tillman (eds.) Proceedings of the First International Conference on Runtime Verification (RV 2010), San Julians, Malta. Lecture Notes in Computer Science, vol. 6418. Germany: Springer Berlin Heidelberg, 2010, pp. 122–135.

[40] Youngjoo Kim, Okjoo Choi, Moonzoo Kim, Jongmoon Baik, Tai-Hyo Kim. Validating software reliability early through statistical model checking. IEEE Software, vol. 30, no. 3, pp. 35–41, May/Jun. 2013.

[41] Jean Quilbeuf, Everton Cavalcante, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, Axel Legay. A logic for the statistical model checking of dynamic

software architectures. In: Tiziana Margaria, Bernhard Steffen (eds.) Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016), Corfu, Greece. Lecture Notes in Computer Science. Switzerland: Springer International Publishing, 2016.

[42] Jeremy S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical report, Queen's University, Kingston, ON, Canada, Mar. 2004.

[43] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04), Newport Beach, CA, USA. New York, NY, USA: ACM, 2004, pp. 28–33.

[44] Nenad Medvidovic. ADLs and dynamic architecture changes. Joint Proceedings of the Second International Software Architecture Workshop and the 1996 International Workshop on Multiple Perspectives in Software Development, San Francisco, CA, USA. New York, NY, USA: ACM, 1996, pp. 24–27.

[45] The Go Programming Language Specification. Available at: https://golang.org/ref/spec

[46] Google Cloud Platform. Available at: https://cloud.google.com/

[47] Ivo Balbaert. The way of Go: A thorough introduction to the Go programming language. Bloomington, IN, USA: iUniverse, 2012.

[48] David Chisnall. The Go programming language phrasebook. Upper Saddle River, NJ, USA: Addison-Wesley/Pearson Education, Inc., 2012.

[49] Mark Summerfield. Programming in Go: Creating applications for the 21st Century. Upper Saddle River, NJ, USA: Addison-Wesley/Qtrac Ltd., 2012.

[50] C. A. R. Hoare. Communicating sequential processes. Upper Saddle River, NJ, USA: Prentice-Hall, 1985.

[51] Axel Legay, Manesh Viswanathan. Statistical model checking: Challenges and perspectives. International Journal on Software Tools for Technology Transfer, vol. 17, no. 4, pp. 369–376, Aug. 2015.

[52] Håkan L. S. Younes, Marta Kwiatkowska, Gethin Norman, David Parker. Numerical vs. statistical probabilistic model checking. International Journal on Software Tools for Technology Transfer, vol. 8, no. 3, pp. 216–228, Jun. 2006.

[53] Mario Lefebvre. Applied Probability and Statistics. USA: Springer New York, 2006.

[54] Håkan Lorens Samir Younes. Verification and planning for stochastic processes with asynchronous events. Doctoral dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2004.

[55] Håkan L. S. Younes, Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In: Ed Brinksma, Kim Guldstrand Larsen (eds.) Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002), Copenhagen, Denmark. Lecture Notes in Computer Science, vol. 2404. Germany: Springer Berlin Heidelberg, 2002, pp. 223–235.

[56] Koushik Sen, Manesh Viswanathan, Gul Agha. Statistical model checking of black-box probabilistic systems. In: Rajeev Alur, Doron A. Peled (eds.) Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004), Boston, MA, USA. Lecture Notes in Computer Science, vol. 3114. Germany: Springer Berlin Heidelberg, 2004, pp. 202–215.

[57] Sumit K. Jha, Edmund M. Clarke, Christopher J. Langmead, Axel Legay, André Platzer, Paolo Zuliani. A Bayesian approach to model checking biological systems. In: Pierpaolo Degano, Roberto Gorrieri (eds.) Proceedings of the 7th International Conference on Computational Methods in Systems Biology (CMSB 2009), Bologna, Italy. Lecture Notes in Computer Science, vol. 5688. Germany: Springer Berlin Heidelberg, 2009, pp. 218–234.

[58] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, Sylvain Peyronnet. Approximate probabilistic model checking. In: Bernhard Steffen, Giorgio Levi (eds.) Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Implementations (VMCAI 2004), Venice, Italy. Lecture Notes in Computer Science, vol. 2937. Germany: Springer Berlin Heidelberg, 2004, pp. 73–84.

[59] Sophie Laplante, Richard Lassaigne, Frédéric Magniez, Sylvain Peyronnet, Michel de Rougemont. Probabilistic abstraction for model checking: An approach based on property testing. ACM Transactions on Computational Logic, vol. 8, no. 4, Aug. 2007.

[60] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association, vol. 58, no. 301, pp. 13–30, Mar. 1963.

[61] Vidyasagar Potdar, Atif Sharif, Elizabeth Chang. Wireless sensor networks: A survey. Proceedings of the 2009 International Conference on Advanced Information Networking and Application Workshops (WAINA'09), Bradford, United Kingdom. USA: IEEE, 2009, pp. 636–641.

[62] Jennifer Wick, Biswanath Mukherjee, Dipak Ghosal. Wireless sensor network survey. Computer Networks, vol. 52, no. 12, pp. 2292–2330, Aug. 2008.

[63] Danny Hughes, Jó Ueyama, Eduardo Mendiondo, Nelson Matthys, Wouter Horré, Sam Michiels, Christophe Huygens, Wouter Joosen, Ka Lok Man, Sheng-Uei Guan. A middleware platform to support river monitoring using wireless sensor networks. Journal of the Brazilian Computer Society, vol. 17, no. 2, pp. 85–102, Jun. 2011.

[64] BBC News. Paris flood warning. Available at http://news.bbc.co.uk/2/hi/science/nature/1782691.stm, Jan. 2002.

[65] FloodList. Brazil floods – Over 200,000 affected in Santa Catarina and Rio Grande do Sul. Available at http://floodlist.com/america/brazil-floods-santa-catarina-rio-grande-do-sul, Oct. 2015.

[66] David De Roure. Floodnet: A new flood warning system. Ingenia, vol. 23, pp. 50–51, Jun. 2005.

[67] Slobodan P. Simonovic. Decision support system for flood management in the Red River Basin. Canadian Water Resources Journal, vol. 24, no. 3, pp. 203–223, Jun. 2004.

[68] Jó Ueyama, Daniel Roy Hughes, Nelson Matthys, Wouter Horré, Wouter Joosen, Christophe Huygens, and Sam Michiels. An event-based component model for wireless sensor networks: A case study for river monitoring. Proceedings of the 28th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC 2010), Gramado, RS, Brazil. Porto Alegre, RS, Brazil: SBC, 2010, pp. 997–1004.

[69] Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, Wouter Joosen. LooCI: A loosely-coupled component infrastructure for networked embedded systems. Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM 2009), Kuala Lumpur, Malaysia. New York, NY, USA: ACM, 2009, pp. 195–203.

[70] Agência Nacional do Petróleo, Gás Natural e Biocombustíveis (Brazil). Anuário Estatístico Brasileiro do Petróleo, Gás Natural e Biocombustíveis 2015 (in Portuguese). Rio de Janeiro, RJ, Brazil: ANP, 2015. Publicly available at: http://www.anp.gov.br/?dw=78135.

[71] Mohammad reza Akhondi, Alex Talevski, Simon Carlsen, Stig Peterson. Applications of wireless sensor networks in the oil, gas and resources industries. Proceedings of the 24th IEEE International Conference on Advanced

Information Networking and Applications, Perth, WA, Australia. USA: IEEE, 2010, pp. 941–948.

[72] Petru Junie, Octav Dinu, Cristian Eremia, Dan Stefanoiu, Catalin Petrescu, Ioan Savulescu. A WSN based monitoring system for oil and gas transportation through pipelines. IFAC Proceedings Volumes, vol. 45, no. 6, pp. 1796–1801, May 2012.

[73] Ivan Stoianov, Lama Nachman, Sam Madden, Timur Tokmouline. PIPENET: A wireless sensor network for pipeline monitoring. Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN'07), Cambridge, MA, USA. New York, NY, USA: ACM, 2007, pp. 264–273.

[74] Ilmad Jawar, Nader Mohamed, Khaleb Shuaib. A framework for pipeline infrastructure monitoring using wireless sensor networks. Proceedings of the 2007 Wireless Telecommunications Symposium (WTS 2007), Pomona, CA, USA. USA: IEEE, 2007, pp. 1–7.

[75] Gbenga Owojaiye, Yichuang Sun. Focal design issues affecting the deployment of wireless sensor networks for pipeline monitoring. Ad Hoc Networks, vol. 11, no. 3, pp. 1237–1253, May 2013.

[76] Flávio E. A. Horita, Maria C. Fava, Eduardo M. Mendiondo, Jairo Rotava, Vladimir C. Souza, Jó Ueyama, João Porto de Albuquerque. AGORA-GeoDash: A geosensor dashboard for real-time flood risk monitoring. In: Starr Roxanne Hiltz, Mark S. Pfaff, Linda Plotnick, Patrick S. Shih (eds.) Proceedings of the 11th International Conference on Information Systems for Crisis Response and Management (ISCRAM 2014), University Park, PA, USA. USA: The Pennsylvania State University, 2014, pp. 309–318.

[77] Jesús M. T. Portocarrero, Flavia C. Delicato, Paulo F. Pires, Nadia Gámez, Lidia Fuentes, David Ludovino, Paulo Ferreira. Autonomous wireless sensor networks: A systematic literature review. Journal of Sensors, 2014.

[78] Danny Hughes, Phil Greenwood, Geoff Coulson, Gordon Blair. GridStix: Supporting flood prediction using embedded hardware and next generation grid middleware. Proceedings of the 2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06), Niagara Falls/Buffalo, NY, USA. Washington, DC, USA: IEEE Computer Society, 2006, pp. 621–626.

[79] Benjamin C. Pierce. Foundational calculi for programming languages. In: Allen B. Tucker, editor. Handbook of Computer Science and Engineering. USA: CRC Press, 1997, pp. 2190–2207.

[80] Robin Milner. A calculus of communicating systems. Lecture Notes in Computer Science, vol. 92. Germany: Springer Berlin Heidelberg, 1980.

[81] Robin Milner. Communicating and mobile systems: The π-calculus. New York, NY, USA: Cambridge University Press, 1999.

[82] Robin Milner. Functions as processes. Mathematical Structures in Computer Science, vol. 2, no. 2, pp. 119–141, Jun. 1992.

[83] Peter Michael Sewell. The algebra of finite state processes. PhD thesis, University of Edimburgh, Edimburgh, Scotland, United Kingdom, 1995.

[84] Calos Canal, Ernesto Pimentel, José M. Troya. Specification and refinement of dynamic software architectures. In: Patrick Donohoe (ed.) Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA. IFIP Series, vol. 12. USA: Springer US, 1999, pp. 107–125.

[85] Alessandro Aldini, Marco Bernardo, Flavio Corradini. A process algebraic approach to software architecture design. United Kingdom: Springer London, 2010.

[86] Flavio Oquendo, Ilham Alloui, Sorana Cîmpan, Herve Verjus. The ArchWare ADL: Definition of the abstract syntax and formal semantics. Technical report, The ArchWare Consortium, Dec. 2002.

[87] Flavio Oquendo. Tutorial on ArchWare ADL – Version 2 (π-ADL Tutorial). Technical report, The ArchWare Consortium, Jun. 2005.

[88] Ron Morrison, Graham Kirby, Dharini Balasubramaniam, Kath Mickan, Flavio Oquendo, Sorana Cîmpan, Brian Warboys, Bob Snowdon, R. Mark Greenwood. Support for evolving software architectures in the ArchWare ADL. Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04), Oslo, Norway. Washington, DC, USA: IEEE Computer Society, 2004, pp. 69–78.

[89] Brian Warboys, Bob Snowdon, R. Mark Greenwood, Wykeen Seet, Ian Robertson, Ron Morrison, Dharini Balasubramaniam, Graham Kirby, Kath Mickan. An active-architecture approach to COTS integration. IEEE Software, vol. 22, no. 4, pp. 20–27, Jul. 2005.

[90] Peter H. Feiler, David P. Gluch. Model-Based Engineering with AADL: An introduction to the SAE Architecture Analysis & Design Language. Upper Saddle River, NJ, USA: Addison Wesley/Pearson Education, Inc., 2013.

[91] Thais Batista, Ackbar Joolia, Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In: Ron Morrison, Flavio Oquendo (eds.) Proceedings of the 2nd European Workshop on Software Architecture (EWSA

2005), Pisa, Italy. Lecture Notes in Computer Science, vol. 3527. Germany: Springer Berlin Heidelberg, 2005, pp. 1–17.

[92] Robert T. Monroe. Capturing software architecture design expertise with Armani. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, Jan. 2001.

[93] Muhammad Atif Javed, Uwe Zdun. A systematic literature review of traceability approaches between software architecture and source code. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14), London, England, United Kingdom. New York, NY, USA: ACM, 2014.

[94] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, vol. 21, no. 4, pp. 314–335, Apr. 1995.

[95] Nenad Medvidovic, David S. Rosenblum, Richard N. Taylor. A language and environment for architecture-based software development and evolution. Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA. New York, NY, USA: ACM, 1999, pp. 44–53.

[96] Flavio Oquendo, Brian Warboys, Ron Morrison, Régis Dindeleux, Ferdinando Gallo, Hubert Garavel, Carmen Occhipinti. ArchWare: Architecting evolvable software. In: Flavio Oquendo, Brian C. Warboys, Ron Morrison (eds.) Proceedings of the First European Workshop on Software Architecture (EWSA 2004), St. Andrews, Scotland, United Kingdom. Lecture Notes in Computer Science, vol. 3047. Germany: Springer Berlin Heidelberg, 2004, pp. 257–271.

[97] Flavio Oquendo. π-Method: A model-driven formal method for architecture-centric software engineering. ACM SIGSOFT Software Engineering Notes, vol. 31, no. 3, pp. 1–13, May 2006.

[98] Eclipse Platform. Available at: http://www.eclipse.org/

[99] Xtext. Available at: https://eclipse.org/Xtext/

[100] ISO/IEC 14977:1996(E). ISO/IEC International Standard for Information Technology – Syntactic metalanguage, Extended BNF. Geneva, Switzerland: ISO/IEC, 1996.

[101] ANTRL. Available at: http://www.antlr.org/

[102] Xtend. Available at: https://www.eclipse.org/xtend/

[103] Patchwork Toolkit – Lightweight platform for the network of things. Available at: https://blog.gopheracademy.com/advent-2014/patchwork/

[104] EMDB – Golang Embedded Programming Framework. Available at: http://embd.kidoman.io/

[105] Gobot – Golang framework for robotics, physical computing, and the Internet of Things. Available at: https://gobot.io/

[106] Jonathan Aldrich, Craig Chambers, David Notkin. ArchJava: Connecting software architecture to implementation. Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, FL, USA. New York, NY, USA: ACM, 2002, pp. 187–197.

[107] Jonathan Aldrich, Craig Chambers, David Notkin. Architectural reasoning in ArchJava. In: Boris Magnusson (ed.) Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002), Malaga, Spain. Lecture Notes in Computer Science, vol. 2374. Germany: Springer Berlin Heidelberg, 2002, pp. 334–367.

[108] Jan Oliver Ringert, Bernhard Rump, Andreas Wortmann. From software architecture structure and behavior modeling to implementations of cyber-physical systems. In: Stefan Wagner, Horst Lichter (eds.) Proceedings of the 6. Arbeitstagung Programmiersprachen (ATPS 2013), Aachen, Germany. Lecture Notes in Informatics, vol. P-215. Bonn, Germany: Gesellschaft für Informatik, 2013, pp. 155–170.

[109] Jan Oliver Ringert, Bernhard Rumpe, Andreas Wortmann. Architecture and behavior modeling of cyber-physical systems with MontiArchAutomaton. Aachen, Germany: RWTH Aachen University, 2014.

[110] Arne Haber, Jan Oliver Ringert, Bernhard Rumpe. MontiArc: Architectural modeling of interactive distributed and cyber-physical systems. Technical report, RWTH Aachen University, Aachen, Germany, 2012.

[111] Zawar Qayyum. Realization of software architectures using a formal language: Towards languages dedicated to formal development based on π-ADL. PhD thesis, Université Bretagne Sud, Vannes, France, 2009 (original title in French: *Concrétisation des architectures logicielles à l'aide d'un langage formel : Vers les langages dédiés au développement formel fondés sur π-ADL*).

[112] Microsoft .NET. Available at: http://www.microsoft.com/net

[113] IEC/IEEE 23271:2012(E). ISO/IEC International Standard for Information Technology – Common Language Infrastructure. Geneva, Switzerland: ISO/IEC, 2012.

[114] PLASMA Lab. Available at: https://project.inria.fr/plasma-lab/

[115] Cyrille Jegourel, Axel Legay, Sean Sedwards. A platform for high performance statistical model checking – PLASMA. In: Cormac Flanagan, Barbara König (eds.) Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012), Tallinn, Estonia. Lecture Notes in Computer Science, vol. 7214. Germany: Springer Berlin Heidelberg, 2012, pp. 498–503.

[116] Amir Pnueli. The temporal logics of programs. Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS'77), Providence, RI, USA. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57.

[117] Zohar Manna, Amir Pnueli. The temporal logic of reactive and concurrent systems: Specification. USA: Springer-Verlag New York, 1992.

[118] Koushik Sen, Mahesh Viswanathan, Gul Agha. On statistical model checking of stochastic systems. In: Kousha Etessami, Sriram K. Rajamani (eds.) Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005), Edinburgh, Scotland, United Kingdom. Lecture Notes in Computer Science, vol. 3576. Germany: Springer Berlin Heidelberg, 2005, pp. 266–280.

[119] Radu Grosu, Scott A. Smolka. Monte Carlo model checking. In: Nicolas Halbwachs, Lenore D. Zuck (eds.) Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Edinburgh, Scotland, United Kingdom. Lecture Notes in Computer Science, vol. 3440. Germany: Springer Berlin Heidelberg, 2005, pp. 271–286.

[120] George S. Fishman. Monte Carlo: Concepts, algorithms, and applications. USA: Springer New York, 1996.

[121] Benoît Boyer, Kevin Corre, Axel Legay, Sean Sedwards. PLASMA-lab: A flexible, distributable statistical model checking library. In: Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, Pedro R. D'Argenio (eds.) Proceedings of the 10th International Conference on Quantitative Evaluation of Systems (QEST 2013), Buenos Aires, Argentina. Lecture Notes in Computer Science, vol. 8054. Germany: Springer Berlin Heidelberg, 2013, pp. 160–164.

[122] Axel Legay, Sean Sedwards. On statistical model checking with PLASMA. Proceedings of the 2014 Theoretical Aspects of Software Engineering Conference (TASE 2014), Changsha, China. Washington, DC, USA: IEEE Computer Society, 2014, pp. 139–145.

[123] S. M. Cho, H. H. Kim, S. D. Cha, D. H. Bae. Specification and validation of dynamic systems using temporal logic. IEE Proceedings – Software, vol. 148, no. 4, pp. 135–140, Aug. 2001.

[124] Raluca Marinescu, Henrik Kaijser, Marius Mikučiois, Cristina Seceleanu, Henrik Lönn, Alexandre David. Analyzing industrial architectural models by simulation and model-checking. In: Cyrille Artho, Peter Csaba Ölveczky (eds.) Proceedings of the Third International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2014), Luxembourg. Communications in Computer and Information Science, vol. 476. Switzerland: Springer International Publishing, 2014, pp. 189–205.

[125] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Robby. Expressing checkable properties of dynamic systems: The Bandera specification language. International Journal on Software Tools for Technology Transfer, vol. 4, no. 1, pp. 34–56, Oct. 2002.

[126] Manuel Mazzara, Anirban Bhattacharyya. On modelling and analysis of dynamic reconfiguration of dependable real-time systems. Proceedings of the Third International Conference on Dependability (DEPEND 2010), Venice/Mestre, Italy. Wilmington, DE, USA: IARIA, 2010, pp. 173–181.

[127] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, Mayank Saksena. Regular model checking for LTL(MSO). International Journal on Software Tools for Technology Transfer, vol. 14, no. 2, pp. 223–241, Apr. 2012.

[128] Alessandro Basso, Alexander Boltov, Artie Basukoski, Vladimir Getov, Ludovic Henrio, Mariusz Urbanski. Specification and verification of reconfiguration protocols in grid component systems. Proceedings of the 3rd IEEE Conference on Intelligent Systems (IS'06), London, United Kingdom. USA: IEEE, 2006, pp. 450–455.

[129] Julien Dormoy, Olga Kouchnarenko, Arnaud Lanoix. Using temporal logic for dynamic reconfigurations of components. In: Luís Soares Barbosa, Markus Lumpe (eds.) Proceedings of the 7th International Workshop on Formal Aspects of Component Software (FACS 2010), Guimarães, Portugal. Lecture Notes in Computer Science, vol. 6921. Germany: Springer Berlin Heidelberg, 2010, pp. 200–217.

[130] Alan Blackwell, Thomas Green. Notational systems – The Cognitive Dimensions of Notations framework. In: John M. Caroll (ed.) HCI models, theories, and frameworks. San Francisco, CA, USA: Morgan Kaufmann Publishers/Elsevier Science, 2003, pp. 103–133.

[131] Daniel Moody. The "Physics" of notations: Toward a scientific basis for constructing visual notations in Software Engineering. IEEE Transactions in Software Engineering, vol. 35, no. 6, pp. 756–779, Nov. 2009.

[132] Barbara Kitchenham, Lesley Pickard. Case studies for method and tool evaluation. IEEE Software, vol. 12, no. 4, pp. 52–62, Jul. 1995.

[133] Per Runeson, Martin Höst, Austen Rainer, Björn Regnell. Case study research in Software Engineering: Guidelines and examples. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2012.

[134] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, Dag I. K. Sjøberg. A systematic review of quasi-experiments in Software Engineering. Information and Software Technology, vol. 51, no. 1, pp. 71–82, Jan. 2009.

[135] Claes Wohlim, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, Anders Wesslén. Experimentation in Software Engineering. Germany: Springer Berlin Heidelberg, 2012.

[136] Krzysztof Czarnecki, Simon Helsen. Feature-based survey of model transformation approaches. IBM Systems Journal, vol. 45, no. 3, pp. 621–645, Jul. 2006.

[137] Phu hong Nguyen. Quantitative analysis of model transformations. Master's Thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2010.

[138] Mark W. Maier. Architecting principles for systems-of-systems. Systems Engineering, vol. 1, no. 4, pp. 267–284, Feb. 1998.

[139] Everton Cavalcante. On the architecture-driven development of software-intensive systems-of-systems. Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Florence, Italy, vol. 2. Washington, DC, USA: IEEE, 2015, pp. 899–902.

[140] Gordon Blair, Nelly Bencomo, Robert B. France. Models@run.time. Computer, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[141] Amel Bennaceur et al. Mechanisms for leveraging models at runtime in self-adaptive software. In: Nelly Bencomo, Robert France, Betty H. C. Cheng, Uwe Aßmann (eds.) Models@run.time: Foundations, applications, and roadmaps. Lecture Notes in Computer Science, vol. 8378. Switzerland: Springer International Publishing, 2014, pp. 19–46.

Appendix A – π-ADL Grammar

This appendix describes the concrete textual syntax of the π-ADL language by using the Extended Backus-Naur Form (EBNF) meta-language [1], a notation for formally describing the context-free grammar of a language, i.e., its syntax. Section A.1 presents the notation elements used hereinafter whereas Section A.2 presents the production rules forming the π-ADL grammar.

A.1 Grammar notation

The EBNF meta-language consists of *terminal symbols*, which are a sequence of one or more characters forming an irreducible element of the language, and non-terminal *production rules* governing how a particular syntactic element can be legally formed in terms of terminal symbols. Syntactic elements have names that are used in production rules and they are distinguished from names and reserved words (keywords) in the language. Furthermore, the EBNF meta-language uses a set of *meta-symbols* summarized in Table A-I.

Table A-I – EBNF meta-symbols

Meta-symbol	Usage
Right arrow (→)	Definition of production rule: $A \rightarrow B$ is read as *A is defined as B*
Pipe symbol (|)	Alternative choice between elements in production rule
Brackets ([and])	Optional occurrence of element in production rule
Asterisk character (∗)	Multiple occurrences of element in production rule
Plus character (+)	At least one occurrence of element in production rule
Ampersand character (&)	Any occurrence order of elements in production rule: *A*&*B* denotes that both *AB* and *BA* sequences of the elements *A* and *B* are valid
Ellipses (…)	Character range
Parentheses	Element grouping

In the EBNF specification of the π-ADL grammar, reserved words and terminal symbols of the language are expressed in a typewriter font whereas names of production rules are typed in the italic form. It is important to highlight that names of production rules and attributes in this specification seek to be representative and self-explanatory.

As an example, consider the three following production rules:

```
ArchitectureDescription →
ArchitecturalElement*
Architecture+
BehaviorDeclaration

ArchitecturalElement → Component | Connector

Component →
component Identifier is abstraction([Parameter (, Parameter)*]) {
    TypeDeclaration*
    ConnectionDeclaration*
    VariableDeclaration*
    [ProtocolDeclaration]
    BehaviorDeclaration
}
```

The *ArchitectureDescription* rule refers to an architecture description composed of a set of architectural elements (defined by the *ArchitecturalElement* rule), at least one architecture (defined by the *Architecture* rule), and exactly one behavior declaration (defined by the *BehaviorDeclaration* rule). An architectural element can be either a component or a connector, respectively defined by the *Component* and *Connector* rules. A component is declared by using the component keyword and it comprises an identifier (represented by the *Identifier* rule). Optionally, a component can also take as input a list of parameters, each one defined by the *Parameter* rule. Within the definition of this architectural element (delimited by braces), one can have in sequence:

- declaration of zero or more types defined by the *TypeDeclaration* rule;
- declaration of zero or more connections defined by the *ConnectionDeclaration* rule;
- declaration of zero or more variables defined in the *VariableDeclaration* rule;
- optional declaration of a protocol defined by the *ProtocolDeclaration* rule; and
- declaration of exactly one behavior defined by the *BehaviorDeclaration* rule.

A.2 π-ADL production rules

ArchitectureDescription →
ArchitecturalElement∗
Architecture+
BehaviorDeclaration

ArchitecturalElement → *Component* | *Connector*

Component →
`component` *Identifier* `is` `abstraction(`[*Parameter*∗ (, *Parameter*)∗]`) {`
  TypeDeclaration∗
  ConnectionDeclaration∗
  VariableDeclaration∗
  [*ProtocolDeclaration*]
  BehaviorDeclaration
`}`

Connector →
`connector` *Identifier* `is` `abstraction`[*Parameter*∗ (, *Parameter*)∗]`) {`
  TypeDeclaration∗
  ConnectionDeclaration∗
  VariableDeclaration∗
  [*ProtocolDeclaration*]
  BehaviorDeclaration
`}`

TypeDeclaration → `type` *Identifier* `is` *ValueType*

ConnectionDeclaration → `connection` *Identifier* `is` *ConnectionMode* `(`*ValueType*`)`

ConnectionMode → `in` | `out`

VariableDeclaration → *Identifier* `is` *ValueType*

ProtocolDeclaration →
`protocol is {`
  `(`*ProtocolAction*∗ `(|` *ProtocolAction*)∗ *ProtocolAction*∗`)` (∗ | +)
`}`

ProtocolAction → `(`∗ `via` *Identifier* *Action* *ValueType* `)`∗

Action → `send` | `receive`

BehaviorDeclaration →

```
behavior is {
    BehaviorClause*
}

BehaviorClause →
  TypeDeclaration
| Prefix
| Choice
| Composition
| Decomposition
| Recurse
| IfThenElse
| Statement
| Inaction
| Iteration

Prefix → InputPrefix | OutputPrefix | SilentPrefix

InputPrefix → via Identifier receive Parameter

OutputPrefix → via Identifier send AbstractExpression

SilentPrefix → unobservable

Parameter → Identifier : ValueType

Choice →
choose {
    BehaviorClause+
    (or BehaviorClause+)+
}

Composition →
compose {
    [(BehaviorClause | ElementInstantiation | ElementReference | VariableRef)+
    (and (BehaviorClause | ElementInstantiation | ElementReference | VariableRef))*]
} [UnificationClause]

UnificationClause →
where {
    (UnificationElem)*
}

UnificationElem → Unification | VariableRef
```

Decompostion → *Identifier* `:=` `decompose` *Identifier*

Recurse → *SelfRecurse*

SelfRecurse → `behavior(`[*AbstractExpression* (`,` *AbstractExpression*)*]`)`

Inaction → done

Iteration →
`iterate` *VariableRef* `by` *Identifier*
[`from` *Identifier* `initially` *InitialValue*]
`accumulate` {
    *BehaviorClause**
}

InitialValue → *Composition* | *EmptyList* | *EmptyChoice* | *AbstractExpression*

EmptyList → `[]`*ValueType*

EmptyChoice → `choose{}`

Statement →
  VariableDeclaration
| *ExplicitProjection*
| *Assignment*
| *FunctionDeclaration*
| *FunctionCall*
| *While*
| *For*
| *Return*
| *Unobservable*

ExplicitProjection → `project` *Identifier* `as` *Parameter* (`,` *Parameter*)*

Assignment → *VariableAssignment* | *CollectionAddition*

VariableAssignment → *VariableRef* = *AbstractExpression*

CollectionAddition → *VariableRef* `add` (*AbstractExpression* | *Unification*)

FunctionDeclaration →
Identifier `is` `function(`[*Parameter* (`,` *Parameter*)*]`)` [`:` *ValueType*] {
    *BehaviorClause**
}

FunctionCall → *Identifier*`(`[*AbstractExpression* (`,` *AbstractExpression*)*]`)`

Return → `return` *AbstractExpression*

IfThenElse →
`if` *AbstractExpression* `then {`
    BehaviorClause∗
`}` *ElseIf*∗
[*Else*]

ElseIf →
`else if` *AbstractExpression* `then {`
    BehaviorClause∗
`}`

Else →
`else {`
    BehaviorClause∗
`}`

While →
`while` *AbstractExpression* `do {`
    BehaviorClause∗
`}`

For →
`for (`*VariableAssignment* `;` *LogicalExpression* `;` *VariableAssignment*`) do {`
    BehaviorClause∗
`}`

Unobservable → `unobservable`

Architecture →
`architecture` *Identifier* `is abstraction(`[*Parameter* (`,` *Parameter*)∗]`) {`
    TypeDeclaration∗
    BehaviorDeclaration
`}`

ElementInstantiation → *Identifier* `is` *FunctionCall*

ElementReference → *Identifier* `is` *AbstractExpression*

Unification → *ConnectionAccess* `unifies` *ConnectionAccess*

ConnectionAccess → *VariableRef* `::` *Identifier*

ValueType → *BaseType* | *ConstructedType* | *Identifier*

$\mathit{BaseType} \rightarrow$
$\mathit{NaturalType} \mid \mathit{IntegerType} \mid \mathit{RealType} \mid \mathit{BooleanType} \mid \mathit{StringType} \mid \mathit{AnyType}$

$\mathit{NaturalType} \rightarrow \texttt{Natural}$

$\mathit{IntegerType} \rightarrow \texttt{Integer}$

$\mathit{RealType} \rightarrow \texttt{Real}$

$\mathit{BooleanType} \rightarrow \texttt{Boolean}$

$\mathit{StringType} \rightarrow \texttt{String}$

$\mathit{AnyType} \rightarrow \texttt{Any}$

$\mathit{ConstructedType} \rightarrow \mathit{Tuple} \mid \mathit{View} \mid \mathit{Set} \mid \mathit{Sequence}$

$\mathit{Tuple} \rightarrow \texttt{tuple[}\mathit{ValueType}\ (\texttt{,}\ \mathit{ValueType})*\texttt{]}$

$\mathit{View} \rightarrow \texttt{view[}\mathit{LabeledType}\ (\texttt{,}\ \mathit{LabeledType})*\texttt{]}$

$\mathit{LabeledType} \rightarrow \mathit{Identifier}\ \texttt{:}\ \mathit{ValueType}$

$\mathit{Set} \rightarrow \texttt{set[}\mathit{ValueType}\texttt{]}$

$\mathit{Sequence} \rightarrow \texttt{sequence[}\mathit{ValueType}\texttt{]}$

$\mathit{AbstractExpression} \rightarrow \mathit{Expression} \mid \mathit{ConstructedValue}$

$\mathit{Expression} \rightarrow \mathit{LogicalExpression}$

$\mathit{LogicalExpression} \rightarrow \mathit{EqualityExpression}\ (\texttt{||} \mid \texttt{\&\&})\ \mathit{EqualityExpression}$

$\mathit{EqualityExpression} \rightarrow \mathit{RelationalExpression}\ (\texttt{==} \mid \texttt{!=})\ \mathit{RelationalExpression}$

$\mathit{RelationalExpression} \rightarrow \mathit{ArithmeticExpression}\ (\texttt{>=} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{<})\ \mathit{ArithmeticExpression}$

$\mathit{ArithmeticExpression} \rightarrow \mathit{Term}\ (+ \mid -)\ \mathit{Term}$

$\mathit{Term} \rightarrow \mathit{Factor}\ (* \mid \texttt{/} \mid \texttt{mod})\ \mathit{Factor}$

$\mathit{Factor} \rightarrow \texttt{(}\mathit{Expression}\texttt{)} \mid \mathit{UnaryExpression} \mid \mathit{AtomicElement}$

$\mathit{UnaryExpression} \rightarrow \texttt{!}\mathit{AtomicElement}$

$\mathit{AtomicElement} \rightarrow \mathit{LiteralElement} \mid \mathit{VariableRef} \mid \mathit{FunctionCall}$

$\mathit{VariableRef} \rightarrow \mathit{Identifier}(\texttt{[}\ \mathit{AbstractExpression}\ \texttt{]})*$

$LiteralElement \rightarrow IntegerLiteral \mid RealLiteral \mid StringLiteral \mid BooleanLiteral \mid SelfLiteral$

$IntegerLiteral \rightarrow Number$

$RealLiteral \rightarrow RealNumber$

$StringLiteral \rightarrow String$

$BooleanLiteral \rightarrow$ `true` | `false`

$SelfLiteral \rightarrow$ `self`

$ConstructedValue \rightarrow FunctionCall \mid TupleValue \mid ViewValue$

$TupleValue \rightarrow$ `tuple[`$AbstractExpression$ (, $AbstractExpression$)`*]`

$ViewValue \rightarrow$ `view[`$Identifier : AbstractExpression$ (, $Identifier : AbstractExpression$)`*]`

$Number \rightarrow$ (`0...9`)+

$RealNumber \rightarrow Number.Number$

$Identifier \rightarrow$ (`a...z` | `A...Z` | `_`) (`a...z` | `A...Z` | `_` | `0...9`)*

$String \rightarrow$ « any ASCII character »

A.3 References

[1] ISO/IEC 14977:1996(E). ISO/IEC International Standard for Information Technology – Syntactic metalanguage, Extended BNF. Geneva, Switzerland: ISO/IEC, 1996.

Appendix B – The π-ADL textual editor

This appendix provides a description about the implementation of a textual editor for the π-ADL language. Section B.1 concerns the main underlying technologies used for developing the editor. In turn, Section B.2 presents the editor itself and its main constituent elements.

B.1 Preliminaries

The choice of using a development environment based on the Eclipse platform [1] to construct the new π-ADL tools was motivated by its widespread use for developing software and good support in terms of open-source tools and frameworks. Moreover, as Eclipse is based on the Java programming language, it takes advantage of the portability capabilities provided by such a language, thereby allowing its use in multiple operating systems and hardware platforms.

Among the most relevant Eclipse frameworks, the Eclipse Modeling Framework (EMF) [2] is an open-source framework used for model-driven software development and it has been the cornerstone of related technologies and other frameworks. EMF provides facilities for generating code and building tools and applications based on structured *models*. The typical workflow for these tasks by using the EMF facilities encompasses the construction of the models, code generation and customization, and the implementation of the application itself. Therefore, EMF can be seen as the middle ground between abstract models and concrete programming artifacts.

The (meta)model used for representing models in EMF is *Ecore*. In an Ecore model, *classes* are model entities with *attributes* (each one with a name and a *data type*) and relationships (*references*) among each other. From these elements, EMF allows using instances of the classes defined in Ecore to describe a model of the system and then generating its respective code. In this perspective, all frameworks, tools, and applications built upon EMF have an underlying model based on the Ecore (meta)model. In the last years, EMF has been the basis for the construction of several tools and frameworks for developing software. Among them, Xtext [3] is a well-known open-source, highly customizable framework for developing domain-specific languages (DSLs). Xtext covers all aspects of a complete language structure by parsing textual models written in such a language and allows generating code from it in another language.

B.2 The Xtext-based π-ADL textual editor

B.2.1 The π-ADL grammar

The main artifact used as input by Xtext for generating the π-ADL infrastructure is a *grammar* specification in the Extended Backus-Naur Form (EBNF) [4]. This grammar is a set of *production rules* (or simply *rules*) describing the form of the elements that are valid according to the language syntax. When compiling this grammar specification, Xtext generates the respective Ecore metamodel and a Java class corresponding to each production rule.

Figure B-1 shows an excerpt of the π-ADL grammar that was specified based on the π-ADL production rules described in Appendix A. Rules start with their respective name followed by a colon and end with a semicolon. In Figure B-1, line 1 indicates that the current grammar reuses an Xtext grammar with common terminal symbols (`org.eclipse.xtext.common.Terminals`), such as identifiers (`ID`) and integer numbers. The *ArchitectureDescription* rule (lines 5 to 9) corresponds to an architecture description itself, which is composed of a set with zero or more architectural elements (represented by the *ArchitecturalElement* rule), at least one architecture declaration (represented by the *Architecture* rule), and exactly one behavior declaration referring to the controlling behavior (represented by the *BehaviorDeclaration* rule). In turn, the *ArchitecturalElement* rule (lines 11 to 13) define that an architectural element can be either a component or a connector, respectively defined by the *Component* and *Connection* rules. The syntax of the specification of a component is defined by the *Component* rule (lines 15 to 23): it is declared by using the `component` keyword and comprises the `name` attribute, which is represented by the terminal `ID` defined in the Xtext terminals grammar. Optionally, a component can also take a list of parameters as input, each one defined by the *Parameter* rule and separated by commas (line 17). Within the definition of the architectural element (delimited by braces), one can have in sequence:

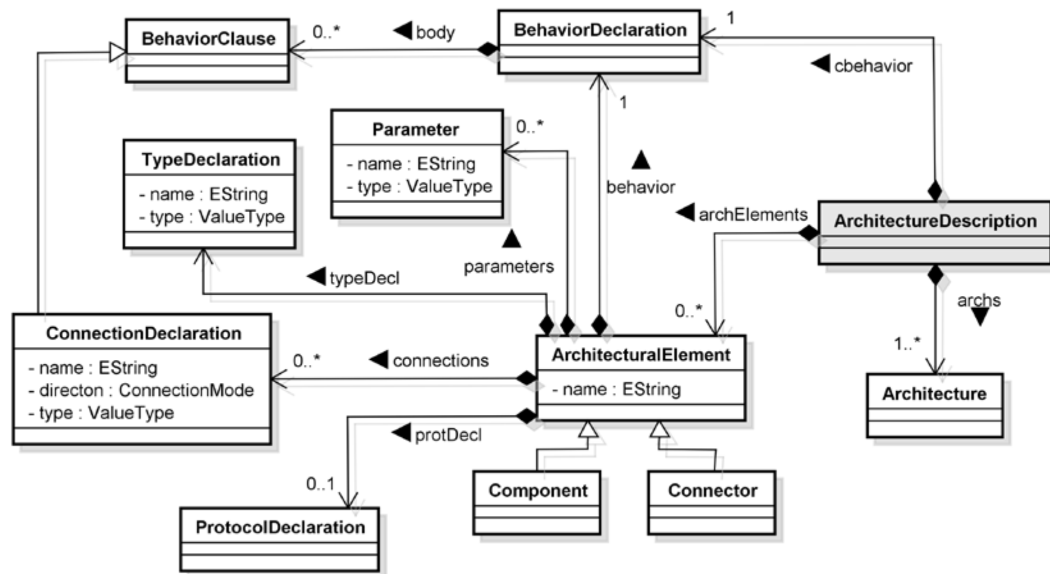
- declaration of zero or more types defined by the *TypeDeclaration* rule (line 18);
- declaration of zero or more connections defined by the *ConnectionDeclaration* rule (line 19);
- optional declaration of a protocol defined by the *ProtocolDeclaration* rule (line 20); and
- declaration of exactly one behavior defined by the *BehaviorDeclaration* rule (line 21).

Figure B-1 – Excerpt of the π-ADL grammar specified in the Xtext framework

```
1:  grammar fr.irisa.archware.PiADL with org.eclipse.xtext.common.Terminals
2:
3:  generate piADL ‘‘http://www.irisa.fr/archware/PiADL’’
4:
5:  ArchitectureDescription:
6:     archElements+=ArchitecturalElement*
7:     archs=Architecture+
8:     cbehavior=BehaviorDeclaration
9:  ;
10:
11: ArchitecturalElement:
12:    Component | Connector
13: ;
14:
15: Component:
16:    ‘component’ name=ID ‘is’ ‘abstraction’
17:    ‘(’ (parameters+=Parameter (‘,’ parameters+=Parameter)*)? ‘)’ ‘{’
18:       typeDecl+=TypeDeclaration*
19:       connections+=ConnectionDeclaration*
20:       protDecl=ProtocolDeclaration?
21:       Behavior=BehaviorDeclaration
22:    ‘}’
23: ;
```

Figure B-2 shows a Class Diagram in the Unified Modeling Language (UML) representing part of the π-ADL metamodel rooted upon the *ArchitectureDescription* class. Despite it is possible to generate an Ecore-based language metamodel from its grammar specification, some developers do not regard this practice as a good one since metamodels are at a higher abstraction level than the concrete syntax model. Nonetheless, it was opted for starting from the π-ADL grammar specification and then generating its corresponding metamodel since such a specification was almost complete from previous versions of the language.

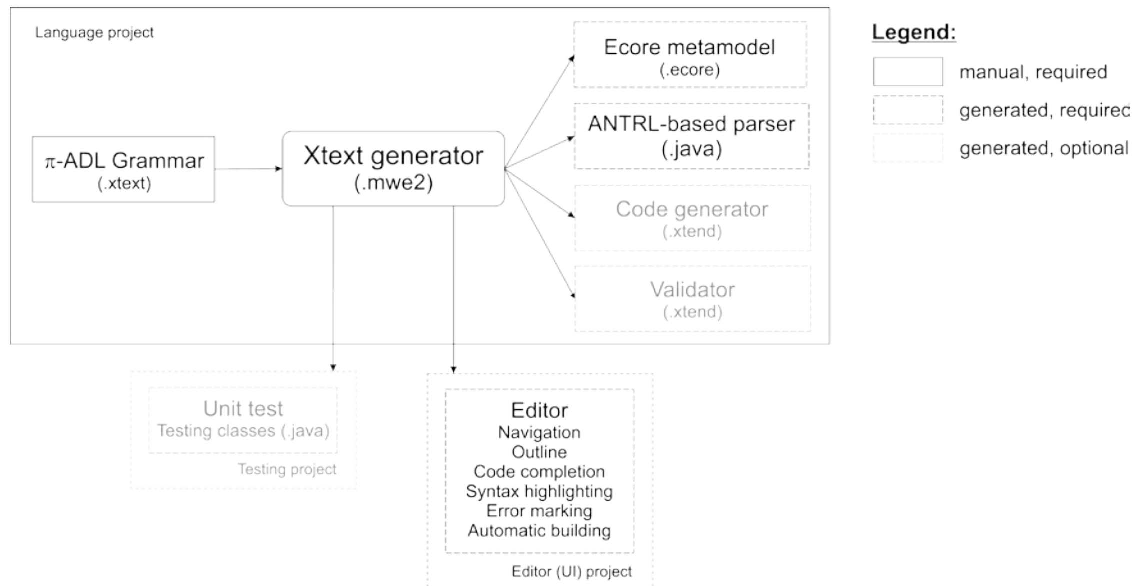
Figure B-2 – UML Class Diagram representing part of the part of the π-ADL metamodel



B.2.2 Automatic generation of the π-ADL infrastructure

In order to automatically generate the π-ADL language infrastructure, Xtext uses a script written in the Modeling Workflow Engine (MWE2) DSL to configure the generation of its artifacts. This script (also automatically created by Xtext) is used by the framework to derive a specification from the π-ADL grammar compatible with the ANTRL parser generator [5], which is used for generating the parser of the language. As depicted in Figure B-3, Xtext generates the following artifacts upon running the script: (i) a Java implementation of a *parser*, which is automatically generated by ANTRL and is responsible for the syntactic analysis of the textual description; (ii) an Ecore-based *metamodel* defining the abstract entities of the language and the relationships among them; (iii) a class for implementing the Go source *code generator*; and (iv) an Eclipse-based *code editor* for assisting the textual description of a software architecture in the π-ADL language. In addition, Xtext creates an *abstract syntax tree* (AST) from the parsed π-ADL textual model and it generates the respective Java classes to persist such an AST.

Figure B-3 – Artifacts generated by Xtext from the π-ADL grammar specification



Although the Java programming language can be used for customizing the generated artifacts, Xtext fosters the use of Xtend [6], a fully Java-interoperable programming language featuring a more compact, easier to use syntax, as well as advanced features such as type inference and lambda expressions. As the AST model needs to be continuously traversed, Xtend provides useful mechanisms for straightforwardly doing this while being easy to use and allowing for a better readable code. Moreover, as Xtend programs are compiled to plain Java code, they can access all of the libraries available in Java, thus allowing these languages to coexist seamlessly.

B.2.3 Validations

Once an architecture description is checked as correct by the parser, it might still have errors as its overall correctness cannot always be determined during the parsing procedure, thereby requiring a *semantic analysis* to be performed on the model. For this purpose, the Xtext framework provides means of constructing *validators*, i.e., classes that contain methods (validation rules) implementing additional constraint checks over the abstract elements of the current model. Despite Xtext provides some default validators (e.g., for checking that two entities have the same name), it also enables developers to implement custom validators by extending the base validation classes that come with the framework.

Xtext performs validation by invoking each Xtend method annotated with the `@Check` directive and passing all instances that have a compatible runtime type to each method. In the body of such methods, the semantic checks are implemented for the element passed as parameter. If a semantic check fails, trigger warning or error

messages appear in the textual editor while making the architecture description. Table B-I summarizes the main error/warning conditions checked by the implemented validation methods.

Table B-I – Error/warning conditions checked by the implemented validation methods

Target element	Warning error/condition	Severity
Architecture description	Duplicate names of components, connectors, and architectures within the architecture description	Error
Architectural element	Architectural element has no declared connections	Warning
Architectural element	Architectural element has no specified protocol	Warning
Architectural element	Inconsistencies between connection declaration and protocol specification (e.g., in terms of data types and/or directions)	Error
Architectural element	Protocol has undeclared connection	Error
Architectural element	Behavior of architectural element is empty	Warning
Type declaration	Name of the declared type starts lower cased	Warning
Architectural element	Inconsistencies between a prefix within a behavior and a connection declaration (e.g., in terms of data types and/or directions)	Error
Variable assignment	Target variable within assignment is undeclared	Error
Explicit projection	Tuple to be projected is undeclared	Error
Function call	Function call refers to an undeclared function	Error
Function declaration	Function is declared with return type and there is no return statement in its body	Warning
Connector	Connector behavior is unobservable	Warning
Choice behavior	Alternative branch in a choice behavior contains no recursion or inaction instruction	Error
Architecture	Duplicate names of architectural element instances	Error
Architecture	Instance refers to an undeclared architectural element	Error
Architecture	Absence of unifications between architectural elements	Error
Architecture	Connection elements in unification are both component or connector instances	Error
Architecture	Unification is not from an output connection to an input connection	Error
Architecture	Unification does not comprise input/output connections of the same type	Error
Architecture	Unification refers to an undeclared connection	Error

B.2.4 Interpreting expressions

In order to describe the behavior of components and connectors, a software architect can make use of *expressions* that are similar to the ones used in programming languages, e.g., logical, relational, equality, and arithmetic expressions. Determining the data types handled by such expressions is important for ensuring that an expression value sent via a connection is the one expected, i.e., its type is equal to the type specified when declaring such a connection. However, type checking cannot be performed during parsing (thus requiring a semantic analysis) and expressions are resolved at runtime, i.e., their value is calculated while they are described. As Xtext is mainly concerned with syntactic analysis and it does not support expression resolution, an *interpreter* and a *validator* were developed in Xtend for handling expressions in π-ADL architecture descriptions.

As the first step, a *type provider* implemented as an Xtend class was developed aiming to provide the type of a given expression. This class contains a set of `typeOf` methods that return the type of a given expression received as input. Four basic situations are possible when determining the type of an expression:

(i) *When the type of the expression does not depend upon the types of its sub-expressions.* Negation, logical, equality, and relational expressions will always return a Boolean value whereas atomic values have their types directly determined by their literals.

(ii) *When the type of expression depends upon the operation and its operands.* In this case, other situations are possible:
   - the division operation will always return a real value, regardless the operands;
   - the multiplication, modulus, and minus operations will return a real value if one of the operands is a real value or an integer value otherwise;
   - the sum operation will return a string value if one of the operands is a string value, a real value if one of the operands is a real value, or an integer value otherwise.

(iii) *When the expression is an assignment expression*. The type of the expression is determined by the declared type of the variable to which the value will be assigned.

(iv) *When the expression is a function call.* The type of the expression is determined by the return type specified when declaring the function.

The expression interpreter of π-ADL was implemented as another auxiliary Xtend class. This interpreter contains a method called `interpret` that receives an expression as input and determines its type by using the `typeOf` methods implemented

in the type provider class. After identifying the atomic values of the operands and converting them to conventional Java primitive types, the interpretation method performs the respective operations given by the operands, similarly to what is done when evaluating an expression by using a typical programming language.

At last, the expression validator of π-ADL was implemented as another Xtend class in order to perform the semantic type checking for expressions. The methods of this class are annotated with the `@Check` directive and then called at runtime in conjunction with the validation methods that check the conditions shown in Table B-I. Table B-II summarizes the main error conditions checked by the implemented expression validation methods. All of these methods trigger error messages when their respective semantic checks fail.

Table B-II – Error conditions checked by the implemented expression validation methods

Target expression	**Error condition**
Negation expression	Operand is not a Boolean value
Logical expression	Both operands are not Boolean values
Equality expression	Both operands are not Boolean values
Relational expression	Both operands are not of the same type or they are Boolean values
Multiplication, division, and modulus operations	Both operands are not numeric values
Minus operation	Both operands are not numeric values
Sum operation	Both operands are not neither numeric nor string values
Variable assignment	The type of the value to be assigned is not equal to the type of the variable (defined when declaring it)
Conditional prefix	The type of the guard is not Boolean
Conditional statement	The type of the guard is not Boolean
While loop	The type of the condition is not Boolean
For loop	The type of the stop condition is not Boolean

B.2.5 Features of the π-ADL textual editor

These are some useful features provided by Xtext to the generated π-ADL textual editor:

– *error and warning alerts* while describing the architecture, thus enabling architects to early detect and fix errors and potential problems on the architecture description as well as allowing for saving time and mental effort to correct these errors;

- *syntax highlighting*, which allows for making distinction between keywords of the language (reserved words) from identifiers allowed for use;
- *auto formatting*, accessed with the *Ctrl* + *Shift* + F keyboard shortcut;
- *content assist* (accessed with the *Ctrl* + *Space bar* keyboard shortcut), which provides suggestions on how to complete a given statement/expression based on the syntactic rules; and
- *automatic build on save*, which allows automatically generating code from the architecture description when it is saved in the language editor.

B.3 References

[1] Eclipse Platform. Available at: http://www.eclipse.org/

[2] Eclipse Modeling Project (EMF). Available at: http://www.eclipse.org/modeling/emf/

[3] Xtext. Available at: https://eclipse.org/Xtext/

[4] ISO/IEC 14977:1996(E). ISO/IEC International Standard for Information Technology – Syntactic metalanguage, Extended BNF. Geneva, Switzerland: ISO/IEC, 1996.

[5] ANTLR. Available at: http://www.antlr.org/

[6] Xtend. Available at: https://www.eclipse.org/xtend/

Appendix C – DynBLTL notation

This appendix describes the concrete textual syntax of the DynBLTL notation by using the Extended Backus-Naur Form (EBNF) meta-language [1], a notation for formally describing the context-free grammar of a language, i.e., its syntax. Section A.1 presents the notation elements used hereinafter whereas Section A.2 presents the production rules forming the DynBLTL grammar.

C.1 Grammar notation

The EBNF meta-language consists of *terminal symbols*, which are a sequence of one or more characters forming an irreducible element of the language, and non-terminal *production rules* governing how a particular syntactic element can be legally formed in terms of terminal symbols. Syntactic elements have names that are used in production rules and they are distinguished from names and reserved words (keywords) in the language. Furthermore, the EBNF meta-language uses a set of *meta-symbols* summarized in Table A-I.

Table A-I – EBNF meta-symbols

Meta-symbol	**Usage**
Right arrow (→)	Definition of production rule: $A \rightarrow B$ is read as *A is defined as B*
Pipe symbol (|)	Alternative choice between elements in production rule
Brackets ([and])	Optional occurrence of element in production rule
Asterisk character (*)	Multiple occurrences of element in production rule
Plus character (+)	At least one occurrence of element in production rule
Ellipses (…)	Character range
Parentheses	Element grouping

In the EBNF specification of the DynBLTL grammar, reserved words and terminal symbols of the language are expressed in a typewriter font whereas names of production rules are typed in the italic form. It is important to highlight that names of production rules and attributes in this specification seek to be representative and self-explanatory.

C.2 DynBLTL production rules

Node → *Identifier*

Connection → *Identifier*.*Identifier*

Function → *Identifier*([*Value* (, *Value*)*])

Value →
Value ArithmeticOperator Value | -*Value* | *Connection* | *Function* | *Node* | *Literal*

Predicate → *Value ComparisonOperator Value* | *Value*

ArithmeticOperator → + | - | / | *

ComparisonOperator → = | != | < | <= | > | >=

Bound → *RealLiteral* `time units` | *IntegerLiteral* `steps`

Property →
`exists` *Identifier* : *Function Property*
| `count` *Identifier* : *Function Property*
| `in` *Bound Property*
| *Property* `until` *Bound Property*
| `isTrue` *Property*
| `not` *Property*
| *Property* and *Property*
| *Property* or *Property*
| *Predicate*

Literal → *IntegerLiteral* | *RealLiteral* | *BooleanLiteral* | *StringLiteral*

IntegerLiteral → *Number*

RealLiteral → *RealNumber*

BooleanLiteral → `true` | `false`

StringLiteral → *String*

Number → (0...9)+

RealNumber → *Number*.*Number*

Identifier → (a...z | A...Z | _) (a...z | A...Z | _ | 0...9)*

String → « any ASCII character »

C.3 References

[1] ISO/IEC 14977:1996(E). ISO/IEC International Standard for Information Technology – Syntactic metalanguage, Extended BNF. Geneva, Switzerland: ISO/IEC, 1996.

Appendix D – List of Publications

D.1 Publications resulted from this work

Everton Cavalcante, Jean Quilbeuf, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, Axel Legay. Statistical model checking of dynamic software architectures. In: Bedir Tekinerdogan, Uwe Zdun (eds.) 10th European Conference on Software Architecture (ECSA 2016), Copenhagen, Denmark. Lecture Notes in Computer Science. Switzerland: Springer International Publishing, 2016 – *CORE Classification (2014)*[20]*: A*

Jean Quilbeuf, **Everton Cavalcante**, Louis-Marie Traonouez, Flavio Oquendo, Thais Batista, Axel Legay. A logic for the statistical model checking of dynamic software architectures. In: Tiziana Margaria, Bernhard Steffen (eds.) Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016), Corfu, Greece. Lecture Notes in Computer Science. Switzerland: Springer International Publishing, 2016 – *Stratum in Qualis-CC (2012)*[21]*: B4*

Everton Cavalcante, Thais Batista, Flavio Oquendo. Supporting dynamic software architectures: From architectural description to implementation. Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA 2015), Montréal, QC, Canada. Washington, DC, USA: IEEE Computer Society, 2015, pp. 31–40 – *CORE Classification (2014): A*

Everton Cavalcante, Flavio Oquendo, Thais Batista. Architecture-based code generation: From π-ADL architecture descriptions to implementations in the Go language. In: Paris Avgeriou, Uwe Zdun (eds.) Proceedings of the 8th European Conference on Software Architecture (ECSA 2014), Vienna, Austria. Lecture Notes in Computer Science, vol. 8627. Switzerland: Springer International Publishing, 2014, pp. 130–145 – *CORE Classification (2014): A*

D.2 Correlated publications

Eduardo Silva, **Everton Cavalcante**, Thais Batista, Flavio Oquendo. Bridging missions and architecture in software-intensive systems-of-systems. Proceedings of the 21st International Conference on Engineering of Complex Computer Systems (ICECCS

[20] CORE Classification obtained from the CORE Conference Portal: http://portal.core.edu.au/conf-ranks/

[21] Qualis-CC Conference Rank retrieved from the Brazilian Coordination for the Improvement of Higher Education Personnel (CAPES) portal: http://goo.gl/nE6olj

2016), Dubai, United Arab Emirates. Washington, DC, USA: IEEE Computer Society, 2016 – *CORE Classification (2014): A; Stratum in Qualis-CC (2012): B1*

Ana Luisa Medeiros, **Everton Cavalcante**, Thais Batista, Eduardo Silva. ArchSPL-MDD: An ADL-based model-driven strategy for automatic variability management. Proceedings of the IX Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2015), Belo Horizonte, MG, Brazil. Washington, DC, USA: IEEE Computer Society, 2015, pp. 120–129 – *Stratum in Qualis-CC (2012): B4*

Porfirio Gomes, **Everton Cavalcante**, Pedro Maia, Thais Batista, Kamilla Oliveira. A systematic mapping on discovery and composition mechanisms for systems-of-systems. Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2015), Funchal, Madeira, Portugal. Washington, DC, USA: IEEE Computer Society, 2015, pp. 191–198 – *Stratum in Qualis-CC (2012): B1*

Everton Cavalcante, Thais Batista, Nelly Bencomo, Pete Sawyer. Revisiting goal-oriented models for self-aware systems-of-systems. Proceedings of the 12th IEEE International Conference on Autonomic Computing (ICAC 2015), Grenoble, France. Washington, DC, USA: IEEE Computer Society, 2015, pp. 231–234 – *Stratum in Qualis-CC (2012): A2*

Everton Cavalcante. On the architecture-driven development of software-intensive systems-of-systems. Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), Florence, Italy, vol. 2. Washington, DC, USA: IEEE, 2015, pp. 899–902 – *CORE Classification (2014): A*; Stratum in Qualis-CC (2012): A1*

Milena Guessi, **Everton Cavalcante**, Lucas B. R. Oliveira. Characterizing architecture description languages for software-intensive systems-of-systems. Proceedings of the 3rd International Workshop on Software Engineering for Systems-of-Systems (SESoS 2015), Florence, Italy. Washington, DC, USA: IEEE, 2015, pp. 12–18

Eduardo Silva, Thais Batista, **Everton Cavalcante**. A mission-oriented tool for system-of-systems modeling. Proceedings of the 3rd International Workshop on Software Engineering for Systems-of-Systems (SESoS 2015), Florence, Italy. Washington, DC, USA: IEEE, 2015, pp. 31–36

Marcelo Benites Gonçalves, **Everton Cavalcante**, Thais Batista, Flavio Oquendo, Elisa Yumi Nakagawa. Towards a conceptual model for software-intensive system-of-systems. Proceedings of the 2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2014), San Diego, CA, USA. Washington, DC, USA: IEEE, 2014, pp. 1605–1610 – *CORE Classification (2014): B; Stratum in Qualis-CC (2012): B2*

Pedro Maia, **Everton Cavalcante**, Porfirio Gomes, Thais Batista, Flavia C. Delicato, Paulo F. Pires. On the development of systems-of-systems based on the Internet of Things: A systematic mapping. In: 2nd International Workshop on Software

Engineering for Systems-of-Systems (SESoS 2014), Vienna, Austria. Proceedings of the 2014 European Conference on Software Architecture Workshops. New York, NY, USA: ACM, 2014

Eduardo Silva, **Everton Cavalcante**, Thais Batista, Flavio Oquendo, Flavia C. Delicato, Paulo F. Pires. On the characterization of missions of systems-of-systems. In: 2nd International Workshop on Software Engineering for Systems-of-Systems (SESoS 2014), Vienna, Austria. Proceedings of the 2014 European Conference on Software Architecture Workshops. New York, NY, USA: ACM, 2014

Jérémy Buisson, **Everton Cavalcante**, Fabien Dagnat, Elena Leroux, Sébastien Martinez. Coqcots & Pycots: Non-stopping components for safe dynamic reconfiguration. Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2014), Marcq-en-Baroeul, France. New York, NY, USA: ACM, 2014, pp. 85-90 – *CORE Classification (2014): B; Stratum in Qualis-CC (2012): B1*

Everton Cavalcante, Ana Luisa Medeiros, Thais Batista. Describing cloud applications architectures. Proceedings of the 7th European Conference on Software Architecture (ECSA 2013), Montpellier, France. In: Khalil Drira (ed.) Lecture Notes in Computer Science, vol. 7957. Germany: Springer Berlin Heidelberg, 2013, pp. 320–323 – *CORE Classification (2014): A*

Eduardo Silva, Ana Luisa Medeiros, **Everton Cavalcante**, Thais Batista. A lightweight language for software product lines architecture description. Proceedings of the 7th European Conference on Software Architecture (ECSA 2013), Montpellier, France. In: Khalil Drira (ed.) Lecture Notes in Computer Science, vol. 7957. Germany: Springer Berlin Heidelberg, 2013, pp. 114–121 – *CORE Classification (2014): A*

Flavia C. Delicato, Paulo F. Pires, Thais Batista, **Everton Cavalcante**, Bruno Costa, Thomaz Barros. Towards an IoT ecosystem. Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems (SESoS 2013), Montpellier, France. New York, NY, USA: ACM, 2013, pp. 25–28.