



HAL
open science

A model-driven framework development methodology for robotic systems

Arunkumar Ramaswamy

► **To cite this version:**

Arunkumar Ramaswamy. A model-driven framework development methodology for robotic systems. Robotics [cs.RO]. Université Paris Saclay (COMUE), 2017. English. NNT : 2017SACL011 . tel-01680004

HAL Id: tel-01680004

<https://pastel.hal.science/tel-01680004>

Submitted on 10 Jan 2018

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.

A Model-Driven Framework Development Methodology for Robotic Systems

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'École nationale supérieure de techniques avancées

École doctorale n°573 Interfaces: approches interdisciplinaires,
fondements, applications et innovation
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, le 05/09/2017, par

M. Arunkumar Ramaswamy

Composition du Jury:

M. Christian Schlegel Professeur, Université de Ulm, Germany	Président
M. Davide Brugali Professeur Associé, Université de Bergame, Italy	Rapporteur
Mme. Thouraya Bouabana-Tebibel Professeur, Ecole Nationale Supérieure d'Informatique, Algeria	Rapporteur
M. Javier Ibanez-Guzman Chargé de recherche, Renault S.A.S, France	Examineur
M. Bruno Monsuez Professeur, ENSTA ParisTech, France	Directeur de thèse
Mme. Adriana Tapus Professeur, ENSTA ParisTech, France	Directeur de thèse

A Model-Driven Framework Development Methodology for Robotic Systems

by

Arunkumar Ramaswamy

Submitted to the Department of Computer Science and System Engineering
on September, 2017, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Most innovative applications having robotic capabilities like self-driving cars are developed from scratch with little reuse of design or code artifacts from previous similar projects. As a result, work at times is duplicated adding time and economic costs. To address this, standardization, benchmarking, and formalization activities in robotics are being undertaken by many technical working groups and independent agencies such as, IEEE, ISO, and OMG. Absence of integrated tools is the real barrier that exists between early adopters of such efforts and early majority of research and industrial community. In addition, Robotic systems are becoming more safety critical systems as they are deployed in unstructured human-centered environments. These software intensive systems are composed of distributed, heterogeneous software components interacting in a highly dynamic, uncertain environment. However, no significant systematic software development process is followed in robotics research. This is a real barrier for system level performance analysis and reasoning, which are in turn required for scalable benchmarking methods and reusing existing software.

The process of developing robotic software frameworks and tools for designing robotic architectures is expensive both in terms of time and effort, and absence of systematic approach may result in ad hoc designs that are not flexible and reusable. Therefore, within the context of architecture design, component development, and their support tools, a coherent practice is required for developing architectural frameworks. Making architecture meta-framework a point of conformance opens new possibilities for interoperability and knowledge sharing in the architecture and framework communities. We tried to make a step in this direction by proposing a common model and by providing a systematic methodological approach that helps in specifying different aspects of software architecture development and their interplay in a framework. As a part of the methodology, we also propose a solution space modeling language for design space exploration by modeling the functional as well as non-functional properties of the components.

Résumé :

La plupart des applications robotiques, telles que les véhicules autonomes, sont développées à partir d'une page blanche avec quelques rares réutilisations de conceptions ou de codes issus d'anciens projets équivalents. Pour répondre à ce problème, les activités de standardisation, d'évaluation, et de formalisation en robotique sont actuellement prises en charge par de nombreux groupes de travail, ainsi que par des agences indépendantes telles que l'IEEE, l'ISO, et l'OMG. L'absence d'outils intégrés rend difficile la communication entre les organismes menant cet effort, et la majorité des chercheurs et industriels. Qui plus est, les systèmes robotiques deviennent de plus en plus critiques, dans la mesure où ils sont déployés dans des environnements peu structurés, et centrés sur l'humain. Ces systèmes à fort contenu logiciel qui utilisent des composants distribués et hétérogènes interagissent dans un environnement dynamique, et incertain. Cependant, aucun processus de développement de logiciel n'est systématiquement suivi dans le cadre de la recherche en robotique, ce qui complique la réflexion au niveau système, ainsi que l'évaluation complète des performances d'un système robotique. Or, il s'agit là d'étapes indispensables pour la mise en place de méthodes d'évaluation extensibles, ainsi que pour permettre la réutilisation de composants logiciels pré-existants.

Le développement de structures logicielles et d'outils de conception d'architectures, orientés pour la robotique, coûte cher en termes de temps et d'effort, et l'absence d'une approche systématique pourrait conduire à la production de conceptions adhoc, peu flexibles et peu réutilisables. De ce fait, dans le cadre de la conception d'architectures, et du développement de composants, et de la mise en place d'outil correspondant, une pratique cohérente est obligatoire pour le développement de structures architecturales. Faire de la meta-structure de l'architecture un point de convergence offre de nouvelles possibilités en termes d'interopérabilité, et de partage de la connaissance, au sein des communautés dédiées à la mise en place d'architectures et de structures. Nous suivons cette direction, en proposant un modèle commun, et en fournissant une approche méthodologique systématique aidant à spécifier les différents aspects du développement d'architectures logicielles, et leurs relations au sein d'une structure partagée. Dans le cadre de cette méthodologie, nous proposons également un langage de modélisation de l'espace de solution pour l'exploration de l'espace de conception, par la modélisation des propriétés fonctionnelles et non-fonctionnelles des composants.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisors Prof. Bruno Monsuez and Prof. Adriana Tapus for their continuous support during my research, for their patience, motivation, enthusiasm, and immense knowledge. Their invaluable help with constructive comments and suggestions throughout my research and thesis writing have contributed to the success of this work. I could not have imagined having better advisers and mentors for my study. I would also like to thank Prof. Adriana Tapus for motivating and inspiring me to organize workshops that helped me become established as part of the research community. The workshops at 'IEEE Systems Man and Cybernetics Conference (SMC 2014)' and 'Robotics: Science and Systems (RSS 2015)' would not have happened without her encouragement and support.

I gratefully acknowledge the funding received towards my research from the Vede-com Institute. I am truly indebted to Dr. Ibanez-Guzman Javier who has been always played a key role in encouraging and mentoring me. I am fortunate to continue my work with him.

I would also like to thank Mr. Karthi Balasubramanian, Assistant Professor, Amrita University for nurturing my interest in robotics during my undergraduate studies. I also convey my deepest gratitude to my parents for their whole hearted cooperation. Most importantly, I wish to thank my loving and supportive wife, Sreelakshmi for providing me with unfailing support, prayers, and continuous encouragement.

I also extend my sincere gratitude to all persons who were in one way or other involved directly or indirectly with the project. Above all I am always thankful to God almighty for giving me good health and mental attitude for completing the thesis.

Contents

1	Introduction	1
1.1	Software Engineering in Robotics	3
1.1.1	Model-Driven Software Engineering	5
1.1.2	MDSE Approaches in Robotics	6
1.2	Brief Overview on Robotic Architectures	14
1.2.1	Architecture Modeling in Robotics	15
1.3	Thesis Context	17
1.4	Thesis Contents and Contributions	18
I	Methodology	20
2	SafeRobots Methodology	21
2.1	Introduction	21
2.1.1	Motivating Example	22
2.2	Conceptual Paradigms in SafeRobots Framework	23
2.2.1	Component-Based Software Engineering	24
2.2.2	Model-Driven Engineering	24
2.2.3	Knowledge-Based Engineering	25
2.3	Developmental Phases in SafeRobots Framework	26
2.4	Overview	27
2.4.1	Knowledge Space	29
2.4.2	Solution Space	30
2.4.3	Solution space to Operational space transformation	30
2.4.4	Operational Space	33

2.5	Related Works	33
2.6	Conclusion and Contributions	34
3	Specification of Non-Functional Properties	35
3.1	Introduction	35
3.2	Non-Functional Properties and Quality of Service	37
3.3	Example of Relevance to Human-Machine Systems	38
3.4	Non-Functional Properties (NFP) Metamodel	40
3.5	Tool Support for NFP specification	42
3.6	Example of NFP specification in an Assistive Lane Keeping System	43
3.7	Related Works	44
3.8	Conclusion	46
3.9	My Contributions	46
4	Solution Space Modeling	47
4.1	Introduction	47
4.2	Motivational Experiment	49
4.3	Solution Space Model	51
4.3.1	Functional Model	53
4.4	Solution Space Model for Lidar Based Vehicle Tracking System	55
4.5	Solution to Operational Space Transformation	60
4.5.1	Transformation Process	61
4.5.2	MDP Model Analysis	62
4.6	Operational Models	67
4.7	Related works	68
4.8	Conclusion	69
4.9	My Contributions	70
5	Architecture Modeling in Operational Space	71
5.1	Introduction	71
5.2	Software Architecture	72
5.2.1	The ISO standard	74
5.2.2	Architecture Framework and Architecture Description	75

5.2.3	Model Kinds in Robotics	76
5.3	Architecture Modeling and Analysis Language	77
5.3.1	AMAL Core Elements	78
5.3.2	Open Semantics Framework	80
5.3.3	Viewpoints and Views	81
5.3.4	Constraint Specification	83
5.3.5	Framework Specification Templates	83
5.3.6	AMAL Specification Formalism	87
5.4	Case Study on an Example Framework	89
5.4.1	TrackX - Framework specification	91
5.4.2	Structural viewpoint specification	93
5.4.3	Coordination viewpoint specification	94
5.4.4	Component-Port-Connector view specification	95
5.4.5	State transition view specification	95
5.5	Related Works	96
5.6	Conclusion	98
5.7	My Contributions	98

II Framework Implementation and Extended Applications 100

6	Framework Implementation	101
6.1	Introduction	101
6.2	Eclipse Modeling Framework	102
6.3	Framework Implementation Process	104
6.4	Conclusion and Contributions	108
7	Extended Applications	109
7.1	Introduction	109
7.2	Existing Framework Development Methods	109
7.3	Case Study 1: Mobile Robot Navigation	111
7.3.1	Requirement Modeling	112
7.3.2	Solution Space Modeling	113

7.3.3	Discussion	118
7.4	Case Study 2: Framework based on Cognitive Architecture	120
7.4.1	Overview on Human-Machine Systems	121
7.4.2	Application: Lane keeping and lane changing assistance system	122
7.4.3	Harel Statecharts	126
7.4.4	Relation between ACT-R Model and Statechart Model	127
7.4.5	Service Oriented Architecture	128
7.4.6	Perception Model	129
7.4.7	Communication Patterns	129
7.4.8	Discussion	131
7.5	Conclusion	136
III	Conclusion	137
8	Conclusion and Future Research Directions	139
8.1	Summary and Contributions	139
8.2	Future Directions	142
8.2.1	Search-Based Software Engineering	142
8.2.2	Non-Functional Property Composition	143
8.2.3	Runtime Models	143
IV	Annexes	144
A	Eclipse Modeling Framework and Supporting Plugins	145
A.1	Eclipse Modeling Framework	145
A.1.1	Metamodeling using Ecore	145
A.1.2	Graphical Modeling Workbench	147
A.1.3	Model Transformation Engines and Generators	150
	Curriculum Vitae	173

List of Figures

1-1	Deviation from linear problem solving caused by wicked problems [1]	4
1-2	A comparison of vertical and horizontal separation of concerns	11
1-3	Architecture Modeling Spectrum in Robotics	16
1-4	Relationship between the concept of Architecture and Framework . .	17
2-1	Informal Model for a mapping system	22
2-2	Developmental Phases and Conceptual Paradigms in SafeRobots Frame- work	24
2-3	SafeRobots Methodology	28
2-4	Transformation from solution space to operational space	31
2-5	Informal model of mobile robot functionality	32
3-1	Models and their Interactions in a typical Human-Machine System . .	39
3-2	NFP Metamodel	41
3-3	Common tooling support with additional editors for having multiple values for a property and a navigator for browsing history of values .	42
3-4	A high level NFP specification of Efficiency property of human driver and automation, and their interactions	44
4-1	Test Vehicle used for vehicle tracking experiment. Lidar and GPS are mounted on top of the vehicle and embedded computers are located in the trunk of the vehicle (see picture inset).	49
4-2	Relationship between the proposed models	52
4-3	Metamodel (left) and its graphical representation (right) of SSML. Dis- patch Gate, Port, and Connector represents the functional aspect and NFP and QoS Profile represents the non-functional aspect.	53

4-4	Metamodel for functional modeling (left) and graphical representation of dispatch gates (right).	54
4-5	Solution space model for tracking system. High level model is shown (top), zero delay gates (G1-5) are inserted to separate the computations. Connector C1 representing data preprocessing is modeled (right)	55
4-6	Connector C2 representing segmentation shown in Figure 4-5 is modeled in the figure	58
4-7	Vehicle Tracking Results: Segmented point clouds (left), detected vehicles (middle), tracked vehicles (right)	59
4-8	SSML Gates and corresponding MDP models	61
4-9	Design-time model analysis	63
4-10	Solution space model for tracking system (top) and the corresponding MDP model (down).	66
4-11	An example of annotated runtime MDP model (top) and the probability analysis of expected utility forecast (down)	67
5-1	Relationship between various concepts related to architecture and framework	74
5-2	Metamodel of Architecture Modeling and Analysis Language (AMAL)	78
5-3	Illustration of AMAL model instance with semantic information . . .	79
5-4	Simple publish-subscribe system represented in AMAL	80
5-5	Screenshot of the model creating methods implementing in Sirius. The figure shows a component creating on the left and a connection creation method on the right. Note that roles are also created while a connector is created.	85
5-6	AMAL Model Relationships	88
5-7	A prototype model of vehicle tracking system designed using the TrackX framework. Structural view (left) and Coordination view (right) of the system is shown	89
5-8	An illustration of the proposed framework topology	90
5-9	Framework model of a TrackX framework	91
5-10	Model relations between Communication Domain Model (CM), Perception Domain Model (PM), ROS Model (RM), and AMAL core elements	94

5-11	Visualization of state transition view and its corresponding AMAL model	96
6-1	Eclipse modeling framework and supporting tools	102
6-2	Structure of the framework with appropriate plugins	108
7-1	Illustration of features discussed in the respective case studies	111
7-2	Requirement model using KAOS notation	112
7-3	A excerpt form solution space model of perception and navigation do- mains	114
7-4	Ecore metamodel of subsumption architecture	115
7-5	Architecture Model in AMAL formalism	117
7-6	Ecore metamodel of HBBA architecture	118
7-7	Architecture for the case study	119
7-8	CPU usage for Subsumption and HBBA architecture	120
7-9	Architecture modeled in the proposed framework	123
7-10	Framework model of driver assistance system using AMAL formalism. Various domains and their relationships are shown.	124
7-11	Ecore diagram of ACT-R metamodel	125
7-12	Ecore diagram of Harel statechart	127
7-13	Ecore diagram of relational model \mathcal{R}_{SC}^{ACTR} between ACT-R model and statechart model	127
7-14	A reference model for service oriented architecture	128
7-15	Ecore diagram of service oriented architecture domain model	129
7-16	Ecore diagram of communication domain model	130
7-17	Control component in AMAL formalism	133
7-18	Production system in ACT-R driver model for lane changing behavior using statechart formalism	135
A-1	Overview of the EMF tool set	146
A-2	Graphical Workbench Specification using Sirius	148
A-3	Basic concept behind Model Transformation	150
A-4	Metamodel for ROS Middleware	152

A-5	Code Snippet showing a M2M ETL template for transforming a AMAL compliant model to ROS Middleware-based model	153
A-6	Code Snippet showing a M2T EGL template for generating C++ code from a ROS + Boost Statecharts model	154

List of Tables

1.1	Feature Comparison of MDSE Approaches in Robotics	10
4.1	Solution comparison for segmentation w.r.t application: Ground truth generation and Autonomous Driving	58
7.1	Metamodel relations between Communication Domain Model (CM), Perception Domain Model (PM), Statechart Model (SM), ROS Model (RM), NFP Model (NFP), SOA Model (SOA), Structural View (SV) and Behavioral View (BV)	132
7.2	An example for estimation of quality parameters while composing multiple domains	136

Abbreviations

AADL Architecture Analysis and Design Language.

ADL Architecture Description Language.

AET Average Execution Time.

AMAL Architecture Modeling and Analysis Language.

AQL Acceleo Query Language.

CBSE Component Based Software Engineering.

CIM Computation Independent Model.

CPC Component-Port-Connector.

DSL Domain Specific Language.

DSM Domain-Specific Model.

EEF Extended Editing Framework.

EGL Epsilon Generation Language.

EMF Eclipse Modeling Framework.

ET Execution Time.

ETL Epsilon Transformation Language.

EVL Epsilon Validation Language.

FAA Federal Aviation Administration.

GMF Graphical Modeling Language.

GRL Goal-oriented Requirement Language.

HBBA Hybrid Behavior-based Architecture.

HRI Human Robot Interaction.

IDE Integrated Development Environment.

IEC International Electrotechnical Commission.

IEEE Institute of Electrical and Electronics Engineers.

ISO International Organization for Standardization.

KAOS Knowledge Acquisition in Automated Specification.

KBE Knowledge Based Engineering.

M2M Model to Model.

M2T Model to Text.

MDA Model-driven Architecture.

MDE Model-driven Engineering.

MDP Markov Decision Process.

MDSD Model-Driven Software Development.

MoC Model of Computation.

NFP Non-Functional Property.

OCL Object Constraints Language.

OEM Original Equipment Manufacture.

OMG Object Management Group.

PCL Point Cloud Library.

PDM Platform Dependent Model.

PFH Point Feature Histogram.

PHD Probability Hypothesis Density.

PIM Platform Independent Model.

PSM Platform Specific Model.

QoS Quality of Service.

ROS Robot Operating System.

SAE Society of Automotive Engineers.

SDK Software Development Kit.

SOA Service Oriented Architecture.

SoC Separation of Concerns.

SSML Solution Space Modeling Language.

TCA Task Control Architecture.

UML Unified Modeling Language.

WCET Worst Case Execution Time.

XMI XML Metadata Interchange.

XML eXtensible Markup Language.

Chapter 1

Introduction

If I have seen further it is by
standing on the shoulders of giants

Sir Issac Newton

A robotic system is a software intensive system that is composed of distributed, heterogeneous software components interacting in a highly dynamic and uncertain environment. According to IEEE 1471-2000 standard, a software-intensive system is "any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole" [2]. However, a major part of the robotics research concentrates on the delivery of "proof of concepts" in order to substantiate the researcher's idea, for example, a robust path planning algorithm or a real-time collision detection system. Typically, these are developed from scratch or by using external code-based libraries [3]. Nevertheless, when such components are composed with other functional modules, the system does not exhibit the expected behavior. Hence, the approach in which different functionalities are integrated, commonly known as architecture of the system, determine the overall behavior of the robot. Therefore, the robot architecture influences the system emergent behavior to a large extent even when the behavior of individual functional components are known [4].

In order to address challenges posed by large scale integration efforts in robotics, many researchers have created a wide variety of frameworks [5]. These frameworks helped to manage complexity to some extent, and facilitated rapid prototyping of

software for experiments, resulting in the many robotic software systems that are currently being used in academics and industry. Each of these frameworks was designed for addressing a particular purpose, perhaps in response to perceived weaknesses of other available frameworks, or to place emphasis on aspects, which were seen as most important in their own development process. However, relying on a given library results in applications that cannot be easily portable.

Issac Newton once stated that "If I have seen further it is by standing on the shoulders of giants" and he acknowledges the process of discovering truth by building on previous discoveries. The statement is highly relevant in this current era of rapid technological progress, where most of the technical progress is built on the foundations of earlier innovations [6]. Most of the innovative robotic projects, e.g., in government funded projects, and systems, where robotics research plays a vital role, e.g., self-driving cars, are built almost from scratch every time with little reuse of design or code artifacts from previous similar projects. This is mainly due to the fact that software components developed are hard-bound to their operational context and thus they are least reusable. Component Based Software Engineering (CBSE) is proposed in Robotics to manage complexity, reusability, and portability. In CBSE, components are standalone systems having predefined interfaces, which perform a specific functionality. Typically, these components use a communication middleware to interact with each other and do not impose any specific architecture on the designer. However, such techniques did not solve the challenges raised by the robotics domain where the systems are deployed in uncertain real-world environment. The core problem is that when such components are merged and/or combined with other functional modules, the system does not always exhibit the expected behavior. The fundamental tenet in this process is to reuse existing works and to benchmark one's invention with others. Scenario and application based benchmarking techniques are often considered in robotics. However, such techniques are not scalable enough considering the open-ended, unstructured, and dynamic environments where robots are deployed. In addition, large application domains in robotics make it more complex. We believe that the replication and benchmarking of such complex systems should be performed at higher abstraction levels and with better formalized process and quantitative evaluation procedures. This is difficult with the absence of formal specification

and specialized tools.

1.1 Software Engineering in Robotics

Software plays a key role in robotics as it is the medium to embody intelligence in the machine [7]. Since robotics and their application are becoming more pervasive than ever, software in these systems is often required to (1) operate in distributed and embedded environment consisting of diverse computational capabilities, (2) interact in various communication paradigms, (3) adapt to changes in an indeterministic environment, and finally (4) should behave rationally. Despite advances in programming languages and supporting tools, code-centric development of such complex systems requires immense effort.

The fundamental tenet of Software Engineering (SE) is that the development and evolution of software systems can be facilitated by adopting a systematic, disciplined, quantifiable approach in each phases of a software application’s life span, from requirements analysis, system architecture definition, and component design to code implementation, testing, and deployment [7]. Despite this understanding, it is still common for an engineering team to develop the decision-making and control system of a robotics system from scratch, only to discover that it is too difficult to separate this software from the rest of the system and reuse it [8]. In the past, the robotics community has been shown reluctance in adopting modern software engineering methods for developing software architectures. This has led to the increased time-to-market and large system integration efforts when such systems are to be used in safety critical applications.

In the last two decades, the robotics research community has seen a large number of middlewares, code libraries, and component frameworks developed by different research laboratories and universities. They facilitate software development by providing infrastructure for communication (e.g., ROS [9]), real-time control (e.g., Orocos [10]), abstract access to sensors and actuators (e.g., Player [11]), algorithm reuse (e.g., OpenCV [12], PCL [13]), and simulation (e.g., Stage [11], Gazebo [14]). To a large extent, these frameworks have helped in rapid prototyping of individual functionalities, but system level analysis still remains an issue. Usually, the system

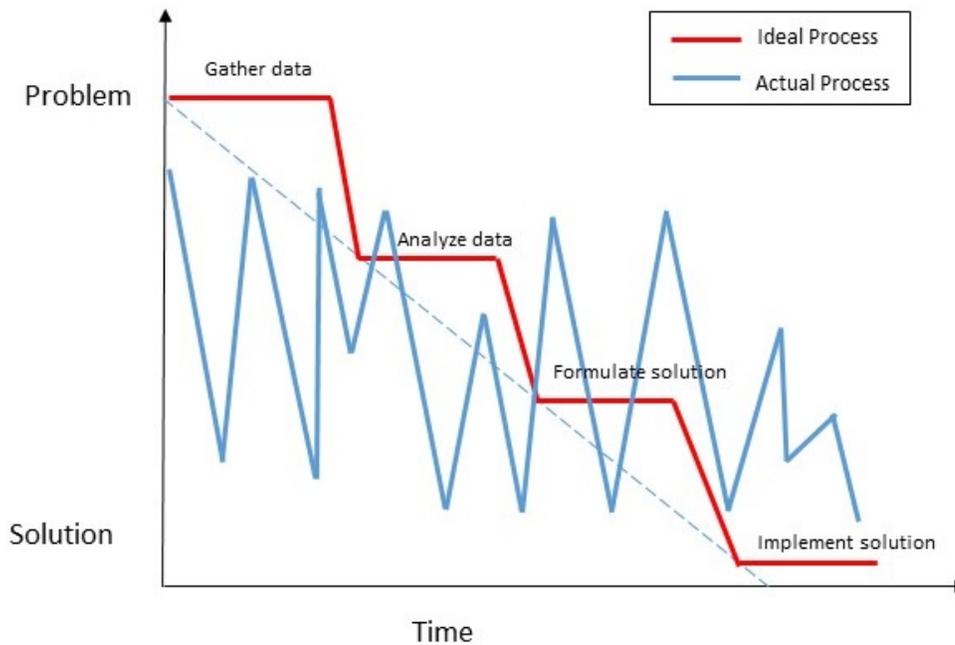


Figure 1-1: Deviation from linear problem solving caused by wicked problems [1]

level properties, such as response time, flexibility, and deployment have been realized as accidental outcomes, rather than a design decision. It is high time that roboticists transform themselves as *system thinkers* in addition to being domain experts.

Motivated by the positive results from the application of Model-driven Software Development (MDS) in other domains, such as automotive, avionics, etc., there is an encouraging trend in its adoption in the robotics domain [15]. MDS helps the domain experts shift their focus from implementation to the problem space. The robotic software architects are attracted by the fact that appropriately selecting the viewpoints and levels of abstraction, the system can be analyzed more efficiently. However, the model-driven work flow cannot directly be applied in the robotics domain. The unpredictability spans over various phases in software development - from requirement specification, system design, implementation, integration, and till it is deployed in real world scenarios. The system cannot be realized in an uni-directional process flow because the solution for a robotic problem cannot be finalized during the design time. It is because neither the problem space nor the target environment can be completely modeled as in embedded systems.

Realizing MDS vision requires addressing a variety of complex interrelated soft-

ware engineering problems that is further aggravated by the issues intrinsic to the robotics domain. The use of MDSD in software lifecycle is considered to be a 'wicked problem' [16]. A wicked problem has multiple dimensions that are related in complex ways and thus cannot be solved by cobbling solutions to the different problem dimensions. In many of the complex robotics systems, the development of software in a linear fashion is not practically possible. It is because the problem is not well understood until a solution is developed. The Figure 1-1, which is inspired from an article by Douglas C. Schmidt, illustrates several iterations between problem understanding and solution that exists during software design, implementation, and testing [1]. To cope with the problem of software complexity MDSD researchers need to develop technologies that developers can use to generate domain-specific software development methodology and associated tools. Such Integrated Development Environment (IDE) should consist of languages and supporting infrastructure that are tailored to the robotic applications. Developing such technologies requires codifying knowledge that reflects a deep understanding of the common and variable aspects of the gap bridging process. Such an understanding can be gained only through costly experimentation and systematic accumulation and examination of experience. Developing such technologies is thus a wicked problem [16].

1.1.1 Model-Driven Software Engineering

In Model-Driven Software Engineering (MDSE), modeling techniques are used to tame the complexity of bridging the gap between the problem domain and the software implementation domain [16]. Although component-based development and model-driven engineering address complexity management, the former adopts a bottom-up approach, while the latter is more top-down in nature [17]. In MDSE, the complexity of the software is managed by using the mechanism called 'separation of concerns (SoC)'. In MDSE approach, abstract models are gradually converted into concrete executable models by a series of systematic transformation processes. Models are designed based on meta-models and domain-specific language(DSLs).

The next section has two objectives:

1. To identify the current model-driven approaches in robotics.

2. Analyze how the identified approaches achieve general modeling related advantages and how effective they are in satisfying robot specific requirements.

We evaluate the identified approaches in robotics by addressing the following questions:

- How is domain knowledge modeled and how it is used in various phases of software development?
- What are the common Separation of Concerns (SoC) that are relevant in robotics and how these SoCs are used for analyzing the desirable properties and facilitate system level reasoning?
- How are the models used at run-time?
- How are the non-functional properties incorporated in the system?
- How are the robotic component specific requirements, such as composability, compositionality, variability, technology neutrality, addressed in these systems?

1.1.2 MDSE Approaches in Robotics

This section provides an overview of some of the MDSD approaches available in robotics. To the best of our knowledge, there are four model based approaches in robotics: BRICS [18], RobotML [19], SmartSoft [15], and V3CMM [20] approach. There are many component-based approaches, in which modeling is mainly used for generating basic skeleton codes rather than using models as a developmental artifact. These four model-based approaches are briefly described below:

BRICS Component Model

BRICS Component model [18] is built upon two complimentary paradigms - Model-driven approach and Separation of Concerns (SoC). The syntax of the model is represented by Component-Port-Connector (CPC) meta-model and their semantics is mapped to the 5Cs - Communication, Computation, Configuration, Coordination, and Composition. The components represent computation and can be hierarchically

composed to represent composite components. A composite component contains a coordinator who is in charge of starting and stopping the computational components. Port represent the type of communication, for example: dataflow, events, service calls, etc. and the connectors connect two compatible ports. The components are configured by using their visible properties, for example: maximum iterations for a planning algorithm. The compositional aspect concerns the interaction between the other 4 concerns. The BRICS approach is built using Eclipse and all the concepts are currently not integrated in the toolchain. The workflow can be roughly summarized as follows:

1. Define the structural architecture by using components, port, and connectors.
2. Define a coordinator for each composite component using state machines.
3. Perform a Model to Text (M2T) transformation to generate executable code (currently Orocos and ROS middleware are supported).

RobotML

RobotML is a DSL for designing, simulating, and deploying robotic applications. It is developed in the framework of the French research project PROTEUS [19]. The domain model consists of architecture, communication, behavior, and deployment metamodels. The architectural model defines the structural design using the CPC model. In addition, it also defines the environment, data types, robotic mission, and platform. The communication model associated with ports defines the type of communication - dataflow port or service port. The behavior model is defined using state machines. Specific activities are associated with states and transitions are mapped to specific algorithms. The deployment model specifies a set constructs that define the assignment of each component to a target robotic middleware or simulator. The workflow is described below:

1. Define the architecture using a component-port-connector diagram.
2. Define the communication policy between components by setting the port attributes.

3. Define the behavioral model of each component using state machines.
4. Create a deployment plan by allocating the components to a middleware or a simulator.
5. Execute Model to Text (M2T) transformation to generate the executable code.

SmartSoft

SmartSoft [15] employs a model-driven approach in creating component skeleton (called *component hull* in SmartSoft terminology) that mediates the external visible services of a component and internal component implementation. The skeleton provides links to four different artifacts: internal usercode, communication to external components, platform-independent concepts such as threads, synchronization, etc., and platform-specific middleware, and operating system.

The communication between external services (interfaces to other components) and internal visible access methods (interface to user code inside component) is based on a set of communication patterns. A set of seven communication patterns are identified that are relevant to robotics systems: `send`, `query`, `push newest`, `push time`, `event patterns`, `state`, and `wiring patterns`. The wiring pattern provides dynamic connection of components at run-time. These patterns provide required abstraction from implementation technologies with respect to the middleware systems. In order to promote loose coupling between components, the objects are transmitted by value and the data are marshaled into a platform-independent representation for transmission. The behavior of the component is specified by an automaton with generic states *Init*, *FatalError*, *Shutdown*, and *Alive*. The *Alive* state can be extended by the user-defined states. The life cycle of the component is managed using these pre-defined states including the fault detection. The workflow can be summarized as follows:

1. Create a Platform-Independent Model (PIM) of a component skeleton with stable interfaces to usercode, externally visible interfaces, and interfaces to SmartSoft framework. The component model contains explicit parameters and attributes, that finally need to be validated while generating platform-specific models (e.g., `wcet: 100 ms [requirement]`).

2. The platform specific information is then added to a PIM and component attributes are refined (e.g., wcet: 80 ms [estimation]).
3. In the deployment phase, a system is designed by wiring the components and system level parameters are extracted with the help of a Platform Description Model (PDM) (e.g., wcet: 85 ms [measurement]).
4. The timing parameters of system are exported to an external tool called Cheddar [21] to analyze for the timing analysis.
5. The PSI is generated from Platform-Specific Model (PSM) using Model to Text (M2T) transformation.

V³CMM

V³CMM component meta-model consists of three complementary views: structural, coordination, and algorithmic views [20]. The structural view describes the static structure of the components, coordination view describes the event-driven behavior of the components and the algorithmic view describes the algorithm executed by each component based on its current state. The structural view consists of component, ports, interfaces and their interconnections. The coordination model is defined using UML state machines, while algorithmic view consist of UML activity diagrams. The workflow is described below:

1. Define the common data types and interfaces.
2. Create the simple and complex component definitions.
3. Design the behavioral model of each component using the UML state machines.
4. Design the algorithmic models using the UML activity diagrams.
5. Link activities to state machines and state machines to components.
6. Execute Model to Model (M2M) transformation to generate UML models and M2T transformation to generate the executable code.

Feature	RobotML[19]	SmartSoft [15]	BCM [18]	V3CMM [20]
SoC	✓	✓	✓	✓
Composability	✗	✗	✗	✗
Compositionality	✗	✗	✗	✗
Static Variability	✓	✓	✓	✓
Dynamic Variability	✗	✓	✗	✗
Component Abstraction	✓	✓	✓	✓
Technology Neutrality	✓	✓	✓	✓
Knowledge Model	✓	✗	✗	✗
System Reasoning	✗	✓	✗	✗
Run-time Models	✗	✗	✗	✗
NFP Model	✗	✓	✗	✗

Table 1.1: Feature Comparison of MDSE Approaches in Robotics

Feature Analysis

An overview of features available in each approach is depicted in Table 1.1. We have identified these required features from our previous research work as detailed in [22]. A brief discussions on these features are provided in the following sections:

Separation of Concerns

In MDSD, the complexity of the software is managed ‘Separation of Concerns (SoC)’. Vertical SoC is built on multiple levels of abstraction. Model Driven Architecture (MDA), a model-driven architecture standardized by OMG specifies four abstraction layers - Computation-Independent Model (CIM), Platform-Independent Model (PIM), Platform-Specific Model (PSM), and Platform-Specific Implementation (PSI) [23]. Horizontal SoC manages complexity by providing different overlapping view-points of the model at the same abstraction level. All of the MDSD approaches in robotics primarily use only two vertical SoC - PIM and PSI. Horizontal SoC is seen only in one of the abstraction layer - PIM. We depict a comparison of SoCs for the four approaches in Figure 1-2.

Composability

A model is said to be composable if its core properties do not change upon integration. In other words, the composability of a model guarantees preservation of its

	RobotML	SmartSoft	BCM	V3CMM
CIM	X	X	X	X
PIM	Architecture, Behavior, Communication, Deployment	SmartComponent (abstract attributes), Communication (7 patterns), Deployment	Computation, Communication, Configuration, Coordination, Composition	Structural, Coordination, Algorithmic
PSM	X	SmartComponent (refined attributes)	X	UML Class Models
PSI	Executable Code (Orocos, Morse)	Executable Code (Corba, Ace)	Executable Code (ROS, Orocos)	Executable Code (Ada 2005)

Figure 1-2: A comparison of vertical and horizontal separation of concerns

properties across integration with other components. A highly composable model can freely move around in the design space and assemble itself to a more complex structure that is semantically correct. However, functional composability such as, semantic correctness and non-functional composability of the composed models are not addressed in any of the approaches. It is very important in robotics since the components are highly heterogeneous in nature in terms of semantics and necessary for providing unambiguous interfaces. For example, the authors of [24], in their proposal of standardization, have provided approximately 24 different ways of representing a geometric relation between rigid bodies and have proposed a DSL based on it [25].

Compositionality

Compositionality allows to deduce properties of a composite component models from its constituent components. It enables hierarchical composition of components and provides correctness of the structure. The reusability of a component is enhanced by providing compile-time guarantees by verifying that formal parameters and actual parameters match regardless of the software module's location. The behavior of the reused components can be predicted in the new configuration and the result of the composition can be analyzed for anomalies. Only syntactic correctness of the architecture is addressed in all the approaches. It is worth mentioning that the works associated with Ptolemy [26] framework provide formal proofs for correctness when heterogeneous actors with different models of computation are composed. The major hindrance in robotics is the lack of standards, however, the robotics domain task force

at OMG [27] and standard committee at IEEE Robotics and Automation society is in the process of standardization. The European project ‘RoSta’ provides standards and a reference architecture for service robots [28]. Most of the approaches previously analyzed does not provide support for the composability and compositionality properties. Smartsoft provides composability at the service level by explicating Quality of Service (QoS) parameters and required resources [29]. This is due to the fact that the component attributes are not explicitly modeled using formal methods.

Static and Dynamic Variability

Static variability is the configuration of the system and dynamic variability is related to the context dependent coordination of the components. Configuration defines which system can communicate with each other and coordination determines when such communication can occur. Configuration can be completely specified during design time while coordination is achieved by allowing variability during design time and run-time dynamic invocation. Static variability and limited context-dependent dynamic variability is provided by the analyzed approaches. SmartSoft models variation points, context, QoS, and Adaptation rules using the Variability Modeling Language (VML) to support run-time variability for service robotics [30]. BRICS uses feature models similar to the one used in Software Product Line (SPL) to specify, implement, and constraint resolution of variabilities and provide graphical models for selecting possible configuration during design time [31]. Recently the authors of [32] have proposed an approach for dynamic variability modeling and its exploitation at run-time for robotic systems.

Technology Neutrality

The component properties and specification should not depend on a specific technology. Software technology neutrality and hardware neutrality to some extent are achieved by many of the already available code-based frameworks, such as, ROS [9], Player project [11], etc. To a larger extent, all the MDSD approaches have achieved middleware independence by using various M2T approaches for generation of executable code.

Modeling Domain knowledge

A primary focus of MDS is the separation of domain knowledge and implementation details. Among the compared approaches, domain knowledge is explicitly modeled only in RobotML. In RobotML, the DSL is designed with the robot domain ontology as the backbone. In fact, the domain concepts that are required while designing the DSL is derived from the ontology. The ontology is used for two purposes - to normalize the robot domain concepts and to act as an inference mechanism during runtime [33]. However, it is not clear from the literature how the ontologies can effectively be used for modeling developmental and run-time models.

Round-tripping Problem

Round-tripping is a major concern of a model-based system, especially if it has multiple abstraction layers and different horizontal SoC. It is a major problem in analyzed approaches because SoC is applied only at the model level. Restricting SoC only to models and in addition only to single abstraction levels worsens the round-tripping effect and reduces the reusability. Providing SoC to models and code can support better traceability and system evolution. Aspect-oriented modeling and template-based techniques can be used to provide an integrated way of dealing with SoC. It will simplify the model development and transformation tasks [34] [35]. Approaches in RobotML and V3CMM provide only loose coupling among different viewpoints, for example, in V3CMM, the uni-directional relationship between structural, coordinational, and algorithmic views have to manually be corrected if there is change in one of the viewpoints.

System level reasoning

SmartSoft and RobotML provide support to some extent to reason about the timing analysis of the models at the system level. Properties such as WCET, Periods, etc, are provided as attributes to components. During the system deployment these properties are exported to an external tool ‘Cheddar’ for schedulability analysis [36]. Semantic reasoning is not yet realized in any approaches because of the lack of standards.

Run-time Models

In robotics, the adaptation of the robotic system to the dynamic environments are in the computational algorithms of the constituent systems. This severely limits the configuration space of such systems. Explicit modeling of the variabilities and variation points during the system design can help find the best possible solution during run-time and can lead to the use of framework supported adaptation mechanisms. Hence, in order to achieve run-time adaptation, explicit models of variation points, variabilities, context or environment and decision mechanisms should be supported by the framework. In [37] run-time models are used for simple scenarios. The authors in [30] have demonstrated how SmartSoft framework with support of VML can be used for run-time adaptation for service robots.

Non-Functional Properties

Non-Functional Properties (NFP) define how a functionality operates, for example, performance, availability, effectiveness, etc. The functional and non-functional attributes of the components are considered to make ‘who does, what, and when’ decisions depending on the operational context. However, non-functional properties are not given sufficient importance compared to that of the functional requirements during the developmental stages. SmartSoft uses NFPs as attributes to a component model and employs those attributes during the development process.

1.2 Brief Overview on Robotic Architectures

The initial work on robot architectures began with two extreme approaches - one is based on sense-plan-act paradigm [38], which is a deliberative approach and the other is based on a purely reactive Brooks’ subsumption architecture [39]. To take advantage of these two extreme approaches, a number of hybrid architectures were then proposed. A notable one is Gat’s three layer architecture that uses controller, sequencer, and deliberator layers to enable the robot to make high level plans and at the same time reactive to sudden events [40]. In NASREM architecture, the perceived information passes through several processing stages until a coherent view of the cur-

rent situation is obtained. A plan is then adopted and successively decomposed by different modules until the desired actions can be directly executed by the actuators [41]. The Task Control Architecture (TCA), developed by Simmons, provides a general framework for controlling distributed robot systems. TCA is essentially a high-level robot operating system with a set of commonly required mechanisms to support distributed communications, task decomposition, resource management, execution monitoring, and error recovery [42].

Majority of the robot architectures found in literature [43], in one form or another consists of the below three distinct characteristics:

- Hierarchical vs. Centralized architectures
- Behavioral vs. Functional systems.
- Reactive vs. Deliberative control

Cognitive architectures such as ACT-R [44] and ICARUS [45] use concepts from artificial intelligence and cognitive psychology to create and understand synthetic agent that support the same capabilities as human and thus makes the robots more pervasive in a social environment. These architectures can be broadly classified as conceptual architectures, though their depth of influence in robotic domain as a whole may be different. For example, the hybrid architectures concentrated mainly on how low level reactive behaviors can be coordinated using high level planning and decisional algorithms, while an expert system that models human driver is valid in very narrow contexts.

1.2.1 Architecture Modeling in Robotics

Typically software architectures are modeled as a collection of interacting components, where low level implementation details are hidden while expressing abstract high-level properties. An architecture model captures multiple abstraction levels or viewpoints that satisfy the requirements of different stakeholders. A hardware engineer would like to see the components allocated to a particular processor, while a system architect would be interested in component topology. A good architecture model facilitates decision making and acts as a mediator between requirements and

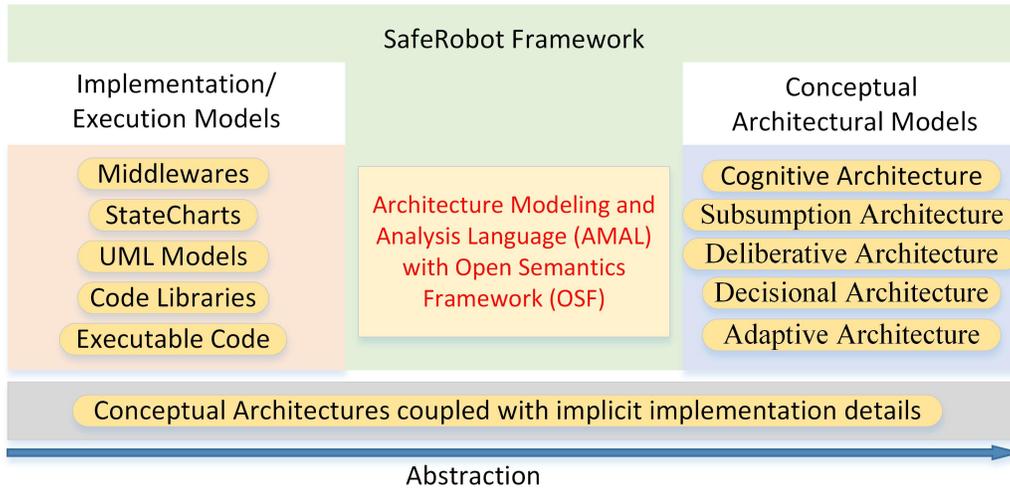


Figure 1-3: Architecture Modeling Spectrum in Robotics

final implementation. Specifically the architecture model plays a critical role in many aspects of software development life-cycle such as requirement specification, system design, implementation, reuse, maintenance, and run-time adaptation [46].

The architectures that concern the execution and implementation aspects lie in the lower end of the abstraction axis. Modeling languages such as UML [47], SysML [48], and Marte [49], model the system that is more closer to the software realizations. Their semantics are mostly implementation-specific and contain semantic models, such as communication patterns, model of computation (MoC). There is another category of architecture that consists of conceptual architectures that are tightly bound to specific implementation models. Architectures such as GenoM [50], and ACT-R [44] provide their software development kits (SDK) to design systems complying with those models. The main advantages of such models is that more stringent validation methods can be applied and can maintain traceability from domain concepts to its implementations. However it takes significant effort to port from one implementation technology to another.

We can view this as a spectrum of models on an abstraction axis as shown in Figure 1-3. One can notice the concentration of architectures on both ends of the spectrum and the significant gap in the middle. Our meta-framework approach, proposed in Chapter 5, tries to fill this gap by acting as a transition architecture model that bridges the gap between conceptual architectures and low level implementation architectures.

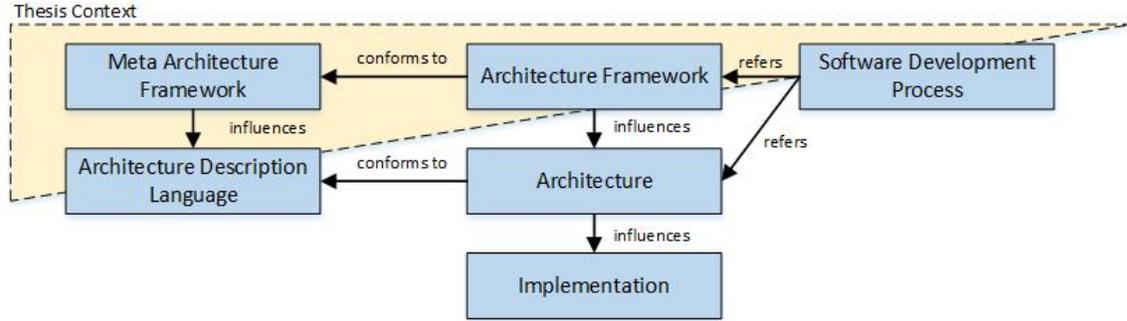


Figure 1-4: Relationship between the concept of Architecture and Framework

1.3 Thesis Context

An architecture framework is a collection of conventions, principles for the description of architectures established within a specific domain of application and/or community of stakeholders [51]. For example, NIST 4D/RCS is a robotic architecture standards framework used by US Department of Defense (DoD) for unmanned vehicle systems [52][53]. Architectures are based on well-defined semantics and conventions provided by the framework. Describing different architectures using the rules provided by the framework helps in standardization at the architecture level. Generally, the architectures are specified using Architecture Description Languages (ADLs). ADLs provides notations and concrete syntax for characterizing software architectures. Typically, the framework also provides tools for parsing, viewing, compiling, analyzing, or simulating architectures specified in their associated description language. The relationship between these concepts is illustrated in Figure 1-4. It also shows the context and influence of this thesis on the those areas.

Based on our comparative survey on existing model-driven frameworks in robotics and qualitative analysis of their features, we found that many of the domain-specific requirements such as architecture level analysis, system reasoning, non-functional property modeling, run-time models, component composition, etc., were addressed differently in these approaches. In addition, it is hard to find a single approach that has features according to one’s requirement. Systematic development process and detailed instructions for building such frameworks and supporting infrastructure have not been studied enough. In addition, there is no clear definition on how OMG’s Model Driven Architecture (MDA) could be realized in a framework.

The contributions from this thesis comprise conceptual methodology and a development approach that facilitates design, analysis, and deployment of framework for robotic systems. The methodology will be based on three well-established software engineering paradigms: Model-Driven Software Development (MDSD), Component-Based Software Engineering (CBSE), and Knowledge-Based Engineering (KBE). The main advantages of our proposed framework are:

1. Rapid development of custom framework for required set of domains.
2. Support for efficient integration of existing heterogeneous architecture styles that accelerates the complex system design.
3. Enables analysis, comparison, and benchmarking of various functional units and architectural paradigms.
4. Homogeneous development environment irrespective of framework or middle-ware and thus promoting faster adoption among software developers and system engineers.

1.4 Thesis Contents and Contributions

The remaining of the thesis is organized as follows:

Chapter 2 introduces the methodology of SafeRobots Framework. It discusses the different conceptual spaces in the framework and drives through the systematic developmental phases in the proposed methodology.

Chapter 3 discusses the concepts of Non-Functional Properties (NFP) and Quality of Services (QoS). The importance of modeling these properties in robotics especially in human-machine systems is explained in this chapter. A metamodel for modeling NFP is also introduced in this chapter.

Chapter 4 identifies several inherent problems in the software design stage of robotics software development. The chapter starts by discussing several challenges faced during design phase of robotic software development. In this direction, different problems

posed during the software development of a robotic software subsystem in an industrial setup are analysed. A domain-specific language that facilitates the design space exploration specific to the field of robotics is also introduced.

Chapter 5 investigates the possibility of a common model for framework specification so as to model heterogeneous architectural paradigms. It reviews and studies different robotic architectures from an abstraction and software engineering point of view. A domain-specific language in the operational space that integrates with our SafeRobots methodology is proposed in this chapter.

Chapter 6 details different tools used and the stages involved in implementing the framework by the developer of the tool. It provides a brief overview on Eclipse Modeling Framework and discusses the selected tools and approaches supporting the framework development.

Chapter 7 employs the proposed framework and methodology to specific applications in robotics. Two case studies are shown so as to demonstrate how a system can be developed and formally specified in our framework. The first case study uses a mobile robot mapping application and shows how two different architectures were developed from the same solution model and by semi-automated model transformations. The second case study is about an application in cognitive architecture domain. A lane keeping and lane changing assistance system is modeled using our framework.

Chapter 8 finally concludes the thesis and provides directions for future work.

Part I

Methodology

Chapter 2

SafeRobots Methodology

Methodology is intuition
reconstructed in tranquility

Paul Lazarsfeld

2.1 Introduction

A software development methodology is a framework that is used to structure, plan, and control the process of developing software for a system. This includes pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application [54]. Although effective techniques for developing safety-critical software are well established, for example, in avionics industry, these techniques are in general designed for projects with long timescale and high staffing levels. It can be unsuitable for use without adaptation in the field of innovative robotics research, where timescale is shorter and, human resource and financial investment are typically much lower [55]. Typically, in robotics domain, one has to deal with a wide variety of sensors and actuators with varying level of capabilities. Adding to this complexity of having heterogeneous hardware devices, robots has to deal with open-ended environment with limited resources [15]. There is a pressing need to tailor proven approaches of software engineering in other domains to the requirements of robotics. In this direction, there is a need for a methodology that do not constraint any specific framework or architecture. In this context, the

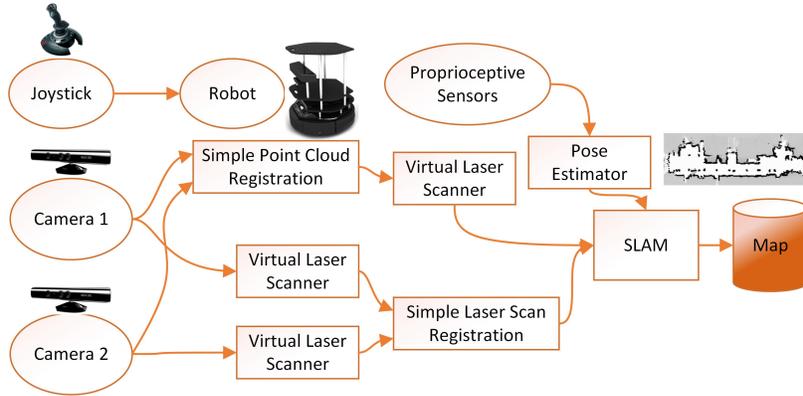


Figure 2-1: Informal Model for a mapping system

purpose of this chapter is:

- To identify the core software engineering methods and developmental phases involved, on which the proposed framework is developed.
- To propose a conceptual methodology for developing software with generic tools for robotics domain.
- To analyze different software developmental phases involved in our methodology.

2.1.1 Motivating Example

Consider the problem of designing a software architecture for a mobile robot that needs to create a map of an indoor environment. The mobile robot is equipped with two ‘Time of Flight’ (ToF) cameras positioned at right angles to each other with an overlapping ‘Field of View’ (FoV). The goal is to develop a SLAM¹ based mapping system using the point cloud data from the two ToF sensors. Assume that a hypothetical library is also given that provides software code implementing the SLAM algorithm. The library also contains a functions that can extract the points on a single plane (Virtual Laser Scanner) and for simple point cloud registration.

Since all the computational algorithms are provided as implemented software components, the primary task of the system developer is to design a data flow structure

¹In robotics, Simultaneous Localization and Mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent’s location within it [56].

connecting these components. Using pragmatic knowledge, this can be achieved by connecting each camera to the virtual laser scanner function and simply summing up the two planar point clouds and then fed it to the SLAM algorithm along with the pose estimates. Another alternative is to combine the raw point cloud from the two cameras applying the provided registration function and then convert the resulting point cloud to the planar laser data. Although, the two approaches seems to be very similar in functionality (creating a map), the system level emergent non-functional properties are different. The non-functional properties of each software component, such as execution time, confidence level, cannot be predicted at this stage because we do not have any data regarding the hardware platform in which it will be executed or whether all these computational components will be executed on a single platform or in a distributed fashion, and there is no information regarding the network bandwidth, latencies, etc. This will influence the rate at which the SLAM algorithm will be executed and it will affect the resolution and confidence level of the resulting map.

An informal representation of these two solutions is illustrated in Figure 2-1. Nonetheless, it is clear that we have used our pragmatism and past experiences to arrive at this mental solution model. This integration method is highly non-deterministic, since the system integration for the same problem with the same ‘ingredients’ provides different results because the knowledge used is in the designer’s mental form. The disparity will be more evident for complex situations or when the system designer’s knowledge is different from the algorithm developer’s knowledge. Since such knowledge is not explicitly encoded anywhere in the design specification, this can hinder the system evolution for large scale projects.

2.2 Conceptual Paradigms in SafeRobots Framework

Building on the identified problems and domain-specific requirements, we propose a conceptual methodological framework called ‘Self Adaptive Framework for Robotic Systems (SafeRobots)’. This section provides an overview of the three orthogonal, yet complementary software engineering methods, on which the proposed methodology is developed (see Figure 2-2).

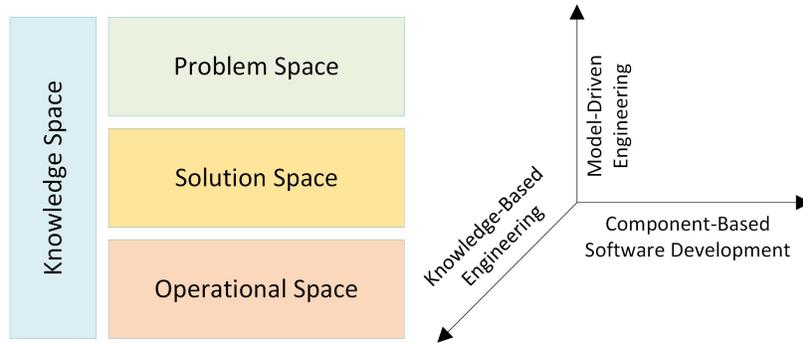


Figure 2-2: Developmental Phases and Conceptual Paradigms in SafeRobots Framework

2.2.1 Component-Based Software Engineering

In Component-Based Software Engineering (CBSE) approach, a software is developed using off-the-shelf components and custom-built components. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only, and can be deployed independently and is subject to composition by third parties [57]. In our previous research paper [58], we have identified specific requirements of components, that are more relevant in robotics: composability, compositionality, static and dynamic variability, component abstraction, and technology neutrality. The main goal of CBSE is to manage complexity and foster software reuse by employing the *divide and rule* strategy. In order to promote reuse, the focus should be on the design and implementation of individual components and on the techniques that maximize component interoperability and flexibility. A more detailed discussion on CBSE in robotics can be found in [59].

2.2.2 Model-Driven Engineering

In Model-Driven Engineering (MDE), modeling techniques are used to tame the complexity of bridging the gap between the problem domain and the software implementation domain [16]. Although component-based development and model-driven engineering address complexity management, the former adopts a bottom-up approach, while the latter is more top-down in nature [17]. In MDE, the complexity of the software is managed by using the mechanism called ‘separation of concerns (SoC)’. Conceptually, it can be classified into vertical and horizontal SoCs. Vertical SoCs

are built on multiple levels of abstraction. Model-Driven Architecture (MDA) [23], a trademarked name for MDE by Object Management Group (OMG), specifies four abstraction models - Computation Independent Model (CIM), Platform Independent Model (PIM), Platform-Specific Model (PSM), and Platform-Specific Implementation (PSI). The computation independent model focuses on the environment of the system, the requirements of the system without any structural or processing details. The platform independent model focuses on the operation of the system while hiding the details regarding the platform, middleware, etc. The platform-specific model is generated using PIM by adding platform specific details. The platform-specific implementation is then generated by using appropriate Model to Text (M2T) transformations. Horizontal SoCs manage complexity by providing different overlapping viewpoints of the model at the same abstraction level. In [60], the authors suggest four horizontal SoCs for large scale distributed systems: coordination, configuration, computation, and communication. In summary, the main challenges in MDE are designing modeling languages at right abstraction level, modeling with multiple overlapping viewpoints, model manipulation, and maintaining viewpoints' consistency [16].

2.2.3 Knowledge-Based Engineering

Knowledge-Based Engineering (KBE) has the potential to change automated reasoning, methodologies, and life cycle of software artifacts. In order to achieve that, domain concepts should be represented at various granularity levels in accordance with multiple abstraction layers. For example, in the case of self-driving cars, for system level reasoning, the knowledge required is regarding the environment, the socio-legal constraints, traffic rules, etc., and for platform independent layers, the knowledge required is about algorithmic parameters, structural configuration, semantics of data, etc. In general, information and knowledge can be represented in symbolic and non-symbolic representation. In the non-symbolic approach, a promising work is proposed by Gardenfors that represents conceptual spaces in the context of cognitive science [61]. Conceptual spaces are simple geometric representations based on quality dimensions, designed for modeling and managing concepts. The quality dimensions are used as framework to assign properties to objects and to specify relationships between

them. The reasoning is mainly based on the distances between points in this conceptual space. In general, the smaller the distance between two objects, the more similar they are. For example, the distance from the concept ‘canny edge detector’ to the concept ‘grayscale image’ is smaller than to the RGB image’. However formalizing this approach with respect to software development is beyond the scope of this thesis.

2.3 Developmental Phases in SafeRobots Framework

The different developmental phases, modeling languages, and models involved in the proposed framework are illustrated in Figure 2-3. The software development process in robotics can be conceptually divided into three spaces: problem space, solution space, and operational space, where each space is supported by the knowledge space. The following developmental phases are involved in the proposed framework:

General Domain Knowledge Modeling

The domain knowledge modeling in this phase is independent of the problem specification or application constraints. The domain concepts can be formally modeled using ontologies, Domain-Specific Languages (DSLs), Knowledge graphs, etc. The models at this level capture the robotic domain specific concepts, meta-data about the computational algorithms and standard interfaces, their structural dependencies, etc. The domain knowledge complements the various application specific development process by providing a knowledge base for abstract concepts such as image, point clouds, links, joints, platform, etc.

Problem Modeling

In this phase of problem modeling, the application-specific requirements, context or environment, are formally modeled. The functional and non-functional requirements are explicitly captured in the problem model.

Problem-Specific Knowledge Modeling

Problem-specific knowledge modeling or solution space modeling is designed with the help of functional requirements from the problem model as *constraints* applied to the domain model. In other words, a solution model captures multiple solutions for the given problem by considering only the functional requirements and given domain knowledge base. The strategy postpones the decisions on non-functional requirements

at a later stage, since such properties can be estimated only when platform-specific decisions are made.

Problem-Specific Concrete Modeling

Problem-specific concrete model, also called operational model, is a reduced subset of a solution model. The reduction is carried out by mapping non-functional requirements of a problem model, and system level and components' non-functional properties such as timings, confidence, resolution levels, etc. If there are multiple solutions that satisfy the constraints, they are modeled as variation points that can be resolved after applying more concrete constraints or during runtime depending on the context.

Executable Code Generation

In this final process, an executable code is generated depending on the platform and the specified middlewares. Several Model to Text (M2T) techniques are available for applying this transformation (see Appendix A for details).

2.4 Overview

SafeRobots is an architecture agnostic framework development methodology, tailored to the needs of robotic domain. It is supported by extensible tools and provides a set of basic metamodels and domain-specific languages, and a formal process for the design and development of software for robotic systems. The methodology comprises four conceptual spaces as shown in Figure 2-3: knowledge space, problem space, solution space, and operational space. The knowledge space is common with the other three spaces and provides abstract knowledge on robotic domain-specific concepts. The models at this level capture the robotic domain-specific concepts, meta-data about the computational algorithms, and standard interfaces, their data structures, etc. The domain knowledge complements the various application-specific development process by providing a knowledge base for abstract concepts such as image, point clouds, links, joints, platform, etc.

The three spaces - problem space, solution space, and operational space are application-specific modeling phases. However, models in solution space can be reused across different applications and hence the solution space models can be seen as do-

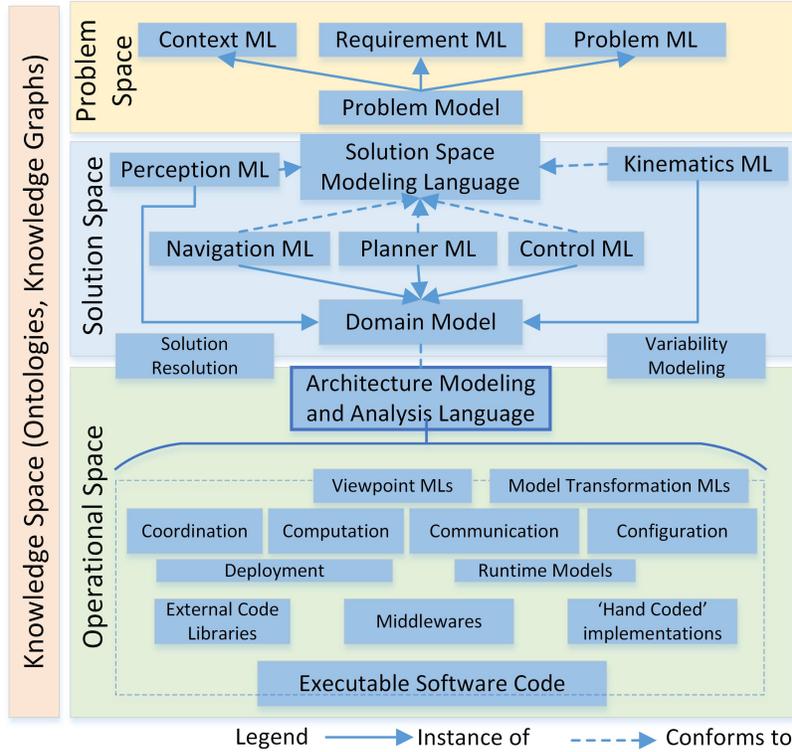


Figure 2-3: SafeRobots Methodology

main models as well. The requirements, goals, and contexts models in problem space capture functional and non-functional requirements. Different solutions for solving specific functionalities are modeled in the solution space. In the solution space, only functional requirements are considered while non-functional requirements are specified as properties in this space. In other words, the models in the solution space satisfy only the functional requirements, however, these models comprise non-functional specifications as well, but it may not comply with the requirements. For example, a non-functional requirement may specify certain confidence level for the data, but the solution space models contains the information on how aforementioned confidence can be evaluated from it models, and it may not necessarily satisfy the confidence requirement. The models in the operational space are derived from the solution space model by applying the non-functional constraints. Concrete architectural models are specified in this operational space. If there are multiple models that satisfy the non-functional requirements, there are considered as variation points, which are dynamically resolved during run-time.

The conceptual spaces in the our approach are hierarchically arranged in such a way that the lower layer uses the knowledge gained in the upper layer. For example, models in solution space comply to the functional requirements from problem space and use the domain knowledge from knowledge space. Non-functional requirements from problem space is applied to solution space models to create operational space models. The models in all these spaces is supported by domain concepts in knowledge space as well. The following section provides a brief description of the three conceptual spaces and modeling - requirement modeling in the problem space, solution space modeling in the solution space, and architecture modeling in the operational space.

2.4.1 Knowledge Space

The knowledge space provides abstract knowledge on robotic domain-specific concepts. The domain knowledge modeling in knowledge space is independent of the problem or application constraints. Our methodology does not recommend any particular way of modeling these concepts. However, some parts of our development uses ontologies to some extend. Ontologies are mainly used in Artificial Intelligence (AI) and Semantic Web communities to represent knowledge as a set of concepts and relationships of a domain [62]. Ontology can be thought of as a knowledge representation approach that represents key concepts with their properties, relationships, rules, and constraints. The IEEE Robotics and Automation Society is sponsoring the working group Ontologies for Robotics and Automation to streamline development trends, work on the harmonization of taxonomies and ontologies, along with the standardization of terms, interfaces, and technologies [63]. The goal is to provide a standard ontology and associated methodology for knowledge representation and reasoning in robotics and automation, together with the representation of concepts in an initial set of application domains [64]. Going one step further, software components and various implementations of algorithms can be represented in the form of knowledge graphs. Some examples include the selection and configuration of sensors, robust software components that implement specific functionalities, such as object detectors, classifiers, etc. For such representations, conceptual spaces introduced by Gardenfors is very promising by presenting a framework for representing information on the conceptual level that provide us with a natural way of representing similar-

ties [65]. The authors of [66] have used such a framework to enable the systematic integration of different knowledge representations required in robotics. Recently, the authors of [67] have extended this framework to represent robotics perception architectures and showed how to model and store ready-to-use perception graphs and to efficiently select the most appropriate perception graph at run time [68].

2.4.2 Solution Space

The solution space for implementing a functionality in robotics domain is large. A complex robotic system cannot be finalized during design-time [15]. Hence, there is a need to support reasoning during development stages by the system designer to mitigate the uncertainty. Nowadays, there is a trend to delay decisions to handle uncertainty at runtime instead of handling it during development-time [69]. Nevertheless, there is a need to define the solutions at a higher abstraction level at which relevant properties and characteristics are expressed. Solution space modeling is performed with the help of functional requirements from the problem model as constraints applied to the concepts modeled in knowledge space. In other words, a solution space model captures multiple solutions for the given problem by considering only the functional requirements and given domain knowledge base modeled in problem independent knowledge space. The strategy is to postpone the decisions on non-functional requirements at a later stage, since such properties can be estimated only when platform-specific decisions are made. The solution model also helps in - system level reasoning, making tradeoffs, documenting decisions, and comparing them, and for formally proving and validating the final implementation.

2.4.3 Solution space to Operational space transformation

Once the solution space model for a given problem is modeled, the next step is to transform it into an operational model. It is to be noted that the solution space model only satisfies the functional requirements and the non-functional requirements are enforced only while it is transformed to the operational model. It is common that there may be more than one solution that satisfies the non-functional requirements, in which case, the best possible solution can only be resolved during runtime. In addition

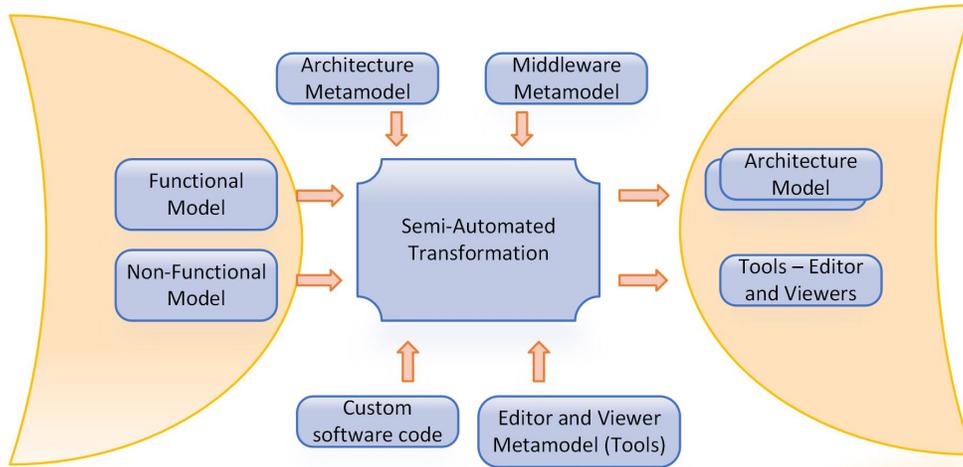


Figure 2-4: Transformation from solution space to operational space

to these solution subsets, there may be some solutions that partially satisfies or may depend on the context variables that can be estimated in real execution scenarios. Hence, the transformation has two stages - design time and runtime.

A reduced form of the transformation process is illustrated in Figure 2-4. The core idea is a systematic semi-automated process to use models from problem space, solution space, and knowledge space to guide the system designer to make principled decisions and trade offs during the system development process. The high level models are functional models, non-functional models, architecture metamodels (conventions and patterns), and middleware metamodels. In the transformation process, the user includes the custom software code, maps the software components to the corresponding architecture models, and generate a system architecture. It is to be noted that models are not only primary artifacts for development, they are primary means by which the different users such as developers and system architects interact and share.

The functional models consist of operations that have to be carried out in order to accomplish a specific functionality. Figure 2-5a shows an informal model of a functional model for navigating a mobile robot. The dark and white circle represents the starting and ending points of the directed graph representation. The model shows that for navigating a mobile robot, four different operations: **Move Forward**,

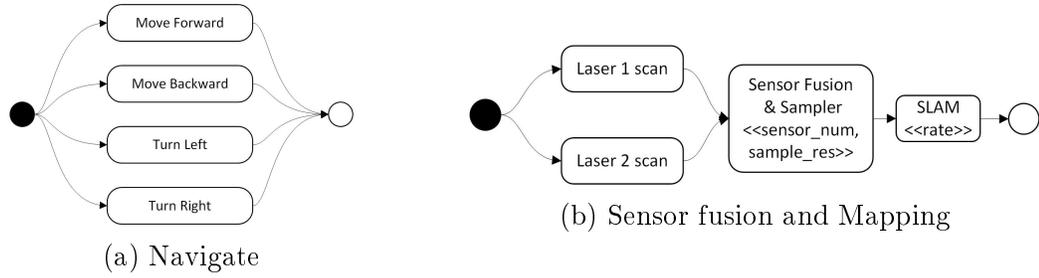


Figure 2-5: Informal model of mobile robot functionality

Move Backward, Turn Left, and Turn Right can be performed. The model also assumes that these operations are independent in the sense that performing one operation do not constraint the other three operations to be activated or deactivated. In other words, in order to have the `Navigate` skill, the four sub-skills are required. The functional model can also model more complex functionalities, for example, Figure 2-5b shows a functional model for a sensor fusion and mapping operation for a mobile robot. The `Sensor Fusion and Sampler` block receives data from two Laser acquisition blocks and sends data to the `SLAM` block. In a certain sense, the model can be viewed as a data flow diagram, but conceptually it shows the relationship between functions in order to achieve more complex functions. Non-Functional model are also linked with these functions that can be in the form of attributes as shown in Figure 2-5b. Non-Functional model captures the non-functional properties of a functionality. Non-functional properties (NFP) define how a functionality operates, for example, performance, availability, effectiveness, etc. For example, the NFP model of a mapping functionality may require that the resolution of the map should be less than a specific value, `res_map < 3cm`. The core idea is to apply constraints from problem space to the models (functional and non-functional) in solution space together with domain concepts from knowledge space to guide the user in the process of system design.

The vision of our methodology is to use the functional and non-functional models to generate the system architecture in a tool-assisted way with core software engineering methods and systematic developmental phases. The semi-automatic transformation process also uses metamodels of architecture, communication middleware, supporting tools, and custom software components to generate the architecture of the system. If the system architect needs to change the middleware, he/she has to

change the middleware metamodel and perform the transformation process to generate the architecture in a different middleware. The process is the same if he/she has to change the architecture, the corresponding metamodel has to be replaced. The architecture metamodel consists of abstract information and constraints related to the architecture.

2.4.4 Operational Space

In the operational space, a concrete architecture of the system is modeled that satisfies the functional and non-functional requirements. Robotics domain is highly heterogeneous with domains ranging from conceptual domain such as perception, planning, control, decision making to computational domain consisting of discrete, continuous; software domain consisting of communication middlewares, operating systems, etc. Hence, the meta-framework architecture should be extensible in order to incorporate different domain models. The composed domain models should be semantically compatible. For example, assume a model incorporates concepts from two domains a and b . In domain a , the modeling element connector represents a computation process, and in domain b , the connector represents an instantaneous transition between two states. These two domains are semantically incompatible unless the conflict between them is resolved, say by assigning the computational process to the component.

The model relationships enable integration of various domains so as to build more complex systems. The support for architectural views manages complexity by promoting separation of concerns (SoC). Our Open Semantic Framework proposed in Chapter 5 helps to modify and extend the semantics to incorporate different domain concepts and to capture the relationships and identify the conflicting domain semantics.

2.5 Related Works

We have discussed different MDSD approaches in Robotics in Chapter 1. In addition, there are many approaches that target specific area of software development life cycle such as, task coordination [70], deployment [71], and integration [72] of software for robotic systems. Recently, SmartMDSD Toolchain v2 was released that provides an

Integrated Development Environment (IDE) for robotics software development that combines a set of DSLs and tools that guides experts and stakeholders through a formalized development process [73]. The authors have used an approach of using models to cover and support the whole life-cycle of robotic components and their evolution from design phase to run-time usage. As an extension of BRICS approach, HyperFlex toolchain proposes a development process that defines how reference architectures can be exploited for building robotic applications. It provides a set of principles and tools that support the software engineers and the robotic developers in the task of designing, reusing, and composing robotic systems [74].

2.6 Conclusion and Contributions

In this chapter, we proposed SafeRobots methodology based on three orthogonal software development approaches: Component-Based Software Engineering, Model-Driven Software Development, and Knowledge-Based Engineering. The developmental phases in this framework can be classified into four phases: General Domain Knowledge Modeling, Problem Specific Solution Modeling, Problem Specific Concrete Modeling, and Executable Code Generation. Functional and Non-Functional Meta-model are proposed to model functional and non-functional aspects of the system. Once the solution space of a problem is modeled, an appropriate solution or a set of solutions is selected depending on the application context and quality requirements. After this resolution, a more concrete solution (system architecture) is modeled in the operational space. A semi-automated transformation will facilitate this process of mapping to the operational space models. A detailed analysis of solution space modeling and framework design in operational space is provided in Chapter 4 and 5, respectively.

Chapter 3

Specification of Non-Functional Properties

Quality is never an accident. It is always the result of intelligent effort.

John Ruskin

3.1 Introduction

A major part of the robotics research concentrates on the delivery of ‘proof of concepts’ in order to substantiate the researchers’ ideas, for example, a robust path planning algorithm or a real-time collision detection system. Typically, these are developed from scratch or by using external code-based libraries. Nevertheless, when such components are merged and/or combined with other functional modules, the system does not always exhibit the expected behavior. This has led to the increased time-to-market and large system integration efforts when such systems are to be used in safety critical applications. The main issue that arises when such heterogeneous components are combined, is in ensuring reliability and safety at the system level. In cyber-physical systems execution correctness encompass both functional and non-functional properties. However, only functional semantics is treated in traditional programming languages. Software modules built on such programming languages becomes unreliable unless it is handled at the component level in the hierarchy.

Adaptable components should provide a way to incorporate Non-Functional Properties (NFP) along with the data that they consume and produce. System engineers take into account the non-functional properties of components to characterize the system architecture. It necessitates a mechanism in which the NFPs propagate in component interactions so that it will assist system engineers to provide a better architecture for target applications. The confidence level of the components does not depend only on the efficiency of the implemented algorithm but also on the context in which it is executed such as resource availability, external disturbance, etc. For example, if a component is implementing localization using adaptable particle filters, the quality and response time depends on the number of particles used, which in turn depends on available memory during execution time, the state uncertainty level [2], etc. The component developer cannot test these properties unless it considers a Worst Case Execution Time (WCET) anticipating the available resources and the component should be incorporated in the anticipated platform. Nonetheless, that will fail the very purpose of reusable component for a reconfiguring system.

Specification and utilization of NFPs are crucial for building quality robotic software architectures. While functional properties decide what the system is supposed to do, the non-functional properties specify the conditions and how a functionality operates. Description of non-functional properties are essential for design analysis and refinement decisions, as well as automatic generation of code and test cases [75]. Usually, these properties are not explicitly dealt during robotic system development because of their complexity, informal way of their requirement specification, high abstraction level, as well as due to the limited support of languages, methodologies, and tools [76]. Hence, it is necessary that a framework developer for robotics domain should provide a facility for the system designer to specify NFP in a more consistent manner. Accordingly, the purpose of this chapter is:

- To introduce the importance of NFPs in Robotics and Human-Machine Interaction systems.
- To structure the specification of NFPs using a common metamodel.
- To provide tooling support for NFP specification that integrates with the SafeRobots methodology.

3.2 Non-Functional Properties and Quality of Service

Non-functional properties define how a functionality operates, for example, performance, availability, effectiveness, etc. QoS is the aptitude of a service for providing a quality level to the different demands of the clients [77]. There is no general consensus in the community about the concepts of NFP and QoS. In [78], the authors define Non-Functional Properties (NFP) as the element that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability; in short, a requirement that specifies constraints on a functional requirement. In the early phases of system development, Non-Functional Properties are considered as requirements that impose certain constraints to be met. Subsequently, during the development process, these properties will become integral part of the system. Non-Functional Requirements are not implemented in the same way as functional ones. NFPs are seen as *by products* when a functionality is implemented. In software engineering terms, usability, integrity, efficiency, correctness, reliability, maintainability, testability, flexibility, reusability, portability, interoperability, performance, etc. constitute NFPs [76]. At the same time, what determines QoS is highly domain specific. For example, throughput and bandwidth determines QoS for a network; performance, fault-tolerance, availability, and security for an embedded system; personality, empathy, engagement, and adaptation for social robots [79]; resource utilization, run-time adaptation for service robots [30]. In [80], the authors identified some non-functional properties (in the form of metrics) for tasks in navigation, perception, management, and manipulation for Human-Robot Interaction. For example, effectiveness of a navigation task can be measured by:

- Percentage of navigation tasks successfully completed
- Coverage of area
- Deviation from planned route
- Obstacles that were successfully avoided
- Obstacles that were not avoided, but could be overcome

Efficiency can be measured by time to complete the task, operator time for task, average time for obstacle extraction, etc.

The management of NFPs in component models is one of the main challenges in the component-based software engineering community. The starting point in their management, namely their specification in a context of component models is not addressed in a systematic way [81]. The purpose of NFPs is to provide additional information about the components, complementing the structural information that is provided by the component model. This additional information is intended to give a better insight in the behavior and capability of the component in terms of reliability, safety, security, maintainability, accuracy, compliance to a standard, resource consumption, and timing capabilities, among many others. In that sense, these properties bridge the gap between the knowledge of what a component does and its actual capabilities. As the system development progresses, the meaning of the NFPs typically changes from 'required' to 'exhibited'. In most cases, the value of the property also might change as knowledge about the system and its target platform changes. For example, a NFP requirement in the requirement specification phase might be considered as a NFP property in the design phase with an estimated value. As the system development progresses towards implementation, the value will be replaced by a measured value [81]. Hence, NFPs shall be allowed to assume multiple values that are valid depending on the context of existence.

3.3 Example of Relevance to Human-Machine Systems

In this section, we discuss why specifying NFP is important for robotic systems that involve human interaction. In a typical Human-Machine Interaction (HMI) system, a task is performed by cooperation of the human and the automation component. The system adopts a cognitive architecture to model human psychology and makes optimum decisions on dynamic task allocation between human and the machine counterpart depending on the context. However, such architectures do not define how these systems are implemented in a software. This is due to the gap in the abstraction axis

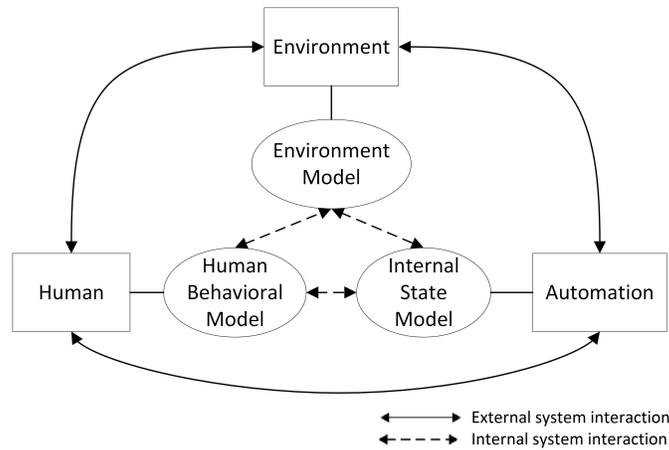


Figure 3-1: Models and their Interactions in a typical Human-Machine System

as detailed in Chapter 1.

In certain tasks, humans outperform automation while in some others, it is the opposite. However, in most of semi-autonomous systems involving humans, for example in cars, many functions that human performs are provided by the automation also. Although the software component and the human counterpart can perform the same functionality, it is the quality expected from the entire system and the context that determines which one should be activated or deactivated in compliance with the regulation authority such as FAA for civil aviation. A typical human-machine system maintains three kinds of models: (a) human model, (b) environment model, and (c) the internal state model of the system as shown in Figure 3-1. The functionality is achieved through the interaction of these three models. Most modeling languages provide support for the description of functional behavior, while the non-functional requirements are normally described using informal comments even though these properties play a vital role in determining the quality of the system. A decision making system estimates the quality provided by the human and that of the automation, and decides the authority for control. For example, attributes such as response time, accuracy, and reliability may comprise the quality of the software component, while alertness, skills, and expectation-intention correlation may decide the quality of the human.

Furthermore, selecting the services based on the quality have advantage not only between human and automation agents but also within the automation system itself.

Since the functionalities are viewed as services it is possible to compose basic low level services in different fashion to provide different functionalities. For example, in cars, voice recognition service can be used in an entertainment system as well as in a navigation system. However, the bottleneck is more related to the business models existing in industries that develop such systems. The standard approach for automobile OEMs is to develop systems by assembling components that have been completely or partly designed and developed by external vendors [82]. Because of the increasing complexity of automobile systems with large number of distributed features, such an approach will also lead to various compositional issues commonly known as *feature interactions*. Therefore, Service Oriented Architectures (SoA) are gradually being adopted in these systems where various functionalities are provided as services and the assembled components are seen as service providers. This software engineering paradigm has many advantages in Human-Machine collaborative work. Advances in human behavior research helps to model various human actions. These actions can be seen as services provided by human, for example, steering, braking, etc, applying acceleration by the driver can be viewed as services provided by the human driver. In some context, humans provide high quality services while in some others the machine counterpart does. For example, in the case of assisted parking, human steering control service is delegated to the machine still retaining the authority over acceleration with the driver.

The problem of dynamic adaptation is to maintain the QoS of the system at a certain level. In order to do that in a system involving humans, the QoS of human models as well as for automation models needs to be specified in a common framework. The challenge is that various attributes that determine a NFP are heterogeneous in nature while considering human and machine models. We can address this problem using our common NFP structure to specify the QoS of human, machine, and their interactions.

3.4 NFP Metamodel

In order to set a general consensus, based on empirical observations, the following assumptions are made: A NFP are determined by a set of non-functional attributes

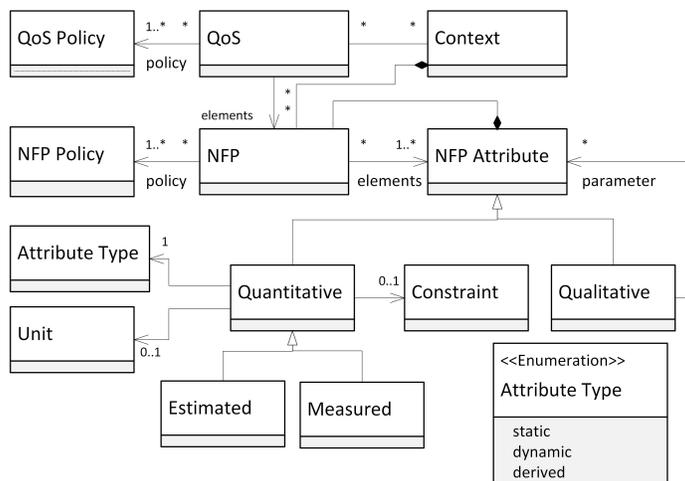


Figure 3-2: NFP Metamodel

(e.g., performance of a object classifier can be estimated by the time required for classification; and its efficiency by the rate of misclassification, etc.). QoS is a high level property for comparing a functionality in different contexts (e.g., QoS of a specific object classification algorithm is better in indoors as compared to outdoor environments). Policies associated with NFP and QoS determine how these properties are estimated from its constituent attributes. Policies are functions that act on a set of attributes, and define how these properties are estimated. Policies can be defined using logic systems, such as a first order logic, predicate logic, etc [83]. In a nutshell, $NFP_Policy(NFP\ attributes)$ defines NFP, $QoS_Policy(NFPs, Context)$ defines QoS of a functionality. However, there is no strict rule to identify whether a property is NFP or QoS. In our work, we use a rule of thumb that a functionality (or system) can have multiple NFPs (e.g., performance, efficiency), but can have only a single QoS property (e.g., $QoS_Policy(performance, efficiency, context)$ defines QoS). In our NFP metamodel terminology, the attributes that determine NFPs are called *QoS attributes*. Each NFP has at least one *QoS profile* associated with it. A QoS profile consists of a set of QoS attributes and a *QoS policy* that defines how the attributes affect the quality of NFP.

Based on the assumptions made in Section 3.2, a metamodel is proposed for specifying the NFP as shown in Figure 3-2. NFP and QoS are the root entities in this metamodel. NFP has at least one NFP Policy and a set of NFP Attributes. Similarly, QoS has at least one QoS Policy and a set of NFPs as its attributes. NFP attributes

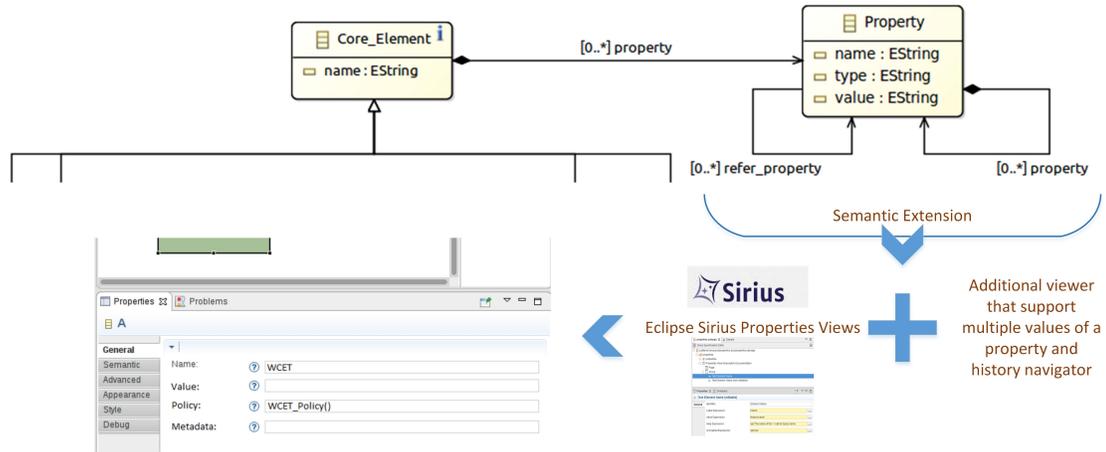


Figure 3-3: Common tooling support with additional editors for having multiple values for a property and a navigator for browsing history of values

can be Quantitative or Qualitative. Quantitative attributes can be directly measured or can be estimated. Quantitative attributes have metric units associated with it, for example: seconds for responsiveness, bits per second for throughput, etc. The quantitative attributes can be static, dynamic, or derived. Static attributes will not change during the course of system operation, for example, pixel density of a camera. Dynamic attributes can change during system operation, for example, frames per second of a camera. Derived attributes can be static or dynamic and depend on the constituent attributes when they are hierarchically composed. Qualitative attributes refer to discrete characteristics that may not be measured directly but provide a high level abstraction that are meaningful in a domain, for example, reliability of a network channel. Sometimes a qualitative attribute needs some quantitative measures also, for instance, round robin scheduler with a refresh rate of 100ms.

3.5 Tool Support for NFP specification

In Chapter 5, we will introduce a framework specification method that supports semantic extension for different domains. This extension framework provides a tool support in the form of viewpoints and property sheets. In order to make use of this tool support, the NFP structure is mapped to the **Property** element as shown in Figure 3-3. However, additional plugins were created for improving usability mainly for two reasons. First, to facilitate that a property can assume multiple values and

secondly, to provide a view that help user to navigate through history of values for a property. In early phases of the system life cycle when a component is being modeled, the properties can be an estimation or even a requirement. The accuracy of the estimation during the development process can be changed, as a result of an increasing amount of information or a change in the way the value is obtained. In the run-time phase (or even in the development phase in some cases), the property value could be measured [81]. For this purpose, appropriate tools are provided to allow a property to assume multiple values.

One of the major ambition of MDE is to provide automated code generation to be executed on a target platform [23]. On one hand it is important to have a structure for specifying the NFPs that can be reviewed at different design phases and that facilitates in verification and validation efforts. On the other hand, certain NFPs cannot be preserved at code level and sometimes they cannot be accurately determined until the execution of the code [84]. To facilitate this process, the tool allows multiple values for an NFP during the development process and helps to incorporate a full round-trip engineering approach in order to evaluate quality attributes of the system by code execution monitoring as well as code static analysis, and then provide back-propagation of the resulting values to modeling level [85].

3.6 Example of NFP specification in an Assistive Lane Keeping System

Assistive lane keeping is a perfect example for dynamic function allocation in Human-Machine system. The automation part is the lateral control of the vehicle to keep the vehicle in the same lane. The system takes control by applying the required torque on the steering wheel in case of lane departure situations. In Chapter 7, we have used this application as case study for specifying framework for developing cognitive systems. NFPs of Human and Machine are formally modeled using the proposed NFP metamodel. In Figure 3-4, efficiency property of human, machine and interaction model is specified. We define QoS for efficiency as the response time in which the human will react to a critical event (here lane departure). Modeling

```

//HUMAN_MODEL:
import control_skill,monitor_skill,decision_skill,memory_skill
NFP: eh:efficiency_human; NFP_ATTRIBUTES: cs:control_skill:derived, ms:
monitor_skill:derived, ds:decision_skill:derived, memory_skill:
derived; NFP_POLICY: eh.cs>thr_cs & eh.ms>thr_ms & eh.ds>thr_ds;
-----
NFP: control_skill; NFP_ATTRIBUTES: prep_time:static:ms, exec_time:static:
ms, kfar_lateral:dynamic, knear_lateral:dynamic, ki_lateral:dynamic,
kfar_speed:dynamic, knear_speed:dynamic, ki_speed:dynamic; NFP_POLICY
: contrl_policy();
-----
NFP: monitor_skill; NFP_ATTRIBUTES: prob_monitor:dynamic, NFP_POLICY:
monitor_policy();
-----
NFP: decision_skill; NFP_ATTRIBUTES: safe_distance:dynamic:m, NFP_POLICY:
decision_policy();
-----
NFP: memory_skill; NFP_ATTRIBUTES: memory:dynamic:, creation_time:static:
sec, decay_rate:static:, NFP_POLICY: memory_policy();
//MACHINE_MODEL:
import platform_capability,sensor_capability,algorithm_parameters
NFP: efficiency_machine; NFP_ATTRIBUTES: response_time; NFP_ATTRIBUTES:
execution_time:static:ms, NFP_POLICY: response_time_policy();
-----
NFP: platform_capability; NFP_ATTRIBUTES: process_load:dynamic:number,
resource_availability:derived, scheduling_policy:qualitative,
NFP_POLICY: decision_policy();
-----
NFP: algorithm_parameters; NFP_ATTRIBUTES: vehicle_velocity:dynamic:kmps,
front_wheel_steering_angle:dynamic:rad, slip_angle:dynamic:rad,
NFP_POLICY: algorithm_policy();
//INTERACTION_MODEL:
import efficiency_human, efficiency_machine
NFP: efficiency_interaction; NFP_ATTRIBUTES: efficiency_human:dynamic:
derived, efficiency_machine:dynamic:derived,
NFP_POLICY: interaction_policy();

```

Figure 3-4: A high level NFP specification of Efficiency property of human driver and automation, and their interactions

the human driver in ACT-R architecture [86] is selected for demonstrating the NFP modeling because of the availability of large number of tunable parameters in models representing various aspects of human driver such as memory, perception, control, monitoring, and decision skills. A detailed description of the model can be found in [87]. The efficiency model of the human driver, machine and their interactions based on the proposed NFP metamodel. The policy functions can be defined based on logic systems, such as a first order logic, predicate logic, etc., by the system designer. An example on how such specifications can be employed in a framework is provided in Chapter 7.

3.7 Related Works

The importance of specifying non-functional properties in computer vision systems is studied in [88]. In [30], the authors showed how to express variability in a robotic system for non-functional properties using a DSL called Variability Modeling Lan-

guage (VML). They have also provided a mechanism to express how a system should adapt at run-time based on those properties and adaptation rules. In [89], a practical means of exploiting non-functional properties in an architectural assembly process with structural constraints are introduced. The authors of [29] have proposed a development process and meta-model that allows the explication, management and analysis of non-functional properties which enables analysis of resource usage and verification of resource constraints. Although there is little literature that deals directly with modeling NFPs in human-machine systems, the works in metrics used in human-machine systems can be used for reference. Metrics for standardization for Human-Robot Interactions (HRI) have been proposed by [90], [91], [80], and [92]. The proceedings of the workshop on 'the Metrics for Human-Robot Interaction' proposed several guidelines for the analysis of human-robot experiments and forwarded a handbook of metrics that would be acceptable to the HRI community [93].

Garlan et al. [94], in their work on Acme, enable architectural adaptation by writing repair strategies which work within an architectural style. Each component declares a number of non-functional properties to be monitored by the running system. If a constraint is violated, a repair strategy is invoked in response. [95] proposes two complementary approaches for using Non-Functional information - process oriented and product oriented. Process oriented approach used NFP to guide the development of software systems. While in the product oriented approach, the non-functional characteristics of the final product are explicitly stated, making it possible to examine if products fall within the constraints of non-functionality. There are three UML profiles that are standardized by the Object Management Group (OMG) that deal with modeling QoS for software components - [96], [97], and [77] profiles. In [98] a QoS ontology for service-oriented systems has been proposed. Our intention is to structure the property specification at a certain abstraction level and to integrate in our framework development methodology. The level of details and constraints on those specification is decided by the framework developer and it is open for him to map the required concepts. In the context of safety in autonomous systems, the authors of [99] describe the computation of probabilities according to different specifications, based on functional and non-functional requirements, through probabilistic model checking.

3.8 Conclusion

Providing a systematic way of NFP specification and integrating it with development process by appropriate tools are essential for efficient framework development process. The importance of Non-Functional properties in robotics and human-machine systems were discussed. Modeling those properties are necessary in architectures where functionality alone cannot be used for making both design time and run-time decisions. Our NFP metamodel provides a generic base for specifying the non-functional aspects of both human and machine models. The main challenge in finding structure for NFP specification is to provide a flexible mechanism to address a large variety of property types and providing a tooling support to manage them. The challenge of dealing with heterogeneous attributes are addressed by categorizing the attributes into profiles hierarchically and then using policies to compare at a higher abstraction level.

3.9 My Contributions

The main contributions in this chapter are enumerated below:

1. The chapter analyzed the importance of specifying and integrating NFPs in Framework development in Robotics domain.
2. A metamodel to structure the specification of Non-Functional Properties is proposed. A common structure for NFP helps in specifying the non-functional aspects of both human and machine models.
3. An extensible tooling support is developed to manage NFP specification and their management.

Chapter 4

Solution Space Modeling

The more of the context of a problem that a scientist can comprehend, the greater are his chances of finding a truly adequate solution.

Russell L. Ackoff

4.1 Introduction

Architecting a robotic system is a science of integrating various independently operable heterogeneous systems such as perception, navigation, planner, controller, etc. Traditionally in robotics, the adaptation of the robotic system to the dynamic environments is embedded in the functionality of the constituent systems, for example, by designing a dynamic path planning algorithm. This practice limits the adaptation within the functional boundaries of the system components. Therefore, the purpose of this chapter is:

- To identify the inherent problems in the software design stage of robotic software development.
- To propose a Domain-Specific Language (DSL) that facilitates the design space exploration specific to the field of robotics

The solution space for implementing a functionality in robotics domain is large. Unlike in research laboratories, where domain experts are involved in software development, in industries there is a clear separation of roles. Domain experts design the system (e.g., using UML class diagrams) and make key decisions on algorithms that are then communicated to the software developer who develops the software code. Robotic experts can oversee various algorithmic solutions, specify abstract software component interfaces, predict high level dependencies, etc.; however, they may not be well versed in best software engineering practices, cannot predict execution time of algorithms, etc. A software developer, sometimes, anticipates and assumes several facets about the target environments while implementing and testing software components that may not be valid in the final system composition. For example, a developer may implement a particular localization algorithm that requires the sensor data to have a certain confidence level and resolution, which may not be satisfied in the target application. The underlying reason is that no formal model is used to analyze and manage the solution space available to various stakeholders. Taking into account several experts' opinion from academics and industry, the robotic system designs are fallible due to these general reasons:

1. Most of the robotic system designs are purely functional, they do not explicitly capture the non-functional aspects, such as, timing properties, etc.
2. The decision on which algorithms to use is decided by the domain expert during the design phase without considering the operational profile of those functionalities and its prerequisites, run-time environment, potential interactions etc.
3. Only functional verification of individual systems is conducted and the performance of these modules cannot be guaranteed when it coexists with other systems, for example, the obstacle detection system may take more time to compute when it is executed along with other systems in the real world scenario.
4. Lack of common ontologies and reluctance of roboticists to accept any standard development process.



Figure 4-1: Test Vehicle used for vehicle tracking experiment. Lidar and GPS are mounted on top of the vehicle and embedded computers are located in the trunk of the vehicle (see picture inset).

4.2 Motivational Experiment

The problems faced and lessons learnt during the system design, software development, and field experiments conducted during a research at Renault were the driving forces to propose a model-based approach. The objective of the experiment was to design and implement a vehicle tracking system using Velodyne HDL-64E lidar sensor. Velodyne lidar is a high definition laser scanning system that generates about a million points per second using its rotating sensor head containing 64 semiconductor lasers as shown in Figure 4-1. The tracking system should detect vehicles in the environment and compute its state - 2D position and velocity. It was foreseen to use the system for the following scenarios: a) Ground truth generation, b) Path planning for autonomous vehicle, c) Traffic surveillance, and d) Map building applications. The experiment was successfully completed, however, in an ad-hoc manner that made it impractical in several other anticipated application scenarios. This section document some of the encountered problems and classifies them into four core categories.

In our work, we employed the classical approach of performing data association on segmented scenes across frames followed by the probabilistic state estimation. The process flow is composed of the following four stages: a) *Data acquisition*: Spatio-temporal point cloud data is acquired; b) *Segmentation*: Points clouds are segmented to object level; c) *Data association*: Point cloud features are computed for segmented

clusters and then associated with objects across frames; d) *State estimation*: Probabilistic inference methods such as Kalman Filters, PHD Filters are applied to estimate the position and velocity of the detected vehicles.

Some of the relevant problems encountered during the design and development of the tracking system are discussed below.

Large number of segmentation algorithms are available in the literature to segment point cloud data. Some of them were dependent on the type of the sensor (e.g., Lidar, Time-of-Flight Cameras), properties of data (e.g., density, resolution, colour, intensity), environment features (e.g., indoor, outdoor, cluttered, flat or sloppy terrain, vegetative land). In addition, the intention to use the tracking system in a variety of applications, such as ground truth generation, and autonomous driving, whose requirements for timing properties, resolution and confidence levels are entirely different, makes it difficult to choose an appropriate algorithm.

The rotating head of lidar takes non-negligible amount of time to capture a frame. If the lidar is mounted on a moving vehicle, the generated 3D point cloud frames get distorted due to this motion. Since this was not properly captured in the initial designs, considerable amount of time was spent on implementing software that later failed in field experiments. One such incorrect component composition was when Point Cloud Histograms (PFH) were used as feature descriptors in the data association step. PFH descriptors captures the spatial distribution of points in the point cloud. When the tracking system was used in moving vehicle without including an undistortion process on the point cloud data, the feature correspondences do not work. The undistortion process requires a high performance inertial localization system to compute the pose of the vehicle at the timestamp of each captured point and applies coordinate transformation to compute the exact 3D point. Such induced requirements and constraints must be captured to make the system more adaptable, for run-time dynamic wiring of software components, and for deriving a product from product line systems.

Context information was unavailable during design time as well as during field experiments. Certain algorithms can be integrated to the system only by using such information, for example, an object classifier needs the road information to help in classifying between vehicles or buildings on the road side, between pedestrians and

parking poles, etc. Several algorithms (ground classification, for instance) require parameter selection based on the context, such as terrain type - sloppy or flat, road width, etc.

Identified Challenges

From the experiment discussed above, and from the previous experiences in developing similar robotic systems, the common challenges can be broadly classified into four categories:

1. **Uncertain problem space:** Ambiguity in requirements due to the desire to reuse the system across various applications.
2. **Large solution space:** Availability of multiple algorithms for implementing a functionality.
3. **Lack of design time context information:** The developer cannot anticipate all the use cases and his/her assumptions are not properly documented.
4. **Incorrect level of abstraction:** Code-centric designs cannot provide the right level of abstraction that promote portability and reusability.

To address these aforementioned problems, we propose a modeling language to formally model the solution space and to specify the quality attributes during design time. The approach is to capture multiple solutions in the model that permits formal analysis, reasoning, and decision making on selecting best possible solution depending on the functional and non-functional properties. The solution model also helps in - system level reasoning, making tradeoffs, documenting decisions, and comparing them, and for formally proving and validating the final implementation.

4.3 Solution Space Model

By critical analysis of the problems described in Section 4.2, the desirable features for a solution space model are deduced: a) It should be a graphical model that can be visually inspected, as well as machine readable; b) It must be a hierarchical model that provides views at different granularity levels; c) Non-Functional Properties (NFP) and Quality of Service (QoS) must be explicitly stated; d) It must capture

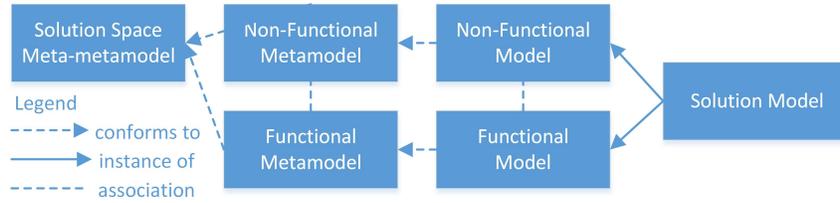


Figure 4-2: Relationship between the proposed models

the uncertainties in problem space and must act as a reference model for developing concrete software models in the operational space. Equally important to the desirable features, the model should not include any implementation specific details, such as communication patterns and programming language dependencies.

In this chapter, a Solution Space Modeling Language (SSML) is proposed. SSML is specified at two abstract levels as shown in Figure 4-2. Solution space meta-metamodel is at the highest level and functional and non-functional metamodels at the lower level in the MDE hierarchy. Figure 4-3 shows the Ecore meta-metamodel diagram and the graphical representation of the primitive elements. In natural language, the syntax can be explained as follows: the solution space consists of Dispatch Gates, Ports, Connectors as the primitive elements. A Dispatch gate consists of number of ports and is associated with a Dispatch Policy. The ports can be ‘in’ or ‘out’ indicated by the `port_type`. A connector is associated with two ports of which one should be an ‘in’ port and the other an ‘out’ port. A connector can have NFP and QoS associated with it.

The semantics of the proposed model is as follows: the dispatch gates represent basic operations for composing different functional computational processes, such as selecting an appropriate data source for a computation, synchronization point, buffering, etc. The dispatch policy associated with it defines the operation of the gate. The data enters or leaves the gates through ports. Ports can be of type ‘in’ or ‘out’ depending on whether the data enters or leaves the gate. A connector connects two semantically compatible ‘in’ and ‘out’ ports. A Connector represents a functional computation and its quality aspects are represented by NFP and QoS.

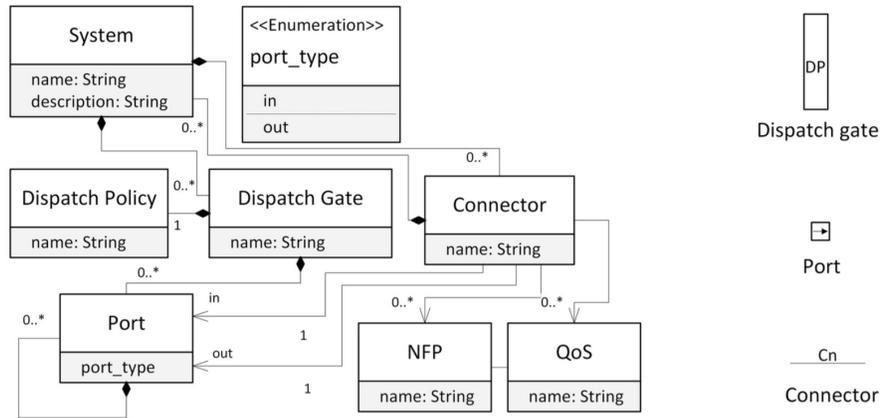


Figure 4-3: Metamodel (left) and its graphical representation (right) of SSML. Dispatch Gate, Port, and Connector represents the functional aspect and NFP and QoS Profile represents the non-functional aspect.

4.3.1 Functional Model

A functional model satisfies the behavioral requirements of the system. A behavioral or functional requirements are those requirements that specify the inputs (stimuli) to the system, the outputs (response) from the system, and the behavioral relationships between them [100]. The functional model in the solution space should not be interpreted as a data flow diagram, but as a relationship diagram between functional concepts.

Functional Metamodel

A functional metamodel is depicted in Figure 4-4 that conforms to the solution space meta-metamodel. This metamodel imposes *soft* constraints on the dispatch policies of the gates and the number of ports [101]. Soft constraints means that the dispatch policies are not concretely defined but only conceptual restrictions are only imposed in the metamodel. The intention is to facilitate the designer to apply application specific policies, at the same time, the gates can be used for automated reasoning of the solution space.

a) *Splitter gate* consists of 1 input port and n output ports. It creates n splits/copies of the input data and transfers to its output ports.

b) *Merger gate* consists of n input ports and 1 output port. It merges the data from n input ports to the output port.

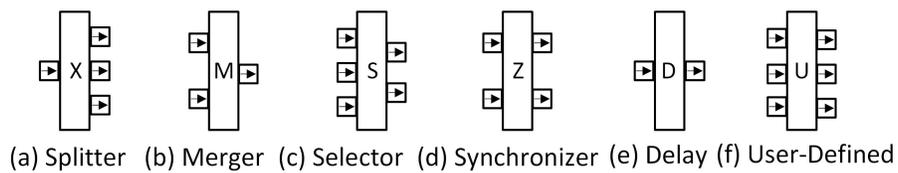
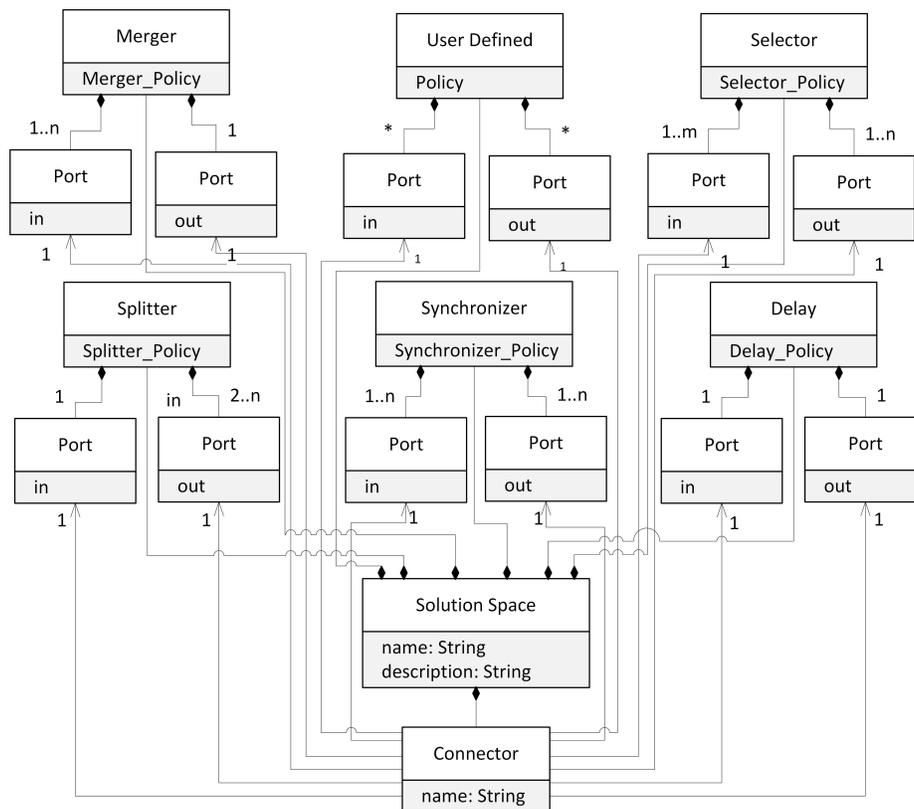


Figure 4-4: Metamodel for functional modeling (left) and graphical representation of dispatch gates (right).

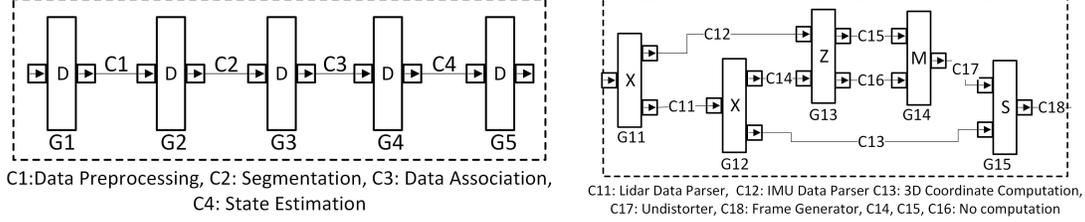


Figure 4-5: Solution space model for tracking system. High level model is shown (top), zero delay gates (G1-5) are inserted to separate the computations. Connector C1 representing data preprocessing is modeled (right)

- c) *Selector gate* consists of m input ports and n output ports. It selects n out of m input data.
- d) *Synchronizer gate* consists of an equal number of input and output ports. It acts as a synchronization point between the different data streams.
- e) *Delay gate* consists of one input and output port. It passes the data in the input port after a time delay and can also act as a data buffer.
- f) *User-defined gate* does not have any constraints on the number of ports and dispatch policy.

4.4 Solution Space Model for Lidar Based Vehicle Tracking System

This section models the solution space of a lidar based vehicle tracking problem using the proposed SSML. Figure 4-5 shows the high level model of a classical approach in tracking described in section 4.2. Four connectors, C1, C2, C3, C4, represent data preprocessing, segmentation, data association, and state estimation process and the zero delay gates are inserted to differentiate between these processes. This model consists of single sequence of processes that do not have multiple solution paths at this hierarchical level. In this context, a solution is referred as an execution path or simply a path in the solution space model. In textual form, it is represented as a sequence of labels indicating gates and nodes in that solution. The path represented by adjacent parenthesis, for example (path 1)(path 2), indicates a mandatory parallel execution path. The lidar data preprocessing solution model captures two solutions: G11-C11-G12-C13-G15-C18 and G11-(C11-G12-C14-G13-C16)(C12-G13-

C15)-G14-C17-G15-C18. The two solutions do the same functionality: converting raw sensor data into point cloud frame. The difference between the two solutions is the quality aspect that changes with context. In this case, context represents the motion of the vehicle and the difference in quality is due to time latency of lidar, as explained in section 4.2. The NFPs that are considered for this process are performance and resource cost. The NFP attributes that model performance are response time, resolution, and average distortion. An example snippet from the NFP model for data preprocessing represented by C11 is shown in Listing 4.1.

```

CONTEXT: vehicle
NFP: vehicle_motion
NFP_ATTRIBUTES: velx:base_velocity_x:msrd:dynamic:mps, vely:base_velocity_y:msrd:
dynamic:mps, velz:base_velocity_z:msrd:dynamic:mps, rr:roll_rate:msrd:dynamic:
degpsec, pr:pitch_rate:msrd:dynamic:degpsec, yr:yaw_rate:msrd:dynamic:degpsec;
NFP_POLICY: vehicle_motion_policy();
*****
NFP: C11.response_time;
NFP_ATTRIBUTES: fps:frame_per_second:msrd:static:int,tp:transmission_speed:msrd:
static:msec;wcet:worst_case_execution_time:est:static:msec;
NFP_POLICY: response_time_lidar_policy();
-----

IMPORT CONTEXT vehicle;
NFP: C11.resolution;
NFP_ATTRIBUTES: ar:angular_resolution:est:static:deg,rps:rotations_per_second:msrd:
static:Hz,tlppr:total_laser_points_revolution:msrd:static:int,pplpr:
points_per_laser_per_revolution:msrd:static:int;
NFP_POLICY: resolution_lidar_policy();
-----

IMPORT NFP C11.response_time,C11.resolution,C11.average_distortion
NFP: C11.performance;
NFP_ATTRIBUTES: rtl:response_time,rl:resolution_lidar;
NFP_POLICY: c11_performance_policy();
-----

IMPORT CONTEXT vehicle,environment;
IMPORT NFP C11.performance,C11.resource_cost;
QoS: C11.QoS
NFP: C11.performance,C11.resource_cost;
QoS_POLICY: C11_QoS_Policy();

```

Listing 4.1: Relevant snippets of the Non-Functional Model of vehicle, environment, and lidar data preprocessing process.

Another significant advantage of modeling functional and non-functional aspects separately is that certain invalid compositions of systems can be found. For example,

referring to the problem described in section 4.2 regarding PFH feature correspondence failure, such anomalies in system composition can be captured without explicitly indicating that Point Feature Histogram (PFH) feature ‘requires’ undistortion process. Listing 4.1 shows an example of non-functional properties associated with Lidar data parser (connector C11 in Figure 4-5). The modeling elements comply to the data structure presented in Chapter 3. Resolution property of lidar data preprocessing process shown in Listing 4.1 using vehicle context to determine the resolution of the point cloud data. Hence, if PFH feature that requires a better resolution is composed without undistortion process, the QoS of the system will be lower since the resolution constraint does not satisfy point cloud resolution level.

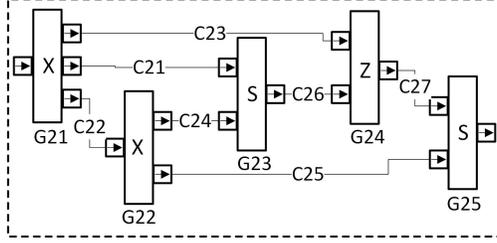
The solution space for segmentation process represented by connector C2 in Figure 4-5 is modeled in Figure 4-6. The objective is to cluster a point cloud frame into smaller clusters in such a way that each cluster represents individual objects. There are multiple data processing steps and algorithms to achieve the goal. In general, a point cloud data can be segmented into clusters using two methods: 1) By directly processing the 3D data and clustering using some criteria such as Euclidean distance or 2) By mapping the point cloud to a 2D image and performing segmentation in 2D space using image processing techniques and then projecting back to the 3D space to compute the final point cloud clusters. Each algorithmic step indicated in Figure 4-6 is briefly explained as follows:

Ground Classifier: The algorithm can classify ground and non-ground points from a point cloud frame. By rejecting the ground points, this algorithm can significantly reduce the number of points for further processing.

2D Projection Method: This method uses (x,y) coordinates of the point cloud and converts it to a two-dimensional binary image. This is best suited for point clouds consisting of non-ground points and hence, it receives data from a ground classifier.

Euclidean Clustering: This algorithm uses Euclidean distance to cluster the point clouds. It can be applied on non-ground point cloud data to generate the final point cloud clusters.

2.5D Projection: This algorithm can convert a point cloud to a 2D grayscale image. Ground points are automatically rejected and there is no requirement for explicitly classifying the ground and non-ground points.



C21: 2.5D Projection, C22: Ground Classifier, C23: No Processing, C24: 2D Projection, C25: Euclidean Clustering, C26: Image Segmentation, C27: Box Mapping

Figure 4-6: Connector C2 representing segmentation shown in Figure 4-5 is modeled in the figure

No.	Execution Path	Variation Points
1	G21-(C21-G23-C26)-(C23)-G24-C27-G25	(C21),
2	G21-(C22-G22-C24-G23-C26)(C23)-G24-C27-G25	(C22-G22-C24),
3	G21-C22-G22-C25-G25	(C24-G24-C27), (C25)

Subpath No.	Variation	Performance		Ground Truth Generation		Autonomous Driving	
		AET	Res.	Stage 1	Stage 2	Stage 1	Stage 2
1	(C21)	48ms	Med	Fail	Pass	Pass	Pass
2	(C22-G22-C24)	107ms	High	Pass	Pass	Fail	Fail
3	(C24-G24-C27)	31ms	Med	Pass	Fail	Pass	Fail
4	(C25)	120ms	Low	Fail	Fail	Fail	Fail

Table 4.1: Solution comparison for segmentation w.r.t application: Ground truth generation and Autonomous Driving

Image Segmentation: Using image processing techniques, connected regions are found in the image and a bounding box is computed for each region.

2D to 3D Box Mapping This uses a list of 2D boxes and the correspondence point clouds and generates 3D point cloud clusters. Some intermediate results are shown in Figure 4-7.

The NFP of segmentation process are modeled in a similar way as previously described in this section. In this example, two contexts are considered; one for a ground truth generation application and another for autonomous driving application. The intention is to reason, analyze, and extract appropriate solutions from the solution space that satisfy the context requirements. If two or more solutions satisfy the requirements, it will be considered as variation point that can be resolved in the design or implementation phase, or dynamically selected during run-time. For simplicity, only the performance property is considered here. The performance is modeled with

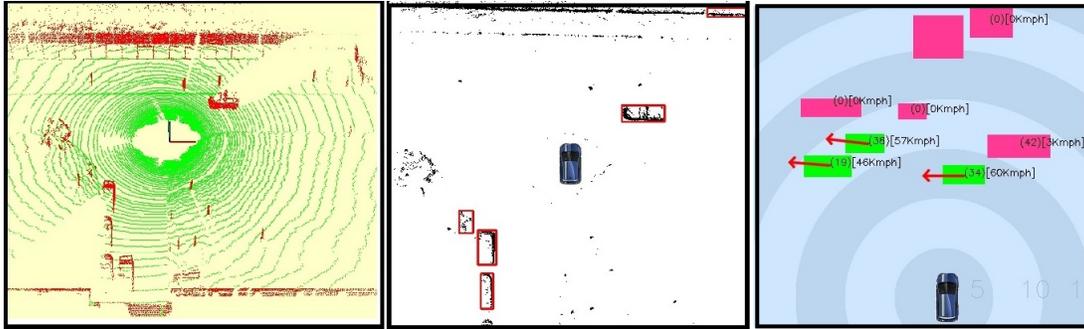


Figure 4-7: Vehicle Tracking Results: Segmented point clouds (left), detected vehicles (middle), tracked vehicles (right)

two NFP attributes - Average Execution Time (AET) and Resolution. AET is computed as the average time taken to execute the computation and it is estimated in the test platform, since an accurate timing information is not required at this stage. Also it is reasonable to compare the AET of different computations on the same platform for primary investigations. The resolution is divided into three levels depending on the grid size of the map: High, Medium, and Low, to facilitate the demonstration. The appropriate solution in different contexts is extracted in a staged process as described below:

1. Find the multiple solutions available in the solution space model.
2. Find the variation sub-paths among the multiple solution paths.
3. List the sub-paths from the variations found in step 2.
4. For a given context, find whether the homogeneous sub-paths satisfy the quality policy (NFP or QoS Policy) of that subpath. A homogeneous sub-paths is a set of sub-paths that perform exactly the same functionality.
5. Consider all the sub-paths satisfied in step 4 and check for any contradictory results, for example, if the solution passed in one homogeneous check and failed in another.
6. If there were any contradictory results, repeat the quality check for the solution at the higher level in the hierarchy.

The solution resolution steps of segmentation model are captured in Table 4.1. There are three execution paths as listed in the table. Two variation points are then found - one variation sub-path between execution path 1 and 2 and the second between 2 and 3. It is to be noted that individual paths that end with a synchronizer gate cannot be considered as separate execution path since all the paths are mandatory for synchronization. The next step is to list out the sub-paths in the variation points. The four sub-paths are indicated in the second half of the table. The sub-path 1 and 2, and 3 and 4 are homogeneous pairs since they represent the same functionality. The policy used were (`Resolution==High`) for ground truth application and (`Resolution==High`) AND (`AET<50ms`) for autonomous driving application. In stage 1, two solutions satisfy the policy and in the next stage, one solution is extracted by considering the policy in that level.

4.5 Solution to Operational Space Transformation

After the solution space model for a given problem is modeled in SSML language, the next stage is to transform it to an operational model. It is to be noted that the solution space model only satisfies the functional requirements and the non-functional requirements are enforced only while it is transformed to the operational model. It is common that there may be more than one solution that satisfies the NFP requirements, in which case, the best possible solution can only be resolved during runtime. In addition to these solution subsets, there may be some solutions that partially satisfy or may depend on the context variables that can be estimated in real execution scenarios. Hence, the transformation has two analysis stages - design time and runtime.

The solution model for point cloud segmentation problem is shown in Figure 4-6. The solution space model is a form of Directed Acyclic Graph (DAC) with functional computations represented as edges, and gates form the nodes of the graph. The non-functional properties associated with the edges are assumed to be fully observable in our study. This is a valid assumption for most of the NFP properties such as, WCET, Required Memory, etc., as these can be estimated using external tools when the execution context is known. This assumption can be extended to qualitative properties

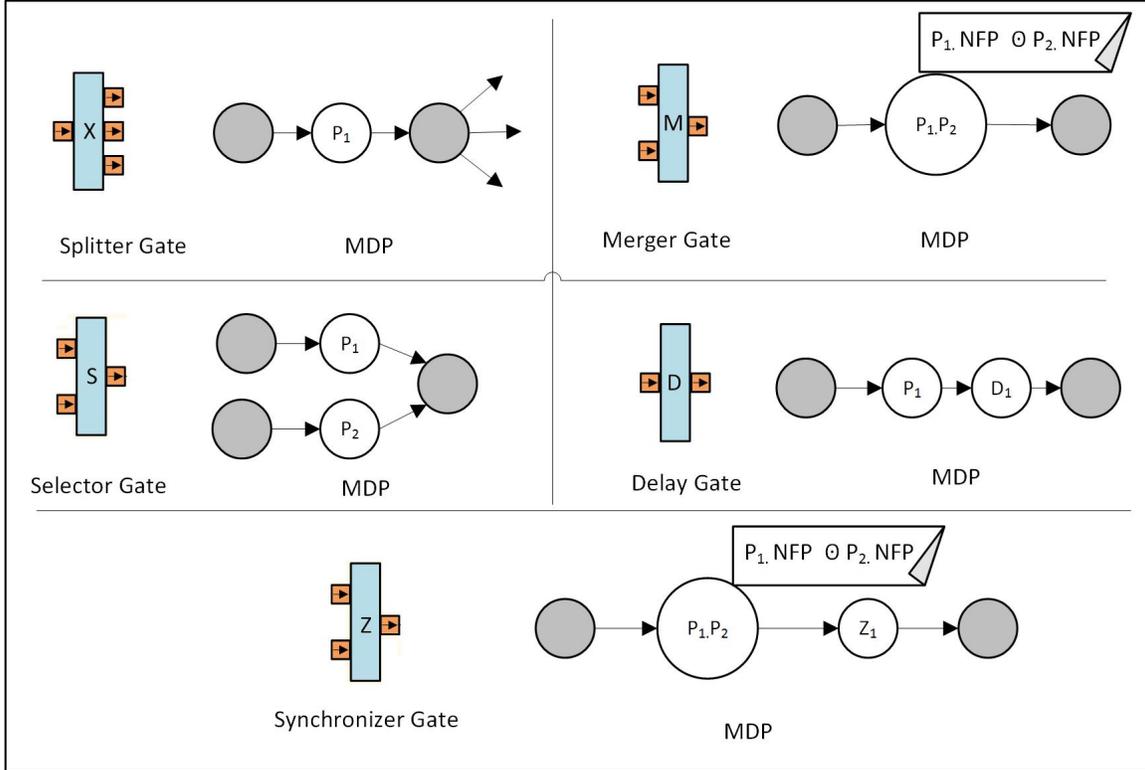


Figure 4-8: SSML Gates and corresponding MDP models

such as, usability, portability, etc., by mapping it to a range of values. For example, portability can be given values in the range 0 to 5, 0 being least portable component and 5 for the component with a high portability. The uncertain environment in which the robot is to be deployed makes the decision process on the best possible solution, stochastic in nature. The graph can be seen as a finite state machine augmented with probabilities and non-deterministic transitions. Hence, this model can be translated to a formal decision model called as Markov Decision Process (MDP). MDP models a sequential decision problem for a fully observable, stochastic environment with Markovian transition model and additive reward. By Markov transition model, we mean that the probability of reaching the next state from the current state depends only on the current state and not on the earlier states [102].

4.5.1 Transformation Process

The solution space model is transformed into MDP model in a systematic approach. A solution space model is composed of gates and connectors as basic modeling elements

that represent decision points and computational functionalities respectively. The connectors are transformed to states and gates are translated to edges connected to these states. Figure 4-8 shows the translation of SSML gates to MDP graphs. The grey states are decision points where multiple edges can be connected. The decision states do not have an associated NFP properties, hence they are also called as non-impact states. The NFP property associated with the connectors in solution space are augmented with the corresponding states in the MDP model. As shown in Figure 4-8, splitter gate is transformed into two decision states and one impact state that represents the connector connected to the input port of the gate. Three edges that originate from the second decision state has an associated probability that should sum to 1. The MDP model corresponding to Merger gate consists of two decision states and one impact state. In this model, the impact state represent two functional computations connected to the input ports of the gate. This means that both functionalities are active in this state and the NFP property associated with this state is estimated by composing the properties of the two functionalities. The MDP model of the selector gate is composed of the same number of impact states corresponding to the number of ports in the gate. In the MDP model of delay gate, an additional impact state D_1 is added that has a *time-based* NFP property. Similarly for the synchronizer gate, the additional state Z_1 models the time delay caused by the data synchronization.

After translating the gates to MDP model, the final MDP model of the complete system is generated by merging the corresponding decision states. Figure 4-10 shows the translated MDP model for the point cloud segmentation solution model. The states are annotated with two kinds of NFP property - Execution time (ET) and Resolution (Res). These are translated directly from the solution model. The transformation of solution space model to MDP model is performed by the SafeRobots tool and is generally hidden from the user.

4.5.2 MDP Model Analysis

The analysis of the MDP model is performed in two stages - design time and runtime. The design time analysis is an offline process to compute a subset of solution space model given the problem model. Runtime analysis is an online process during the

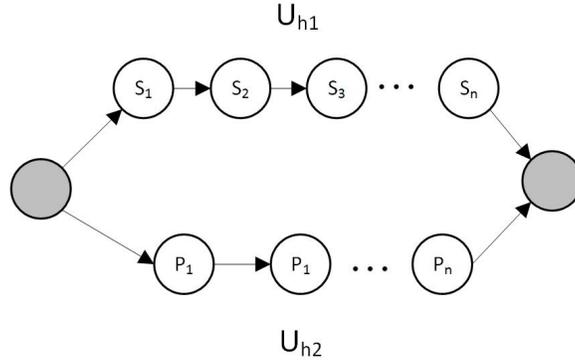


Figure 4-9: Design-time model analysis

system execution and is a pre-requisite for the self-adaptation of the system.

Design-time Analysis

The functional and non-functional requirements are modeled in the problem space of the framework. The NFP requirements are modeled in domain-specific language, which is presented in Chapter 3. However, for simplicity, in this chapter we specify them as simple name-value pairs. For example, estimated execution time of a functional computation is denoted as $ET : 20ms$.

Due to the stochastic nature of the execution context, a fixed sequence of actions may not work. There should be a policy that specifies what should be performed next from any given state. The overall goal of the system is to maximize the utility that depends on the sequence of states selected. We attach a bounded reward $R(s)$ to each state. The rewards are specified in terms of the NFP property attached to the state. For decision states, rewards are not specified since they do not influence the overall utility directly. In this chapter, we use the term *reward* in general, in some case it can represent a cost, for e.g. execution time (ET). This depends on how the requirement is specified, $ET < 100ms$ denotes ET as a cost and usability, $U > 3$ presents it as a reward. The policy, π should yield expected utility value that is satisfied by the requirement. For example, assume that a policy π_x can provide a expected reward in terms of execution time of $\langle 21, 44 \rangle$ for a particular subset of solution space model. Due to the stochastic nature of the system, the expected reward will always be a range of values. Only the overall utility of each path is considered during the

design-time analysis of the model.

The expected utility, U_h for each model path can be computed as:

$$U_h([s_0, s_1, s_2, \dots, s_n]) = R(s_0) \oplus R(s_1) \oplus R(s_2) \oplus \dots \oplus R(s_n) \quad (4.1)$$

Where, $R(s)$ is the reward for state s , \oplus denotes a composition operator. Figure 4-9 illustrates the utility values for a model consisting of two paths. The composition scheme depends on the type of non-functional property used as rewards. For example, ET property can be simply composed by adding the values of all the states in the sequence, while most of qualitative properties have more complex composition schemes. In many cases, the composition of NFP properties depend on the external environment and the system architecture. For example, the composition of component performance may depend on scheduling policies and the system configuration. Currently, the common practice is to define the composition based on some constraints/assumptions. For example, static memory usage is considered to be additive give that there is no concurrency in assembly execution [103]. The utility values are computed for all the available paths in the solution space model. Only those paths that satisfy the requirements are then selected. For example, if the requirement specifies the $ET < 100ms$, only those paths with utility value of ET less than 100ms are selected for the runtime resolution.

Runtime Analysis

The MDP model generated after the design time analysis is a subset of solution space that satisfies the non functional requirements. However, the best possible solution can be selected only during runtime due to the environment dependency. During the runtime analysis each NFP reward is replaced by the expected utility value from that state to the terminal state in form of $\langle \min(U_h), \max(U_h) \rangle$. The Figure 4-11 shows an example of a runtime MDP model in which each functional state is augmented with the expected utility forecast. The utility forecast estimation is performed using a probabilistic model checker called PRISM, which is a tool used for formal modeling and analysis of systems that exhibit random or probabilistic behavior [104]. For example, the state s_2 in Figure 4-11 is augmented with a expected utility forecast of $\langle 89, 134 \rangle$, which means that the ET property will have a value in that range from

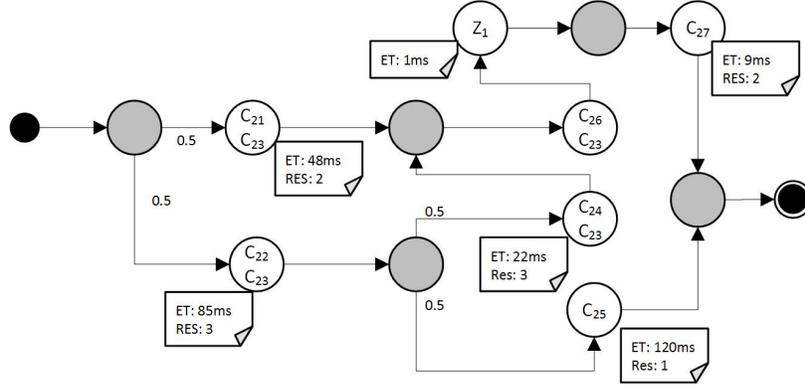
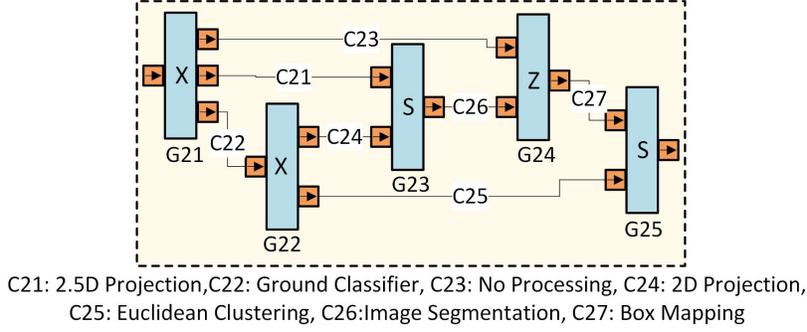


Figure 4-10: Solution space model for tracking system (top) and the corresponding MDP model (down).

state s_2 to the terminal state. While executing the system, the model also maintains a context data structure that comprises of the real value of the NFP property for the current state. For example, if the functionality associated with state s_1 is being executed, the data structure contains the execution time already taken to reach that state, say $ET_{current}$. Therefore, the available time for reaching the terminal state will be $ET_{req} - ET_{current}$, where ET_{req} is the required ET specified by the requirement. In the example shown in Figure 4-11, there are two execution path after the state s_1 is executed. The decision on the execution path depends on the context data structure and the expected utility forecast of the next states. Here, we model the forecast $\langle \min(U_h), \max(U_h) \rangle$ as uniform probability distribution as shown in Figure 4-11. The available time $ET_{req} - ET_{current}$ should be within $\min(U_h)$ and $\max(U_h)$ in order to reach the terminal state. Hence, the probability at which the execution path reaches the terminal state satisfying the requirements is given by the area under the shaded region shown in Figure 4-11.

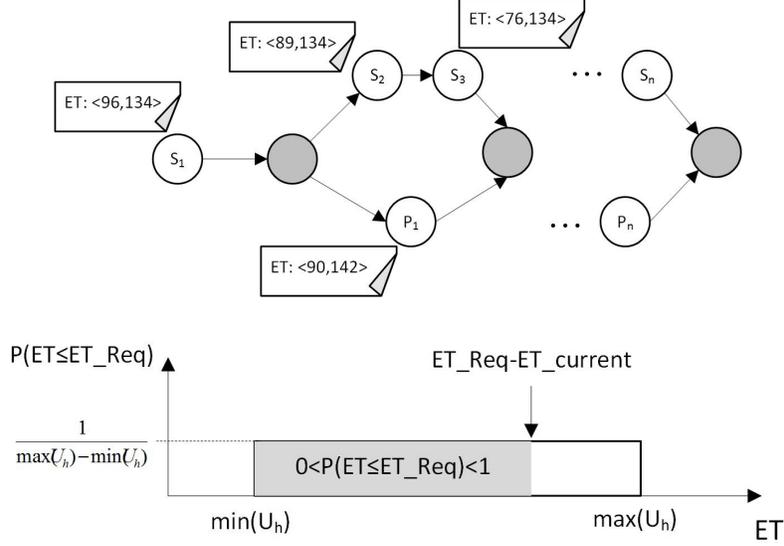


Figure 4-11: An example of annotated runtime MDP model (top) and the probability analysis of expected utility forecast (down)

The area can be computed as:

$$\begin{aligned}
 P(ET < ET_{req}) &= P(ET_{s_i} \leq ET_{req} - ET_{current}) \\
 &= \frac{(ET_{req} - ET_{current}) - \min(U_h)}{\max(U_h) - \min(U_h)}
 \end{aligned} \tag{4.2}$$

The execution branch with the highest probability of satisfying the requirement will be selected at the decision point. We have explained the process with the help of ET as the reward, however this can be extended to any quantitative property as reward with proper composition operation as discussed earlier.

4.6 Operational Models

The SafeRobots component model adopts the separation of concerns approach proposed by the BRICS framework [105]. The component semantics is mapped to the 5Cs - Communication, Computation, Configuration, Coordination, and Composition. In addition, the operation space also consist of architectural models in the form of meta model that specifies the constraints and knowledge regarding the configuration of the final system. The solution space model to operational model transformation

performs this semantic mapping to the 5Cs. However, the configuration-coordination pattern uses internally the MDP models and the mechanisms discussed previously for runtime coordination and dynamic invocation of computational components. Finally, the operational model is automatically translated to executable code using Model to Text (M2T) transformation. The real advantage of our process is that the self-adaptation capability is introduced before architectural decisions are made and they handled by the framework tool and hence, it enables evaluation and benchmarking of robotic architecture in an efficient manner.

4.7 Related works

By learning from the shortcomings of code-based approaches, the software engineering community in robotics is gradually moving towards Model-Driven Software Development (MDSD) approach [15]. Modeling solution space using SSML can be seen as a complementary approach to many existing methods in the robotics. Software product line approach is popular in software engineering community to enhance product quality and reduce developmental cost by promoting constructive reusability [106]. Recently, the authors of [107] have adapted this approach in robotic domain by providing feature resolution and transformation steps. However, one should have already identified well defined boundaries for variation points and a reference architecture for adapting this approach. In addition, many DSLs are proposed in robotics domain for deployment, simulation [19], component creation [108], etc. All these languages reside in the operational space, our proposed SSML can complement and facilitate a smooth transition from problem space to operational space.

Similar to our transformation approach, the authors of [109] have used a probability-based functionality adaptation for distributed mobile applications. However, they mainly address the problem of unexpected high response time and faults during the runtime of the system alone. Current research in self-adaptive systems concerns on the effective utilization of models during runtime. In such situations, the runtime models should reflect the most up-to-date information in order to perform online analysis and reasoning. There are some interesting reference architectures available from self-adaptive systems and models@runtime communities [110]. In [111] authors

have proposed a technique to explicitly document the existence of uncertainty about how architectural decision contribute towards satisfying non-functional properties in the form of soft goals. The technique allows developers to deal with uncertainty during both development time and runtime [112].

4.8 Conclusion

This chapter focused on the problems faced while developing software for robotic systems. An experience report on developing a lidar based vehicle tracking system in an industrial context is used for motivational purpose. The problems were classified into four categories: uncertain problem space, large solution space, lack of design-time context information, and abstraction issue. In this chapter, Solution Space Modeling Language is proposed to address the multiple solution problem and to formally specify NFP during system design. Solution space modeling can expand this design space, help finding the best possible solution, and also permit to perform run-time adaptation of the system. Solution model helps in early analysis of quality attributes, to identify variations and acts as a bridge between problem and implementation space.

More research is required on formal methods for specifying composable policies of Gates, NFP, and QoS in the proposed SSML language. Once the solution space of a problem is modeled, an appropriate solution or a set of solutions are selected depending on the application context and quality requirements. A transformation model will facilitate this process of mapping to the operational space models, for example, discrete timing properties of gates can be employed for process allocation, selecting scheduling policies, etc. In Part III of this thesis, we discuss the tooling support based on Eclipse IDE for graphical and textual editing of SSML models and using Eclipse plug-in feature for semantic enrichment of the model.

4.9 My Contributions

In this chapter, we started by discussing several challenges faced during design phase of robotic software development. Subsequently, we analysed different problems posed during the software development of a robotic software subsystem in an industrial context (intelligent vehicles). The following contributions are made in this chapter.

- We studied common reasons that makes the robotic system designs fallible by taking into account the previous experience of robotics experts in various experiments in academics and industry,
- The common challenges are then broadly classified into four categories: uncertain problem space, large solution space, lack of design time context information, and incorrect level of abstraction.
- To address these aforementioned problems, we proposed a modeling language - Solution Space Modeling Language (SSML), to formally model the solution space and to specify the quality attributes during design time.
- The relevancy of the model is demonstrated by modeling the solution space of vehicle tracking problem. NFP and QoS Policies facilitated the quality flow across functionalities at different granularity levels.
- The resolution of solution space might not always result in a static operational model. Typical the resolution process performed during development time result in a subset of solutions that are modeled as variation points in the architecture. This is attributed mainly due to certain context-based properties that can be estimated only during runtime. We formally define the solution space to operational space transformation process and employs a probabilistic approach to resolve the solution model during design time and execution time. These operations are handled automatically by the framework tool and are independent of any particular robotic framework.

Chapter 5

Architecture Modeling in Operational Space

It is always easier to destroy a complex system than to selectively alter it.

Roby James

5.1 Introduction

Despite the research that spanned almost three decades, the robotics community has not converged to a single or a minimal set of efficient architectures [113]. Although most of the existing architecture styles can be classified into three categories: hierarchical, behavioral, and hybrid; it is very common that each robotic system designer or research group builds their own architecture from scratch considering only their short-term immediate requirements. The design, simulation, and programming of robotics systems is challenging as expertise from multiple domains needs to be integrated conceptually and technically. In addition, complex robotic systems typically involve software components that use a variety of mathematical models for problem solving, for e.g., state machines for robot control [114], stochastic Petri Nets for navigation [115]. Furthermore, as an emerging solution for handling complex and evolving software problems, several Domain Specific Languages (DSLs) have been

proposed for specific functional domains in robotics [116]. They indeed help to raise the level of abstraction through the use of specific concepts that are closer to the respective domain concerns and facilitate validation and analysis [117]. A typical process for designing a robot architecture involves 'mix and match' of such architecture paradigms, mathematical models, DSLs, and implementation technologies. Without any common conformance model, this process is expensive both in terms of time and effort, and absence of systematic approach may result in adhoc designs that are not flexible and reusable. Therefore, within the context of architecture design, component development and their support tools, a coherent practice is required for developing architectural frameworks. The purpose of this chapter is:

- To investigate the possibility of a formal meta-framework model so as to model frameworks that support heterogeneous domains related to robotics.
- To propose a formal way to specify the interactions between different architecture concepts (e.g., state transition system [118] and process control system [100]) and paradigms (e.g., behavior-based [119] and cognitive architecture [120]) in the operational space.

5.2 Software Architecture

The unprecedented growth of software systems in size and complexity has led to new opportunities, but also to increased challenges in designing and specifying the system. Architecture frameworks and architecture description languages are created as a way to capture the conventions and common practices of architecting and the description of architecture within each application domain.

A software architecture helps in several purposes such as:

1. To recognize and reuse common paradigms so that their high level relationships and interactions can be clearly understood.
2. To describe the system topology and guide the decisions leading to the system structure.
3. To make principled choices among design alternatives.

4. To specify high-level properties, which are essential for the analysis of the system.

Definitions of terms that are central to our concept development and further discussions are given below.

Basic Definitions

The following terms and definitions have been adapted from the international standard for describing the architecture of a software-intensive system - ANSI/IEEE Std 1471-2000 [51]. The next section provides more details about this standard.

- A *system* is a collection of components organized to accomplish a specific function or set of functions.
- The *architecture* of a system is the system's fundamental organization, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.
- The *architecture framework* are a collection of conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders.
- The *concern* defines an interest in a system relevant to one or more of its stakeholders. A concern pertains to any influence on a system in its environment, including developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences.
- An *architecture view* expresses the architecture of the system-of-interest in accordance with an architecture viewpoint (or simply, viewpoint).

There are some arguments and discussions in the community regarding the definitions and their relationships between the concept of architecture description, framework, and meta-frameworks [121][122][123]. In order to position our work and motivate its importance, the relationships illustrated in Figure 5-1 are followed in the rest of the chapter. Architecture conforms to an architecture framework and is specified using an Architecture Description Language (ADL). Architecture addresses known concerns for known stakeholders for the system of interest. Architecture frameworks

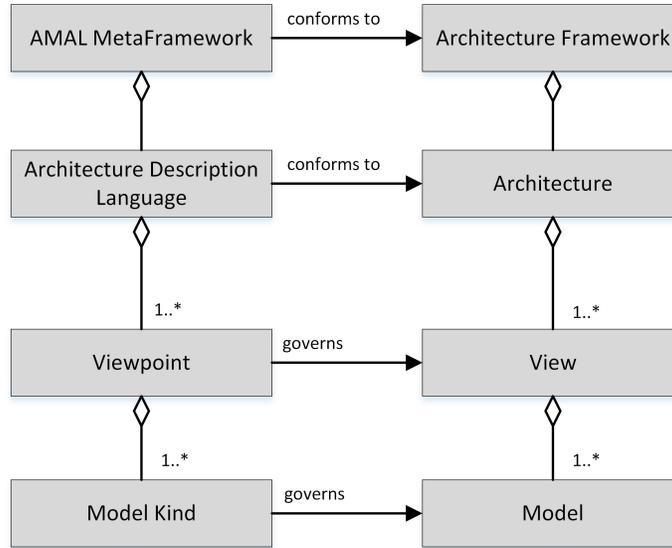


Figure 5-1: Relationship between various concepts related to architecture and framework

introduce a level of indirection such that the stakeholders for system architecture are not known when the framework is defined. However, common practice indicates that framework developers often have in mind known or established stakeholders within the domain of the framework. These stakeholders motivate the set of architecture related concerns that the architecture framework will focus on. A conforming architecture will identifies these concerns and it directly lead to a set of views to be included. Viewpoints governs these views and it establishes notations, model kinds, tools, techniques and methods to be used while creating models. Model kind are conventions for a type of modeling, for e.g., data flow diagrams, Petri nets, and state machines. Various model kinds in robotics and the relationship between viewpoints and views are discussed in Section 5.2.3 and 5.3.3 respectively.

5.2.1 The ISO standard

IEEE 1471:2000, Recommended Practice for Architectural Description of Software-intensive Systems, was the first formal standard by IEEE computer society in the direction for incorporating architectural thinking into IEEE standards and to establish a conceptual framework and vocabulary for talking about architectural issues of systems [2]. Although the idea of an architecture framework was implicit within the

2000 edition, the notion was not defined in the standard. Later a revision proposal was made to define architecture framework explicitly [123]. The proposed definition is:

An architecture framework establishes a common practice for creating, organizing, interpreting and analyzing architectural descriptions used within a particular domain of application or stakeholder community.

ISO adopted the IEEE standard in 2007 as ISO/IEC 42010:2007 [124]. Subsequently ISO and IEEE produced a joint revision, published as ISO/IEC/IEEE 42010 [51], System and software engineering - Architecture description. The first edition of ISO/IEC/IEEE 42010 cancels and replaces ISO/IEC 42010:2007 which has been technically revised. This International standard is used to establish a coherent practice for developing architecture descriptions, architecture frameworks and architecture description languages within the context of a life cycle and its processes.

The fundamental goal of an architecture framework is to codify a common set of architecture practices within a community: for the sake of understandability, commonality and synergy - reducing the need for individual architects to re-invent the wheel; and to promote interoperability. To achieve this goal, the Standard establishes its minimal requirements on architecture frameworks in terms of their content and presentation. Unless otherwise mentioned, in the rest of this chapter, the term 'international standard' refers to ISO/IEC/IEEE 42010:2011(E) [125], Recommended Practice for Architectural Description of Software-intensive Systems.

5.2.2 Architecture Framework and Architecture Description

The international standard defines architecture framework as conventions, principles, and practices for the description of architectures established within a specific domain of application and/or community of stakeholders [51]. Architecture frameworks and architecture description languages (ADLs) are two mechanisms widely used in architecting. The uses of architecture frameworks include: creating architecture descriptions; developing architecture modeling tools and architecting methods; and establishing processes to facilitate communication, commitments and interoperation across multiple projects.

A well-defined architecture allows to reason about system properties at a high level [126]. Typical properties include interface compatibility between components, interaction protocols, performance, resource consumption, conformance to standards, and reliability. Despite its relevance, the practice of architecture description in robotics, is not much used or appreciated. According to the international standards, an architecture description is a work product to express an architecture.

In the life-cycle of the system, an architecture description has many uses:

- a basis for system design and development activities.
- a basis to analyze and evaluate an alternate implementation of an architecture.
- documenting essential aspects of a system, such as, intended use and its environment, principles, assumption, and constraints to guide future system evolution.
- to guide architecture decisions, their rationales and implications.
- specifying a group of systems sharing common features (such as architectural styles, reference architectures and product line architectures)

Architecture descriptions are used to create, utilize, and manage systems to improve communication and co-operation, enabling them to work in an integrated, coherent fashion. Architecture frameworks and architecture description languages are being created as assets that codify the conventions and common practices of architecting and the description of architectures within different application domains.

5.2.3 Model Kinds in Robotics

According to the International standard, conventions for a type of modeling are called model kind. Example of models kinds that are common in robotics are state machines, component-based software architectures, control diagrams, and probabilistic models. *State machines* model the discrete behavior of a robot control system. It decides what activities must be running in the system in concurrent ways, and based on which events the system must switch its overall behavior to another set of concurrent activities. The structure of these switches is modeled by the states being connected through transitions. Structural hierarchy abstracts away how the system reacts to a set of events [127].

A *Bayesian Network (BN)* is a directed acyclic graph $G = (V, E)$ with nodes representing a set of RVs $X_1, X_2, \dots, X|V|$, and edges denoting conditional dependence relationships between random variables. In a nutshell, the structure of the BN encodes the assertion that each node is conditionally independent of its non-descendants, given its parents [128]. BNs are widely used in robotics for localization, and for several learning algorithms [129].

Knowledge networks such as the semantic web, and conceptual graphs are widely used to enable sharing and reuse of knowledge by specifying the terms and relationships among them [130]. In robotics, these kind of models are used to represent knowledge-linked semantic object maps in order to provide a range of information for robots to accomplish complex tasks [131]. In such models, nodes represent facts, data, etc., and edges represent relationships.

Control diagrams such as Cartesian position control and other data flow models such as Simulink [132] consists of node that represent certain functions and edges represents their input/output relationships.

Component based software architectures are approaches used to compose software systems from off-the-shelf and custom components [133]. In such models, nodes represent a piece of software with contractually specified interfaces that implement robotic functionality and the connector represents their interactions [7].

One can see that all the aforementioned models are some form of Hierarchical Graphs with certain additional properties. This is an important realization since our language proposed in the next section has a basic formalism on which the semantic and other properties are added according to the concerned domain.

5.3 Architecture Modeling and Analysis Language

A well-defined architecture framework is a key component for architecture description. Architecture Modeling and Analysis Language (AMAL) together with specific viewpoints and supporting tools forms the core of the operational space in SafeRobots methodology.

A key outcome of IEEE 1471 - Recommended Practice for Architectural Description of Software-Intensive Systems, was the introduction of architecture viewpoints

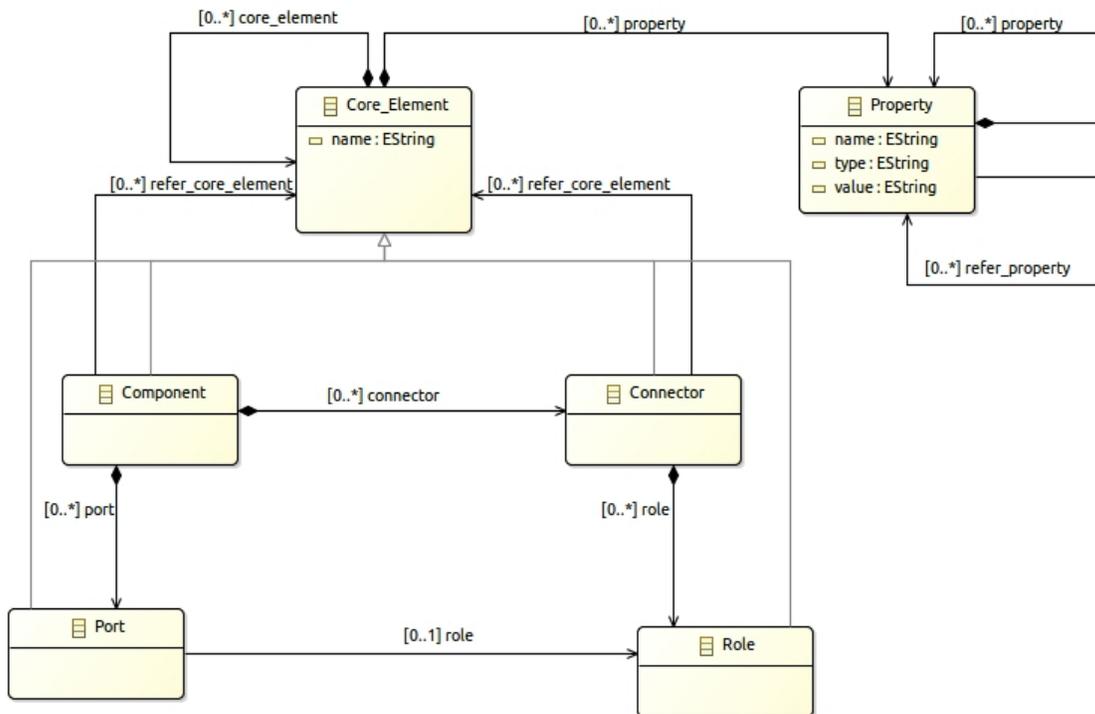


Figure 5-2: Metamodel of Architecture Modeling and Analysis Language (AMAL)

to codify best practices for the creation and use of architecture views within an architecture description. A viewpoint specifies the architecture concerns to be dealt with, the stakeholder addressed, and the languages, models, methods and techniques used to create, interpret, and analyze any view resulting from applying that viewpoint. A conceptual overview of different constructs is shown in Figure 5-1. The approach we have taken is to define a set of basic primitives upon which to construct different domain models.

5.3.1 AMAL Core Elements

The structure of AMAL is defined using four core elements: Component, Port, Connector, and Role. The semantics of the core elements are extended using properties as detailed in the next section. Figure 5-2 shows relationships between AMAL elements using Ecore diagram. The AMAL metamodel provides minimal syntactic rules using constraints, for example, a port cannot be connected to another port without a connector. The description of each core element is as follows:

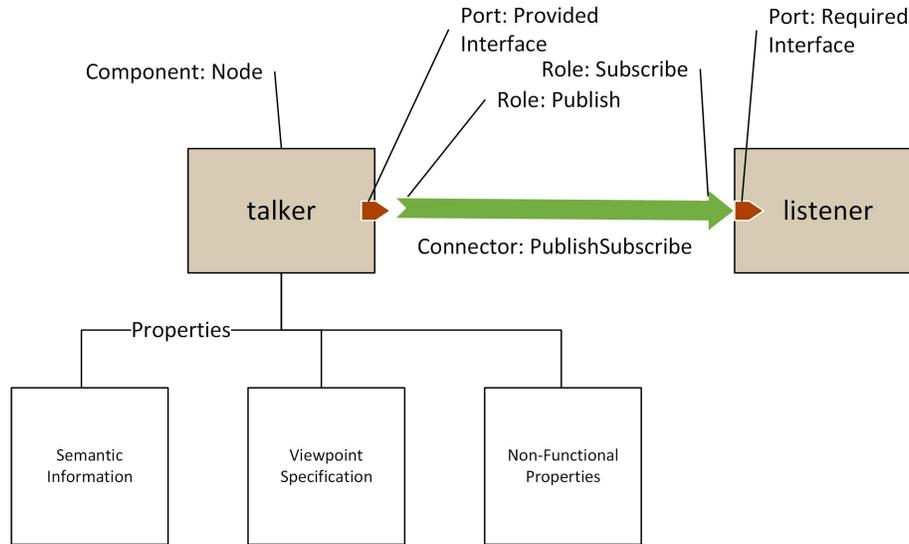


Figure 5-3: Illustration of AMAL model instance with semantic information

- 1) *Component*: The component denotes the computation or a physical entity of the architecture. Components can hierarchically compose other components. A component can be atomic, representing an indivisible unit; or composite, representing a collection of components or an entire system itself.
- 2) *Port*: A Port represents the interface of the component. An interface is the interaction point of the component with the external environment. They are the external visible parts of the component that may facilitate data communication, monitoring, and reasoning of the component composition. A port can also represent viewpoints associated with the components. A detailed explanation of viewpoints is given in Section 5.3.3.
- 3) *Connector*: A connector represents the interaction between the components and are identified as the building blocks of the architecture. Depending on the domain semantics it can represent a data stream, event connections, state transitions, etc.
- 4) *Role*: A role represents the interface of the connectors similar to ports is for components. For example, a remote procedure call (RPC) connector consists of two roles: a callee and a caller role, an event broadcaster connector consists of one broadcaster role and arbitrary number of receiver roles. When connectors connect two components, roles are associated with compatible ports.

As a very simple example, Figure 5-3 depicts a system containing two nodes: talker

```

Component talker
{
    Property Node : semantic_element;
    Port talker_out : @data_connection.publish
    {
        Property Interface: semantic_element = Provided;
    }
}
Component listener
{
    Property Node : semantic_element;
    Port talker_in : @data_connection.subscribe
    {
        Property Interface: semantic_element = Required;
    }
}
Connector data_connection
{
    Property publishsubscribe : semantic_element;
    Role publish;
    Role subscribe[];
}

```

Figure 5-4: Simple publish-subscribe system represented in AMAL

and listener, communicating using publish subscribe mechanism. For illustrative purpose, each element of the system is showed as `core_element: semantic_element` in the Figure 5-3. For example, In the domain that this system represents, a component is a node, a connector represents a publish subscribe communication protocol with associated roles taking the form of publish or subscribe accordingly. The ability to refine the semantics of the core elements is facilitated by `Property` element in AMAL.

5.3.2 Open Semantics Framework

Property associated with core elements of AMAL facilitates the annotation mechanism for extending the semantics of the model elements. For example, it can contain information on its semantic information, non-functional properties, the graphical representation of the model element, etc. The contents of the property are not interpreted by AMAL. It has to be defined by the domain models. A deployment domain may use non-functional property associated with the component to dynamically allocate resources.

Figure 5-4 shows the textual representation of the publish-subscribe system. For

simplicity, only the semantic information is shown as associated property in the model. The attributes of the Property element are shown using the syntax - `Property name: type = value`. For convenience, we represent the property as `{name, type, value}` in the text. It is to be noted that the textual form of the model is shown only for illustrative purpose. The model creation and manipulation is accomplished using the tools associated with specific viewpoints. However, it is important for the viewpoint or tool developer to understand how it will be captured in the model. For example, if the user click on a tool to create a node in the system, the tool will create a component with associated property as `{Node,semantic_element,}`. Furthermore, a property can represent complex properties by composing more properties and can refer to other properties according to the AMAL metamodel. This mechanism along with viewpoints and tools provides the framework with different viewpoints, customized tools and rich visualizations. The following two sections discuss the concept of viewpoints and views, and constraint specification.

5.3.3 Viewpoints and Views

A Viewpoint of a system is a work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns. The concept of view and viewpoints is central to the International Standard. A viewpoint is a way of looking at systems; a view is the result of applying a viewpoint to a particular system of interest. In other words, a view is a description of the system relative to a set of concerns from a certain viewpoint. Similar to the use of modules and packages to manage the complexity of system, we employ viewpoints to manage the complexity of framework. Furthermore, the views can be seen as constructs for the management of architecture. The relationship between the aforementioned notions can be seen as follows:

viewpoints : framework :: views : architecture :: packages : system

AMAL does not enforce any particular views or viewpoints, but provides facilities for specifying viewpoints and further to create tooling support for view creation, interpretation, modification, etc. AMAL strongly encourages the practice of defining

and selecting viewpoints according to the stakeholder's concerns and treating viewpoints as first-class elements of architecture. In the framework modeled using AMAL, each view is governed by exactly one viewpoint. Each view conforms to one set of conventions, but can be possibly using multiple model kinds. However, for specific purposes, viewpoints can be combined in any manner.

There are two different approaches for creating views: constructive and projective approaches. In the constructive approach, views of the system based on model kinds are created individually. These views are then synthesized to an overall model. Model correspondences are performed for integrating views from multiple model kinds. In the projective approach, the views are derived from the overall model. AMAL strongly recommends the projective approach. This is to avoid the problem of view integration and view consistency. Since the semantics of AMAL is open, new model kinds can be extended from the meta-model and hence projective approach is a more convenient approach. Currently, constructive approach has not been applied using AMAL formalism and is not further discussed in this thesis.

Motivated from the benefits of *Separations of Concerns (SoC)* in addressing the inherent complexity in large software intensive systems, multiple views are being used in software architecture. However, they introduce the problem of view integration and model consistency. The notion of multiple views appeared in one of the earliest work titled "Foundations for the study of software architecture" by Perry and Wolf [134]. Although the concept of view is not defined during that time, it states that:

Three important views in software architecture are those of processing, data, and connections ... all three views are necessary and useful at the architectural level.

Most of the views are not independent or fully orthogonal. Elements of one view are connected to elements in another view, following certain rules and heuristics. A general technique called model correspondences introduced by ISO 42010, is used for expressing relations between views. Model Correspondence rules are constraints on two or more architecture models, which is enforced on a model correspondence.

5.3.4 Constraint Specification

The model instance that conforms to AMAL metamodel is not refined enough to provide all the relevant aspects of a specification. The AMAL metamodel itself is developed as an Ecore model which is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints cannot be described in natural language as they will always result in ambiguities. In order to write unambiguous constraints, formal languages called Object Constraints Language (OCL) is used [135].

OCL is a formal language that has been developed as a business modeling language within the IBM Insurance division, and has its roots in the Syntropy method. OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition).

OCL can be used for a number of different purposes. In our context, we use it to specify invariants on model elements in AMAL metamodel and for to describe pre- and post conditions on Operations and Methods. An OCL expression is an invariant of the type and must be true for all instances of that type at any time. For example, if in the context of the Connector element in AMAL model, the following expression would specify an invariant that the number of roles must always be more than 2:

```
context Connector inv:  
self.roles > 1
```

5.3.5 Framework Specification Templates

This section provide templates for documenting framework specification, viewpoints, and views. The template consists of a set of slots or information items followed by a brief description of its intended content, guidance for developing that content, and in some cases appropriate examples are given. Not every slot is needed for specifying the

framework. These templates are designed in such a way as to claim the conformance with the provisions of the International Standard ISO/IEC/IEEE 42010:2011(E).

Framework Template

Framework name: The name of the framework or phrase for identification.

Viewpoint overview: An abstract or brief overview of the framework and its key features.

AMAL Formalism: The definition of the framework in AMAL formalism, which is detailed in Section 5.3.6

System stakeholders: A listing of system stakeholders expected to be users of this framework

Concerns: A listing of the architecture-related concerns that are required for this framework.

Viewpoints: A listing of viewpoints available for the users.

Examples: This section provides examples for the framework developer.

Notes: Any additional information that users of this views might need or find helpful.

Sources: Identify the sources for this views, if any, including author, history, literature reference, etc.

Viewpoint Template

Viewpoint name: The name of the viewpoint or phrase for identification.

Viewpoint overview: An abstract or brief overview of the viewpoint and its key features.

Concerns: A listing of architecture related concerns framed by this viewpoint. This is a critical information as it decides whether the viewpoint is valid or not at the current abstraction level. At the tool level, when the user selects a model element, the current list of valid viewpoints are visible to the user. This usually takes the form of preconditions in viewpoint designing tool. In our framework Accleo Query Language (AQL) is used for defining the precondition. AQL is

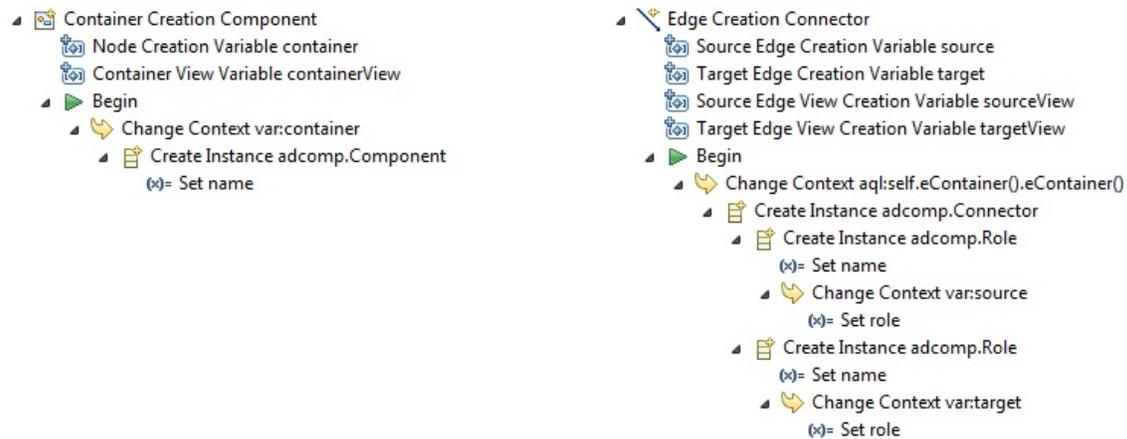


Figure 5-5: Screenshot of the model creating methods implementing in Sirius. The figure shows a component creating on the left and a connection creation method on the right. Note that roles are also created while a connector is created.

a language used to navigate and query EMF models. An example precondition for a viewpoint that shall be valid for a model element that represents a state machine is shown below:

```
[self.property->select(myprop | myprop.type =
'semantic_element')->first().name='statemachine'/]
```

The given AQL query checks whether the property associated with the selected model element is {statemachine,semantic_element,}

Typical Stakeholders: A listing of the system stakeholders expected to be users or audiences for views prepared using this viewpoint.

Views: The identifying name or phrase of the view governed by this viewpoint.

Examples: This section provides examples for the framework developer.

Notes: Any additional information that users of this viewpoint might need or find helpful.

Sources: Identify the sources for this viewpoint, if any, including author, history, literature reference, etc.

View Template

View name: The name of the viewpoint or phrase for identification.

View overview: An abstract or brief overview of the viewpoint and its key features.

Model Kinds: Each model kind specified by this view is identified in this section. For each model kind used, describe the convention to structure the properties associated with AMAL core elements. Key modeling resources that the view makes available including how the model elements are visually rendered by the view and determine the vocabularies for constructing the view. These includes the tools for operations such model creation, deletion, making connections, editing properties, etc.

For example, for specifying the source and target of a connector are specifying as below in AQL:

```
domain class: amal.connector
source finder expression:
[self.eGet('role')->first().eInverse()/]
target finder expression:
[self.eGet('role')->last().eInverse()/]
```

For better understanding, model kinds can also be presenting in the form of metamodels specifying the structure and vocabulary.

A model kind may be documented in a number of ways such as:

1. by specifying a metamodel that defines its core elements and their relationships.
2. by providing a pseudo code that conforms to AMAL metamodel by specifying the associated properties.
3. via a language definition or by reference to existing modeling language.
4. or by combination of these methods.

Operation on views: Operations define the methods to be applied to views or to their models. Operations can be divided into categories:

- *Creation methods:* These are the means by which the model elements represented by this view are created. This is usually in the form of process guidance and are specific to the tool used for creating viewpoint. Since we adopt Sirius tool, the operations are implemented using the user interface provided by the tool.
- *Interpretive methods:* These are the means by which view are to be understood by the system stakeholders.
- *Analysis methods:* These are used to check, reason and transform, predict, apply and evaluate results from this view.
- *Design methods:* These are used to realize or construct systems using information from this view.

Examples: This section provides examples for the framework developer.

Notes: Any additional information that users of this views might need or find helpful.

Sources: Identify the sources for this views, if any, including author, history, literature reference, etc.

5.3.6 AMAL Specification Formalism

We adopt a formalism to specify the framework. The formalism defines the participating domain models, viewpoints, views, and their relationships.

Definition 1: *An architectural framework X , that conforms to AMAL formalism is a tuple $\langle M_X, DM, IM, \mathcal{R}_X^{AMAL}, \mathcal{R}_{AMAL}^{DM}, \mathcal{R}_{AMAL}^{IM} \rangle$, where*

- M_X is the model defined in framework X .
- DM is the domain model where the framework X has conceptual relationships.
- IM is the implementation model that framework X supports.

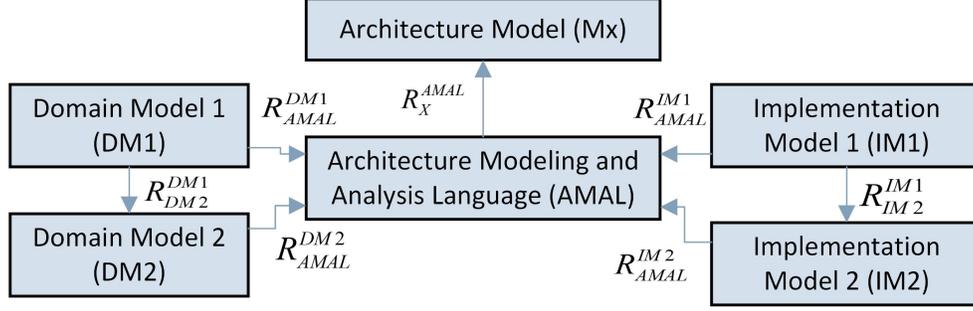


Figure 5-6: AMAL Model Relationships

- \mathcal{R}_X^{AMAL} is a relation that associates AMAL model elements to the framework X .
- \mathcal{R}_{AMAL}^{DM} is a relation that associates model elements from DM to AMAL model elements.
- \mathcal{R}_{AMAL}^{IM} is a relation that associates model elements from implementation model IM to AMAL model elements.

Robotics domain is heterogeneous with domains ranging from conceptual domain such as perception, planning, control, decision making; computational domain consisting of discrete, continuous; software domain consisting of communication middlewares, operating systems, etc. Hence, the meta-framework architecture should be extensible in order to incorporate different domain models. The composed domain models should be semantically compatible. For example, assume a model incorporates concepts from two domains a and b . In domain a , the modeling element connector represents a computation process, and in domain b , the connector represents an instantaneous transition between two states. These two domains are semantically incompatible unless the conflict between them is resolved, say by assigning the computational process to component.

In Figure 5-6, model relationships among multiple domain models and implementation models, in AMAL formalism are shown.

Definition 2: The viewpoints V_n , of an architectural framework X is a set of tuple $\langle M_{V_1}, M_{V_2}, \mathcal{R}_{V_2}^{V_1}, \mathcal{R}_{V_1}^{V_2} \rangle$, where

- M_{V_1} is the model defined in viewpoint V_1 .
- M_{V_2} is the model defined in viewpoint V_2 .

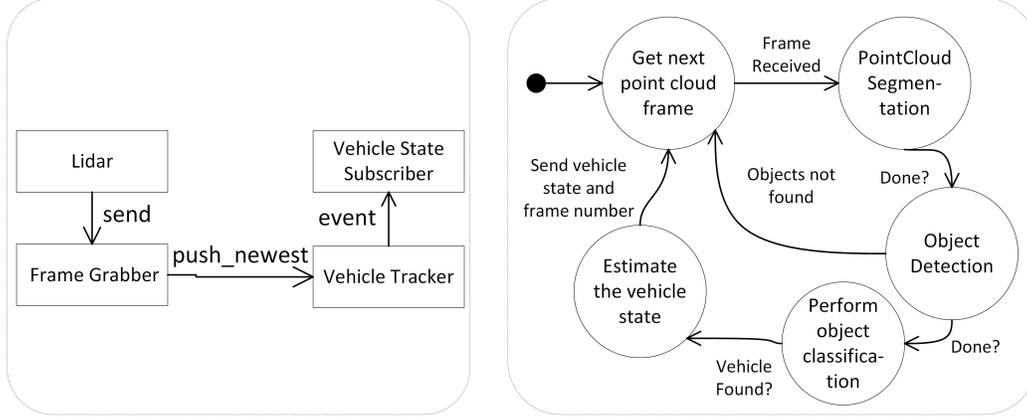


Figure 5-7: A prototype model of vehicle tracking system designed using the TrackX framework. Structural view (left) and Coordination view (right) of the system is shown

- $\mathcal{R}_{V_2}^{V_1}$ is a relation that associates model elements from V_1 to V_2 .
- $\mathcal{R}_{V_1}^{V_2}$ is a relation that associates model elements from V_2 to V_1 .

Architectures are specified as a collection of viewpoints and their relationships among them. A viewpoint isolates independently solvable aspect of a system in order to manage complexity. For example, a deployment view of the system addresses only the concerns related to initiating the execution of components in a particular order. The relation between the viewpoints is an important requirement for an analyzable architecture. The relationships are loosely coupled (uni-directional) or tightly coupled (bi-directional). For example, a coordination view that models ‘when components should communicate’ should be aware of the configuration view that models ‘who communicates with whom’ of a system, while configuration view need not know about the coordination model.

5.4 Case Study on an Example Framework

In this section, we describe how a hypothetical framework (say ‘TrackX’) can be designed using our meta-framework. TrackX is a framework for designing software system for tracking vehicles using lidar. It uses concepts and algorithms from perception and object tracking domain. We have selected only minimal concepts from these domains for the simplicity of explanation. The framework shall support model-

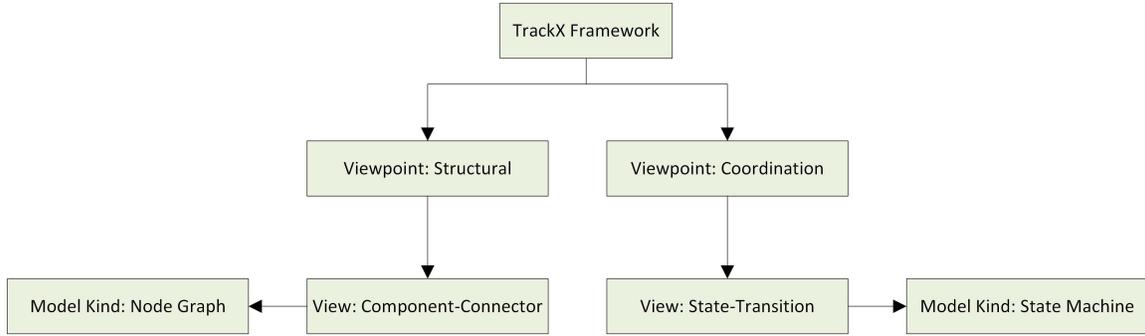


Figure 5-8: An illustration of the proposed framework topology

ing a system using two complementary views: structural viewpoint and coordination viewpoint. The structural view models the interconnection of various computational algorithms and its communication aspects, and the coordination view models the coordination of these computations using state transition formalism. TrackX supports ROS based implementation and can be simulated using Gazebo simulator.

An example model of system in the TrackX framework is shown in Figure 5-7. A high-level structural model and its coordination model of a vehicle tracking system modeled in TrackX framework. The structural model consists of four components: Lidar, Frame grabber, Vehicle tracker, and a Vehicle state subscriber. The component denotes a computational process and can be hierarchically composed, i.e., vehicle tracker component can be expanded to represent the system a lower abstraction level with more detailed view of algorithms involved. The connector represents the communication pattern (using connector label) between the connected components. The coordination view models the system behavior using states and transitions. The states represents the activation/deactivation of computational processes in structural view and transition coordinate the state changes. In order to concentrate on the overall elements of framework specification in the operational space, in this case study, details regarding the formal semantics of state transition model and activation/deactivation of computational processes in the states will not be further discussed. Furthermore, code generation techniques and AMAL to implementation model transformation details are discussed in Part 3 of this thesis.

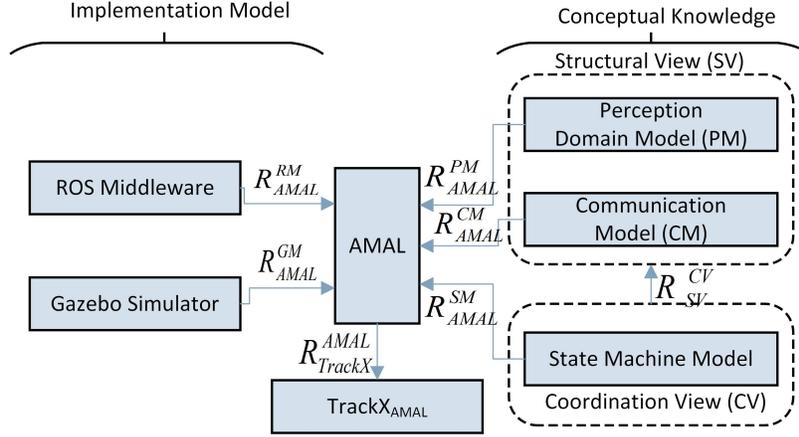


Figure 5-9: Framework model of a TrackX framework

5.4.1 TrackX - Framework specification

Framework name: TrackX

Framework overview: TrackX is a framework for designing software system for tracking vehicles using lidar. An illustration of different viewpoints and associated views are shown in Figure 5-8. TrackX requires knowledge from three conceptual domains and two implementation domains. The conceptual domains are Perception, Communication, and State Machine Formalism, and implementation domains are ROS middleware and Gazebo simulator.

Viewpoints: structural, coordination.

AMAL Formalism: The TrackX framework in AMAL formalism is defined as:

With reference to the domain models, the framework can be defined as:

$$TrackX_{AMAL} \Rightarrow \langle M_{TrackX}, PM, CM, RM, GM, \mathcal{R}_{AMAL}^{TrackX}, \mathcal{R}_{AMAL}^{PM}, \mathcal{R}_{AMAL}^{CM}, \mathcal{R}_{AMAL}^{RM}, \mathcal{R}_{AMAL}^{GM}, \mathcal{R}_{PM}^{CM} \rangle.$$

With reference to the required viewpoints, the framework can be defined as:

$$TrackX_{AMAL} \Rightarrow \langle M_{SV}, M_{CV}, \mathcal{R}_{SV}^{AMAL}, \mathcal{R}_{CV}^{AMAL}, \mathcal{R}_{CV}^{SV} \rangle.$$

A pictorial representation of the relationship between different domains is shown in Figure 7-10. Each element is explained as follows:

M_{SV} denotes the model in the structural view (SV) of the system.

M_{CV} : denotes the model in the coordination view (CV) of the system.

Different domain models and their relationships are described below:

Perception Model (PM): Perception model captures the conceptual knowledge of the perception domain. It contains abstract knowledge (solution space) regarding various computational algorithms, its dependencies, etc. The domain model specified in SSML contains meta-data on algorithms, their execution sequence, non-functional properties, expected quality of service, their constraints, etc.

Communication Model (CM): Communication models capture the knowledge regarding several patterns that cover the communication requirements for component interactions. In this example, we use the communication patterns proposed by the authors of [15]. It consists of a minimal set of communication patterns required by robotic software component interaction: send, query, push_newest, push_timed, and event. It supports communications such as synchronous, asynchronous, publish/subscribe, client/server, and dynamic wiring.

ROS Model (RM) and Gazebo Model (GM) represent the implementation model of ROS middleware and Gazebo simulator respectively. The model of ROS is captured as metamodels. It contains meta-information regarding supported communication pattern, deployment, etc.

\mathcal{R}_{AMAL}^{CM} represents the relation between communication model and the elements in the AMAL model. Figure 5-10 shows the mapping between CM model elements to model element: **Connector** in the structural view. Intuitively, the communication patterns are mapped to connectors in SV. It also details the the number of roles that each connector is associated with, depending on the patterns, for example, a connector that represent push_timed can have 1 publisher role and ‘n’ number of subscriber role.

\mathcal{R}_{AMAL}^{PM} represents the relation between perception model and the structural view in AMAL formalism. Figure 5-10 shows the mapping between PM model elements to model elements in structural view. Since the PM is modeled by SSML language, the mapping is between abstract elements of SSML with that of AMAL. In SSML, the connector represents the computation process, Dispatch_Gate represent the interconnection between computations, Ports represent abstract data types and NFP represent the non-functional property of the algorithm. The relation table maps $Connector^{SSML}$ to $Component^{SV}$, $Dispatch_Gate^{SSML}$ to $Connector^{SV}$.

\mathcal{R}_{PM}^{CM} represents the relation between communication domain and perception domain

model. It captures the semantic compatibility between the domain and conflicts (if any) in the relationship. For example, the previous two relation model maps communication pattern in CM and Dispatch_Gate in PM to the same model element Connector in SV. The relation \mathcal{R}_{PM}^{CM} specifies that these two concepts are semantically compatible since Dispatch_Gate and communication pattern represent the interaction between computations in their respective domains.

\mathcal{R}_{AMAL}^{CV} represents the relation between state machine formalism in the coordination view and the model elements in AMAL formalism. Figure 5-10 shows the mapping between SM model elements to model elements in coordination view. States are mapped to component and transitions to connectors in relationship model. It is to be noted that components in the structural view and coordination view need not be semantically compatible since it resides in two different views.

\mathcal{R}_{SV}^{CV} represents the relation from coordination view to structural view. A component (state) in CV is mapped to arbitrary number components in the SV. It means that a state influence (activate/deactivate) one or more components in structural view. In addition, the relationship is unidirectional from CV to SV denoting loose coupling between the two views. It is recommended and in fact comparatively easy to provide only a unidirectional mapping because the structural model can express different behavior depending on the coordination model.

\mathcal{R}_{AMAL}^{RM} represent the relation from ROS middleware implementation model to AMAL formalism. For convenience, we map the model elements from structural view model elements in AMAL to ROS since coordination view do not influence the ROS node interconnection structure directly. \mathcal{R}_{AMAL}^{GM} and \mathcal{R}_{GM}^{RM} are not shown explicitly as Gazebo internally depends on ROS middleware and hence have the similar relation as \mathcal{R}_{AMAL}^{RM} . $\mathcal{R}_{TrackX}^{AMAL}$ captures the relations that are not explicitly captured by other model relationships. In this case, all the relations have been implicitly captured by the model relationships.

5.4.2 Structural viewpoint specification

Viewpoint name: Structural viewpoint

Viewpoint overview: The structural view models the interconnection of var-

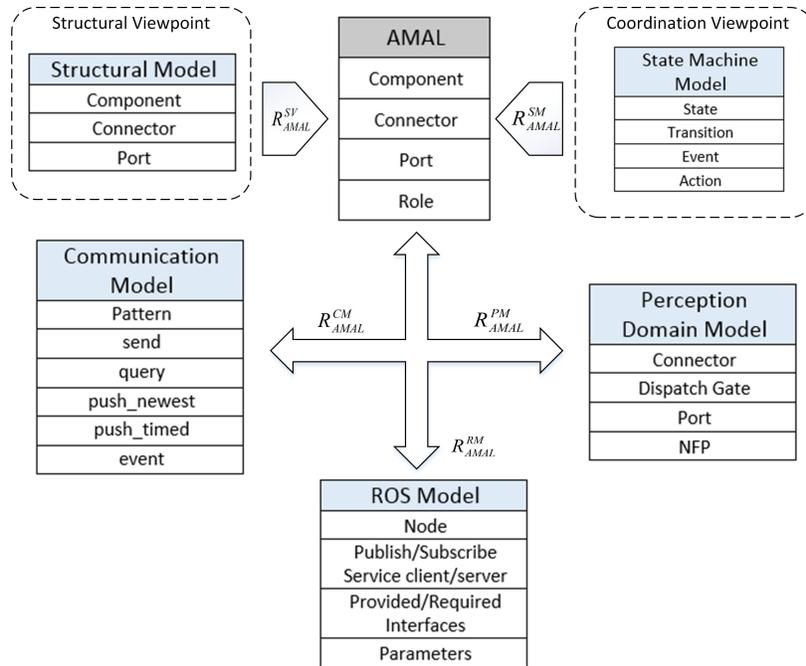


Figure 5-10: Model relations between Communication Domain Model (CM), Perception Domain Model (PM), ROS Model (RM), and AMAL core elements

ious computational algorithms and its communication aspects. The component denotes a computational process and can be hierarchically composed, i.e., a high-level component can be expanded to represent the system a lower abstraction level with more detailed view of algorithms involved. The connector represents the communication pattern (using connector label) between the connected components.

Concerns: Component-based system architecture.

Views: Component-Port-Connector view

5.4.3 Coordination viewpoint specification

Viewpoint name: Coordination viewpoint

Viewpoint overview: The coordination view models the coordination of these computations using state transition formalism. It captures the system behavior using states and transitions. The states represents the activation/deactivation of

computational processes in structural view and transition coordinate the state changes.

Concerns: System Behavior

Views: State-transition view

Examples: Figure 5-11 shows an example view of the model.

5.4.4 Component-Port-Connector view specification

View name: Component-Connector view

View overview: The structure of the system is modeled as nodes and their interconnections using edges.

Model Kinds: Node Graph

Operation on views:

- *Creation methods:* The tools for creating models shall be provided in tools palette. The model element creation of nodes and connectors will be provided. The model deletion operation shall be implemented as key bindings.
- *Design methods:* The view shall allow the creation of sub nodes and hence the tool shall disable the capability to create composite nodes.

5.4.5 State transition view specification

View name: State-transition view

View overview: The behavior of the system is modeled using state machine formalism.

Model Kinds: State Machine

Operation on views:

- *Creation methods:* The tools for creating models shall be provided in tools palette. The model element creation of states and transitions will be provided. The model deletion operation shall be implemented as key bindings.

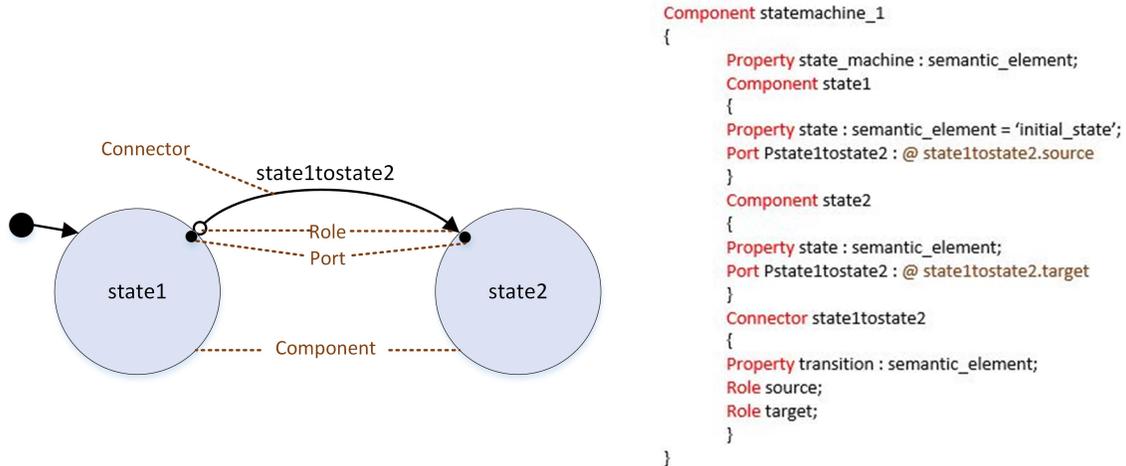


Figure 5-11: Visualization of state transition view and its corresponding AMAL model

- *Design methods:* The view does not allow to create substates and hence the tool shall disable the capability to create hierarchical states.

Examples: An example of the state transition view and the corresponding model is shown in Figure 5-11.

5.5 Related Works

Architecture Description Languages

ADLs developed in terms of the international standard include Rapide [136], Wright [137], SysML [138], and ArchiMate [139]. Our approach of providing common syntax with semantic content, whose semantic enrichment is provided specialized sub-domain specific language is conceptually related to that of ACME [140]. ACME is an interchange language for software architecture that provides structural core that represents commonalities between various Architectural Description Languages (ADL) [141]. ACME uses annotations to add semantic information by sub-languages. However, the objective is to serve as a common representation for software architectures and that permits the integration of diverse collection of independently developed architecture analysis tools. The Architecture Analysis and Design Language (AADL) is a modeling language standardized by Society of Automotive Engineers (SAE) to specify and analyze software architectures for complex real-time embedded systems.

Views and Viewpoints

There are number of existing approaches utilizing viewpoints and views in software systems architecture. The earliest work on viewpoints appeared in Ross' Structured Analysis in 1977 [142]. Many first generation software engineering techniques used functional and data viewpoints. Nuseibeh, Kramer and Finkelstein treated viewpoints as first class entities, with associated attributes and operations [143]. Clements et al. introduced the term *viewtype* for categorization of viewpoints [144]. Three categories of viewpoints are described in their work: module, component, and connector, and allocation viewtypes. Kruchten's 4+1 view model takes four views as its starting point [145]. The integration of these viewpoints is accomplished through a fifth viewpoint which is a set of scenarios used to validate the other view and their interactions. The authors of [146] investigated the composition of heterogeneous styles in architectures. They characterized a style (which is actually a view in our terminology) as a collection of constraints on the structure, behavior, and resource usage of the components and connectors in a software system.

Architecture Meta-Frameworks

The authors of [147] introduced Meta Architecture Description Language (MADL) to define, comment, document, compare architectures, in particular semiformal architectures. Their main proposal is for unifying ADLs in the context of software architectures and to provide reflexivity in architecture metamodeling. However our work is at the framework level and we introduced common primitive elements with semantic extensibility to manage complexity of multiple viewpoints and views. Rich Hilliard introduced *decorative stance* as an alternative to constructive views in frameworks. Our work is more similar to this approach. The approach is to decorate a primary representation with attributes pertaining to other concerns, rather than separate concerns. This is to address multiple-view problem in architectures, that lead different specification describe different, but overlapping issues. In another work, behavioral semantics are integrated in structural formalism [148]. Our method does not enforce any such specific views and it entirely depends on the framework developer. The Practical Architecture Method prescribes no particular view; instead the determination of useful views is part of the architect's work and is driven by concerns related to the specific system [149].

5.6 Conclusion

Making architecture meta-framework a point of conformance opens new possibilities for interoperability and knowledge sharing in the architecture and framework communities. We tried to make a first step in this direction by proposing common model and provided a systematic approach that helps in specifying different aspects and their interplay in a framework. The multi-domain architecture modeling helps to build integrated intelligent robotic systems. We have presented an architecture description language Architecture Modeling and Analysis Language to model system architectures based on heterogeneous architectural paradigms. The semantic extensibility feature of AMAL helps to refine the semantic knowledge, and to specify inter-domain relationships. It provided a homogeneous development environment irrespective of framework or middleware and thus promoting faster adoption among software developers and system engineers.

5.7 My Contributions

The contributions of my thesis brought in this chapter are:

1. Formalized the robotic framework design and development process in order to build custom frameworks and integrate existing best practices in architecture development.
2. A meta-framework language, AMAL is proposed that provides a minimal set modeling elements.
3. The Open Semantic Framework helps to modify and extend the semantics to incorporate different domain concepts and to capture the relationships and identify the conflicting domain semantics.
4. The model relationships enables integration of various domains in the framework and support building complex robotic systems.
5. Our approach of framework specification and development is suitable to compare various architectures. Describing various architecture with the same formalism facilitates their comparison and analysis.

6. Our approach of having minimal primitive elements with associated properties promotes reuse of software
 - *reuse of model to model transformation code*: Our framework is defined as a collection of viewpoints and views, and the relationship between the domain models. New frameworks can be created with different set of domain models and mixing it in different fashions. Since all the domain models use the same set of primitive elements, the model to model transformation code can be easily reused in the new framework.
 - *reuse of code generation templates*: The models needs to be transformed to executable code at some point of time in the development process. Such model to text transformation templates can be reused across frameworks.
 - *reuse of graphical editors*: The tools required for model parsing and visualization can be reused. The views are generated by applying viewpoints to the model. The associated tools for model creation and manipulation can be easily reused. For example, in the new framework, if there is a viewpoint to model state machine, this can be reused from another framework if it uses the same model kind.
 - *reuse of infrastructure*: The usefulness of the framework is considerably increased if it provides facilities for logging, messaging, tracing, debugging, etc. Such supporting tools can be easily imported and reused in our approach.
7. The potential benefit of having a common base for the models is developing *application and platform neutral* meta-architecture. It promotes reuse in system modeling in robotics and in the development of reusable tools and generic algorithms.
8. Irrespective of the framework, the *look and feel* of the graphical interfaces will be the same. The tools provided with the frameworks provides homogeneous user interfaces and thus promoting faster adoption among users.

Part II

Framework Implementation and Extended Applications

Chapter 6

Framework Implementation

Give us the tools and we will finish
the job

Winston Churchill

6.1 Introduction

In a model driven approach, the vision is that models become artifacts to be maintained along with the code. This will only happen if the benefit obtained from producing the models is considerably higher and the effort required to keep them in line with the code is considerably lower than the current practice. Models are valuable as tools for abstraction, for summarizing, and for providing alternative perspectives. The value is greatly enhanced if models become tangible artifacts that can be simulated, transformed, checked etc., and if the burden of keeping them in step with each other and the delivered system is considerably reduced. Tooling is essential to maximize the benefits of having models, and to minimize the effort required to maintain them. Specifically, more sophisticated tools are required than those in common use today, which are in many cases just model editors. In this chapter, we detail the different tools used and the stages involved in implementing the framework by the developer of the tool. Section 6.2 provides a brief overview on Eclipse Modeling Framework and discusses about selected tools supporting the framework development. Different stages in our framework implementation process are detailed in Section 6.3.

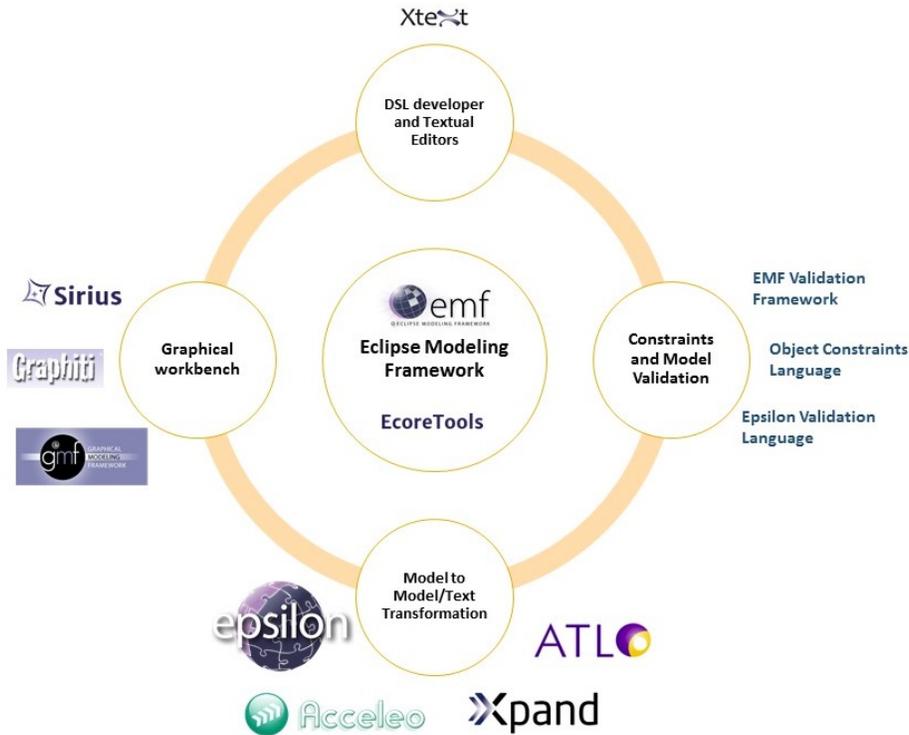


Figure 6-1: Eclipse modeling framework and supporting tools

6.2 Eclipse Modeling Framework

Eclipse is an open source software project dedicated to providing a robust, full-featured, and commercial-quality platform for developing and supporting highly integrated software engineering tools [150]. The Eclipse platform defines a set of frameworks and common services that collectively make up the "integrationware" required to support a comprehensive tool integration platform. Except the small Eclipse runtime kernel, all the platform components are plug-in tools integrated seamlessly through predefined extension points [151]. Fundamentally, Eclipse is a framework for plug-ins. Beside its runtime kernel, the platform consists of the workbench, workspace, help, and team components. Other tools plug into this basic framework to create a usable application. Plug-ins can also define new extension points for others to extend. For example, a group of plugins implements the workbench user interface.

The Eclipse Modeling Framework (EMF) is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification point of view as described in XML Metadata

Interchange (XMI), EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor [152]. Associated with EMF, there is an ecosystem of plugins that assist the developer during different phases of model based tool development as illustrated in Figure 6-1. A brief overview on the selected tools is given below. Detailed discussion on each tool is provided in Appendix A for reference.

Ecore Tools for Metamodeling

Ecore Tools plugin provides a complete environment to create, edit, and maintain Ecore models. This component facilitates handling of Ecore models with a Graphical Ecore Editor and bridges to other existing Ecore tools. It is a powerful tool for designing Model-Driven Architecture (MDA), which can be used as a starting point for software development. Typically, the process is to define the objects (of type EClass) in the domain of application, their attributes, and their relationships. In addition, specific operations that belong to these objects are also defined using the EOperation model element. By default, EMF will generate skeletons, or method signatures which can be modified according to our use.

Sirius for designing graphical workbench

Sirius allows to create custom graphical modeling workbenches by leveraging the Eclipse Modeling technologies, including EMF and Graphical Modeling Framework (GMF) [153][154]. The modeling workbench created is composed of a set of Eclipse editors (diagrams, tables and trees) which allow the users to create, edit and visualize EMF models. All graphical characteristics and behaviors can be configured with a minimum technical knowledge. This description is dynamically interpreted to materialize the workbench within the Eclipse. Since no code generation is involved, the specifier of the workbench can have instant feedback while adapting the description. Once completed, the modeling workbench can be deployed as a standard Eclipse plugin. In our reference implementation, Sirius is used for developing graphical user interfaces and viewpoint development.

Epsilon for Model transformation and Code Generation

Epsilon provides consistent and interoperable languages for common model-driven engineering activities such as code generation, model validation and model transfor-

mation [155]. All languages in Epsilon build on top of a common expression language that promotes code across your model-to-model transformations, code generators, validation constraints etc .

6.3 Framework Implementation Process

In this section, we detail different stages in which the framework developer can specify, implement, and deploy the framework tool.

Stage 1: The first step is to specify the framework using the templates and formalism proposed in Chapter 5. This process identifies the domains involved, required viewpoints in the framework, target platform and middleware, etc. The domain models and their relationships are formally specified using our AMAL based approach. Based on the artifacts generated in this stage, the developer creates appropriate Eclipse plugins to implement the framework.

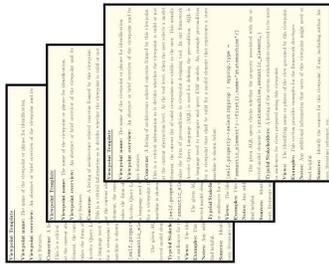
Stage 2: In this stage, the developer implements the framework specification. As discussed in the previous section, Eclipse Modeling Framework is used for implementing the framework.

Stage 2.1: The developer creates meta models of identified domain models involved in the framework. This can be performed in two ways:

1. By creating an Ecore metamodel and mapping its elements with the respective elements in AMAL metamodel. For this procedure, the developer develops the metamodel based on Ecore and generates necessary java templates using the tools by EMF. The developer then develops Model transformation tool to transform the model to our AMAL compatible model. However, we recommend the second procedure, as it eliminates the need to create a model transformation tool.
2. By using the facility provided by Sirius to create model elements directly based on AMAL metamodel. For when the user create a component that represents a state machine, the property of the component will be having name as *semantic_element* and value as *state_machine* as shown below.

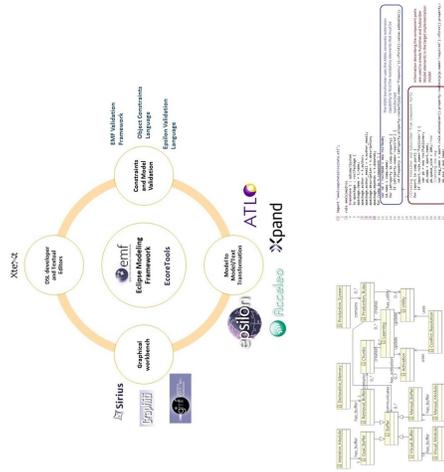
Property state_machine : semantic_element

Stage 1



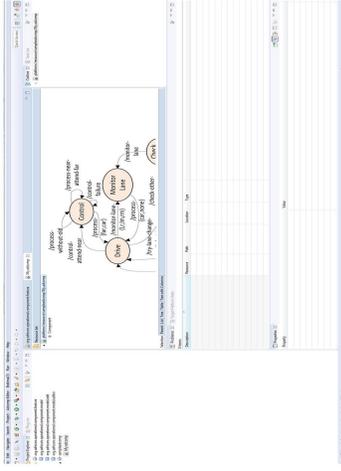
Framework Specification

Stage 2



Eclipse Tools, Metamodels, Transformation Templates

Stage 3



Tool Deployment

Figure 6-2: An Overview of the tool development process

Sirius supports dynamic changes of tool designs even while the development is in progress and the developer can take advantage of this feature to develop tools much faster.

Stage 2.2: The developer provides necessary constraints on the model elements. There are three options available to perform validation on the constraints:

1. **Object Constraints Language (OCL):** Although many approaches have been proposed to enable automated model validation, the OCL is the de facto standard for capturing constraints in modeling languages specified using object-oriented meta modeling technologies. While its powerful syntax enables users to specify meaningful and concise constraints, its purely declarative and side-effect free nature introduces a number of limitations in the context of a contemporary model management environment. In OCL, structural constraints are captured in the form of invariants. Each invariant is defined in the context of a meta-class of the metamodel and specifies a name and a body. The body is an OCL expression that must evaluate to a Boolean result, indicating whether an instance of the meta-class satisfies the invariant or not. Execution-wise, the body of each invariant is evaluated for each instance of the meta-class and the results are stored in a set of $\langle \text{Element}, \text{Invariant}, \text{Boolean} \rangle$ triplets. Each triplet captures the Boolean result of the evaluation of an Invariant on a qualified Element [135].
2. **Sirius Validation Mechanism:** Sirius allows to define custom validation rules, which will only be applied while the user creates the model in the graphical modeler. To define validation rules the developer first create a Validation element inside the diagram, and then add one or more Semantic Validation Rule or View Validation Rule. Both kinds of rules are similar, but semantic rules check the structure of the underlying semantic model, which view validation rules can check the structure of the representation itself. When a rule is violated, a marker will appear on the diagram on the problematic elements and in the Problems view [156].
3. **Epsilon Validation Language (EVL):** EVL can be used to specify and evaluate constraints on models of arbitrary metamodels and modelling technologies.

EVL constraints are quite similar to OCL constraints. However, EVL also supports dependencies between constraints (e.g. if constraint A fails, don't evaluate constraint B), customizable error messages to be displayed to the user and specification of fixes, which users can invoke to repair inconsistencies. Also, as EVL builds on Epsilon Object Language, it can evaluate inter-model constraints (unlike OCL)[155].

In our case, we suggest to use any of the aforementioned methods to specify constraints depending on usecase. We primarily used Sirius validation mechanism as it is easy to use without any external dependencies. In some cases, OCL is also used for specifying the constraints specifically while creating a standalone metamodel. EVL is mainly used for validation during model transformation process.

Stage 2.3: Implementing the graphical modeling workbench is the process that consumes most of the time according to our experience. Initially, we used Graphiti [157] for creating the graphical user interfaces. Later, we switched to Sirius that have inbuilt support for implementing the viewpoints, different graphical options, and validation mechanisms.

Stage 3: Eclipse is an open platform and it is designed to be easily and infinitely extensible by third parties. At the core is the Eclipse SDK, various products/tools around this SDK. A plugin is a small unit of Eclipse Platform that can be developed separately. It must be noted that all of the functionalities of Eclipse are located in different plugins. The resulting framework that we implemented is a set of plugins as shown in Figure 6-2. In order to promote reusability, the developer should to make it more modular in the form of several plugins. For example, if the framework has a structural viewpoint and behavioral viewpoint, these will be implemented as two separate plugins. The specific viewpoint will be available to the user only if the corresponding plugin is installed. This also helps in mix and match of frameworks according to the user's need. The plugins can also be provided in the form of a repository, from where the user selects and installs according to the requirement. The only constraint is that necessary relationship and model transformations should be accordingly available in the tool.

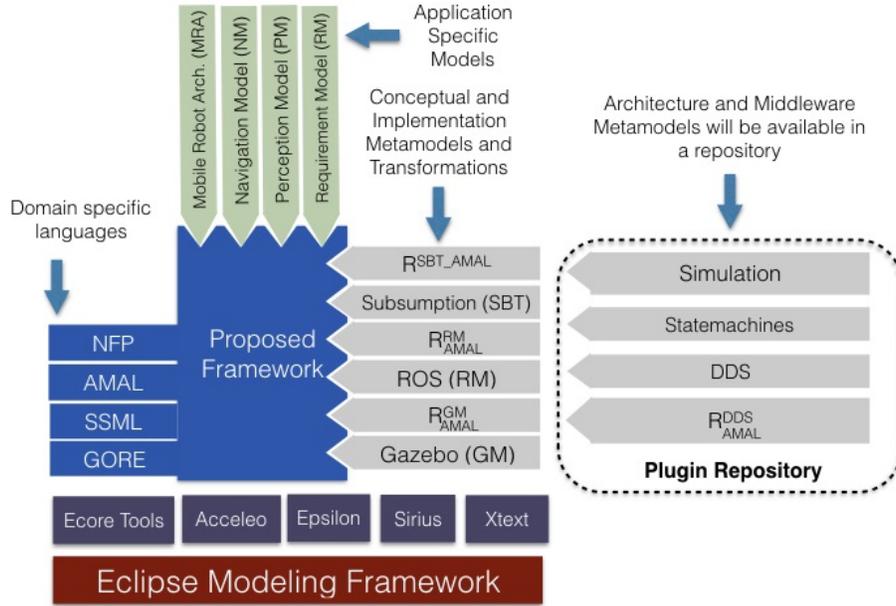


Figure 6-2: Structure of the framework with appropriate plugins

6.4 Conclusion and Contributions

In this chapter, an overview on the Eclipse Modeling Framework and several supporting tools were provided. Available tools and technologies in different stages of framework development are also discussed. Several macro and micro processes are required for systematic engineering of different models. Macro process outlines the order in which models are developed, transformed, and coordinated. Micro process is mostly concerned with producing a particular model. We have detailed these processes based on our reference implementation and suggestions based on our experiences. We have carefully avoided certain tools in the process which requires high level of knowledge in the domain of Java object oriented language, EMF, and Eclipse plug-in development. Our recommended tools simplifies the product, reduces design time and rapidly increases the overall productivity of building the framework. However, it is to be noted that our methodology does not enforce any particular formal process in developing the model. This can be considered as one of our future works.

Chapter 7

Extended Applications

7.1 Introduction

Framework design and tools development that support the framework are expensive in terms of development time and expertise. Therefore, frameworks should be developed only when many applications are going to be developed within a specific problem domain, allowing to save time by reusing the tools and to recoup the time invested to develop them. The purpose of this chapter is to provide some insights on framework development using example applications in which our methodology is applied. The chapter is organized as follows: In Section 7.2 we discuss some of the existing methods for framework development. Section 7.3 and 7.4 discuss two case studies in which specific processes from our methodology are detailed.

7.2 Existing Framework Development Methods

Framework Development based on Generalization

When a framework and support tools are designed, the main concern is to recognize things that should be kept flexible. These are called the hot spots of the framework [158]. In order to identify variant parts, some of the following questions should be answered [159]:

1. Which concepts of the problem domain exist in variants and should be treated uniformly?

2. Is it possible to find a concrete concept that can be generalized?
3. Which parts of the system might change?
4. Where might a user want to hook custom code into the framework? and How?

The author of [160] suggests a two-phase design method to build an initial version of a framework; the first phase is called problem generalization and the second phase is called framework design. Problem generalization starts from the specification of a representative application of the intended framework, and generalizes it in a sequence of steps into the most general form. During the second phase the generalization levels of the previous phase are considered in reverse order leading to an implementation for each level. One of the limitations of this method is that essential tasks to problem generalization or framework design are not defined concretely. Also, concrete guidelines to identify hot spots are not described in this method.

Framework Development based on Application Experiences

This method is a pragmatic framework development approach [161]. The first step is to develop some candidate applications in the problem domain. Then, the next step is to identify the common features in these applications and extract these into a framework. To evaluate whether the extracted features are the right ones, redevelop the these applications based on the framework. The advantage of this method is that the framework development is easy, because we extract commonalities from pre-built applications through previously building applications. Whereas it requires much of time to build framework in the case of complex or large domain, and it is difficult to evaluate whether extracted commonalities are the right ones.

Framework Development based on Domain Analysis

The first activity is to analyze the problem domain so as to identify and understand well-known abstractions in the domain [161]. This is similar to the one discussed for lane keep assistance example in Chapter 5. Analyzing the domain requires analyzing existing applications and the analysis of existing applications will also take a large portion of the budget. After the abstractions have been identified, the next step is to develop the framework together with a test application and modify the framework if necessary. Subsequently, a second application based on the framework is developed. Then, through domain analysis, commonalities are identified. In this direction, we

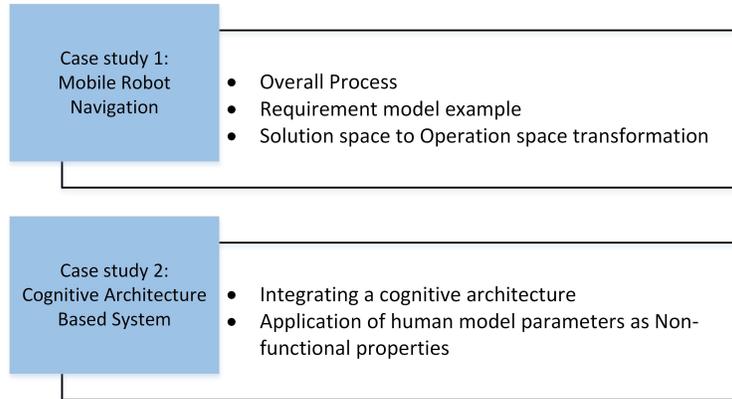


Figure 7-1: Illustration of features discussed in the respective case studies

can extract well-defined abstract concepts, whereas this method requires much of time and budget because of domain analysis.

In this chapter, we present two applications in which our methodology and framework development process are explained. We have adopted a mix of the above three approaches for framework development in our case study. We start with problem analysis based on the case study where the required domains are identified, and then we try to generalize using past experiences in framework specification to finally to develop the tool. Specifically, our two case studies addresses the following aspects:

- Our first case study provides an overall process of framework development from requirement specification to code generation. Requirement modeling in problem space is provided as an example in this case study. For experimental purpose, an example transformation process from solution model to a basic architecture model in operational space is also provided.
- Second case study focuses on how frameworks based on cognitive architecture can be specified using our approach. We show how non-functional properties are used in human behavior modeling and their interaction with the help of an example of lane keeping and assistance system.

7.3 Case Study 1: Mobile Robot Navigation

In this section, we will use a mobile robot navigation example to demonstrate how a robotic system can be developed and formally specified in our SafeRobots methodology.

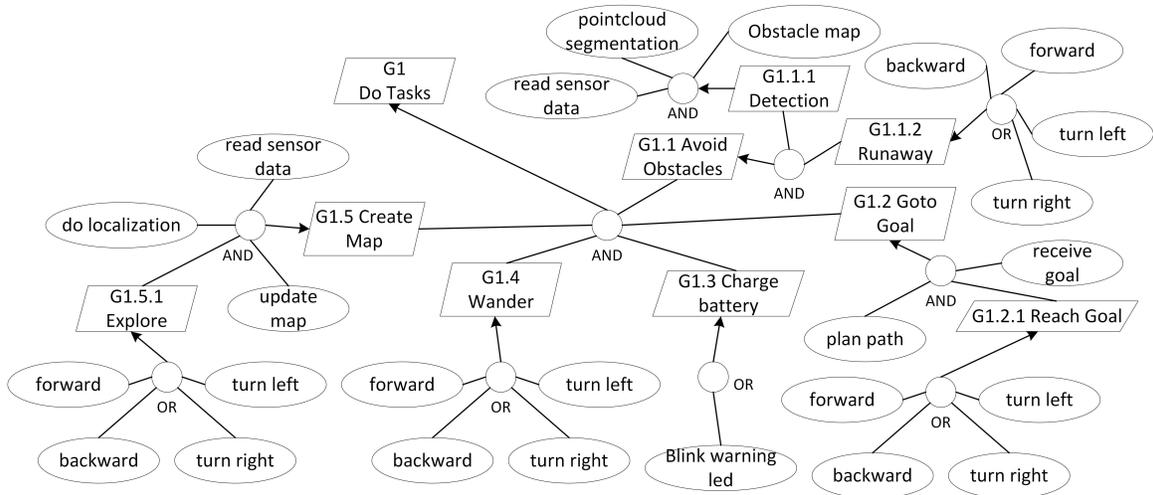


Figure 7-2: Requirement model using KAOS notation

The robot is an indoor differential robot equipped with depth camera for perception and a bumper sensor. The mobile robot has to perform different tasks such as mapping, go to a user-specified goal position by avoiding obstacles, and blinking led in case of low battery. The robot should accept the user-specified goal position only if it has sufficient confidence on its own position and the goal position has been already explored. If the robot does not receive any goal position, it should explore new places and continue the mapping process. The robot should wander aimlessly avoiding obstacles if the mapping process is completed. The system should be based on ROS middleware [9] and the developer intends to use Gazebo simulator [14].

We assume that the knowledge space of the SafeRobots already defines domain specific knowledge, such as the structure of laser range scan, image properties, etc., in the form of knowledge graphs and ontologies. The following section discusses different processes, such as requirement modeling, solution space modeling, and architecture modeling corresponding to the problem space, the solution space, and the operational space of the SafeRobots framework, respectively.

7.3.1 Requirement Modeling

The functional and non-functional requirements of the mobile robot are modeled using KAOS notation [162]. It is a hierarchical structure in which the goals are refined into conjoint subgoals (AND refinement) or a combination of disjoint subgoals (OR

refinement). The goals are decomposed until the leaf node (shown as ellipses) that represents a requirement [69]. We use an extended version of KAOS notation in which the non-functional requirements can be represented as nodes or by attaching a NFP model to a functional goal node. Each functional goal can be associated with a non-functional model specified using the NFP language. For example, the requirement says that the robot should navigate to the goal position (goal G1.2) only if goal position in map is explored and the confidence in the pose of the robot with respect to the map is above a threshold level.

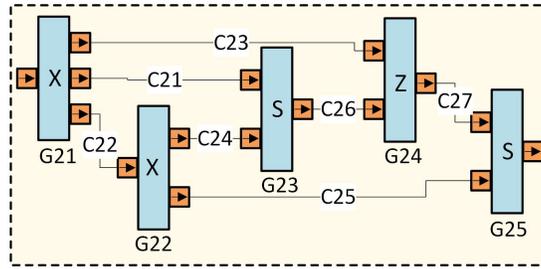
```
import goal, robot;
NFP: (G1.2).valid;
NFP_ATTRIBUTES: goal.pose, robot.pose;
NFP_POLICY: map(goal.pose).is_explored() AND robot.pose.isConfident();
```

Listing 7.1: NFP model for goal - G1.2 in requirement model

This requirement is specified using NFP language as shown in Listing 7.1 and is attached to the G1.2 goal in the requirement model. The model specifies a `valid` property to goal G1.2, and the `NFP_POLICY` specifies how it is evaluated based on the `NFP_ATTRIBUTES` - pose of the goal and robot.

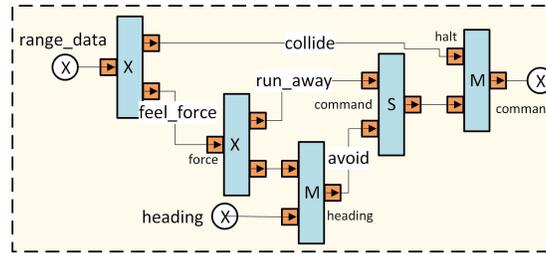
7.3.2 Solution Space Modeling

The solution space model captures the design space available for a given domain or functionality. Figure 7-3b shows the solution space model in SSML language for a functionality from perception domain and navigation domain. The Figure 7-3a models the design space for a point cloud segmentation function (subgoal of requirement G1.1.1). The model formally specifies four different solution paths in which point clouds can be segmented. We have already discussed this model in Chapter 4. Similarly, Figure 7-3b shows a model for obstacle avoidance functionality from the navigation domain. This is the real advantage that the domain model can be reused, instead of reusing just software code libraries. It is to be noted that the solution model only comply with the functional constraints and the non-functional properties are specified as such in the model. In other words, the NFPs specified in the requirement model are not imposed on the SSML model. The NFP constraints are imposed only in the next phase, i.e. architecture modeling. The strategy is to



C21: 2.5D Projection, C22: Ground Classifier, C24: 2D Projection, C25: Euclidean Clustering, C26:Image Segmentation, C27: Box Mapping

(a) Point cloud segmentation domain model



(b) Obstacle avoidance domain model

Figure 7-3: A excerpt form solution space model of perception and navigation domains

postpone the decision on NFPs on a stage, which is closer to the implementation where more information on platform, communication middleware, etc., are known.

The final executable models generated during the architecture modeling process are called operational models, and they usually comprise a reduced subset of solution space model. The reduction is carried out by considering the required system level non-functional properties such as response time, confidence, resolution levels, etc, as constraints to the solution space model. We have already shown a reduction process based on MDPs in Chapter 4. If there are multiple solutions that satisfy the constraints, they are modeled as variation points that can be resolved during runtime when more contextual information is available. The following sections describe how the architecture for the case study can be implemented using a simple subsumption architecture and a more complex Hierarchical Behavior-Based Architecture (HBBA).

Subsumption Architecture

A layered reactive control architecture called subsumption architecture was introduced by Brooks in 1985 [39]. It is primarily based on the decomposition of robot

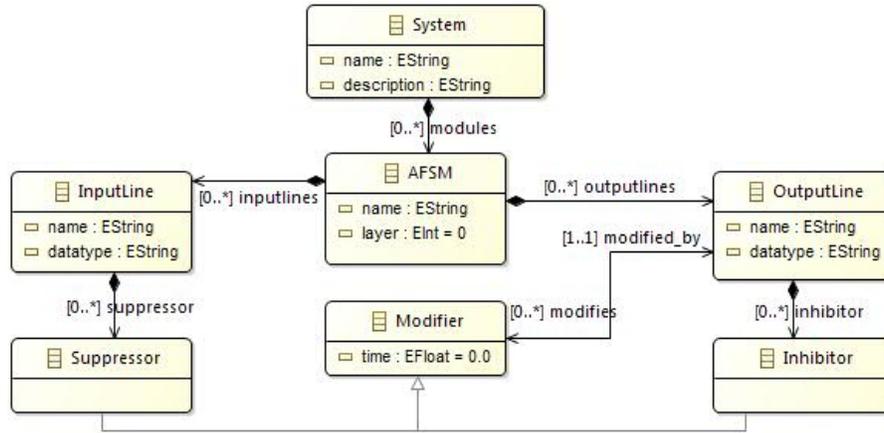


Figure 7-4: Ecore metamodel of subsumption architecture

control problem into special task achieving modules. Required behaviors can be generated by composing the modules at different competency levels in a layered fashion. A level of competency achieves a set of behaviors that can be overridden or constrained by a higher level of competency. Each module is a finite state machine augmented with some instance variable that can hold data structures. There are number of input and output lines for carrying messages, that are associated with each module. An output line from one module can be connected to one or more input lines of other modules. The output line of a module can also terminate at output site of other modules inhibiting the messages on that line for a specific period of time. Similarly, it can terminate in input site of other modules suppressing the usual message and replacing it.

Architecture Metamodel

A formal model of the subsumption architecture using Ecore metamodel is shown in Figure 7-4. It defines that the system comprises a number of modules that are identified by an unique name. It has an integer attribute named *layer* that indicates the layer to which it belongs to. The input and output lines have name and datatype as its attributes. The outline can be associated with a modifier that can inhibit or suppress an output or input line, respectively. The time attribute of the modifier represents the time period in which the modification happens.

AMAL formalism

The Subsumption-based architecture of our mobile robot use case requires knowledge from three conceptual domains and two implementation domains. The conceptual domains are Perception, Navigation, and Subsumption-based control, and implementation domains are ROS middleware and Gazebo simulator. The framework of our mobile robot architecture in AMAL formalism is defined as:

$$\text{System}_{AMAL} \Rightarrow \langle M_{MRA}, PM, NM, SBT, RM, GM, \mathcal{R}_{AMAL}^{PM}, \mathcal{R}_{AMAL}^{NM}, \mathcal{R}_{SBT}^{AMAL}, \mathcal{R}_{AMAL}^{RM}, \mathcal{R}_{AMAL}^{GM}, \mathcal{R}_{GM}^{RM} \rangle.$$

where, M_{MRA} is the system architecture model that performs the tasks mentioned in the requirement model; PM, NM are the solution space model pertaining to perception and navigation domains; SBT is the architecture model of subsumption architecture; RM, GM are domain models of ROS and Gazebo; and their relationship among these models and with AMAL model are also included in the specification.

Figure 7-5 shows a pictorial representation of this formal specification. The relationships define the conceptual mapping from one model to another. For example, the relationship, \mathcal{R}_{SBT}^{AMAL} , defines that a module in subsumption architecture is a component in AMAL model; and the relationship \mathcal{R}_{AMAL}^{RM} defines that a component in AMAL is a *node* in ROS middleware. Therefore, in a system point of view, the module in subsumption architecture is mapped to the *ros node*. The transformation process from solution space model to the architecture model using these relationship specification. However, they cannot be completely automated, and it is a human-assisted process since there are architecture specific decisions to be taken. For example, in this case the information related to the layer at which the behavior belongs is not available in the SSML model and the designer has to manually include it in the transformation process.

Hierarchical Behavior-Based Architecture

The Hybrid Behavior-Based Architecture (HBBA) is a robot control framework that was originally developed for the IRL-1 humanoid robot [163]. HBBA combines two robot control paradigms: behavior-based control (“Think the way you act”), and Hybrid control (“Think and act concurrently”) [164]. HBBA unifies both paradigms by adding layers on top of a behavior-producing modules (referred to as Behaviors), al-

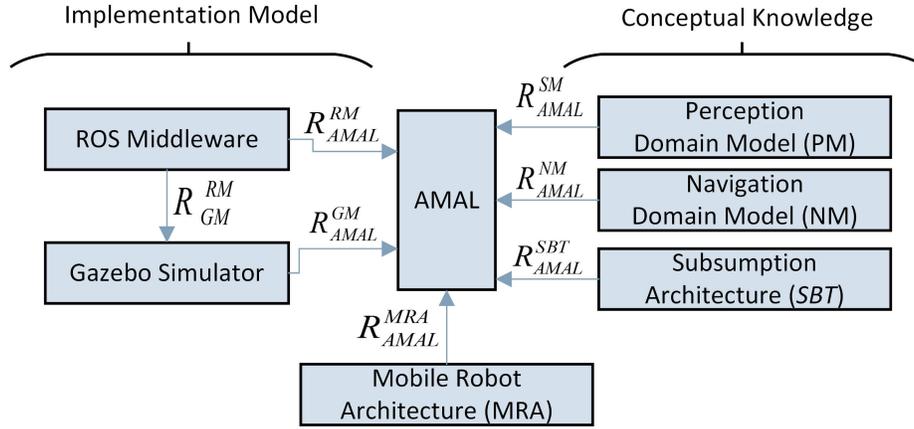


Figure 7-5: Architecture Model in AMAL formalism

lowing Perception and Behavior modules to be selected and configured according to the Intentions of the system. These Intentions are derived from Desires, which represent the satisfaction or inhibition of Intentions as generated by Motivations. Just like Behaviors, Motivations are distributed processes from which a decision can emerge at the Organization Layer. The Intention Workspace, situated at the Coordination Layer, gathers all Desires to infer the Intentions of the robot, mainly by determining which specific modules in the Behavioral Layer must be activated. Like in the Behavioral Layer, conflicting Desires are selected on a priority basis according to their intensity.

Interaction performance can be affected by an unbalanced allocation of computing resources. For instance, acceptable delays in vocal interaction can be difficult to satisfy if computationally expensive modules are simply added to the architecture: it may result in a robot that can navigate efficiently, but may not perform well engaging and interacting with people, or *vice versa*. To solve this, human-inspired selective attention [165] takes place in HBBA by perceptual filtering of Perception modules and by configuration and activation of Behaviors. Computing resources allocation can therefore be prioritized according to the current Intentions of the robot.

Architecture Metamodel

A formal model of HBBA using Ecore metamodel is shown in Figure 7-6. It defines that the system primarily consists of modules and knowledge. Task, Desire, Intention,

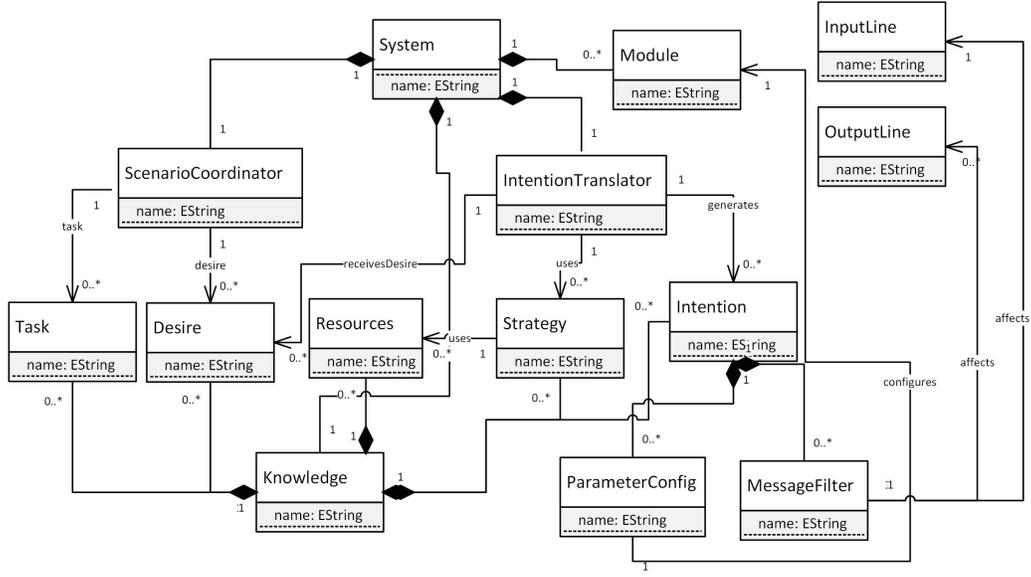


Figure 7-6: Ecore metamodel of HBBA architecture

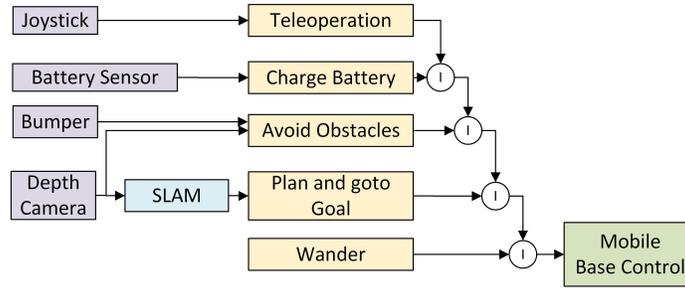
Strategies, and Resources are a part of the knowledge. Scenario Coordinator and Intention Translator are special types of modules. Scenario Coordinator is the only Motivation module in this instance of HBBA. It receives tasks and generates Desires, such as performing simultaneous localization and mapping (SLAM) and going to a specific place on the generated map. Intention Translator generates Intentions from Desires based on Strategies and Resources constraints describing the capabilities of the robot. An Intention can configure parameters of a module and affects the input and output lines of the module by changing the state of Message Filter modules.

AMAL formalism

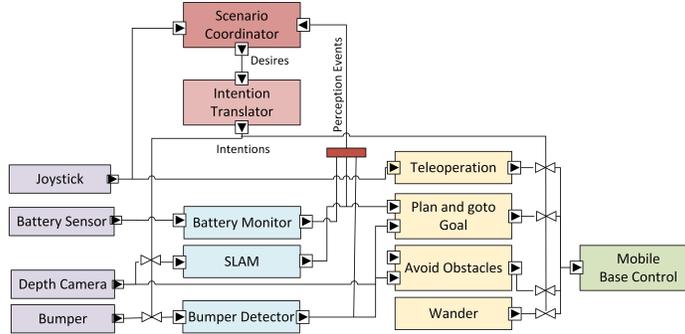
The architecture based on HBBA uses a similar model as proposed for subsumption based architecture. In this case, the subsumption model is replaced by the HBBA model and \mathcal{R}_{SBT}^{AMAL} defines its relation with the AMAL model.

7.3.3 Discussion

The operational model for our mobile robot use case in subsumption and HBBA architecture are shown in Figure 7-7. These are the final executable model after human-assisted transformation from solution space model. During this transformation process, the decision is made, which is specific with respect to the target architec-



(a) Operational Model in Subsumption Architecture



(b) Operational Model in HBBA Architecture

Figure 7-7: Architecture for the case study

ture. For example, priorities are decided for the behaviors in the case of subsumption architectures and default desires for the HBBA architectures. It is also to be noted that there is no explicit *Charge Battery* behavior in HBBA architecture shown in Figure 7-7b, as it is handled by the desire that generates *Planning* Intentions with appropriate parameters configured. Using the results of the domain analysis, the framework specification templates based on AMAL formalism were created. Since we have targeted ROS middleware in the case study, the generated files are a set of ROS launch files. The computational components are generated as ROS Nodes with protected area in which the users can insert their custom code.

Figure 7-8 illustrates the CPU usage for subsumption and HBBA architectures. The average resources usage for subsumption architecture is significantly higher than that of the HBBA architecture due to the fact that all behaviors are active all the time. However, for the HBBA, the behaviors are activated only when the required desires are present. The higher resource usage for HBBA as seen in the Figure 7-8 occurs when a *GoTo* Desire with maximum intensity is generated by Scenario Coordinator, which in turn activates the planning and navigation components required to reach the goal

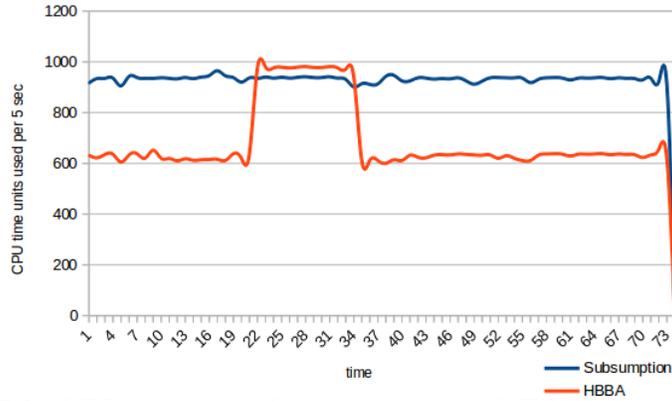


Figure 7-8: CPU usage for Subsumption and HBBA architecture

pose. The main intention of our case study is not to benchmark the architectures, but to demonstrate the process in which different domains are identified and how their interactions are formally specified.

7.4 Case Study 2: Framework based on Cognitive Architecture

Cognitive models are very complex to develop and understand, and the systems based on cognitive paradigms are hard to analyze by system architects without expertise in cognitive architectures. In real world applications, cognitive models have to be integrated with other computational models, such as in robotic systems and other intelligent systems. One reason why cognitive architectures are not yet effective in industrial applications is that most of the cognitive control uses a production system paradigm¹. Production systems are used in cognitive architectures to model cognitive processes that act on memory and employ conflict resolution mechanisms to create new production rules and memory elements. When such cognitive architectures are integrated in existing systems, there is no formal way to analyze and validate the system, since the control flow cannot be explicitly represented [167]. Systematic approaches and tools are required so as to increase the breadth of influence of cognitive models and to prove their capabilities in addressing much more complex tasks, and

¹A production is a rule that collectively form an information processing model of some cognitive task or a range of tasks. Each production represents a retrieval of knowledge from long-term memory. Production systems are a major formalism for modelling integrated cognitive architecture [166]

building robust human-machine systems. In this case study, a framework for designing service oriented architecture is developed using our approach. We use a lane keeping and lane change assistance system as an example application. Specifically, we concentrate on the formal definition of different aspects of the framework. We explain how multiple domains are incorporated using our AMAL formalism. We also emphasize the importance of modeling Non-Functional properties in human-machine systems using this case study. In the next section, an overview on human-machine systems are provided and it shows how modeling of NFP and QoS, helps in dynamic adaptation of the system. In Section 7.4, we use a Lane keeping and lane changing assistance system as an application to demonstrate our approach. It is to be noted that our intention is to demonstrate framework specification that facilitate cognitive architecture based system, and sometimes during the discussions, we tend to go into the details of the architecture and the domain itself. Hence, the merits and demerits of the architecture is of no concern in our case study.

7.4.1 Overview on Human-Machine Systems

In earlier times, function allocation in human-machine systems was primarily a design decision. Consider the case of cruise control in today's cars, the steering control is allocated to the human driver while the automation is responsible for applying acceleration. This is an example for static function allocation. Some of the traditional strategies for function allocation are: (a) assigning each function to the most capable agent, (b) allocating to machine every function that can be automated, and (c) applying an appropriate allocation policy [168]. However, in adaptive human-machine systems the function allocation is a run-time problem. Consider the case of auto-pilot taking control of an aircraft in emergency situations. In such systems, there is some kind of control sharing and trading that happens. The first example of adaptive cruise control is a control sharing case, while the second one is a control trading problem. In the latter case, the software takes the control of system with or without the consent of human.

7.4.2 Application: Lane keeping and lane changing assistance system

Lane keeping and lane changing assistance in automobiles is a perfect application in which human and automation have to be integrated seamlessly. Research in intelligent transportation systems has improved the vehicle's ability to sense, understand, make decisions, and ultimately act on its own accord in its operating environment. The desire for better driver assistance and machine intelligence drives automobile manufactures to pack more sensors and more computation into their vehicles in a cost-effective manner. Hence, the vehicle architecture designer has to integrate various domain concepts and subsystems that have their own design paradigms and constraints. In this case study, we intend to design a framework that supports building an architecture for a system that provides assistance for lane keeping and lane changing. The automation part is the lateral control of the vehicle to keep the vehicle in the same lane and the assistance is provided by applying a thrust on the steering wheel. The system takes control by applying the required torque on the steering wheel in case of lane departure situations. The driver behavior and external environment have to be continuously monitored in order to predict the driver's intention. Specifically, the system has to predict whether the driver is trying to change the lane or he/she is erroneously departing from the lane. Most of the commercially available lane keeping assistance systems predict the driver's behavior during the maneuver execution phase. But there are promising results in research, that predicts the driver's intention in the decision phase by modeling cognitive processes behind the behavior. The theoretical basis for our cognitive model of the driver is based on the works of D. Salvucci [87, 169, 170].

An overview on the application framework

The standard approach for automobile OEMs is to develop systems by assembling components that have been completely or partly designed and developed by external vendors [82]. Because of the increasing complexity of automobile systems with large number of distributed features, such an approach will also lead to various compositional issues commonly known as *feature interactions*. Therefore, Service Ori-

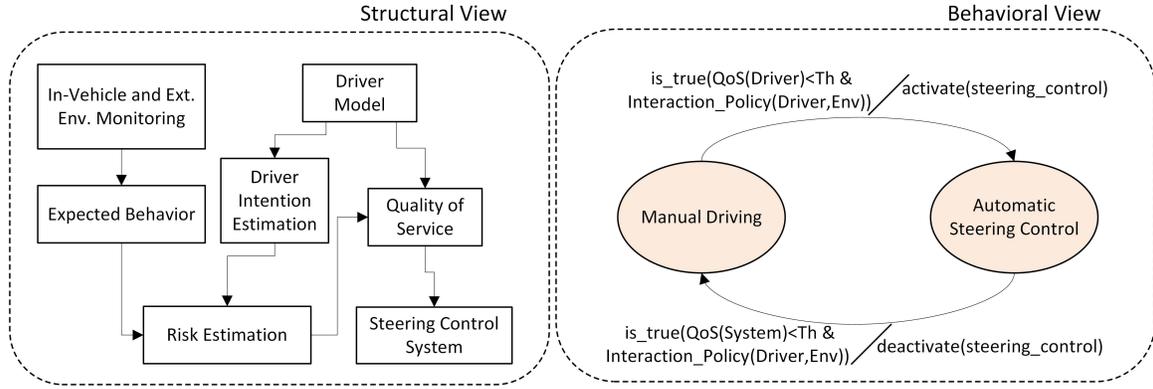


Figure 7-9: Architecture modeled in the proposed framework

ented Architectures (SoA) are gradually being adopted in these systems where various functionalities are provided as services and the assembled components are seen as service providers. This software engineering paradigm has many advantages in Human-Machine collaborative work. Advances in human behavior research helps to model various human actions. These actions can be seen as services provided by the human, for example, steering, braking, applying acceleration by the driver can be viewed as services provided by the human driver. In some contexts, humans provide high quality services while in some others the machine counterpart does. For example, in the case of assisted parking, human steering control service is delegated to the machine still retaining the authority over acceleration with the driver. Furthermore, selecting the services based on the quality has advantage not only between human and automation agents but also within the automation system itself. Since the functionalities are viewed as services, it is possible to compose basic low level services in different fashion to provide different functionalities. For example, in cars, voice recognition service can be used in an entertainment system as well as in a navigation system.

In the target application of our proposed framework, we design a SoA for lane keeping and lane changing assistance system based on AMAL formalism. The proposed framework supports designing the architecture by using two complementary views - structural view and behavioral view. An example of an architecture modeled by using this framework is shown in Figure 7-9. The structural view models the interconnection of various computational algorithms and its communication aspects,

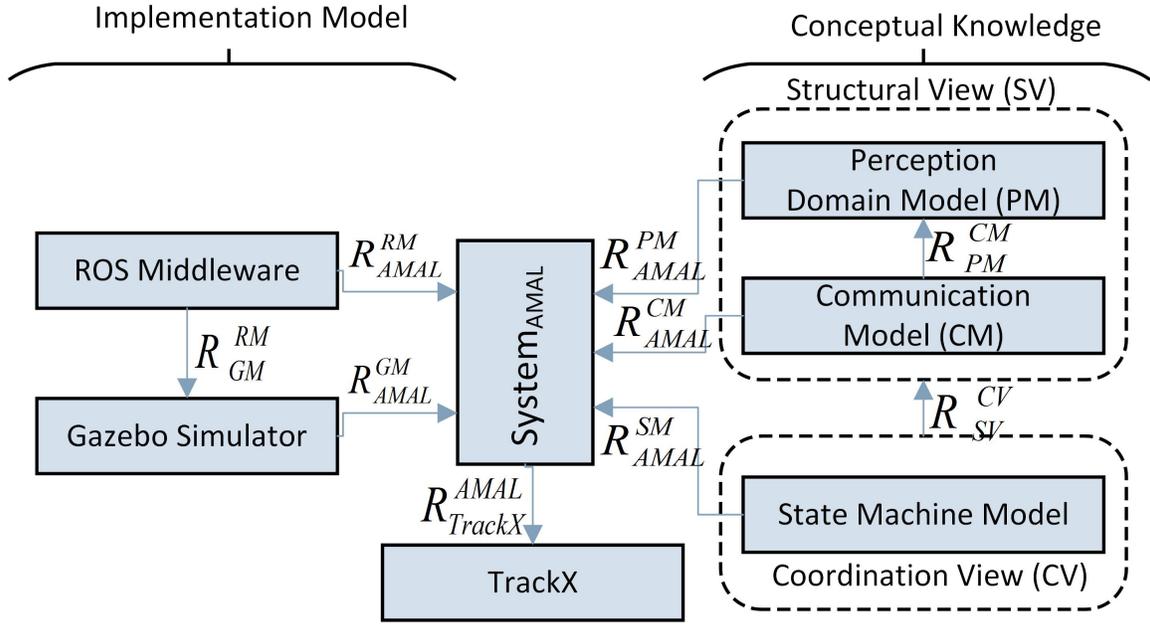


Figure 7-10: Framework model of driver assistance system using AMAL formalism. Various domains and their relationships are shown.

and the behavioral view models the coordination of these computations using state machine formalism. In our example, the behavioral view shown in Figure 7-9, models two states: manual driving and automatic steering control. Quality of Service of Driver and the `interaction_policy()` coordinate when automation has to take control and when it should be deactivated. The main intention of this case study is to show the architectural model, details regarding interaction policies are beyond the scope of this study.

Figure 7-10 illustrates various domains and their relationships involved in the architecture modeled in AMAL formalism. Domain models are specified using meta-models. The various domains and their formal models in the proposed framework are described in the following sections.

ACT-R Cognitive Architecture

The ACT-R cognitive architecture is based on rigid theory of human cognition and provides a computational framework that constraints the models that are cognitively plausible. It consists of four modules - Intentional, Declarative, Visual and Manual Modules. A central production system communicates with these modules through

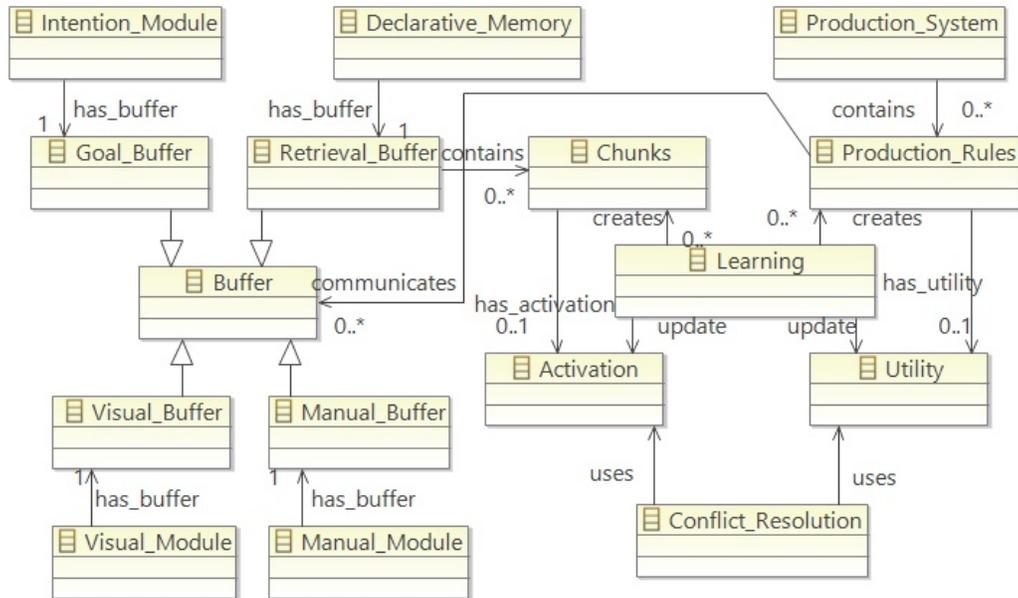


Figure 7-11: Ecore diagram of ACT-R metamodel

associated buffers. Visual and Manual modules represent the perception and motor schema, respectively. Intentional module and Declarative module represent memory stores for goals and facts (termed as ‘chunks’), respectively. The production system represents procedural memory that consists of condition-action rules that uses the respective buffers of the modules to ‘fire’ the actions. The actions can add or modify chunks in declarative memory, create a new sub-goal and issue commands to visual and manual modules. Each chunk has sub-symbolic parameters, such as chunk activation that represents the relative ease with which the chunk can be recalled. The activation level decay over time to replicate forgetting aspects of human. Similarly, the procedural rule has real-values quantities called utilities that represent the cost and probability of reaching the goal if that rule is chosen. In addition, learning mechanisms affects the activation and utilities, and creates new chunks and production rules. Conflict resolution mechanism uses activation and utilities to choose firing when multiple production condition matches [86].

ACT-R Metamodel

Figure 7-11 shows the structure of the ACT-R 6.0 framework as an Ecore diagram. The Ecore model shown in Figure 7-11 also contains constraints such as the production

system always communicates to modules through buffers. The constraints that are not expressed by Ecore model are specified using Object Constraints Language (OCL) [171]. OCL constraints on the size of Retrieval Buffer and activation level of merged chunks are shown in Listing 7.2. The first constraint states an invariant that the buffer can hold a maximum of MAX_RB number of elements at any time. The second formalism states that when two chunks are merged, the activation level of the resulting chunk is the sum of the activation of the added chunks.

```

context: Retrieval_Buffer
inv: self.size >= MAX_RB
context: merge(chunk_source:Chunk, chunk_target:Chunk)
def: chunk_target.activation += chunk_source.activation

```

Listing 7.2: OCL constraints on buffer size and activation level of merged chunks

7.4.3 Harel Statecharts

The Harel statechart [118] is a formalism of state machines and state diagrams for the specification and design of complex discrete-event systems, that include the notion of hierarchy, concurrency, and communication.

Harel Statecharts Metamodel

Figure 7-12 shows an Ecore model of minimalist variation of Harel statecharts, rFSM proposed by [70], for defining execution and interaction semantics of robotic tasks. It consists of three model elements: states, transitions, and connectors. A virtual element name *node* is used to simplify the concepts. State can be composite or atomic, depending on whether it has leaf nodes or not. An active node consists of exactly one leaf node that must be active. The Boolean function *guard* determines whether a transition can occur and *effect* function is executed when transition occurs. The effect function can represent computation and, hence consumes time. The functions associated with states are executed during the in-state *do* activity. The action *entry* and *exit* are executed upon entering and leaving the state.

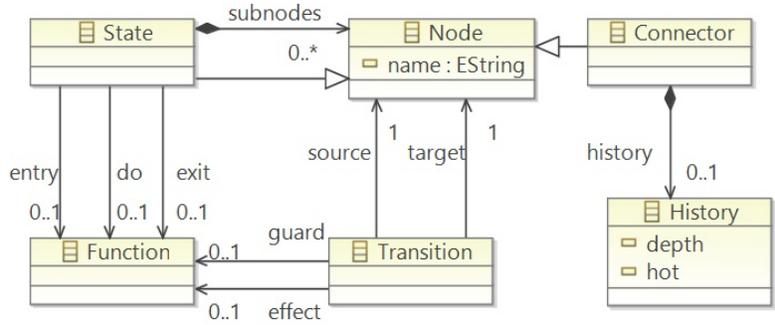


Figure 7-12: Ecore diagram of Harel statechart

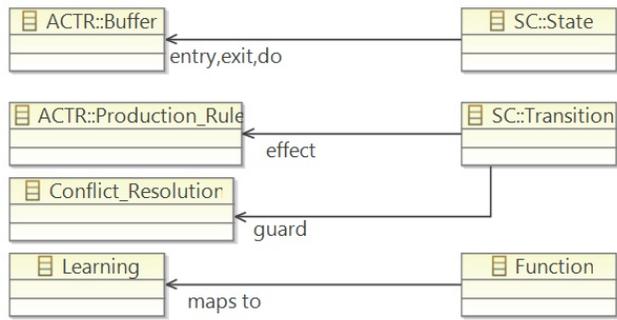


Figure 7-13: Ecore diagram of relational model \mathcal{R}_{SC}^{ACTR} between ACT-R model and statechart model

7.4.4 Relation Model \mathcal{R}_{SC}^{ACTR} between ACT-R Model and Statechart Model

The relation model \mathcal{R}_{SC}^{ACTR} between the ACT-R and statechart formalism is shown in Figure 7-13. The *state* represents the state of buffers, for example, goals, chunks, etc. The entry, exit, and do behavior can affect the state of the buffers, in turn affects the four modules in ACT-R. The state transition represents the production rule. The guard of transition is mapped to conflict resolution, that determines which production rule to fire (effect of transition). The function is mapped to Learning mechanism, that can affect the activation and utilities associated with chunks and production rules respectively. The Ecore model in Figure 7-13 represents these relations and the internal mechanism, such learning affects activation are specified by their respective metamodels.

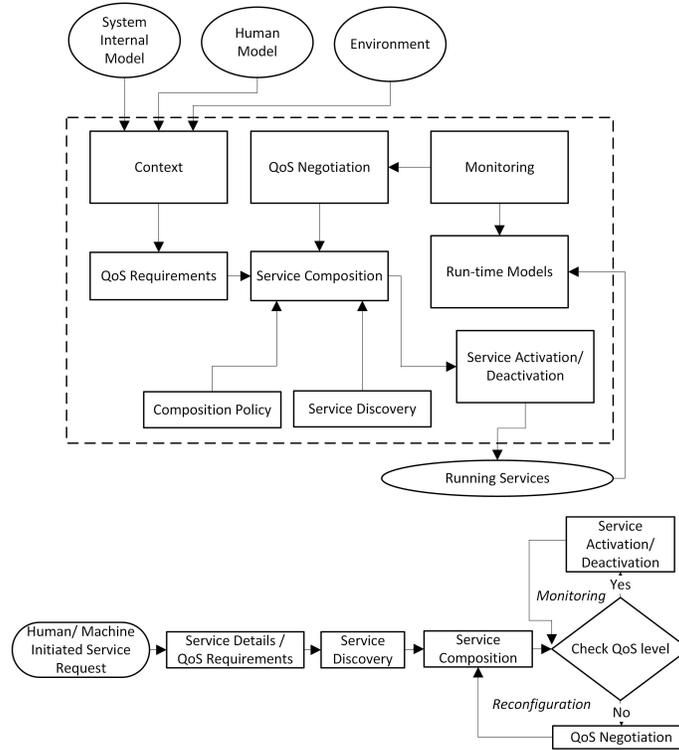


Figure 7-14: A reference model for service oriented architecture

7.4.5 Service Oriented Architecture

Figure 7-14 shows an abstract architecture of a service oriented system. The reference architecture is used here only as a way to explain the concepts hiding all the technology and platform specific details. The architecture is comprised of a number of low level services that provide support for service discovery, QoS negotiation, supervisors, etc.

Figure 7-14 also shows a typical sequence of activities in the system. The request for service can be generated either from human or machine. The service description with all the required parameters and required levels of QoS is computed with respect to the context. For example, emergency braking service will be at a high level while a music streaming service will be at a lower level. Appropriate services found from a repository are composed to check for conflict, redundancy etc. and QoS is computed for the composed service. The composed service is verified with QoS specification and the required service is activated. A supervisor continuously monitors the quality level of the running services and reconfiguration is done when anomalies are detected. Figure 7-15 shows the metamodel of service oriented architecture.

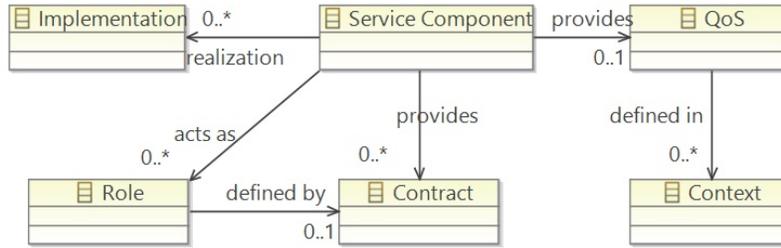


Figure 7-15: Ecore diagram of service oriented architecture domain model

7.4.6 Perception Model

The Perception model captures the conceptual knowledge of the perception domain. It contains abstract knowledge (solution space) regarding various computational algorithms, its dependencies, etc. In this domain model, we have utilized Solution Space Modeling Language (SSML) to model multiple solutions.

7.4.7 Communication Patterns

Distributed components exchange data, message, or events through specific communication protocols, such as Remote Method Invocation (RMI), Message Passing, etc. A communication pattern defines the communication mode, formally define accessor methods and hides all the middleware, platform-specific synchronization issues. It always consists of two complementary parts named service requestor and service provider representing a client/server, master/slave or publisher/subscriber relationship [15].

Communication Pattern Domain Model

Communication domain models capture the knowledge regarding several patterns that cover the communication requirements for component interactions. In this case, we use the communication patterns proposed by the authors of [15]. It consists of a minimal set of communication patterns required by robotic software component interaction: send, query, push_newest, push_timed, and event. It supports communications such as synchronous, asynchronous, publish/subscribe, client/server, and dynamic wiring. Figure 7-16 shows the metamodel of communication pattern domain used in our case study.

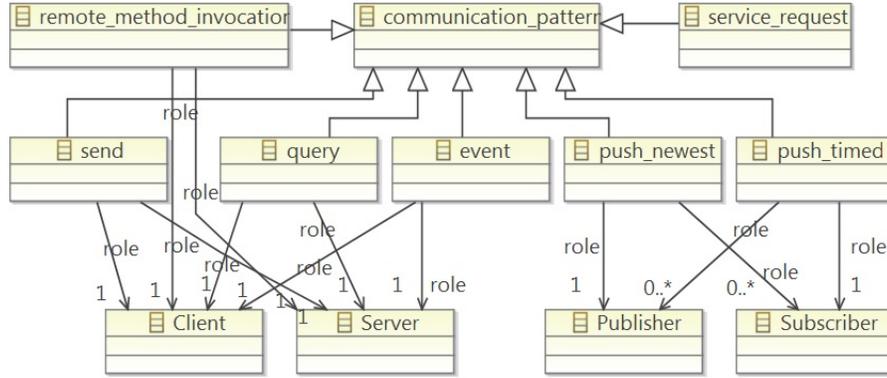


Figure 7-16: Ecore diagram of communication domain model

Relation Models

Similar to relation model between ACT-R and statechart formulated in Section 7.4.4, the relation between domain models can be specified. Table 7.1 shows the relations that maps model elements across different domain models that was described in previous sections. For better understanding, some of the relation models is coupled together, for example, \mathcal{R}_{AMAL}^{BV} and \mathcal{R}_{AMAL}^{SC} shown as separate relation in Figure 7-10 is coupled in form of \mathcal{R}_{BV}^{SC} in Table 7.4.4 to represent the relation between behavioral view and statechart formalism.

\mathcal{R}_{SV}^{CM} represents the relation between Communication Model (CM) and the Structural View (SV) in AMAL formalism. Table 7.1a shows the mapping between CM model elements to model elements in structural view. Intuitively, the communication patterns are mapped to connectors in SV. Table 7.1a also lists the number of roles that each connector is associated with, depending on the patterns, for example, a connector that represent **push_timed** can have 1 publisher role and 'n' number of subscriber role.

\mathcal{R}_{SV}^{PM} represents the relation between Perception Model (PM) and the structural view in AMAL formalism. Table 7.1b shows the mapping between PM model elements to model elements in structural view. Since the PM is modeled by SSML language, the mapped is between abstract elements of SSML with that of AMAL. In SSML, the connector represents the computation process, Dispatch_Gate represent the interconnection between computations, Ports represent abstract data types and NFP represents the non-functional property of the algorithm. The relation table maps

$Connector^{SSML}$ to $Component^{SV}$, $Dispatch_Gate^{SSML}$ to $Connector^{SV}$, and the rest as shown in the Table 7.1b.

\mathcal{R}_{PM}^{CM} represents the relation between communication domain and perception domain model. It captures the semantic compatibility between the domain and conflicts (if any) in the relationship. For example, the previous two relation model maps communication pattern in CM and $Dispatch_Gate$ in PM to the same model element $Connector$ in SV. The relation \mathcal{R}_{PM}^{CM} shown in Table 7.1c specifies that these two concepts are semantically compatible since $Dispatch_Gate$ and communication pattern represent the interaction between computations in their respective domains. In the similar fashion, the relationships between model elements between the domain models are shown in Table 7.1.

7.4.8 Discussion

In this section, we highlight different aspects on our multi-domain architecture formalism for control component in Human driver model. To provide a theoretical basis for our discussions, a brief overview of the driver model is discussed below. More detailed description can be found in [87].

Human Driver Behavior Model: Human driver model in ACT- R architecture consists of control, monitoring, and decision making components. The control component is responsible for low level perception and, lateral and longitude control. Lateral control is based on two features - near point and far point. The near point represents the vehicle's current lane position. It is measured at 10m from the vehicle's center. The far point can be: (a) the vanishing point of a straight road, (b) the tangent point for the curve, and (c) the lead vehicle. The control law for steering angle can be expressed as [169] :

$$\Delta\varphi = k_{far}\Delta\theta_{far} + k_{near}\Delta\theta_{near} + k_I\theta_{near}\Delta t \quad (7.1)$$

where φ is the steering angle, θ_{far} and θ_{near} are visual angles of near and far points, respectively, $\Delta\varphi$, $\Delta\theta_{far}$, $\Delta\theta_{near}$, and Δt are the difference of the respective parameters with respect to the last cycle. In short, the control law imposes three constraints on the steering angle: a steady far point ($\Delta\theta_{far} = 0$), a steady near point ($\Delta\theta_{near} = 0$),

Communication Model		Structural View
Communication Pattern		Connector
Pattern	Relationship	Roles
send	client/server	client(1), server(1)
query	client/server	client(1), server(1)
push_newest	publisher/subscriber	publisher(1), subscriber(n)
push_timed	publisher/subscriber	publisher(1), subscriber(n)
event	client/server	client(1), server(1)

(a) \mathcal{R}_{SV}^{CM}

Perception Domain Model	Structural View
Connector	Component
Dispath_Gate	Connector
Port	Port
NFP	Property

(b) \mathcal{R}_{SV}^{PM}

Communication Model	Perception Model
Communication Pattern	Dispatch Gate

(c) \mathcal{R}_{PM}^{CM}

Statechart Model	Behavioral View
State	Component
Transition	Connector

(d) \mathcal{R}_{BV}^{SC}

Behavioral View	Structural View
Component	Component[n]

(e) \mathcal{R}_{SV}^{BV}

SOA	AMAL
QoS	Property
Service_component	Component
Contract	Connector
Role	Role

(f) \mathcal{R}_{AMAL}^{SOA}

NFP Model	AMAL
NFP	Property
QoS	Property

(g) \mathcal{R}_{AMAL}^{NFP}

ROS Middleware Model	AMAL
Node	Component
Publish/Subscribe & Service client/server	Connector
Provided/Required Interfaces	Port
Parameters	Property

(h) \mathcal{R}_{AMAL}^{RM}

Table 7.1: Metamodel relations between Communication Domain Model (CM), Perception Domain Model (PM), Statechart Model (SM), ROS Model (RM), NFP Model (NFP), SOA Model (SOA), Structural View (SV) and Behavioral View (BV)

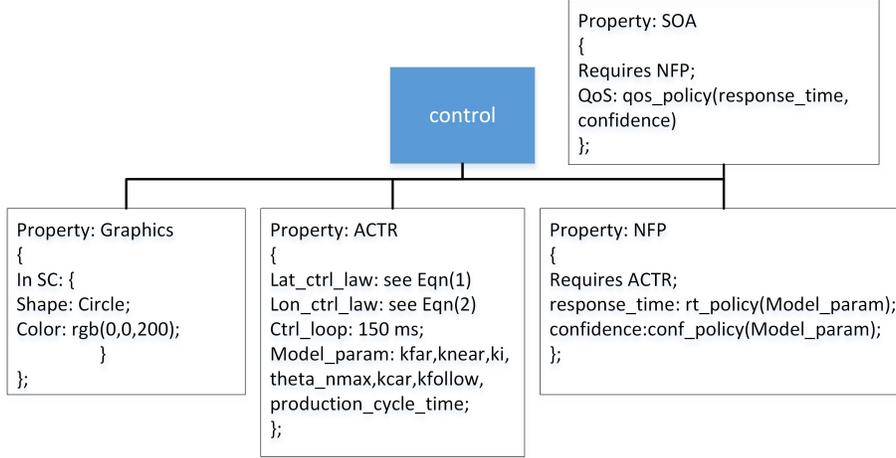


Figure 7-17: Control component in AMAL formalism

and a near point at the center of the lane ($\theta_{near} = 0$). Similarly, the longitude control law can be expressed as:

$$\Delta\psi = k_{car}\Delta thw_{car} + k_{follow}(thw_{car} - thw_{follow})\Delta t \quad (7.2)$$

where ψ is the acceleration value, thw_{car} is the time headway to the lead vehicle, Δthw_{car} is the difference of thw_{car} with respect to the previous cycle, and thw_{follow} is the time headway for following a lead vehicle. The monitoring model is defined by a random probability measure ($p_{monitor}$) that checks left or right lane, and forward and backward with equal likelihood. The decision model is defined by the safe distance (d_{safe}) and the lead vehicle distance that is considered as safe by the driver. The memory is modeled as total number of references that can be stored in declarative knowledge, decay time for the memory, and memory creation time.

Figure 7-17 shows the specification of control component in AMAL formalism. It shows four properties corresponding to four different domains - Graphics, ACT-R, Non-functional property, and service oriented architecture. The domain models and their relations were discussed in the previous sections. The graphics property is used for visualization by the modeling tool, for example, control component will be shown as circular nodes while modeling graphically in statecharts domain. The property name, for instance ACT-R determines the domain in which the properties are valid. The properties are specified using domain-specific language and they can be parsed

only using the domain knowledge. For example, the property ACT-R shown in Figure 7-17 defines two control laws for lateral and longitudinal control, and the execution time for the control loop, model parameters.

Figure 7-18 shows the production system of ACT-R driver model proposed for lane changing behavior in statechart formalism. Notice the control component is visualized as circular node in this domain. This is due to the `Graphics` property defined for that domain. The state represents the goal and the transitions that originate from the state represents the production rules for that goal.

```

REQUIRES ACTR;
import control_skill,monitor_skill,decision_skill,memory_skill
NFP: performance;
NFP_ATTRIBUTES: cs:control_skill:derived, ms:monitor_skill:derived, ds:decision_skill
:derived, memory_skill:derived; NFP_POLICY: eh.cs>thr_cs & eh.ms>thr_ms & eh.ds>
thr_ds;
-----
NFP: control_skill; NFP_ATTRIBUTES: prep_time:static:ms, exec_time:static:ms,
kfar_lateral:dynamic, knear_lateral:dynamic, ki_lateral:dynamic, kfar_speed:
dynamic, knear_speed:dynamic, ki_speed:dynamic; NFP_POLICY: contrl_policy();
-----
NFP: monitor_skill; NFP_ATTRIBUTES: prob_monitor:dynamic, NFP_POLICY: monitor_policy
();
-----
NFP: decision_skill; NFP_ATTRIBUTES: safe_distance:dynamic:m, NFP_POLICY:
decision_policy();

```

Listing 7.3: NFP model of performance of driver model

The NFP domain is valid in the implementation (see `implementation` model element in SOA metamodel) domain, i.e., in this case when the human model is viewed as a piece of software. For example, NFP property shown in Figure 7-17 specifies two policies, `rt_policy()` and `conf_policy()`, that act on model parameters derived from ACT-R domain, to define the `response_time` and `confidence` respectively. This NFP model is visible when the control component is implemented in software, in a formal way, when the control component is associated with an `implementation` domain element in SOA metamodel. Listing 7.3 shows the NFP model for performance of the driver model in the NFP domain-specific language.

The QoS property is defined for a service component in the SOA domain when the software realization is used as a service. Notice that `qos_policy(response_time, confidence)` shown in Figure 7-17 uses the response time and the confidence prop-

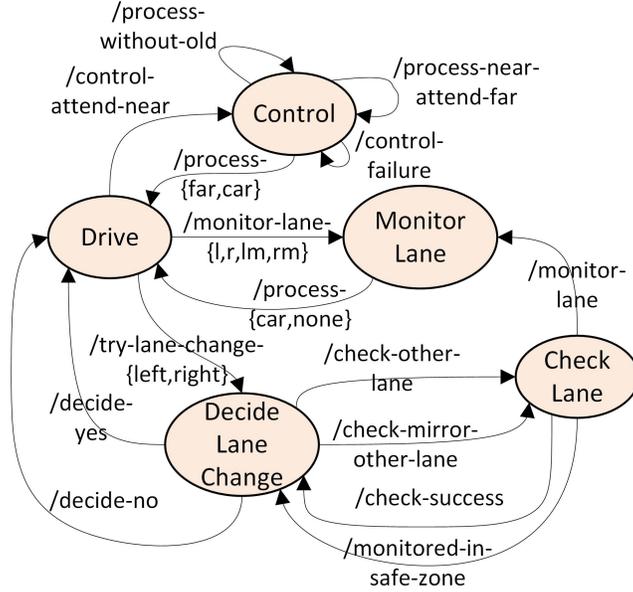


Figure 7-18: Production system in ACT-R driver model for lane changing behavior using statechart formalism

erty, to define the QoS property. In short, non-functional properties are viewed as model parameters in ACT-R domain, as NFP in the implementation domain, and as QoS in service oriented architecture domain. For example, the authors of [170] have proposed a ‘model tracing’ methodology for detecting lane change intention. The QoS policies of this model in our approach is listed in Table 7.2. In ACT-R domain, the model parameters are `production_cycle_time` and `lane_change_score`. The `production_cycle_time` is the cycle time (nominally 50ms) needed to fire a production rule. The `lane_change_score` is the probability of lane changing behavior by the human driver. It is computed as shown below:

$$Score = \frac{\log S(LK)}{\log S(LC) + \log S(LK)} \quad (7.3)$$

where LK is the lane keeping model, LC is lane changing model, and S is the similarity score between model simulation and the observed human data (refer to [170] for more theoretical details). Two non-functional properties are defined, when this model is realized in software - `response_time` and `confidence`. The `response_time` is defined as $3 \times \text{production_cycle_time}$ represented the time required for firing three productions rules (nominally 150ms). Two rules for encoding near point and

far point, and one to issue motor commands as shown in Figure 7-18. When this software implementation is used as a service in SOA domain, these non-functional properties are employed to compute the QoS. Table 7.2 shows that QoS is the sum of normalized `response_time` and `confidence`. Hence, the main idea is to determine the relevant parameters in higher abstraction level (QoS in SOA domain) using independent low-level domain-specific parameters (model parameters in ACT-R domain).

Domain Viewpoint	Visibility Domain	Policy
Model Parameters	ACT-R	{production_cycle_time, lane_change_score}
NFP	Software Implementation	{rt_policy:response_time = 3 X production_cycle_time, conf_policy:confidence = lane_change_score}
QoS	SOA	QoS = [normalize(response_time, base) + confidence]

Table 7.2: An example for estimation of quality parameters while composing multiple domains

7.5 Conclusion

In this chapter, we started by discussing some of the existing methods for framework development. Two applications in which our methodology and framework development process were then explained. We have adopted a mix of the three approaches for framework development in our case study. We started with problem analysis based on the case study and subsequently, required domains were identified, and then we tried to generalize using past experiences in framework specification. Our first case study provided an overall process of framework development from requirement specification to framework development in operational space. For experimental purpose, an example transformation process from solution model to a basic architecture model in operational space is also provided. Second case study focused on how frameworks based on cognitive architecture can be specified using our approach. We showed how non-functional properties are used in human behavior model and their interaction with the help of a case study on lane keeping and assistance system.

Part III

Conclusion

Chapter 8

Conclusion and Future Research Directions

8.1 Summary and Contributions

The process of developing robotic software frameworks and tools for designing robotic architectures is expensive both in terms of time and effort, and the lack of a systematic approach may result in adhoc designs that are not flexible and reusable. Accordingly, we identified the current model-driven approaches in robotics and we analyzed how these approaches achieve general modeling related advantages and how effective they are in satisfying robotic domain specific requirements. Based on our comparative survey on existing model-driven frameworks in robotics and qualitative analysis of their features, we found that many of the domain-specific requirements such as architecture level analysis, system reasoning, non-functional property modeling, run-time models, component composition, etc., were addressed differently in these approaches. Hence, it is hard to find a single approach that has features according to one's requirement. In addition, it is difficult to reuse these solutions encoded in tools, model transformations, and middleware modules. Systematic development process and detailed instructions for building such frameworks and supporting infrastructure have not been studied enough. In this direction, the thesis proposes a conceptual methodology and development approach that facilitates specification, design, and deployment of framework for robotic systems.

We analyzed the importance of specifying and integrating NFPs in Framework development in Robotics domain. Providing a systematic way of NFP specification and integrating it with development process by appropriate tools are essential for efficient framework development process. The importance of Non-Functional properties in robotics and human-machine systems were discussed with an example from cognitive architecture domain. Modeling those properties are necessary in architectures where functionality alone cannot be used for making both design time and run-time decisions. Our NFP metamodel provides a generic base for specifying the non-functional aspects of both human and machine models. The main challenge in finding structure for NFP specification is to provide a flexible mechanism to address a large variety of property types and providing a tooling support to manage them. The challenge of dealing with heterogeneous attributes are addressed by categorizing the attributes into profiles hierarchically and then using policies to compare at a higher abstraction level.

We studied common reasons that make the robotic system designs fallible by taking into account the previous experience of robotics experts in various experiments in academics and industry. For motivational purpose, we used an experience report on developing a lidar based vehicle tracking system in an industrial context. To address the identified problems, we proposed a modeling language - Solution Space Modeling Language (SSML), to formally model the solution space and to specify the quality attributes during design time. Solution space modeling can expand this design space, help finding the best possible solution, and also permit to perform run-time adaptation of the system. Solution model helps in early analysis of quality attributes, to identify variations and acts as a bridge between problem and implementation space. The resolution of solution space might not always result in a static operational model. Typically, the resolution process performed during development time results in a subset of solutions that are modeled as variation points in the architecture. This is attributed mainly due to certain context-based properties that can be estimated only during runtime. We formally defined the solution space to operational space transformation process and employed a probabilistic approach to resolve the solution model during design time and execution time. These operations are handled automatically by the framework tool and are independent of any particular robotic framework.

The specification of architecture framework is intended to facilitate its development and to establish the relationships between different domain models in a formalized manner. We saw that most of the discussed model kinds in robotics are some form of Hierarchical Graphs with certain additional properties. This is an important observation since our metamodel has a basic formalism on which the semantic and other properties are added according to the concerned domain. Making architecture meta-framework a point of conformance opens new possibilities for interoperability and knowledge sharing in the architecture and framework communities. The first step in this direction is made by proposing common model and by providing a systematic approach that helps in specifying different aspects and their interplay in a framework. We emphasized on the infrastructure modularity and the reuse of transformation tools and middleware modules. The uniform definition and underlying common conformance model of architecture viewpoints and coordinated collection of these viewpoints can promote reuse of tools and techniques to the robotic communities using these frameworks. It is also possible to integrate existing tools based on Eclipse that has a well-formed meta model by mapping its elements with that of our AMAL model.

As the community find better ways to abstract and standardize concepts in robotics, Domain Specific Languages (DSLs) will become more and more attractive. By mapping these concepts with our meta level models, it is possible to save considerable amount of time for development tools by reusing system infrastructure code. We believe that combining the benefits of integrating existing solutions with benefit of having common underlying structure result in more mature tools. In addition, by specifying the relationship with other domain models in an already existing framework, it is possible to integrate these DSLs with less effort. We also discussed how our methodology can be employed to model domain knowledge and how they can be applied in intelligent software tools and processes, to develop complex robotic systems. The main challenge is to adopt the domain model at the appropriate granularity to assist the system designer in systematic software development process to develop efficient and reusable software for robotic systems.

8.2 Future Directions

8.2.1 Search-Based Software Engineering

Search-Based Software Engineering (SBSE) reformulates software engineering as a search problem [172]. Our experience indicates that the existence of large solution space available for system designers leads to poor software quality. In Chapter 4, we showed that multiple algorithms are available for implementing a functionality in robotics and early decisions assuming future operating conditions tend to degrade the software quality. Once the solution space for a given problem is modeled in our SSML, computational search techniques can be applied to evaluate the system objectives and to converge to a reduced set of solutions. We showed one method using a probabilistic method. However, more research is required in this direction, especially when the system consists of humans and robots with a specific goal to accomplish. For example, the system's objective to maximize its Quality of Service (QoS) can be defined as:

$$\text{Maximize } \sum_{i=1}^k \text{QoS}(m_i) \quad (8.1)$$

where m_i is an abstract term that represents an algorithm (algorithmic sequence), model (human or automation), or a control flow. We have seen the benefits of modeling NFP of human driver, automation and their interaction in the case study of assistive lane keeping system for vehicles. The QoS of the m_i can be defined as:

$$\text{QoS}(m_i) = \sum_{j=1}^n c_{ij} x_{ij} \quad (8.2)$$

where, c_{ij} are application-specific coefficients and x_{ij} are model parameters. It is an open research question how adaptation can be made in real world complex systems. There exist some interesting approaches based on ontologies and genetic programming in the domain of Service Oriented Architectures (SOA) [173], but these are still in their infancy.

8.2.2 Non-Functional Property Composition

The NFP specification provides a structure using a formal language and helps to manage those properties using the associated tools. The NFP models are utilized in the three spaces - problem, solution, and operation spaces; and are used to make developmental time decisions and runtime QoS resolution. The NFP policies that are defined by the user are used to compose and compare these properties. Compositions of NFPs are based on different composition theories, and, in addition, they are often not only the result of compositions of component properties, but also depend on other elements of a particular system architecture or even its environment. For example, determining the composition of component performance may depend on the scheduling policies and the system architecture. According to [174], NFPs can be classified in categories depending on the composition domains (i.e. type of parameters that determine the composition). However, due to the highly heterogeneous nature of these properties, there is no formally defined method to compose the components and estimate the emergent properties [175]. However, more research is required in this direction.

8.2.3 Runtime Models

The resolution of solution space might not always result in a static operational model. Typically the resolution process performed during development time result in a subset of solutions that are modeled as variation points in the architecture. This is attributed mainly due to certain context-based properties that can be estimated only during runtime. In these situations, the runtime models should reflect the most up-to-date information in order to perform online analysis and reasoning. In this direction, reference architectures are available from self-adaptive systems and models@runtime research [110]. A detailed discussion on addressing uncertainty in models can be found in [69]. A proposal for adapting the Eclipse Modelling Framework (EMF), for a more dynamic usage of models in the context of Models@Runtime is already made by the authors of [176]. Innovative approaches in reasoning on the presence of uncertainty and its influence during runtime needs to be studied.

Part IV

Annexes

Appendix A

Eclipse Modeling Framework and Supporting Plugins

A.1 Eclipse Modeling Framework

Eclipse is an open source software project dedicated to providing a robust, full-featured and commercial-quality platform for developing and supporting highly integrated software engineering tools [150]. The Eclipse platform defines a set of frameworks and common services that collectively make up the "integrationware" required to support a comprehensive tool integration platform. Except the small Eclipse runtime kernel, all the platform components are plug-in tools integrated seamlessly through predefined extension points [151]. Fundamentally, Eclipse is a framework for plug-ins. Besides its runtime kernel, the platform consists of the workbench, workspace, help, and team components. Other tools plug into this basic framework to create a usable application. Plug-ins can also define new extension points for others to extend. For example, a group of plug-ins implements the workbench user interface.

A.1.1 Metamodeling using Ecore

According to the Eclipse Foundation, the core EMF framework includes a metamodel (Ecore) for describing models and run-time support for the models, including change notification, persistence support with default XMI serialization, and a reflective API for manipulating EMF objects generically. In other words, Ecore defines the structure

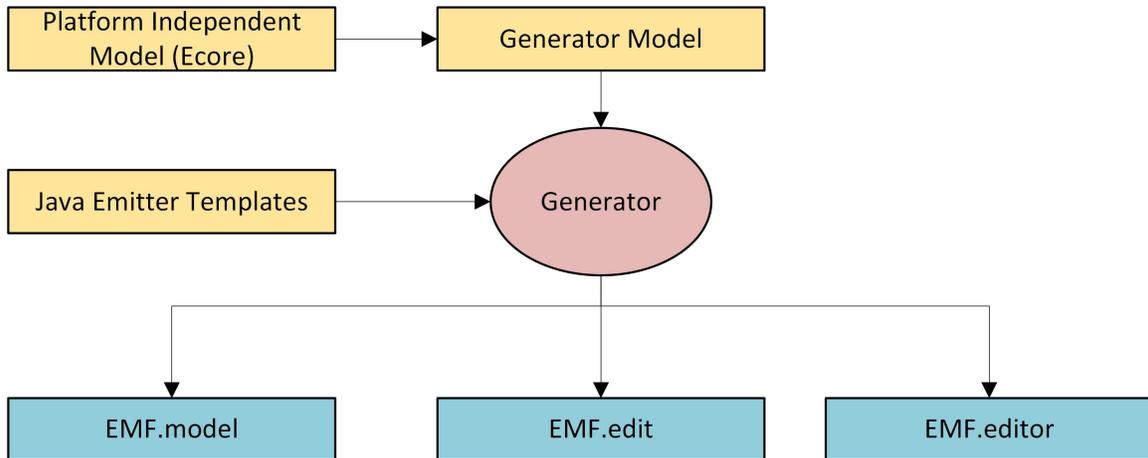


Figure A-1: Overview of the EMF tool set

of core models, which define the structure of the models developers use to maintain application data.

There are four Ecore classes needed to represent a model:

1. **EClass** is used to represent a modeled class. It has a name, zero or more attributes, and zero or more references.
2. **EAttribute** is used to represent a modeled attribute. Attributes have a name and a type.
3. **EReference** is used to represent one end of an association between classes. It has a name, a boolean flag to indicate if it represents containment, and a reference (target) type, which is another class.
4. **EDataType** is used to represent the type of an attribute. A data type can be a primitive type like int or float or an object type.

Basic templates and support code is generated from the Ecore model using the transformation engine provide by the EMF as illustrated in Figure A-1. Ecore and its XMI serialization, is the center of the EMF world. An Ecore model can be created from any of at least three sources: a UML model, an XML Schema, or annotated Java interfaces. Java implementation code and, optionally, other forms of the model can be generated from an Ecore model.

A.1.2 Graphical Modeling Workbench

Building a very flexible graphical editor for editing the models is really a labor-intensive task. EMF provides an object graph for representing models, as well as capabilities for (de)serializing models in a number of formats, checking constraints, and generating various types of tree editors for use in Eclipse. The Graphical Editor Framework (GEF) and Draw2D provide the foundations for building graphical views for EMF and other model types [150]. The Graphical Modeling Framework (GMF), by encapsulating GEF and Draw2D, provides a tool for creating graphical editor with a high degree of flexibility. Creation of editor in GMF is often complex and highly depends on Java, XML and Eclipse plug-in knowledge.

To implement graphical interfaces and tools for our methodology, we choose a graphical modeling workbench, Sirius, whose concepts of viewpoints and view lies closer to our approach. In addition, the structure of our framework description templates discussed in Chapter 5 makes it easier for the framework developer to implement it.

Sirius

Sirius framework is built on top of Graphical Modeling Framework (GMF) and uses interactive editors called "modelers" to create, visualize and edit models. Depending on the required visual representations, Sirius supports three different dialects (kinds of representations): diagrams (graphical modelers), tables, and trees (hierarchical representations), but new dialects can be added through programming [156]. Sirius provides possibility of analysis, roles and concerns of same data using different viewpoint on the same domain model. Sirius provides tools to specify the viewpoints which are relevant for user business domain which are usually specified as a EMF metamodel. Due to Sirius uses domain specification, which is not strictly in the scope of Sirius, it provides a graphical modeler for creating a DSM, which defines concepts and their relations in the abstract. After defining DSM models, Sirius allows easily creation of specific concrete representations of these models, and representations can be presented in more than one diagrams, tables, matrices (cross-tables) or hierarchies (trees) [157].

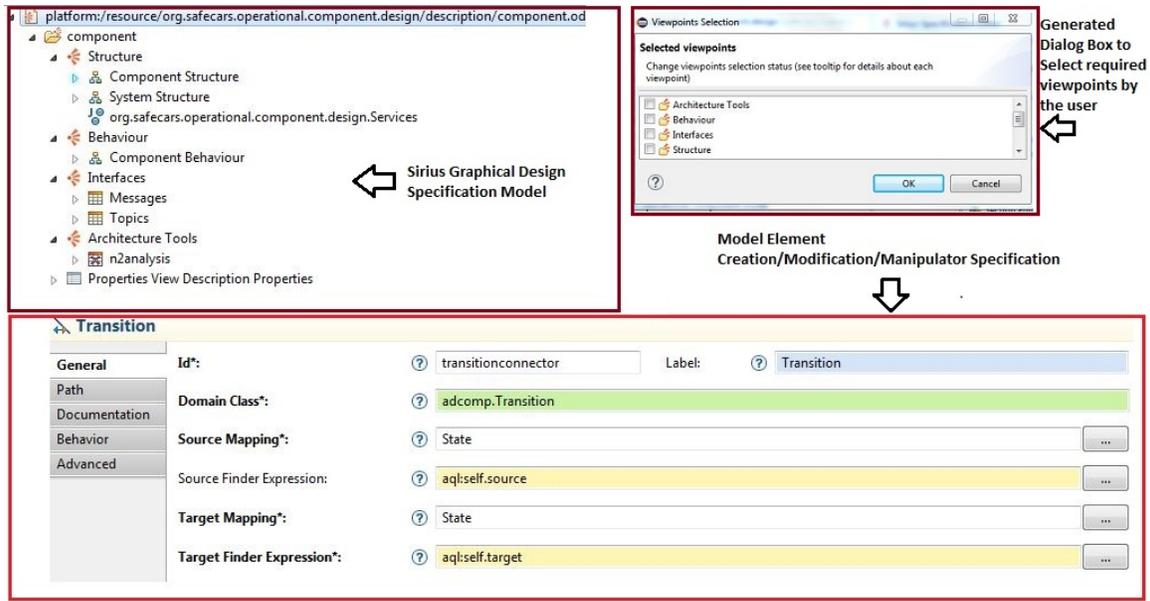


Figure A-2: Graphical Workbench Specification using Sirius

The representations are not static, and they complete modeling environments where user can create, modify and validate their designs. It can be logically organized in categories (viewpoints), which can be able or disabled by end-user, with purpose to provide a different, logically consistent, view on the same model. It can be concluded that Sirius simplifies the product, reduces design time and rapidly increases the overall productivity of building a domain-specific graphical editor. It uses Aceleo [177] as recommended language for defining expressions. By using a Java class as Java Extensions and Aceleo queries, defined in .mtl files, Sirius supports customization according to the particular user needs in form of service methods which is available inside all the representations defined in the viewpoint. Considering that Sirius encapsulates GMF, user can customize the program code on GMF level too. However, this is an advanced feature, because user must have a deep knowledge of GMF.

The five main concepts on which Sirius is based are stored using Viewpoint Specification Model (VSM). In the Sirius terminology, the following concepts are defined:

- viewpoint is a core element which is a logical set of representation specifications and representation extension specifications.
- representation is a group of graphical construction which represent domain data. It also describes the structure, appearance and behavior of models. There are four

representations (dialects) available: diagrams, tables, matrices and trees.

- mapping strongly depends of dialect and identifies a sub-set of the semantic model element that should appear in a representation and indicates how they should be represented.
- style is used to configure the visual appearance of the elements
- tool describes behaviors mapping.

Sirius works with models which describe semantics of editors - structure, appearance and behavior of dedicated representations and associated tools. For creation of editor, no Java code is necessary. A main disadvantage is need for interpreted expressions which will be evaluated at runtime to provide a behavior specific to domain and representations. An expression can be written in Aceleo [177], OCL [135] or Java language.

Property sheet views are widely used in our framework for editing the component properties. The Property Sheet view is used to display the properties of the current selection and modify their values. Views must implement the `IViewPart` interface and can be contributed to the workbench by extending the `org.eclipse.ui.views` extension point. However, more easier ways are currently available in the eclipse ecosystem. During the initial part of the framework development, we used Eclipse Exented Editing Framework (EEF) for creating property sheets. EEF is explained in the next section. Later, a new feature for property views was introduced in Sirius version 4.0 with many features like complex styling, validation, context etc. Now we suggest to use property views provided by Sirius for creating property sheets.

Exented Editing Framework

Extended Editing Framework (EEF) provides advanced editing components for the properties of EMF elements and a default generation based on standard metamodels.

Most of the concepts of the language can be configured using expressions based on various interpreters. By using EEF along with Sirius, all the interpreters can be leveraged and makes it available in Sirius and it is possible to use it out of Eclipse Sirius, with our own interpreter. By using EEF with AQL, we can navigate very

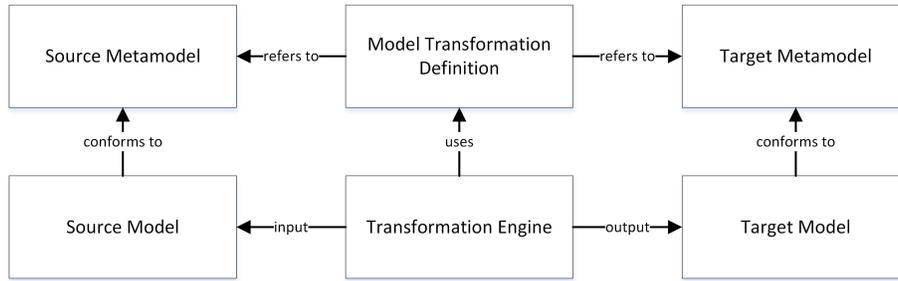


Figure A-3: Basic concept behind Model Transformation

easily in the concepts of your models to compute what to display and edit. In order to be as powerful to build a form-based user interface as Sirius is for diagrams, the language used by EEF is not linked to the EMF-based meta-models of the specifier.

A.1.3 Model Transformation Engines and Generators

The very basic concept of a model transformation on the highest level of abstraction is to translate one model to another model. Model translations can be of two types - endogenous and exogenous model transformation. For an endogenous model transformation we take a source model expressed in a modeling language and produce a target model expressed in the same modeling language. While an exogenous model transformation translates a source model expressed in one modeling language into a target model expressed in another modeling language [178]. It is essential that these models remain consistent, and therefore both the source and target model have to conform to their corresponding meta-models. Figure A-3 represents the basic concepts of a model transformation. The two concepts, transformation language and transformation engine are provided by model transformation environment. The main idea behind changing two models are to read a source model and write a target model. The transformation engine executes a set of guidelines provided by a transformation language that express how the target model is constructed. These guidelines is created from meta-data that are defined in the source and target meta-model to create an executable environment for the transformation engine

Transformation engines and generators analyze certain aspects of models and then synthesize various types of artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations. The ability to syn-

thesize artifacts from models helps ensure the consistency between application implementations and analysis information associated with functional and QoS requirements captured by models. This automated transformation process is often referred to as "correct-by-construction," as opposed to conventional handcrafted "construct-by-correction" software development processes that are tedious and error prone [1].

In our SafeRobots Framework, Model to Model (M2M) and Model to Text (M2T) transformation engines are widely used in many processes. We use Epsilon for defining templates for model transformation engines. Epsilon, standing for Extensible Platform of Integrated Languages for mOdel maNagement, is a platform for building consistent and interoperable task-specific languages for model management tasks such as model transformation, code generation, model comparison, merging, refactoring and validation. Epsilon is a family of languages and tools for code generation, model-to-model transformation, etc., that work out of the box with EMF and other types of models. M2M and M2T templates are discussed in the following sections.

Model to Model Transformation

Model to model transformation templates are defined using Epsilon Transformation Language (ETL). ETL provides all the standard features of a transformation language but also provides enhanced flexibility as it can transform many input to many output models, and can query/navigate/modify both source and target models. More specifically, ETL can be used to transform an arbitrary number of input models into an arbitrary number of output models of different modelling languages and technologies at a high level of abstraction. ETL adopts a hybrid style and features declarative rule specification using advanced concepts such as guards, abstract, lazy and primary rules, and automatic resolution of target elements from their source counterparts. Also, as ETL is based on EOL reuses its imperative features to enable users to specify particularly complex, and even interactive, transformations [155]. Figure A-5 show a snippet from the M2M ETL template for transforming a AMAL compliant model to ROS Middleware compliant model. In this case the target model conforms to the ROS metamodel as shown in Figure A-4. ETL transformations are organized in modules (EtlModule). A module can contain a number of transformation rules (TransformationRule). Each rule has a unique name (in the context of the module)

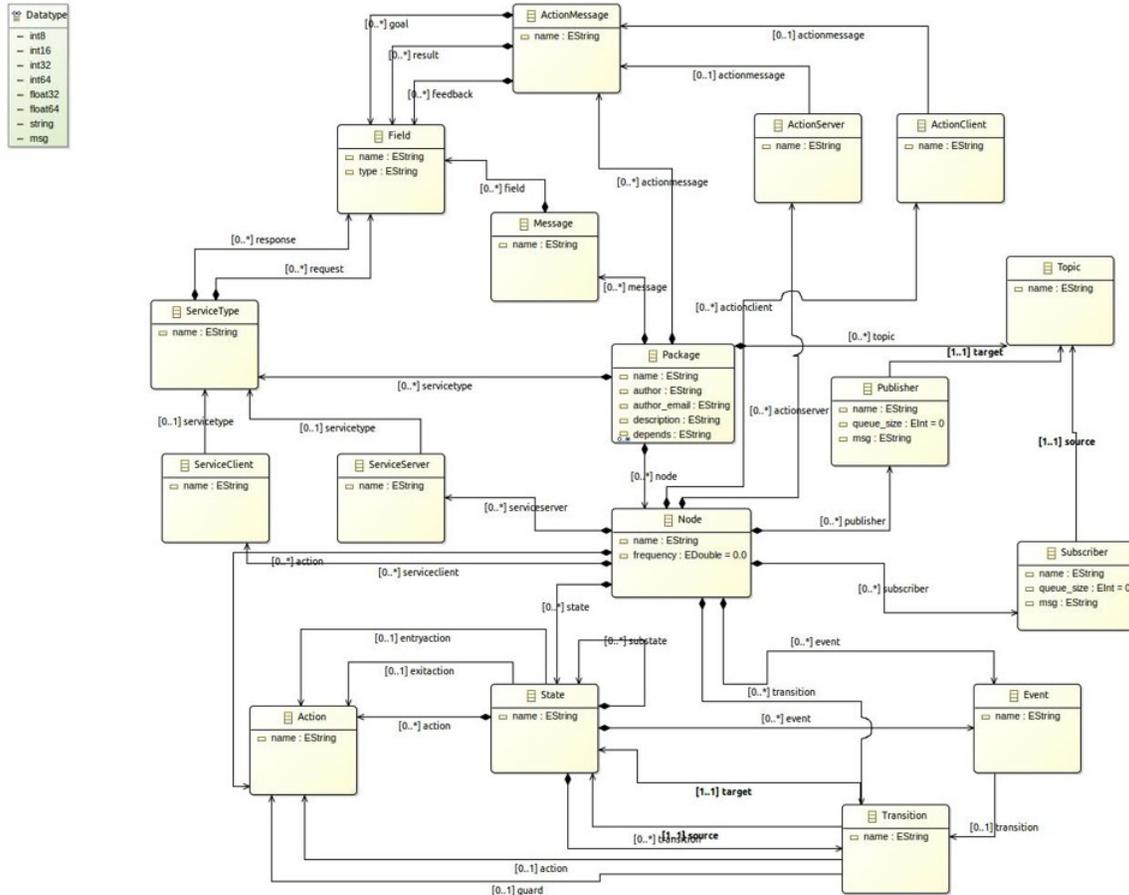


Figure A-4: Metamodel for ROS Middleware

and also specifies one source and many target parameters. A transformation rule can also extend a number of other transformation rules and be declared as abstract, primary and/or lazy.

Certain transformation are more complex and may have more than one intermediate model and may require user assistance for including missing information or for making decisions. For instance, solution space model are transformed in a user assisted way to operational models. This process comprises of transforming SSML models to intermediate MDP models and then, finally to operational model in a user assisted manner.

```

1 import "amalcompstate2rosstate.etl";
2
3 rule amalcomp2ros
4   transform t : adcomp!System
5   to apackage : ros!Package {
6     apackage.name = t.name;
7     apackage.author = t.author;
8     apackage.author_email = t.author_email;
9     apackage.description = t.description;
10    apackage.depends = t.depends;
11    for (comp in t.component ) {
12      var nd : ros!Node = new ros!Node;
13      nd.name = comp.name;
14      for (aProperty in comp.property) {
15        if (aProperty.name='required') {
16          nd.frequency = (aProperty.property->select(p|p.name='frequency')->first().value.asDouble());
17        }
18      }
19      //Creating Publisher and Subscriber from Component Ports
20      for (aport in comp.port) {
21        if (aport.role.name='Publisher') {
22          var pb = new ros!Publisher;
23          pb.name = aport.name;
24          pb.queue_size = 100;//tbd
25          //setting the msg
26          var msg_name = aport.role.eContainer().property->select(p|p.name='required')->first().property.;
27          pb.msg = msg_name;
28        }
29      }
30    }
31  }

```

The M2M transformer uses the AMAL semantic extension capability to find the mandatory elements that must be transformed

Information describing the component ports are used to create Publisher and Subscriber Model elements in the target implementation model

Figure A-5: Code Snippet showing a M2M ETL template for transforming a AMAL compliant model to ROS Middleware-based model

Model to Text Transformation

Code generation is a Model to Text transformation that translates a source model that is described by a DSL and produce a target model that usually is described by a general purpose programming language, such as Java or C++.

The code generation templates in SafeRobots framework are defined in Epsilon Generation Language (EGL). EGL provides a language tailored for model-to-text transformation (M2T). EGL can be used to transform models into various types of textual artefact, including executable code (e.g. Java), reports (e.g. in HTML), images (e.g. using DOT), formal specifications (e.g. Z notation), or even entire applications comprising code in multiple languages (e.g. HTML, Javascript and CSS). EGL is a template-based code generator (i.e. EGL programs resemble the text that they generate), and provides several features that simplify and support the generation of text from models, including: a sophisticated and language-independent merging engine (for preserving hand-written sections of generated text), an extensible template system (for generating text to a variety of sources, such as a file on disk, a database server, or even as a response issued by a web server), formatting algorithms (for producing generated text that is well-formatted and hence readable), and traceability

<pre> 46 namespace sc = boost::statechart; 47 namespace mpl = boost::mpl; 48 49 /// 50 namespace [%=node.name()]_namespace{ 51 52 class [%=node.name()]_config 53 { 54 }; 55 56 class [%=node.name()]_data 57 { 58 // autogenerated: don't touch this class 59 public: 60 //input data 61 [%for (sub in node.subscriber) { %] 62 [%sub.msg%] in_[%=sub.name%]; 63 [%] 64 //output data 65 [%for (pub in node.publisher) { %] 66 [%pub.msg%] out_[%=pub.name%]; 67 [%] 68 [%for (pub in node.publisher) { %] 69 bool out_[%=pub.name%]_active; 70 [%] 71 //service clients 72 [%for (serviceclient in node.serviceclient) { %] 73 ros::ServiceClient [%=serviceclient.name%]_client; 74 [%] 75 76 }; 77 78 template< class T > 79 boost::intrinsic_ptr< T > MakeIntrinsic(T * pObject) 80 { 81 return boost::intrinsic_ptr< T >(pObject); 82 } 83 84 typedef std::allocator< void > [%=node.name()]Allocator; 85 typedef sc::fifo_scheduler<> [%=node.name()]Scheduler; 86 </pre>	<pre> 2 3 void *do_[%=state.class_name()%(void *data) 4 { 5 [%=state.node_name()%([%=state.node_name()%(object = ([%=state.node_name()%(6 [%=out.startPreserve("/%", "%/", "do activity", true)%] 7 ROS_INFO("[%=state.node_name()%(("[%=state.name()%(Do Activity"); 8 [%=out.stopPreserve()%(9 pthread_exit(NULL); 10 } 11 12 [%=state.class_name()%(::~[%=state.class_name()%(my_context ctx): my_base(ctx) 13 { 14 [%=out.startPreserve("/%", "%/", "entry", true)%] 15 ROS_INFO("[%=state.node_name()%(::~[%=state.name()%(Entered"); 16 [%=out.stopPreserve()%(17 pthread_create(&thread_[%=state.class_name()%(, NULL, do_[%=state.class_name()%(, 18 //pthread_join(thread_[%=state.class_name()%(, NULL); 19 } 20 21 [%=state.class_name()%(::~[%=state.class_name()%(22 { 23 [%=out.startPreserve("/%", "%/", "exit", true)%] 24 ROS_INFO("[%=state.node_name()%(::~[%=state.name()%(Exiting"); 25 [%=out.stopPreserve()%(26 pthread_cancel(thread_[%=state.class_name()%(); 27 } 28 29 [%var transition_event = false;%] 30 [%for (action in state.action) { %] 31 [%transition_event = false;%] 32 [%for (event in state.event) { %] 33 [%if (event.transition.action.name = action.name){%] 34 [%transition_event = true;%] 35 void [%=state.class_name()%(::~[%=action.name%](const [%=event.name%] & evt) 36 { 37 //ROS_INFO("[%=action.name%]"); 38 [%=out.startPreserve("/%", "%/", "action", true)%] 39 [%=out.stopPreserve()%(40 } 41 [%] 42 [%] </pre>
--	--

Figure A-6: Code Snippet showing a M2T EGL template for generating C++ code from a ROS + Boost Statecharts model

mechanisms (for linking generated text with source models). EGL provides language constructs that allow M2T transformations to designate regions of generated text as protected. Whenever an EGL program attempts to generate text, any protected regions that are encountered in the specified destination are preserved. Within an EGL program, protected regions are specified with the preserve(String, String, String, Boolean, String) method on the out keyword. The first two parameters define the comment delimiters of the target language. The other parameters provide the name, enable-state and content of the protected region, as shown below.

```

[%=out.preserve("/%", "%/", "anId", true,
//user code here
%]

```

Bibliography

- [1] D. C. Schmidt, "Model-driven engineering," *IEEE Computer Society*, vol. 39, no. 2, p. 25, 2006.
- [2] R. Hilliard, "IEEE-STD-1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems," *IEEE*, vol. 12, no. 16-20, 2000. [Online]. Available: <http://standards.ieee.org>
- [3] A. Harris and J. M. Conrad, "Survey of popular robotics simulators, frameworks, and toolkits," in *Southeastcon, 2011 Proceedings of IEEE*. IEEE, 2011, pp. 243–249.
- [4] M. J. Mataric, "Designing emergent behaviors: From local interactions to collective intelligence," in *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, 1993, pp. 432–441.
- [5] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*. IEEE, 2008, pp. 736–742.
- [6] S. Scotchmer, "Standing on the Shoulders of Giants: Cumulative Research and the Patent Law," *The Journal of Economic Perspectives*, pp. 29–41, 1991.
- [7] D. Brugali and E. Prassler, "Software Engineering for Robotics [From the Guest Editors]," *IEEE Robotics & Automation Magazine*, vol. 16, no. 1, pp. 9–15, 2009.
- [8] N. Medvidovic, H. Tajalli, J. Garcia, I. Krka, Y. Brun, and G. Edwards, "Engineering Heterogeneous Robotics Systems: A Software Architecture Based Approach," *IEEE Computer Society*, vol. 44, no. 5, pp. 62–71, 2011.

- [9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an Open-source Robot Operating System,” in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.
- [10] H. Bruyninckx, “Open robot control software: the OROCOS project,” in *International Conference on Robotics and Automation ICRA.*, vol. 3. IEEE, 2001, pp. 2523–2528.
- [11] B. Gerkey, R. T. Vaughan, and A. Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems,” in *Proceedings of the 11th international conference on advanced robotics*, vol. 1, 2003, pp. 317–323.
- [12] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O’reilly.
- [13] R. B. Rusu and S. Cousins, “3D is here: Point cloud library (PCL),” in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2011, pp. 1–4.
- [14] N. Koenig and A. Howard, “Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3. IEEE, 2004, pp. 2149–2154.
- [15] C. Schlegel, T. Haßler, A. Lotz, and A. Steck, “Robotic Software Systems: From Code-Driven to Model-Driven Software Development,” in *International Conference on Advanced Robotics (ICAR)*. IEEE, 2009, pp. 1–8.
- [16] R. France and B. Rumpe, “Model-driven Development of Complex Software: A Research Roadmap,” in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.
- [17] M. Tornngren, D. Chen, and I. Crnkovic, “Component-based vs. Model-based Development: A Comparison in the Context of Vehicular Embedded Systems,” in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2005, pp. 432–440.

- [18] M. Klotzbuecher, N. Hochgeschwender, L. Gherardi, H. Bruyninckx, G. Kraetzschmar, D. Brugali, A. Shakhimardanov, J. Paulus, M. Reckhaus, H. Garcia *et al.*, “The BRICS Component Model: A Model-Based Development Paradigm For Complex Robotics Software Systems,” in *28th ACM Symposium on Applied Computing (SAC), Coimbra, Portugal*, 2013.
- [19] S. Dhoub, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, “RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications,” in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2012, pp. 149–160.
- [20] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez, “V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development,” *Journal of Software Engineering for Robotics*, vol. 1, no. 1, pp. 3–17, 2010.
- [21] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, “Cheddar : a Flexible Real Time Scheduling Framework,” in *ACM SIGAda Ada Letters*, vol. 24, no. 4. ACM, 2004, pp. 1–8.
- [22] A. Ramaswamy, B. Monsuez, and A. Tapus, “Component Based Decision Architecture for Reliable Autonomous Systems,” in *International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, 2013, pp. 605–610.
- [23] *OMG: Model-Driven Architecture*, Object Management Group Std. [Online]. Available: <http://www.omg.org/mda/>
- [24] T. De Laet, S. Bellens, R. Smits, E. Aertbeliën, H. Bruyninckx, and J. De Schutter, “Geometric Relations between Rigid Bodies: Semantics for Standardization,” *IEEE Robotics and Automation Magazine*, 2012.
- [25] T. De Laet, W. Schaekers, J. de Greef, and H. Bruyninckx, “Domain Specific Language for Geometric Relations between Rigid Bodies targeted to robotic applications,” *arXiv:1304.1346*, 2013.
- [26] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A framework for simulating and prototyping heterogeneous systems,” 1994.

- [27] T. Kotoku and M. Mizukawa, “Robot Middleware and its Standardization in OMG-Report on OMG Technical Meetings in St. Louis and Boston,” in *SICE-ICASE, 2006. International Joint Conference*. IEEE, 2006, pp. 2028–2031.
- [28] RoSta, “Robot Standards and Reference Architectures,” <http://www.robot-standards.eu/index.php?id=8>, accessed January 30, 2014.
- [29] A. Steck and C. Schlegel, “Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development,” *arXiv:1009.4877*, 2010.
- [30] J. F. Inglés-Romero, A. Lotz, C. V. Chicote, and C. Schlegel, “Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties,” *arXiv preprint arXiv:1303.4296*, 2013.
- [31] L. Gherardi and D. Brugali, “An eclipse-based Feature Models toolchain,” in *Proc. of the 6th Workshop of the Italian Eclipse Community (Eclipse-IT)*, 2011.
- [32] A. Lotz, J. F. Inglés-Romero, D. Stampfer, M. Lutz, C. Vicente-Chicote, and C. Schlegel, “Towards a stepwise variability management process for complex systems: A robotics perspective,” in *Artificial Intelligence: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2017, pp. 2411–2430.
- [33] G. Lortal, S. Dhouib, and S. Gérard, “Integrating Ontological Domain Knowledge into a Robotic DSL,” in *Models in Software Engineering*. Springer, 2011, pp. 401–414.
- [34] A. Solberg, D. Simmonds, R. Reddy, S. Ghosh, and R. France, “Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development,” in *29th Annual International on Computer Software and Applications Conference, 2005. COMPSAC 2005.*, vol. 1. IEEE, 2005, pp. 121–126.
- [35] V. Kulkarni and S. Reddy, “Separation of Concerns in Model-driven Development,” *Software, IEEE*, vol. 20, no. 5, pp. 64–69, 2003.
- [36] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, “Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering,” in

- International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2010, pp. 324–335.
- [37] J. F. Inglés Romero, C. Vicente Chicote, B. Morin, and O. Barais, “Using Models@ Runtime for Designing Adaptive Robotics Software: an Experience Report,” 2010.
- [38] P. E. Agre and D. Chapman, “What are plans for?” *Robotics and autonomous systems*, vol. 6, no. 1, pp. 17–34, 1990.
- [39] R. A. Brooks, “A Robust Layered Control System for a Mobile Robot,” *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.
- [40] E. Gat *et al.*, “On three-layer architectures,” *Artificial intelligence and mobile robots: case studies of successful robot systems*, vol. 195, 1998.
- [41] J. S. Albus, R. Quintero, and R. Lumia, “Overview of NASREM: The NASA/NBS standard reference model for telerobot control system architecture,” *NASA Technical Report*, vol. 95, p. 12854, 1994.
- [42] R. Simmons and D. Apfelbaum, “A task description language for robot control,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3. IEEE, 1998, pp. 1931–1937.
- [43] A. A. Medeiros, “A survey of control architectures for autonomous mobile robots,” *Journal of the Brazilian Computer Society*, vol. 4, no. 3, 1998.
- [44] J. R. Anderson, M. Matessa, and C. Lebiere, “ACT-R: A theory of higher level cognition and its relation to visual attention,” *Human-Computer Interaction*, vol. 12, no. 4, pp. 439–462, 1997.
- [45] P. Langley and D. Choi, “A unified cognitive architecture for physical agents,” in *Proceedings of the National Conference on Artificial Intelligence*, vol. 21, no. 2. MIT Press, 2006, p. 1469.
- [46] D. Garlan, “Formal modeling and analysis of software architecture: Components, connectors, and events,” in *Formal Methods for Software Architectures*. Springer, 2003, pp. 1–24.

- [47] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.
- [48] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Elsevier, 2011.
- [49] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard, “Marte: Also an uml profile for modeling aadl applications,” in *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*. IEEE, 2007, pp. 359–364.
- [50] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, “An architecture for autonomy,” *The International Journal of Robotics Research*, vol. 17, no. 4, pp. 315–337, 1998.
- [51] ISO, “Systems and Software Engineering-Architecture Description,” Tech. Rep., 2011.
- [52] H.-M. Huang, J. Albus, J. Kotora, and R. Liu, “Robotic Architecture Standards Framework in the Defense Domain with Illustrations Using the NIST 4D/RCS Reference Architecture,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, vol. 3. IEEE, 2003, pp. 2415–2420.
- [53] J. S. Albus, “4D/RCS: A reference model architecture for intelligent unmanned ground vehicles,” in *AeroSense 2002*. International Society for Optics and Photonics, 2002, pp. 303–310.
- [54] “Selecting a development approach,” Web Article, Feb. 2005. [Online]. Available: <https://www.cms.gov/research-statistics-data-and-systems/cms-information-technology/xlc/downloads/selectingdevelopmentapproach.pdf>
- [55] P. Varley, “Techniques for development of safety-related software for surgical robots,” *IEEE Transactions on information technology in biomedicine*, vol. 3, no. 4, pp. 261–267, 1999.

- [56] T. Bailey and H. Durrant-Whyte, “Simultaneous localization and mapping (SLAM): Part II,” *IEEE Robotics & Automation Magazine*, vol. 13, no. 3, pp. 108–117, 2006.
- [57] C. Szyperski, *Component software: beyond object-oriented programming*. Pearson Education, 2002.
- [58] A. Ramaswamy, B. Monsuez, and A. Tapus, “Formal models for cognitive systems,” in *International Conference on Advanced Robotics (ICAR)*. IEEE, 2013.
- [59] D. Brugali and P. Scandurra, “Component-Based Robotic Engineering (Part I) [Tutorial],” *Robotics & Automation Magazine, IEEE*, vol. 16, no. 4, pp. 84–96, 2009.
- [60] M. Radestock and S. Eisenbach, “Coordination in evolving systems,” in *Trends in Distributed Systems CORBA and Beyond*. Springer, 1996, pp. 162–176.
- [61] P. Gärdenfors, *Conceptual Spaces: The Geometry of Thought*. MIT press, 2004.
- [62] S. Bechhofer, “OWL: Web ontology language,” in *Encyclopedia of Database Systems*. Springer, 2009.
- [63] T. Haidegger, M. Barreto, P. Gonçães, M. K. Habib, S. K. V. Ragavan, H. Li, A. Vaccarella, R. Perrone, and E. Prestes, “Applied ontologies and standards for service robots,” *Robotics and Autonomous Systems*, vol. 61, no. 11, pp. 1215–1223, 2013.
- [64] C. Schlenoff, E. Prestes, R. Madhavan, P. Goncalves, H. Li, S. Balakirsky, T. Kramer, and E. Miguelanez, “An IEEE standard ontology for robotics and automation,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2012, pp. 1337–1342.
- [65] P. Gardenfors, “Conceptual spaces as a framework for knowledge representation,” *Mind and Matter*, vol. 2, no. 2, pp. 9–27, 2004.
- [66] A. Chella, M. Frixione, and S. Gaglio, “A cognitive architecture for artificial vision,” *Artificial Intelligence*, vol. 89, no. 1-2, pp. 73–111, 1997.

- [67] N. Hochgeschwender, S. Schneider, H. Voos, and G. K. Kraetzschmar, “Declarative specification of robot perception architectures,” in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 291–302.
- [68] S. Blumenthal, N. Hochgeschwender, E. Prassler, H. Voos, and H. Bruyninckx, “An approach for a distributed world model with QoS-based perception algorithm adaptation,” in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2015, pp. 1806–1811.
- [69] H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Wätzoldt, and S. Clarke, “Living with Uncertainty in the Age of Runtime Models,” in *Models@run.time*. Springer, 2014, pp. 47–100.
- [70] M. Klotzbücher and H. Bruyninckx, “Coordinating robotic tasks and systems with rFSM Statecharts,” *JOSER: Journal of Software Engineering for Robotics*, vol. 3, no. 1, pp. 28–56, 2012.
- [71] N. Hochgeschwender, L. Gherardi, A. Shakhirmardanov, G. K. Kraetzschmar, D. Brugali, and H. Bruyninckx, “A model-based approach to software deployment in robotics,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2013*. IEEE, 2013, pp. 3907–3914.
- [72] J.-A. Fernández-Madrigo, C. Galindo, J. González, E. Cruz-Martín, and A. Cruz-Martín, “A software engineering approach for the development of heterogeneous robotic applications,” *Robotics and Computer-Integrated Manufacturing*, vol. 24, no. 1, pp. 150–166, 2008.
- [73] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegel, “The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software,” *Special Issue on Domain-Specific Languages and Models in Robotics, Journal of Software Engineering for Robotics (JOSER)*, 2016.
- [74] L. Gherardi and D. Brugali, “Modeling and Reusing Robotic Software Architectures: the HyperFlex toolchain,” in *IEEE International Conference on Robotics*

- and Automation (ICRA 2014)*. Hong Kong, China: IEEE, May 31 - June 5 2014.
- [75] B. Suleiman, V. Tasic, and E. Aliev, “Non-functional property specifications for WRIGHT ADL,” in *8th IEEE International Conference on Computer and Information Technology*. IEEE, 2008, pp. 766–771.
- [76] L. Chung and J. C. S. do Prado Leite, “On Non-Functional Requirements in Software Engineering,” in *Conceptual modeling: Foundations and applications*. Springer, 2009, pp. 363–379.
- [77] MARTE, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*, 1st ed., Object Management Group, June 2011.
- [78] I. Jacobson, G. Booch, and J. E. Rumbaugh, *The unified software development process—the complete guide to the unified process from the original designers*. Addison-Wesley, 1999.
- [79] A. Tapus, M. J. Mataric, and B. Scassellati, “Socially assistive robotics,” *IEEE Robotics and Automation Magazine*, vol. 14, no. 1, p. 35, 2007.
- [80] A. Steinfeld, T. Fong, D. Kaber, M. Lewis, J. Scholtz, A. Schultz, and M. Goodrich, “Common metrics for human-robot interaction,” in *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*. ACM, 2006, pp. 33–40.
- [81] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic, “Integration of extra-functional properties in component models,” in *International Symposium on Component-Based Software Engineering*. Springer, 2009, pp. 173–190.
- [82] A. Sangiovanni-Vincentelli and M. Di Natale, “Embedded system design for automotive applications,” *Computer*, vol. 40, no. 10, pp. 42–51, 2007.
- [83] N. S. Rosa, P. R. Cunha, and G. R. Justo, “Process(NFL): a language for describing non-functional properties,” in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. IEEE, 2002, pp. 3676–3685.

- [84] D. Cancila, R. Passerone, T. Vardanega, and M. Panunzio, "Toward correctness in the specification and handling of non-functional attributes of high-integrity real-time embedded systems," *IEEE Transactions on industrial informatics*, vol. 6, no. 2, pp. 181–194, 2010.
- [85] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Round-trip support for extra-functional property management in model-driven engineering of embedded systems," *Information and Software Technology*, vol. 55, no. 6, pp. 1085–1100, 2013.
- [86] N. A. Taatgen, C. Lebiere, and J. R. Anderson, "Modeling paradigms in ACT-R," *Cognition and multi-agent interaction: From cognitive modeling to social simulation*, pp. 29–52, 2006.
- [87] D. D. Salvucci, "Modeling driver behavior in a cognitive architecture," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 48, no. 2, pp. 362–380, 2006.
- [88] S. Fenn, A. Mendes, and D. M. Budden, "Addressing the non-functional requirements of computer vision systems: a case study," *Machine Vision and Applications*, vol. 27, no. 1, pp. 77–86, 2016.
- [89] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "Exploiting non-functional preferences in architectural adaptation for self-managed systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 431–438.
- [90] D. R. Olsen and M. A. Goodrich, "Metrics for evaluating human-robot interactions," in *Proceedings of PERMIS*, vol. 2003, 2003, p. 4.
- [91] J. L. Burke, R. R. Murphy, D. R. Riddle, and T. Fincannon, "Task performance metrics in human-robot interaction: Taking a systems approach," DTIC Document, Tech. Rep., 2004.
- [92] J. A. Saleh and F. Karray, "Towards generalized performance metrics for human-robot interaction," in *2010 International Conference on Autonomous and Intelligent Systems (AIS)*. IEEE, 2010, pp. 1–6.

- [93] C. R. Burghart and A. Steinfeld, "Proceedings of Metrics for Human-Robot Interaction, a Workshop at ACM/IEEE HRI," School of Computer Science, University of Hertfordshire, Hatfield, UK, Tech. Rep. Technical Report 471, 2008.
- [94] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the first workshop on Self-healing systems*. ACM, 2002, pp. 27–32.
- [95] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *Software Engineering, IEEE Transactions on*, vol. 18, no. 6, pp. 483–497, 1992.
- [96] QoSFT, *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification*, version 1.1 ed., Object Management Group, April 2008.
- [97] SPT, *UML Profile for for Schedulability, Performance, and Time Specification*, version 1.1 ed., Object Management Group, January 2005.
- [98] G. Dobson, R. Lock, and I. Sommerville, "QoSOnt: a QoS ontology for service-centric systems," in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2005, pp. 80–87.
- [99] J. Morse, D. Araiza-Illan, J. Lawry, A. Richards, and K. Eder, "Formal specification and analysis of autonomous systems under partial compliance," *arXiv preprint arXiv:1603.01082*, 2016.
- [100] A. M. Davis, *Software requirements: objects, functions, and states*. Prentice-Hall, Inc., 1993.
- [101] A. Ramaswamy, B. Monsuez, and A. Tapus, "Formal models for cognitive systems," in *16th International Conference on Advanced Robotics (ICAR), Montevideo, Uruguay*. IEEE, November 2013, pp. 1–8.
- [102] S. Russell and P. Norvig, "Artificial Intelligence - A modern approach," *Prentice-Hall*, vol. 25, 1995.

- [103] S. Sentilles, “Managing extra-functional properties in component-based development of embedded systems,” Ph.D. dissertation, Pau, 2012.
- [104] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of probabilistic real-time systems,” in *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.
- [105] D. Vanthienen, M. Klotzbuecher, and H. Bruyninckx, “The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming,” *JOSE: Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 17–35, 2014.
- [106] P. Heymans and J.-C. Trigaux, “Software product lines: State of the art,” 2003.
- [107] L. Gherardi, “Variability modeling and resolution in component-based robotics systems,” Università degli studi di Bergamo, 2013.
- [108] L. Manso, P. Bachiller, P. Bustos, P. Núñez, R. Cintas, and L. Calderita, “RoboComp: a tool-based robotics framework,” in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2010, pp. 251–262.
- [109] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli, “Managing non-functional uncertainty via model-driven adaptivity,” in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 33–42.
- [110] N. Bencomo, R. France, B. H. C. Cheng, and U. Abmann, *Models@run.time, Foundations, Applications, and Roadmaps*. Springer, 2014.
- [111] K. Welsh and P. Sawyer, “Understanding the scope of uncertainty in dynamically adaptive systems,” in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2010, pp. 2–16.
- [112] K. Welsh, P. Sawyer, and N. Bencomo, “Run-time resolution of uncertainty,” in *19th IEEE International Conference on Requirements Engineering*. IEEE, 2011, pp. 355–356.

- [113] D. Kortenkamp, R. Simmons, and D. Brugali, “Robotic systems architectures and programming,” in *Springer Handbook of Robotics*. Springer, 2016, pp. 283–306.
- [114] B. Argall, B. Browning, and M. Veloso, “Learning to select state machines using expert advice on an autonomous robot,” in *IEEE International Conference on Robotics and Automation*. IEEE, 2007, pp. 2124–2129.
- [115] G. Kim and W. Chung, “Navigation behavior selection using generalized stochastic petri nets for a service robot,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 4, pp. 494–503, 2007.
- [116] A. Nordmann, N. Hochgeschwender, and S. Wrede, “A survey on domain-specific languages in robotics,” in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 195–206.
- [117] C. Lesire, D. Doose, and H. Cassé, “Validation of real-time properties of a robotic software architecture,” in *6th National Conference on Control Architectures of Robots*, 2011, p. 7 p.
- [118] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.
- [119] R. C. Arkin, *Behavior-based robotics*. MIT press, 1998.
- [120] H.-Q. Chong, A.-H. Tan, and G.-W. Ng, “Integrated cognitive architectures: a survey,” *Artificial Intelligence Review*, vol. 28, no. 2, pp. 103–130, 2007. [Online]. Available: <http://dx.doi.org/10.1007/s10462-009-9094-9>
- [121] D. Emery and R. Hilliard, “Every architecture description needs a framework: Expressing architecture frameworks using ISO/IEC 42010,” in *Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*. IEEE, 2009, pp. 31–40.

- [122] A. Smeda, M. Oussalah, and T. Khammaci, "MADL: Meta Architecture Description Language," in *Third ACIS International Conference on Software Engineering Research, Management and Applications*. IEEE, 2005, pp. 152–159.
- [123] D. Emery and R. Hilliard, "Updating IEEE 1471: architecture frameworks and other topics," in *Seventh Working IEEE/IFIP Conference on Software Architecture WICSA*. IEEE, 2008, pp. 303–306.
- [124] "ISO/IEC 42010: Systems and software engineering, Recommended practice for architectural description of software-intensive systems," International Standardization Organization, Tech. Rep., 2007.
- [125] I. Standard, "IEEE: ISO/IEC/IEEE 42010: Systems and Software Engineering - Architecture Description," *Proceedings of Technical Report*, 2011.
- [126] D. Garlan and D. E. Perry, "Introduction to the special issue on software architecture," *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 269–274, 1995.
- [127] E. Scioni, N. Huebel, S. Blumenthal, A. Shakhimardanov, M. Klotzbuecher, H. Garcia, and H. Bruyninckx, "Hierarchical Hypergraphs for Knowledge-centric Robot Systems: a Composable Structural Meta Model and its Domain Specific Language NPC4," *Journal of Software Engineering for Robotics*, vol. 7, no. 1, pp. 55–74, 2016.
- [128] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein, "BayesStore: Managing large, uncertain data repositories with probabilistic graphical models," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 340–351, 2008.
- [129] A. Doucet, N. De Freitas, K. Murphy, and S. Russell, "Rao-blackwellised particle filtering for dynamic bayesian networks," in *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 2000, pp. 176–183.
- [130] H. Zender, O. M. Mozos, P. Jensfelt, G.-J. M. Kruijff, and W. Burgard, "Conceptual spatial representations for indoor mobile robots," *Robotics and Autonomous Systems*, vol. 56, no. 6, pp. 493–502, 2008.

- [131] M. Tenorth, L. Kunze, D. Jain, and M. Beetz, “Knowrob: Map-knowledge linked semantic object maps,” in *10th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. IEEE, 2010, pp. 430–435.
- [132] F. Piltan, M. H. Yarmahmoudi, M. Shamsodini, E. Mazlomian, and A. Hosainpour, “PUMA-560 robot manipulator position computed torque control methods using Matlab/Simulink and their integration into graduate nonlinear control and Matlab courses,” *International Journal of Robotics and Automation*, vol. 3, no. 3, pp. 167–191, 2012.
- [133] D. Brugali, A. Brooks, A. Cowley, C. Cote, A. C. Dominguez-Brito, D. Letourneau, F. Michaud, and C. Schlegel, “Trends in component-based robotics,” in *Software Engineering for Experimental Robotics*. Springer, 2007, pp. 135–142.
- [134] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.
- [135] *Object Constraint Language*, Object Management Group Std. Version 2.4, 2014. [Online]. Available: <http://www.omg.org/spec/OCL/2.4>
- [136] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and analysis of system architecture using rapide,” *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–354, 1995.
- [137] R. Allen and D. Garlan, “A formal basis for architectural connection,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 3, pp. 213–249, 1997.
- [138] OMG, “Systems modeling language (OMG SysML) specification,” *Object Management Group, OMG Available Specification (September 2007)*, 2007.
- [139] M.-E. Iacob, H. Jonkers, M. M. Lankhorst, and H. A. Proper, *ArchiMate 1.0 Specification*. Zaltbommel: Van Haren Publishing, 2009.

- [140] D. Garlan, R. Monroe, and D. Wile, “ACME: An Architecture Description Interchange Language,” in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 159–173.
- [141] P. C. Clements, “A survey of architecture description languages,” in *Proceedings of the 8th international workshop on software specification and design*. IEEE Computer Society, 1996, p. 16.
- [142] D. T. Ross, “Structured analysis (SA): A language for communicating ideas,” *IEEE Transactions on software engineering*, no. 1, pp. 16–34, 1977.
- [143] B. Nuseibeh, J. Kramer, and A. Finkelstein, “A framework for expressing the relationships between multiple views in requirements specification,” *IEEE Transactions on software engineering*, vol. 20, no. 10, pp. 760–773, 1994.
- [144] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting software architectures: views and beyond*. Pearson Education, 2002.
- [145] P. B. Kruchten, “The 4+ 1 view model of architecture,” *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995.
- [146] A. Abd-Allah and B. Boehm, “Reasoning about the composition of heterogeneous architectures,” *USC Center for Software Engineering, Computer Science Department, Univ. of Southern California*, 1995.
- [147] T. A. Smeda and M. Oussalah, “Meta architecting: Towards a new generation of architecture description languages,” *Journal of Computer Science*, vol. 1, no. 4, pp. 454–460, 2005.
- [148] R. J. Allen, “A formal approach to software architecture.” Tech. Rep., 1997.
- [149] D. E. Emery, R. F. Hilliard II, and T. B. Rice, “Experiences applying a practical architectural method,” in *International Conference on Reliable Software Technologies*. Springer, 1996, pp. 471–484.
- [150] E. Merks, R. Eliersick, T. Grose, F. Budinsky, and D. Steinberg, “The Eclipse Modeling Framework,” *Addison Wesley*, p. 37, 2003.

- [151] Z. Yang and M. Jiang, “Using Eclipse as a tool-integration platform for software development,” *IEEE Software*, vol. 24, no. 2, 2007.
- [152] F. Budinsky, *Eclipse modeling framework: a developer’s guide*. Addison-Wesley Professional, 2004.
- [153] Eclipse and Graphical Modeling Framework, “Gmf.” [Online]. Available: <https://www.eclipse.org/modeling/gmp/>
- [154] V. Vijovi, M. Maksimovi, and B. Perisi, “Sirius: A rapid development of dsm graphical editor.” *IEEE*, 2014, pp. 233–238.
- [155] D. Kolovos, L. Rose, A. Garcia-Dominguez, and R. Paige, “The Epsilon Book (2010),” 2012.
- [156] *Sirius Specifier Manual*. [Online]. Available: <http://www.eclipse.org/sirius/doc/>
- [157] V. Vujovi, M. Maksimovi, and B. Periei, “Comparative analysis of DSM graphical editor frameworks: Graphiti vs. Sirius,” in *23rd International Electrotechnical and Computer Science Conference*, 2014, pp. 7–10.
- [158] W. Pree, “Meta patterns, a means for capturing the essentials of reusable object-oriented design,” *Object-oriented programming*, pp. 150–162, 1994.
- [159] Y. J. Yang, S. Y. Kim, G. J. Choi, E. S. Cho, C. J. Kim, and S. D. Kim, “A UML-based object-oriented framework development methodology,” in *Software Engineering Conference*. *IEEE*, 1998, pp. 211–218.
- [160] K. Koskimies and H. Mössenböck, “Designing a framework by stepwise generalization,” *Software Engineering*, pp. 479–498, 1995.
- [161] D. Wilson and S. Wilson, “Writing frameworks-capturing your expertise about a problem domain,” *Tutorial Notes, OOPSLA*, 1993.
- [162] A. Van Lamsweerde *et al.*, “Requirements engineering: from system goals to UML models to software specifications,” 2009.

- [163] F. Ferland, D. Létourneau, A. Aumont, J. Frémy, M.-A. Legault, M. Lauria, and F. Michaud, “Natural interaction design of a humanoid robot,” *Journal of Human-Robot Interaction, Special Issue on HRI Perspectives and Projects from Around the Globe*, vol. 1, no. 2, pp. 14–29, 2012.
- [164] M. Matarić and F. Michaud, “Behavior-based systems,” in *Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008.
- [165] D. E. Broadbent, *Perception and Communication*. Pergamon Press, London, UK, 1958.
- [166] R. M. Young, “Production systems in cognitive psychology,” *International encyclopedia of the social and behavioral sciences*, pp. 12–143, 2001.
- [167] S. Leuchter, L. Nekrasova, and L. Urbas, “Software engineering for cognitive modeling: Visualization of production systems.”
- [168] T. Inagaki, “Adaptive automation: Sharing and trading of control,” *Handbook of cognitive task design*, vol. 8, pp. 147–169, 2003.
- [169] D. D. Salvucci and R. Gray, “A two-point visual control model of steering,” *Perception-London*, vol. 33, no. 10, 2004.
- [170] D. D. Salvucci, H. M. Mandalia, N. Kuge, and T. Yamamura, “Lane-change detection using a computational driver model,” *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 49, no. 3, pp. 532–542, 2007.
- [171] J. Warmer and A. Kleppe, *The object constraint language: getting your models ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003.
- [172] M. Harman, “The role of artificial intelligence in software engineering,” in *Proceedings of the First International Workshop on Realizing AI Synergies in Software Engineering*. IEEE Press, 2012, pp. 1–6.
- [173] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, “An approach for QoS-aware service composition based on genetic algorithms,” in *Proceedings of*

the 7th annual conference on Genetic and evolutionary computation. ACM, 2005, pp. 1069–1075.

- [174] I. Crnkovic, M. Larsson, and O. Preiss, “Concerning predictability in dependable component-based systems: Classification of quality attributes,” in *Architecting Dependable Systems III*. Springer, 2005, pp. 257–278.
- [175] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, “A classification framework for software component models,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011.
- [176] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jezequel, “An Eclipse modelling framework alternative to meet the models@runtime requirements,” in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 87–101.
- [177] *Acceleo*. [Online]. Available: <https://www.eclipse.org/acceleo/>
- [178] P. Barvik and Y. Lamo, “Model to Model Transformation Tool for the DPF Workbench,” 2013.

Arunkumar Ramaswamy

No. 01.31, 7 rue des Louvieres
MONTIGNY LE BRETONNEUX
78180 France

Email: arun@arunkumarr.co.in

Connect: fr.linkedin.com/in/arunkumarramaswamy

Objective	I am passionate about taking technology beyond closed walls of research laboratories to real world scenarios for solving complex problems.
Summary	I am a researcher at Renault Research Division. My research interests include system engineering, cyber-physical systems, mobile robots, robotic software architectures, autonomous driving, computer vision, and embedded systems.
Education	<p>PhD Candidate at the ENSTA ParisTech, University of Paris-Saclay. Topic: A Model-Driven Framework Development Methodology for Robotic Systems Fellowship by Vedecom Institute, France</p> <p>Dual Degree - European Master in Advanced Robotics – Erasmus Mundus Program Master of Engineering in Robotics and Control, Warsaw University of Technology, Poland, Rank 1 Master in Control, Robotics, Signals, and Images, Ecole Centrale de Nantes, France</p> <p>Bachelor Degree in Electronics and Communication, Amrita University, India</p>
Experience	<p>Sept 2016 - Present System Engineer & Software Architect, Autonomous Driving, Renault Research Division, France</p> <p>Jan. 2013 - Apr. 2013 Doctoral Course in Management School of International Management, Ecole des Ponts ParisTech, France <i>Courses:</i> Business Negotiations, Finance, Marketing, Operations/Quality and Project Management, Entrepreneurship, Networked and Virtual Organizations, Business strategy.</p> <p>Mar. 2012 - Aug. 2012 Research Intern - Renault S.A.S, France <i>Topic:</i> Construction of 3D Ground Truth for the evaluation of Driving Assistance System in Urban environment. <i>Algorithms:</i> 2.5D Projection, Ground Cancellation, Euclidean Clustering, Point Feature Histogram, Kalman Filter. <i>Libraries:</i> PCL, OpenCV, MRPT, Eigen. <i>Software Tools:</i> Visual Studio 2008, Matlab, QT, RT Maps. <i>Apparatus:</i> Velodyne HDL 64E Lidar, IMU, GPS, senger Vehicle.</p> <p>Sept. 2008 - Aug. 2010 Engineer - Honeywell Technology Solutions, India</p> <ul style="list-style-type: none">Worked on Boeing 747-8 Next Generation Flight Management System software. Performed Unit and Integration testing for multiple modules mainly in Airline Operations Control package in C++ that uses a new Software Product Line approach in development.Worked on Health Monitoring of Firefighters (Proof of Concept for Patenting) Designed and implemented a microcontroller-based embedded system to measure blood pressure and heart beat rate and transmits the results to a software application using a RF channel.
Tools	<p>Software: Linux, Matlab, Visual Studio 2008, Eclipse RCP, Eclipse Plugin Development, Xilinx ISE, Proteus, RTMaps, Mplab IDE. Languages/Frameworks: High Proficiency in ROS, OpenCV, PCL, MRPT, Player/Stage, Aria, VHDL, C, C++, Latex. Experienced with VC++, Java, .Net, DBMS Concepts, PL/SQL Programming, QT, Boost libraries.</p> <p>Hardware: PIC Family of Microcontrollers, Assembly language, Embedded C, I2C, RS232, SPI Communication Protocols, Digital design using VHDL, Driver Circuit Design for DC, Stepper and Servo Motors, Turtlebot, Pioneer, Seekur Jr, Sensor drivers for Velodyne LIDAR, IMU.</p>
Professional Activities	<p>Robot Design Competition Co-Chair at International Conference on Social Robotics (ICSR 2015), Paris, France</p> <p>Organized and Chaired the Workshop at Robotics: Science and Systems (RSS 2015), Rome, Italy Title: Abstraction and Synthesis of Correct-by-Construction Robotics Software: Reuniting Formal Methods with Model-Driven Software Engineering.</p> <p>Organized Workshop at IEEE Systems Man and Cybernetics Conference (SMC 2014), USA, Title: System Engineering Human-Centered Intelligent Vehicles.</p>

Publications

Book Chapters

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Formal Specification of Robotic Architectures for Experimental Robotics. In Fabio P Bonsignorio (ed.). *Metrics of sensory motor coordination and integration in robots and animals*. Springer Cognitive Systems Monographs series. (Accepted)

Refereed Journal Articles

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Solution space modeling for robotic systems. *Journal for Software Engineering Robotics (JOSER)* 5(1):89–96.

Refereed Conference Proceedings

Ramaswamy, A., Monsuez, B., & Tapus, A. An Extensible Model-Based Framework for Robotics Software Development. In *Robotic Computing (IRC), IEEE International Conference on* (pp. 73-76).

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Model-Driven Self-Adaptation of Robotics Software using Probabilistic Approach. In *European Conference on Mobile Robots (ECMR)*.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Architecture Modeling and Analysis Language for Designing Robotic Architectures. In *International Conference on Control, Automation, Robotics and Vision (ICARCV), Singapore..*

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. AI Dimensions in Software Development for Human-Robot Interaction Systems. In *AAAI Fall Symposium Series, AI for Human-Robot Interaction, Washington DC, USA*.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. SafeRobots: A Model Driven Framework for Developing Robotic Systems. In *International Conference on Intelligent Robots and Systems (IROS), Chicago, USA, 1517–1524*.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Model-Driven Software Development Approaches in Robotics Research. In *Proceedings of the International Conference on Software Engineering (ICSE), Hyderabad, India*.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. SafeRobots: A Model-Driven Approach for Designing Robotic Software Architectures. In *International Conference on Collaboration Technologies and Systems (CTS), Minneapolis, USA, 131–134*.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Modeling Non-Functional Properties for Human-Machine Systems. In *2014 AAAI Spring Symposium Series, Formal Verification and Modeling in Human-Machine Systems, Palo Alto, USA*.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Formal models for cognitive systems. In *16th International Conference on Advanced Robotics (ICAR), Montevideo, Uruguay*.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Component based decision architecture for reliable autonomous systems. In *International Conference on Collaboration Technologies and Systems (CTS), San Diego, USA, 605–610*.

Karthi Balasubramanian, Arunkumar Ramaswamy, Jinu Jayachandran, Vishnu Jayapal, Bibin A Chundatt and Joshua D Freeman. Object recognition and obstacle avoidance robot. In *Control and Decision Conference, 2009. CCDC'09. Chinese, 3002–3006*.

Refereed Conference Posters

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Addressing Multi-Domain Integration Challenge in Robotics Using Model-Based Approach. In *Proceedings of the 2014 Models Conference, Valencia, Spain*. October 2014.

Arunkumar Ramaswamy, Bruno Monsuez and Adriana Tapus. Addressing Multi-Domain Integration Challenge in Robotics Using Model-Based Approach. In *Proceedings of the 2014 Models Conference, Valencia, Spain*. October 2014.

Technical Talks (excluding conference paper talks)

Title: Model-Based System Engineering for autonomous cyber-physical systems,

Venue: International Conference on Software & Systems Engineering and their Applications, May 2015, Paris

Technical Reports

Arunkumar Ramaswamy. Generation of Ground Truth using 3D Laser Scanner for Urban Environments. Master Thesis, European Master in Advanced Robotics (EMARO), Ecole Centrale de Nantes, France, Warsaw University of Technology, Poland, September 2012.

Arunkumar Ramaswamy. Artificial Intelligent Robot (AIBOT). Undergraduate Thesis, Department of Electronics and Communication Engineering, Amrita University, India, May 2008.

Titre : Une méthodologie de développement de structure logicielle orientée modèle pour les systèmes robotiques

Mots clés : Robotique, ingénierie logicielle pilotée par les modèles, architecture de logiciel, cadre d'architecture, architecture robotique

Résumé : La plupart des applications robotiques, telles que les véhicules autonomes, sont développées à partir d'une page blanche avec quelques rares réutilisations de conceptions ou de codes issus d'anciens projets équivalents. Qui plus est, les systèmes robotiques deviennent de plus en plus critiques, dans la mesure où ils sont déployés dans des environnements peu structurés, et centrés sur l'humain. Ces systèmes à fort contenu logiciel qui utilisent des composants distribués et hétérogènes interagissent dans un environnement dynamique, et incertain. Or, il s'agit là d'étapes indispensables pour la mise en place de méthodes d'évaluation extensibles, ainsi que pour permettre la réutilisation de composants logiciels pré-existants.

Le développement de structures logicielles et d'outils de conception d'architectures, orientés pour la robotique, coûte cher en termes de temps et d'effort, et l'absence d'une approche systématique pourrait conduire à la production de conceptions adhoc, peu flexibles et peu réutilisables. Faire de la meta-structure de l'architecture un point de convergence offre de nouvelles possibilités en termes d'interopérabilité, et de partage de la connaissance, au sein des communautés dédiées à la mise en place d'architectures et de structures. Nous suivons cette direction, en proposant un modèle commun, et en fournissant une approche méthodologique systématique aidant à spécifier les différents aspects du développement d'architectures logicielles, et leurs relations au sein d'une structure partagée.

Title : A model-driven framework development methodology for robotic systems

Keywords : Robotics, model-driven software engineering, software architecture, architecture framework

Abstract : Most innovative applications having robotic capabilities like self-driving cars are developed from scratch with little reuse of design or code artifacts from previous similar projects. As a result, work at times is duplicated adding time and economic costs. Absence of integrated tools is the real barrier that exists between early adopters of standardization efforts and early majority of research and industrial community. These software intensive systems are composed of distributed, heterogeneous software components interacting in a highly dynamic, uncertain environment. However, no significant systematic software development process is followed in robotics research.

The process of developing robotic software frameworks and tools for designing robotic architectures is expensive both in terms of time and effort, and absence of systematic approach may result in ad hoc designs that are not flexible and reusable. Making architecture meta-framework a point of conformance opens new possibilities for interoperability and knowledge sharing in the architecture and framework communities. We tried to make a step in this direction by proposing a common model and by providing a systematic methodological approach that helps in specifying different aspects of software architecture development and their interplay in a framework.