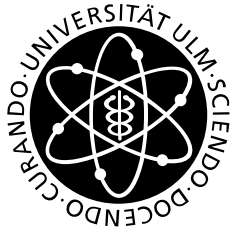
Kooperatives Promotionskolleg "Cognitive Computing in Socio-Technical Systems"
der Universität Ulm und der Technischen Hochschule Ulm



universität ulm

Variability Management in a Robotics Software Ecosystem Based on Dependency Variability Graphs for Modeling, Composing and Resolving Multi-Dimensional Variability Spaces

Dissertation zur Erlangung des Doktorgrades Dr.rer.nat. der Fakultät für Ingenieurwissenschaften, Informatik und Psychologie der Universität Ulm

Timo Blender

aus Illertissen

2022

Amtierende Dekanin: Prof. Dr. Anke Huckauf

Gutachter: Prof. Dr. Christian Schlegel
Gutachter: Prof. Dr. Karsten Berns (Technische Universität Kaiserslautern)

Tag der Promotion: 01.02.2023

Abstract

Service robots are complex, software-intensive systems that must handle their tasks in a robust and reliable way in dynamic environments. Due to these complexities, structures and tools must exist that support the development of service robot systems for service robot applications sensibly. The resulting reduced effort of the development shall justify the deployment of robot technologies for many users from an economical perspective. This is the goal of a robotics software ecosystem.

A robotics software ecosystem relies on a building block principle. That means, service robot systems can be composed out of reusable software building blocks. Likewise, the principle of separation of roles is of central importance so that the development process can be divided to different persons with different tasks according to their respective expertise.

Different building blocks and systems respectively have different properties (in particular non-functional properties) and various applications have various requirements related to these properties. The matching of these two forms of variability taking into account the variability of an environment is the task of variability management in a robotics software ecosystem. By this, the development of service robotic systems is not just supported somehow but requirement-related.

This thesis provides a general method for the variability management in a robotics software ecosystem and includes the modeling of building blocks and their properties, the composition of the properties of composed systems in dependence of the available configuration variants and states of the environment, the specification of requirements as well as the determination of an adequate (composed) variant according to the current situation.

The application of the models that can be created based on the methodological blueprint by building block developers allow for an automatic configuration of customized robotic systems according to the requirements of the end users. In addition, the principles of composition and separation of roles supported by the general method ensure that the effort of creating such models for the building block developers is significantly reduced. The thesis thus contributes to the general goal of a robotics software ecosystem mentioned above.

Zusammenfassung

Serviceroboter sind komplexe, softwareintensive Systeme, die ihre Aufgaben robust und zuverlässig in dynamischen Umgebungen bewältigen müssen. Aufgrund dieser Komplexitäten müssen Strukturen und Werkzeuge existieren, die die Entwicklung von Servicerobotersystemen für Serviceroboteranwendungen sinnvoll unterstützen. Der daraus resultierende reduzierte Aufwand der Entwicklung soll den Einsatz von Robotertechnologie für viele Anwender aus ökonomischer Sicht rechtfertigen. Dieses Ziel hat ein Robotik Software Ökosystem.

Ein Robotik Software Ökosystem setzt auf ein Baukastenprinzip, das heißt, Servicerobotersysteme können aus wiederverwendbaren Softwarebausteinen zusammengesetzt werden. Ebenfalls ist das Prinzip der Rollentrennung von zentraler Bedeutung, so dass der Entwicklungsprozess auf verschiedene Personen mit unterschiedlichen, ihrer Expertise entsprechenden, Aufgaben verteilt ist.

Unterschiedliche Bausteine beziehungsweise Systeme besitzen unterschiedliche Eigenschaften (insbesondere nicht-funktionale Eigenschaften) und verschiedene Anwendungen haben verschiedene Anforderungen bezüglich dieser Eigenschaften. Das Abgleichen dieser beiden Formen von Variabilität unter Berücksichtigung der Variabilität einer Umgebung ist die Aufgabe von Variabilitätsmanagement in einem Robotik Software Ökosystem. Damit wird die Entwicklung von Servicerobotersystemen nicht nur irgendwie, sondern anforderungsbezogen unterstützt.

Diese Arbeit liefert eine allgemeine Methode für das Variabilitätsmanagement in einem Robotik Software Ökosystem und umfasst das Modellieren von Bausteinen und deren Eigenschaften, das Zusammensetzen der Eigenschaften für zusammengesetzte Systeme in Abhängigkeit der vorhandenen Konfigurationsvarianten und Zuständen der Umgebung, das Spezifizieren der Anforderungen sowie das Ermitteln einer angemessenen, der aktuellen Situation entsprechenden (zusammengesetzten) Variante.

Die Modelle, die auf Basis der methodischen Blaupause durch Bausteinentwickler erstellt werden können, ermöglichen eine automatische Konfiguration von maßgeschneiderten Robotiksystemen entsprechend der Anforderungen von Endnutzern. Darüberhinaus sorgen die durch die allgemeine Methode unterstützten Prinzipien der Komposition und Rollentrennung dafür, dass der Aufwand der Entwicklung solcher Modelle für die Bausteinentwickler maßgeblich reduziert wird. Damit trägt die Arbeit zu dem oben genannten allgemeinen Ziel eines Robotik Software Ökosystems bei.

Acknowledgements

First of all, I would like to thank Prof. Dr. Christian Schlegel for supervising this work and for his targeted, useful advice during the last years. I thank Prof. Dr. Karsten Berns at Technical University of Kaiserslautern for agreeing to be the second reviewer of this work. Thank you to all former and current employees of the Service Robotics Research Center for the cooperation and support. I would like to thank Matthias Lutz in particular who has been a great help with all sorts of questions for many years. I also say thank you to my family and friends who are now finally freed from asking whether I am finally done. A very special thanks to my parents for their continuous support in many sorts of day-to-day matters, allowing me to focus on this work.

Contents

I Introduction and Related Work **9**

1 Introduction **11**
1.1 Research Question . . . 14
1.2 Contributions . . . 15
1.2.1 Method . . . 15
1.2.2 Integration/Implementation . . . 15
1.2.3 Application . . . 15
1.2.4 Summary . . . 18

2 Related Work **21**
2.1 Methodological Basics . . . 21
2.1.1 Component-Based Software Engineering in Robotics . . . 21
2.1.2 Model-Driven Engineering/Model-Driven Software Development . . . 22
2.1.3 Software Product Lines . . . 22
2.1.4 Decision Theory . . . 23
2.1.5 Graph Theory . . . 23
2.1.6 Constraint Satisfaction Problems . . . 24
2.1.7 Hierarchical Planning . . . 24
2.1.8 Control Architectures in Robotics . . . 24
2.1.9 Robotics Software Ecosystem . . . 25
2.2 Approaches for Variability Management in Different Domains . . . 26
2.2.1 General Software-Intensive Domains . . . 26
2.2.2 Robotics Domain . . . 28

II Method and Integration/Implementation **31**

3 Towards Variability Management in a Robotics Software Ecosystem **33**
3.1 Building Block Models . . . 33
3.2 Vision of a Holistic Variability Management Workflow . . . 34
3.2.1 Providers (Variability Enrichment) . . . 34
3.2.2 Users (Variability Reduction) . . . 34
3.3 Variability Management Based on Building Block Models . . . 36
3.3.1 Example-driven Problem Analysis . . . 37
3.3.2 Overview and Outline of the Methodology . . . 39

4 Dependency Variability Graph Basics **45**
4.1 A Special Hypergraph as Foundation . . . 45
4.2 From a Hypergraph to a Block-Port-Connector Representation . . . 45
4.3 Blocks and Ports . . . 47
4.3.1 Block Internal Variability Management . . . 47
4.3.2 Additional Port Characteristics . . . 51
4.3.3 Basic Block-Port Patterns: Production (PROD) and Aggregation (AGGR) . . . 53
4.4 Basic System Aspects . . . 57
4.4.1 Connection Constraints . . . 57
4.4.2 Split and Merge Situations and the Extended Production Pattern . . . 58
4.4.3 System Resolution . . . 58

5 Dependency Variability Graph Block-Port Modeling Patterns **65**
5.1 Relative Production (RPRO) . . . 65
5.1.1 RPRO with Preference Output . . . 65
5.2 Comparison Filter (COMF) . . . 67
5.3 Absolute Production (APRO) . . . 67
5.3.1 Precondition . . . 69
5.3.2 Core . . . 69
5.3.3 Final Output . . . 70
5.3.4 APRO with Constraint Output . . . 72
5.4 Multiple Aggregation (MAGR) . . . 72
5.5 Transformation (TRAN) . . . 73
5.6 Contradiction (CONT) . . . 75
5.7 Pareto Transformation Contradiction Composite (PTCC) . . . 75

6 Dependency Variability Graph Systems **81**
6.1 Composition Relationships . . . 81
6.1.1 Comparison Filter (COMF) . . . 81
6.1.2 Relative Production (RPRO) . . . 82
6.1.3 Absolute Production (APRO) . . . 82
6.1.4 Multiple Aggregation (MAGR) . . . 83
6.1.5 Transformation (TRAN) . . . 83
6.1.6 Contradiction (CONT) . . . 83
6.1.7 Pareto Transformation Contradiction Composite (PTCC) . . . 84
6.2 Further System Aspects . . . 84
6.2.1 Complete and Partial DVGs . . . 84
6.2.2 Source Spaces . . . 84
6.2.3 Resolution Views . . . 84
6.2.4 Determining and Interpreting the VSS . . . 85
6.2.5 Resolution Trade-Off Strategies . . . 87
6.3 Comparison With Other Structures . . . 89
6.3.1 Modeling Value Trees . . . 90
6.3.2 Modeling Decision Trees . . . 92

7 Integration/Implementation **95**
7.1 Data Object Definition and VSP Initialization . . . 95
7.2 Building Block . . . 97
7.2.1 VariabilityEntitySet . . . 97
7.2.2 Port . . . 100
7.2.3 Decomposition . . . 100
7.3 DVG . . . 102
7.3.1 Initialization (INIT) . . . 105
7.3.2 Extended Production (EPROD) . . . 106
7.3.3 Comparison Filter (COMF) . . . 106
7.3.4 Relative Production (RPRO) . . . 107
7.3.5 Absolute Production (APRO) . . . 107
7.3.6 Simple Absolute Production (SAPRO) . . . 107
7.3.7 Multiple Aggregation (MAGR) . . . 107
7.3.8 Transformation (TRAN) . . . 108
7.3.9 Contradiction (CONT) . . . 108
7.3.10 Pareto Transformation Contradiction Composite (PTCC) . . . 108
7.3.11 Equivalence Filter (EQUF) . . . 109
7.4 Variability Entities of Multiple Building Block Instances . . . 109
7.5 Towards Standardization and Automated Matching . . . 109

III Applications and Conclusion **111**

8 Building Block Use Cases **113**
8.1 Component: CDL . . . 115
8.1.1 The Building Block Model . . . 117
8.1.2 The DVG Model . . . 117
8.2 Skill: Navigation . . . 123
8.2.1 The Building Block Model . . . 123
8.2.2 The DVG Model . . . 124
8.3 Task: Order Picking . . . 133
8.3.1 The Building Block Model . . . 133
8.3.2 The DVG Model . . . 138
8.4 Collaborative Task: Shared Order Picking . . . 141
8.4.1 The Building Block Model . . . 141
8.4.2 The DVG Model . . . 144

9 Conclusion **151**
9.1 Future Work . . . 152
9.1.1 Extended Production (EPROD) . . . 152
9.1.2 Complex Data Objects . . . 152
9.1.3 Meta Properties . . . 152
9.1.4 Improving Planning Capabilities . . . 153
9.1.5 Probability . . . 153
9.1.6 Causality Cycles . . . 153
9.1.7 Integration in SmartSoft . . . 153

9.1.8 Systematic Support of Multi-Robot Tasks 154
9.2 Final Remark 154

List of Figures

1.1 Idealized vision of a robotics software ecosystem from the perspective of a pure end user. . . . 12
1.2 Illustrative example for the general motivation of this work: Developing robotic applications beyond pure task achievement (possible nowadays in robotic systems engineering) by considering and balancing additional properties. . . . 13
1.3 Structure and contributions of this work. . . . 16
1.4 Overview of the contributions of this work in combination with the three tiers in a robotics software ecosystem. . . . 17
1.5 Variability management in a robotics software ecosystem by modeling, composing and resolving VPSs of building blocks. . . . 19

3.1 The three dimensions of variability in a robotics software ecosystem. . . . 33
3.2 Vision of a holistic variability management workflow in a robotics software ecosystem. 35
3.3 Schematic illustration of the explanatory example. . . . 37
3.4 Overview of the presented DVG method. . . . 40
3.5 Basics of the DVG method: From a hypergraph to a small set of basic patterns based on a block-port-connector representation. . . . 40
3.6 The set of block-port modeling patterns determined in this work. Templates to compose DVG systems. . . . 41
3.7 DVG systems based on composition of block-port modeling patterns. . . . 42
3.8 The MDE/MDSD implementation of building block modeling and associated DVG system modeling. . . . 43

4.1 Valid example of a special hypergraph structure serving as the foundation of a DVG. 46
4.2 Example for the transformation of a hypergraph to a block-port-connector representation. . . . 46
4.3 Basic internal structure of a block and its ports. . . . 47
4.4 The production operator calculates the cartesian product of the IVSPs of a block. . . . 48
4.5 The aggregation operator collects the IVSPs of a block. . . . 49
4.6 Applying the value composition operator to the resulting combinations of the production operator. . . . 50
4.7 The internal structure of a slot. In addition to the VI, each slot maintains a header that captures its dependencies. . . . 51
4.8 Graphical notation for relevant combination of PCs, VTs and RTs. . . . 52
4.9 Possible transitions for the VT of a standard port between its modeling state, pre-resolution state and post-resolution state. . . . 53

4.10 Schematic illustration of the (a) PROD and (b) AGGR pattern. 54
4.11 Example of a PROD block subject to a SAMS. 59
4.12 Slot-specific view of the produced combinations for the concrete example in figure 4.11. The blue colored combinations are invalid. 60
4.13 Schematic illustration of the extended PROD pattern. 61

5.1 The RPRO pattern (a) with a SP as output and (b) with a PP as output. 67
5.2 The COMF pattern. 68
5.3 The APRO pattern (a) with a SP as output and (b) with a CP as output. 70
5.4 The internal structure of the APRO pattern. 71
5.5 The MAGR pattern. 73
5.6 The TRAN pattern. 74
5.7 The CONT pattern. 77
5.8 The PTCC pattern. 77

6.1 The EQUF pattern. 89
6.2 The meta model of a value tree. 91
6.3 An abstract example model of a value tree based on the meta model in figure 6.2. . . 92
6.4 DVG system representation for the value tree example model in figure 6.3. 93
6.5 Own illustration of a decision tree example from [Cle96]. 93
6.6 The decision tree example from figure 6.5 modeled as a special partial DVG system (a composition of MAGR patterns). 94

7.1 Overview of the MDSD/MDE approach of this work for modeling and resolving building blocks. 96
7.2 The *DataObjectDef* meta model. 97
7.3 The *VSPInit* meta model. 98
7.4 The *BuildingBlockDescription* meta model. 99
7.5 A schematic illustration about the overall number of variants related to the processing of *Input*s depending on different *InputRelationship*s. 101
7.6 *Pattern* part of the *DVG* meta model. 103
7.7 *Port* part of the *DVG* meta model. 104
7.8 *ValueCOp* part of the *DVG* meta model. 105

8.1 The composition relationships of the different building block examples described in this chapter. 114
8.2 The offline part of the CDL approach described as a DVG. 116
8.3 The building block model of the CDL component. 118
8.4 The DVG system model of the `CDL` building block from figure 8.3. 120
8.5 The building block model of the `Navigation` example. 125
8.6 (a) The building block model of the `MobileBase`. (b) The associated DVG system model. 127
8.7 (a) The building block model of the `Laser`. (b) The associated DVG system model. 128
8.8 (a) The building block model of the `Camera`. (b) The associated DVG system model. 129
8.9 The DVG system model of the `Navigation` building block 134
8.10 Example for a context and requirement specification of a potential end user of the `Navigation` building block. 135
8.11 The building block model of the `OrderPicking` example. 136

8.12 The DVG model of the `OrderPicking` building block 139
8.13 General representation of the workflow based on two task nets (Transportation and Manipulation) executed on two different service robots (Robotino and Larry) 142
8.14 The service robots available in this example. Larry (left) is able to handle the Manipulation task, while Robotino is able to handle the Transportation task 143
8.15 The building block model of `SharedOrderPicking`. 145
8.16 The DVG model of `SharedOrderPicking`. 147
8.17 Robotino approaching `PickingPlace`. 148
8.18 Execution of `PickAndPlace` by Larry. 148
8.19 Robotino executing `UnloadToStation`. 149

List of Tables

4.1 Meaning of the notation used in the different algorithm descriptions. 54

6.1 Composition Relationships between the modeling patterns considering their SPs: Is pattern A composable with pattern B? 82

8.1 Meaning of the *VariabilityEntiti*es modeled in the `CDL` building block in sense of an absolute representation of the associated *DataObjectDef*. 119
8.2 Meaning of the *VariabilityEntiti*es modeled in the `Navigation` building block in sense of an absolute representation of the associated *DataObjectDef*. 126
8.3 Meaning of the *VariabilityEntiti*es modeled in the `OrderPicking` building block in sense of an absolute representation of the associated *DataObjectDef*. 137
8.4 Abbreviations used in the equations and their meaning. 138

Dependency Variability Graphs Glossary

AGGR The **Aggr**egation basic pattern. 1, 51–53, 68, 70, 71, 87, 90, 127

APRO The **A**bsolute **Pro**duction modeling pattern. 2, 65, 67–69, 80–82, 105, 117, 119, 127, 129, 130

BIVM **B**lock-**I**nternal **V**ariability **M**anagement. 39, 47, 49, 51–53, 55, 59–61, 64–67, 71, 73, 74, 76, 83, 85, 86, 88

COMF The **Com**parison **F**ilter modeling pattern. 2, 65–67, 79–84, 104, 120, 121, 129, 130

CONSS **Con**figuration **S**ource **S**pace. 82–84, 90, 120, 129, 130, 137, 138, 149

CONT The **Cont**radiction modeling pattern. 2, 72–75, 80–83, 88, 106, 120, 129, 130

CP **C**onstraint **P**ort. 2, 49, 50, 65, 67, 68, 72, 79, 100, 103

CPD **C**ommon **P**ort **D**ependency. 55, 72

DO **D**ata **O**bject. 3, 47, 51, 53, 65, 70, 72, 79, 87, 93, 136

DVG **D**ependency **V**ariability **G**raph. 1–3, 15, 39–41, 43, 45, 46, 60, 61, 63, 79, 82–88, 90, 91, 93, 100, 103, 113, 117, 118, 120–122, 124–131, 133, 136–138, 141, 144, 147–150

EPROD The **E**xtended **Prod**uction basic pattern. 55, 59, 63, 65, 72, 73, 104, 130, 148

EQUF The **Equ**ivalence **F**ilter modeling pattern. 2, 87, 88, 107

IVSP **I**nput **V**ariability **Sp**ace. 47, 48, 55, 63, 65, 67, 72, 73

MAGR The **M**ultiple **Aggr**egation modeling pattern. 2, 70, 71, 80–84, 87, 90, 92, 105, 119, 127, 129

OVSP **O**utput **V**ariability **Sp**ace. 47, 48, 55, 60, 63, 65, 67, 72, 73, 75, 83, 87, 117

PC **P**ort **C**lass. 1, 49, 50

PP **P**reference **P**ort. 2, 49, 50, 63, 65, 72, 74–76, 100, 103, 106

PROD The **Prod**uction basic pattern. 1, 2, 51, 52, 55, 56, 58, 63, 65, 72, 82, 85–87

PTCC The **P**areto **T**ransformation **C**ontradiction **C**omposite modeling pattern. 2, 73, 75, 76, 80–83, 106

RPRO The **R**elative **Pro**duction modeling pattern. 2, 63–65, 67, 68, 80–82, 105, 127, 130

RT **R**epresentation **T**ype. 1, 49–51, 53, 63, 65, 72, 73, 79–82, 87, 103, 148

SAMS **S**plit **a**nd **M**erge **S**ituation. 1, 2, 55, 56, 58, 63, 65, 72, 79, 80, 148

SP **S**tandard **P**ort. 2, 5, 49, 50, 65, 68, 79, 80, 100, 103

SR **S**ystem **R**esolution. 60, 61, 83, 85

STASS **Sta**te **S**ource **S**pace. 82–84, 149

TRAN The **Tran**sformation modeling pattern. 2, 72, 73, 80–82, 88, 106, 127, 129, 130

VI **V**alue **I**nstance. 1, 3, 47–51, 55, 77, 83, 86, 87, 99, 103, 105, 106, 113, 136, 137

VPS **V**ariability **P**roblem **S**pace. 1, 15, 18, 19, 39, 41, 50, 60, 83, 88, 111, 147

VSP **V**ariability **Sp**ace. 39, 47, 48, 50–53, 55, 59, 64–66, 68, 70–74, 76, 83–86, 88, 93, 99, 100, 104–108, 113, 117, 133, 137, 147–149

VSS **V**ariability **S**olution **S**pace. 15, 39, 41, 50, 83, 111

VT **V**ariability **T**ype. 1, 49–51, 53, 63, 65, 67, 68, 70, 72, 79–83, 86, 87, 90, 105

Part I

Introduction and Related Work

Chapter 1

Introduction

In recent years, the field of service robotics cleared the first hurdles to evolve from a basic research topic to become one of the practical key contributors to the fourth industrial revolution. Service robotic applications are no longer just pure laboratory prototypes but started to be part of many commercial applications in different domains with convincing perspectives regarding economy and sustainability [BBF+16]. A service robot is characterized by its ability to perform its tasks autonomously and robustly even in complex and open-ended environments. This requirement makes the engineering of a service robotic system to a great challenge from a software point of view in general and will persist as long as every system must be developed from scratch over and over again. This development effort is especially high and problematic for complex applications and it is one reason why commercial applications are unfortunately typically still limited to single, rather simple tasks (vacuum cleaner robots, lawn mower robots). However, multipurpose service robots able to fulfill a high diversity of tasks robustly and flexibly and at the same time regarding the requirements of a user are not yet ready for the market.

The idea of a *robotics software ecosystem* is to reduce the software effort of developing robotic applications/systems significantly by relying on separation of roles and on composition: We put together such systems out of existing robotic functions offered as reusable software building blocks in a marketplace. Under the assumption that the whole diversity of available robotic technologies, algorithms and basic skills are accessible as building blocks in a robotics software ecosystem, the development process of service robotic applications can be reduced to the appropriate composition of these building blocks according to the respective requirements. Hence, the idealized vision of a robotics software ecosystem from the perspective of a pure end user can be described as illustrated in figure 1.1. The end user can treat the ecosystem as a black box and just specifies its application needs to the ecosystem. Based on that specification, the ecosystem should then provide a ready to use, customized system.

Building blocks must be described meaningfully in order to understand which one is of use with respect to fulfilling a user's requirements. This means there must exist a corresponding model that explicates all the relevant properties of a building block. Besides pure functional aspects, *non-functional properties (NFPs)* are often important features of requirements and are thus also decisive for the description of building blocks. However, such building block descriptions are often not trivial because building blocks typically exhibit different forms of internal variability. This variability must be explicated accordingly and put in relation to the resulting properties and their possible conflicts/trade-offs in order to be able to compare it sensibly with the requirements. The management of these relationships is also the key for the appropriate composition of these properties. Properties of different

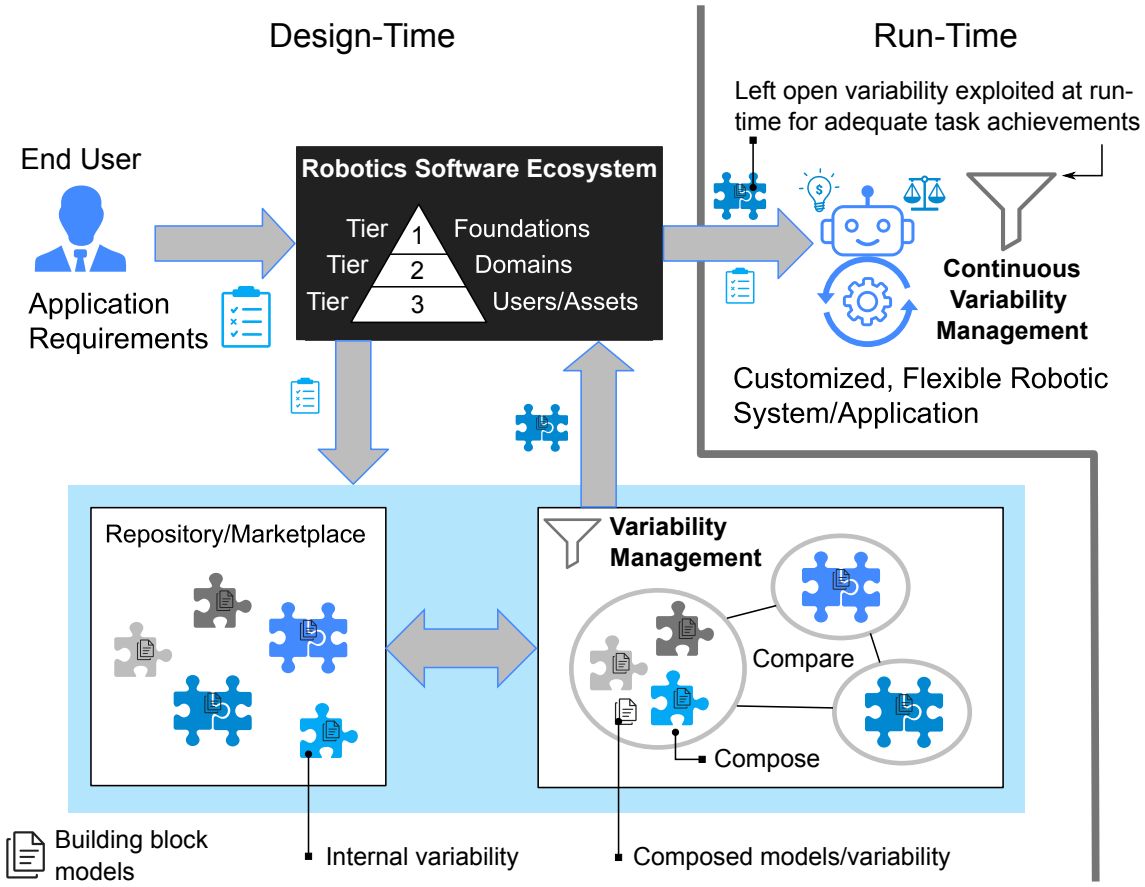


Figure 1.1: Idealized vision of a robotics software ecosystem from the perspective of a pure end user. More information about the different tiers in a robotics software ecosystem can be found in [SLLS21].

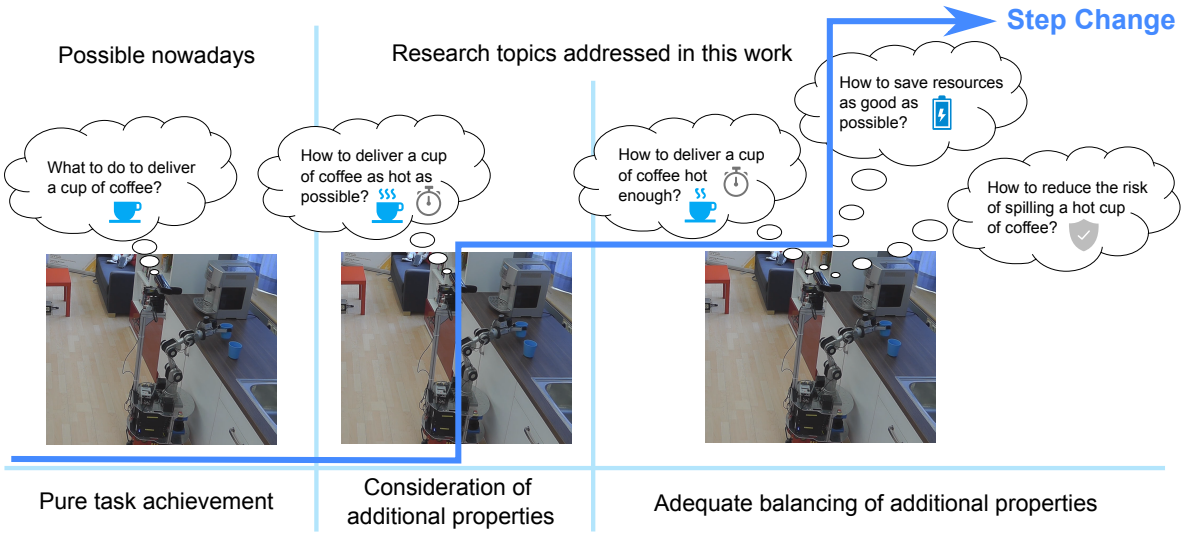


Figure 1.2: Illustrative example for the general motivation of this work: Developing robotic applications beyond pure task achievement (possible nowadays in robotic systems engineering) by considering and balancing additional properties.

building blocks must be composed if their building blocks are combined to realize a new higher order building block representing a more powerful function. In that case we also need a model for the properties of that new, composed building block and the new properties of this new model will be influenced by additional relationships between the properties of its constituent parts and their respective internal relationships. The same principle applies for conflicts of that new building block for which we need to determine aggregated utility values to characterize all the possible variants of a composed building block by taking into account the desired preferences of a user's requirement to that building block. The final outcome would be then the provision of an optimal or at least adequate variant of a building block for the respective user. If we now assume that every building block available in the marketplace of a robotics software ecosystem as well as every possible and meaningful composition of these building blocks are described and interlinked consistently in such a way, we close the circle to the vision sketched in figure 1.1: The marketplace becomes a huge building block and all the building blocks inside can be considered as potential variants to fulfill a user's requirement.

It is important to note that requirements to a service robotic application are not always static at design-time (flexibility is often expected from service robots, requirements vary from task to task, in particular for multi-purpose service robots). Furthermore, the fulfillment of requirements by certain building blocks and composed building blocks respectively are often not determinable at design-time because properties of involved building blocks can exhibit complex dependencies on external variability occurring in dynamic environments. For that reason, it is important that robotic systems still possess sufficient variability at run-time from which then variants can be selected in order to react adequately to upcoming situations. For that, it must be known in principle at which point in time which variant has which properties and which of them is the most useful one taking into account the current state of the environment and the current requirements.

A central concept in this work is *adequacy*. We use this concept for 3 different aspects of a model:

1) **Adequacy of the application (of a model)**
Service robots are able to balance external user constraints and their own internal needs in task ex-

ecution by exploiting models: Properties which are not constrained by external user requirements are left open to let the service robot to decide upon. For a user it is fully satisfying if the required properties are fulfilled good enough (i.e. stay within the given bounds). This enables a service robot at the same time to consider its internal needs, for example not spending more resources than necessary.

2) **Adequacy of the creation (of a model)**
The quality of a created model (completeness and accuracy) correlates with the invested effort: The more precise the building blocks are analyzed, the better models can be created for these building blocks by using the tools based on the methodological blueprint of this work.

3) **Adequacy of the solution (of a model)**
Every solution which considers the external user constraints as stated in 1) as part of applying a model and specifying requirements to it is basically considered as satisfying regardless of whether this solution is close or far away from a global optimum. With that, the complexity of finding a solution can be reduced because there is no general need to explore the whole (potentially huge) solution space.

We use the term *Variability Management* throughout this work to capture the essential aspects of the topics to be addressed. Furthermore our definition aims to create a common, appropriately abstracted baseline with respect to related work and all the different kinds of software-intensive systems where the management of variability plays a major role. For these two purposes, variability management is defined as follows:

A method for the explication of variability of a system, for determining properties of its variants and for the (automated) calculation of custom variants for specific requirements, integrated with and implemented within the boundary conditions imposed by a specific domain.

Figure 1.2 describes based on an illustrative example what is typically possible these days in system-/software-engineering of service robotics and what are still research topics. Addressing these research topics would represent a significant step change in the field of service robotics engineering and would make the sketched idealized vision of a robotics software ecosystem a bit more tangible.

1.1 Research Question

Based on the introduced vision of a robotics software ecosystem and our definition of variability management in software-intensive systems, the resulting research question of this work is:

How can we realize variability management in a robotics software ecosystem?

In combination with our definition of variability management, this research question can be broken down as follows:

1) **Method:** What is a suitable method for the explication of variability of robotic functions/building blocks, for determining (composed) properties of its variants and for the (automated) calculation of custom (composed) variants for specific requirements?

2) **Integration/Implementation:** How to realize such a method in combination with the boundary conditions imposed by established principles of a robotics software ecosystem?

3) **Application:** Illustrating and evaluating the significant added value by concrete service robotic use cases based on the implemented methodology.

With respect to our general definition of variability management, robotic functions and building blocks respectively are treated as special systems in this work for which we want to realize variability management. Our consideration of building blocks is generic, i.e. not limited to a specific level of technical implementation such as functions (in the context of programming languages), components, skills and tasks (systems of components), or whole applications. In general, we assume the principle of hierarchical (de)composition of building blocks which can be theoretically repeated indefinitely where in the end a concrete robotic application can be represented as one abstract building block. In general system theory, this basic principle is known as *hierarchical order* [Ber69]. Other terms are *nested hierarchy* or *containment hierarchy* with *compositional containment hierarchy* as a special sub type [ZV17].

1.2 Contributions

This work contributes a method, an integration/implementation of the method and applications of the implemented method. Figure 1.3 shows the structure of this work and the contributions. Figure 1.4 provides another view on these contributions in combination with the 3 tiers in a robotics software ecosystem.

1.2.1 Method

The *Dependency Variability Graph* (DVG) method - a formal, graph-based approach for modeling, composing and resolving multi-dimensional *Variability Problem Space*s (VPSs) of real robotic systems (building blocks) in complex environments by a special combination of different methods and structures. It applies the principle of modularization to reduce the complexity both of (i) modeling variability problem spaces and (ii) the implementation of a consistent resolution procedure determining a *Variability Solution Space* (VSS).

1.2.2 Integration/Implementation

Realization of the DVG method in combination with the principles of a robotics software ecosystem (separation of roles, composition of building blocks and *Model-Driven Engineering* (*MDE*)/*Model-Driven Software Development* (*MDSD*)). The implementation using MDE/MDSD is a blueprint for modeling building blocks, i.e. it allows to model building blocks and also allows to model, compose and resolve the VPSs of these building blocks by the implemented DVG method. That outcome represents the realization of a blueprint for variability management in a robotics software ecosystem. The great resulting benefit is a substantial simplification of developing customized service robotic applications for specific requirements.

1.2.3 Application

We present and discuss concrete individual building block models realized by our implementation for selected service robotic use cases and illustrate the resulting added value of our method that allows a

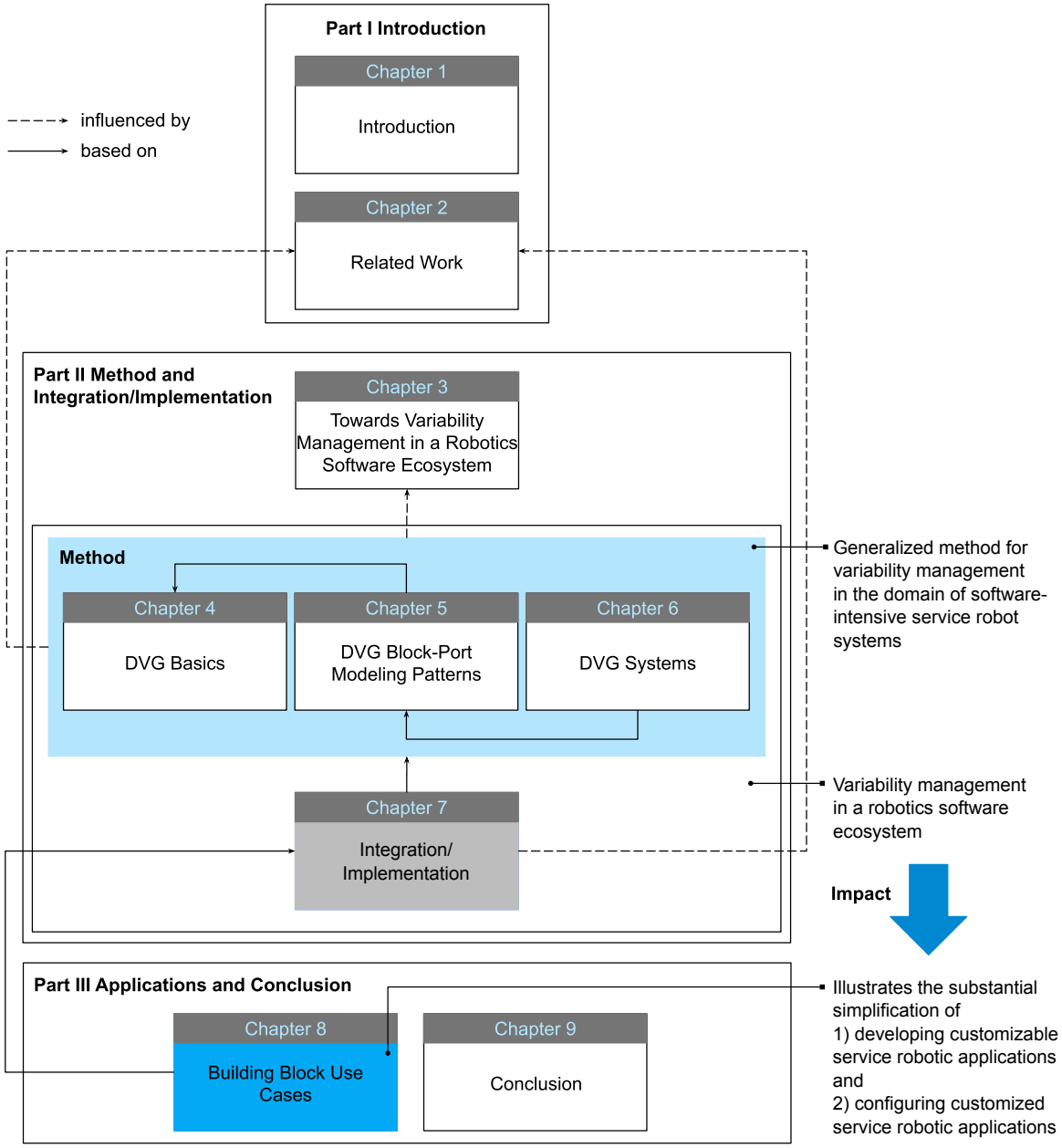


Figure 1.3: Structure and contributions of this work.

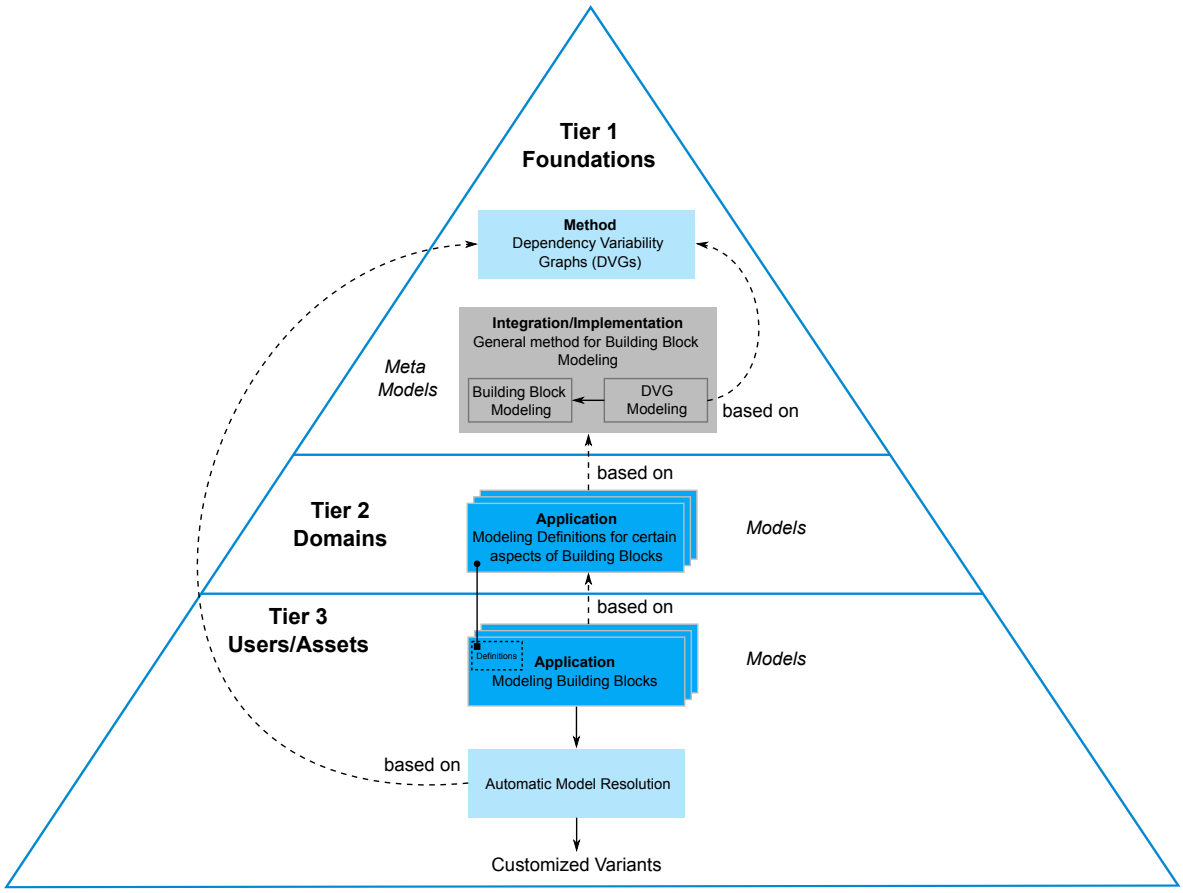


Figure 1.4: Overview of the contributions of this work in combination with the 3 tiers in a robotics software ecosystem. More information about the different tiers in a robotics software ecosystem can be found in [SLLS21]. Note that establishing domain definitions is also possible for individual building blocks at tier 3 in this work. In that case, the building blocks represent their own domain in some way. Although that is not entirely in the spirit of a robotics software ecosystem, this is important for right now to illustrate the proposed methodology in combination with our generic consideration of building blocks in a time where the robotics software ecosystem is still in an early stage. That means standardization efforts can not have taken place yet. Furthermore, it still needs to be clarified which aspects of building block models as presented in this work should be prescribed at tier 2 and which not. The answer to that question is not easy and answering it is not part of this work because that again depends on standardization efforts which can only be made by a community.

robot to determine automatically customized execution variants based on different requirement specifications. For example, we show how a service robot becomes capable of acting resource adequate at run-time by balancing external task goals and its internal needs. These applications illustrate the substantial simplification of realizing tailored robotic applications.

1.2.4 Summary

Shortly summarized, we realize variability management in a robotics software ecosystem by modeling, composing and resolving VPSs of building blocks to provide optimal or at least adequate variants for customized requirements and situations (figure 1.5).

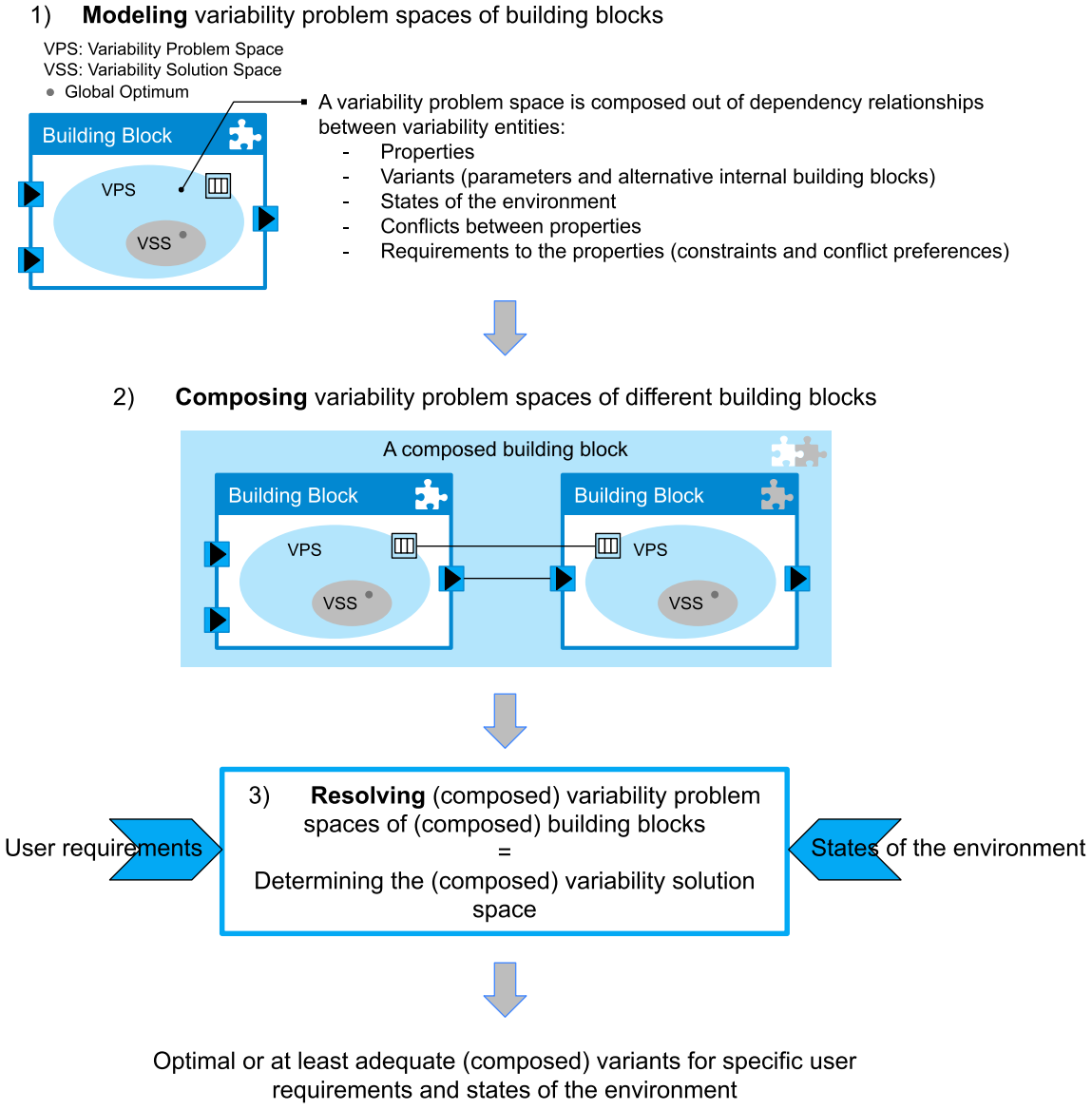


Figure 1.5: Variability management in a robotics software ecosystem by modeling, composing and resolving VPSs of building blocks.

Chapter 2

Related Work

Related work is divided into two parts: In the first part we present some basic principles and general methods that serve as foundations or boundary conditions for the method and context of this work and/or are relevant with respect to different concrete approaches addressing relevant aspects of variability management in different domains which are summarized in the second part. A boundary condition means here an established principle, method or approach that is an essential part of another, larger principle, method or approach. Its part in that whole has undoubted reasons justified by many advantages. Hence, the associated structures of that boundary condition must always be considered when the whole is further expanded to include new aspects.

2.1 Methodological Basics

In this section, we introduce basics for the method of this work and/or for other approaches discussed in the next section.

2.1.1 Component-Based Software Engineering in Robotics

Component-Based Software Engineering (CBSE) is a common principle for the modularization of software into independent units (components) interacting with one another via as few firmly defined interfaces as possible. If the interfaces are service-oriented [SLLS21], the resulting loose coupling enables the service-based composition of (software) components to (software) systems because changes within individual elements do not impact the other parts and thus guarantees the interchangeability of components.

The general *block-port-connector* [Wik21] principle is used to model component-based systems in robotics:

- A block represents the internal calculations of a block and has ports.
- Ports define the interface for the associated calculations of a block. There are input ports (specifying what is required from the outside to perform the calculations) and output ports (specifying what is provided to the outside based on the performed calculations).
- Connectors interlink input ports of blocks with output ports of other blocks.

CBSE has been established in the robotics domain and is an essential boundary condition of a robotics software ecosystem and therefore relevant for this work. CBSE has also been established

in many other domains in which software-/system-engineering is an important part. Hence, it is also a boundary condition for many other approaches that address variability management in different domains.

2.1.2 Model-Driven Engineering/Model-Driven Software Development

Model-Driven Engineering (MDE) or *Model-Driven Software Development (MDSD)* relies on models as the central abstraction element to develop software systems. Models can exist at different abstraction levels and a model of a specific abstraction level must always conform to the structures of the models of the respective superordinate levels. A typical convention is to add a *meta* prefix to the *model* for each abstraction level, so that a *model* conforms to a *meta model* and a *meta model* in turn conforms to a *meta meta model* and so on. The ultimate goal of MDE/MDSD is to generate source code (more complexity) from the created models of less complexity in order to simplify the process of developing software systems. More and varied information about MDE/MDSD can be found in the literature, for example [SVC06].

MDE/MDSD is now widely used in various domains where software- and system-engineering are a decisive part. In the area of embedded systems, matured approaches relying on MDE/MDSD are for example *Architecture Analysis & Design Language (AADL)* [FGH06] and *Modeling and Analysis of Real Time and Embedded Systems (MARTE)* [MAR19]. The *System Modeling Language (SysML)* [Sys19] addresses systems in general. Because of this general focus, it misses semantic concretizations that must be defined first before the approach can be applied. In contrast, *SmartMDSD* [SLLS16] is a more concretized approach defined for separation of roles and composition which is suitable for the robotics domain [SLLS21]. Another example in the robotics domain is *RobotML* [DKS+12]. Many approaches presented in the next section rely on MDE/MDSD as technological basis to implement variability management in their software-intensive domain. The Robotics 2020 Multi-Annual Roadmap (MAR) states: “Model-driven software development (MDSD) and DSL (domain specific languages) are core technologies required in order to achieve a separation of roles in the robotics domain while also improving compose-ability, system integration and also addressing non-functional properties.” [euR15, 245]. MDE/MDSD is an essential technology for realizing a robotics software ecosystem.

2.1.3 Software Product Lines

Software Product Lines (SPL) [BC05] in its original form are the results of a systematic approach for variability management during a software development process that clearly separates generic, reusable core parts of a software product from individual feature variants based on explicit variation points. The goal is to reduce the development costs through reuse while deriving customer-specific software products by appropriate binding of the variation points. A popular form of representing SPLs are feature models [KCH+90]. A basic feature model contains a discrete set of features organized in a hierarchy and the childs are constrained by a set of specific relations with respect to its parent feature. Extended variants of feature models allow the definition of further constraints to increase the expressiveness of this modeling approach [Bat05]. Based on the modeled relationships and constraints it is then possible to derive automatically suitable product variants. *Dynamic Software Product Lines (DSPL)* extend the SPL principle to run-time variant adaptation in order to perform better in complex and changing environments [HHPS08]. The SPL/DSPL approach with special forms and extensions of feature models are often used in different domains to realize variability management. The typical relationships used in feature models to express valid combinations of features are applied in this work

to the relationships between different input ports of building blocks. This expresses the variability of processing different valid combinations of these inputs that influence the properties of the building block and thus also the system properties.

2.1.4 Decision Theory

Decision theory deals with the mathematical systematic deduction of rational decisions in decision problems. This includes inter alia: Systematic structuring of decision problems and a quantification of the available alternatives regarding its properties as well as different mathematical methods for preference specifications and its calculation/definition of usefulness that are applied to modeled decision problems in order to determine appropriate variants. Thus, decision theory is a relevant basis for variability management in general because they share many commonalities. Furthermore, mathematical rational decisions are often desired in technical systems that have to deal with complex decision problems.

Value Trees

Saaty [Saa77] proposes hierarchical structures for representing concrete decision problems as part of his *Analytic Hierarchy Process (AHP)* [Saa90]. According to that, an overall decision making goal can be decomposed into different levels of criteria, sub-criteria, attributes and alternatives. The works in [MH99, MH00] and [GSG01] use the term *Value Tree* for this kind of hierarchical problem organization and they present tools for modeling value trees and to perform a value tree analysis. The tool provided in [MH99, MH00] also allows to apply different methods for *Multi-Criteria Decision Making (MCDM)* in a value tree to specify preferences for criteria and to calculate suitable decisions based on that. A comparison of different MCDM methods can be found in [ZSWD98, TSSR98, VH13]. Value tree analysis is often applied in political, economical or environmental issues (e.g. see [BKP15, ZSS18, THv18]) to support the decision making process. The hierarchical structures of value trees are an aspect that is captured and supported by the general DVG method presented in this work.

Decision Trees

A decision tree represents a certain structuring of a decision problem where discrete variant branches of decisions and chance events can be arranged in a tree-like structure. More information about decision trees can be found in [Cle96]. The structure of nested branches and the differentiation between decisions and events are aspects that are captured and supported by the general DVG method presented in this work.

2.1.5 Graph Theory

Graph theory deals with the mathematical analysis of structure and properties of graphs. Graphs consist of nodes connected by edges and are important, formal models for many practical applications in computer science. A special form of a graph is a hypergraph, whose edges can connect more than 2 nodes in contrast to classical graphs. Hierarchical hypergraphs are suitable for modeling different kinds of systems in robotics [SHB+16]. They combine hypergraphs with the concept of hierarchical order.

Hierarchical hypergraphs are the scientific grounding of a robotics software ecosystem [SLLS21]. Since we address variability management in a robotics software ecosystem, the fundamental structure of the approach presented in this work is a special form of a hierarchical hypergraph.

2.1.6 Constraint Satisfaction Problems

Many problems in computer science can be treated as *Constraint Satisfaction Problems (CSPs)* [Kum92]. Given a set of variables with a finite domain and a set of constraints defined on those variables, solving a CSP means to determine value assignments for each variable so that all constraints are fulfilled [Kum92]. An overview of algorithms and applications of CSPs is given in [BPS99]. An extended form of a CSP is a *Constraint Satisfaction Optimization Problem (CSOP)* that additionally includes an optimization function that determines a numerical value for each valid solution of a standard CSP ([Tsa95] as cited in [Bar99]). Some approaches for variability management are treated as a CSP or a CSOP. The DVG method of this work does the same: We apply the principle of domains of variables when explicating variability and we model dependency relationships between the variables. Finally, it is sometimes relevant to consider certain constraint requirements during the resolution of a DVG. In general, constraints like for example value range specifications are a natural way for an end user to express its requirements. Essentially, the resolution of a DVG can be considered as a CSP or a special CSOP depending on the specified model (CSOP if one or several conflict of objectives are defined).

2.1.7 Hierarchical Planning

Hierarchical planning applies the principle of hierarchical task decomposition to define a special planning problem whose goal is to determine a sequence of primitive tasks (actions) based on given compound tasks to be executed [BAH19]. The imposed hierarchy can be treated as additional guidance to solve the planning problem which improves speed and scalability in contrast to classical planning [GA15]. There exists a wide variety of formalisms for hierarchical planning (e.g. *HTN*) with different extensions [BAH19]. Special task nets combined with a special control architecture (see next section) are an essential part of a robotics software ecosystem to model and coordinate run-time behavior of robotic systems. This combination can be considered as a special form of hierarchical planning and is therefore relevant for this work.

2.1.8 Control Architectures in Robotics

A control architecture in robotics organizes a specific structure to manage the execution of robot actions in a systematic way. They evolved from the *sense-plan-act* paradigm (too inflexible in dynamic environments), over behavior-based approaches (hierarchy and dependency problem of behaviors, e.g. the *subsumption architecture* [Bro86]) to hybrid architectures [Sch04]. Hybrid architectures combine the advantage of goal-directed planning and reactive, behavior-based modules (skills) by introducing an additional sequencing layer between these two parts [Sch04]. Therefore, they are often denoted as three-layer architectures. The sequencer is responsible to manage the individual skills with respect to the current context, i.e. activating/deactivating, setting parameters and reacting to failures [Sch04] but it also allows to encode knowledge about the execution order of these skills in a hierarchical fashion by constructing task nets (e.g. *RAP* [Fir89]) in advance of the execution. This concept has a lot of similarities with hierarchical planning and its decomposition principle. In this context, the sequencing layer with its encoded knowledge can be interpreted as a special form of guidance to

solve a compound task in the context of robotic control architectures that always resolves to one or several (parallel) sequences of skill interactions. For high-level goals or situations that are not known in advance, the topmost planning layer is used. It depends on the concrete implementation of a three-layered architecture whether the planner instructs the sequencer (e.g. *3T* [PBJFG+97]) or vice versa (e.g. *ATLANTIS* [Gat91] or *SmartSoft* [SW99, Sch04]). More information about control architectures in robotics can be found in [Pir97]. A special *3T* control architecture is integral part of a robotics software ecosystem.

2.1.9 Robotics Software Ecosystem

A robotics software ecosystem is an interaction of different principles and methods to guide the development process of robotic systems and manage their execution. Its main principles are separation of roles and composition of commodity building blocks. Important methods for realizing a robotics software ecosystem are CBSE, MDE/MDSD and a special *3T* control architecture. Stampfer [Sta18] proposes 3 layers in a robotics software ecosystem for 3 types of involved roles (top-down): (i) Tier 1: Ecosystem Drivers, (ii) Tier 2: Domain Experts and (iii) Tier 3: Ecosystem Users. Each tier must conform to its next higher level. The *RobMoSys* project [rob] aims to establish an *EU Digital Industrial Platform for Robotics* based on this idea of a robotics software ecosystem. The expected perspectives of such an approach are both of technical (e.g. ease of use, re-usability, traceable/predictable properties, certified systems) and commercial nature (e.g. shorter development time, cost reduction, shorter time to market, mass customization)[1]. Selected contributions to tier 1 of the RobMoSys ecosystem are:

- The concept of *service definitions* [Sta18] to ensure composability between the ports of different software components by combining a *communication pattern* [Sch04] with a *communication object* [Sch04] to define a common agreement about an interaction and the option to define additional *properties*.
- A composition workflow based on CBSE, *Service-Oriented Architecture (SOA)* and *service definitions*
- The principle of *cause-effect chains* [Lot18] for modeling component task chains and the integration with a performance analyzer to conduct a *Compositional Performance Analysis (CPA)* at the system level
- Structures for interfacing between behavior-related skills and tasks and software component internal coordination aspects[2] [3] [4]

SmartSoft

The *SmartSoft* toolchain [SLLS16] is a model-driven framework to support the development of service robotic systems and conforms to the ecosystem structures defined in the *RobMoSys* project. It is based on the *SmartSoft* framework [SW99, Sch04] that is a component based software approach for

[1] https://robmosys.eu/user-stories/

[2] https://robmosys.eu/wiki/composition:task:start

[3] https://robmosys.eu/wiki/general_principles:architectural_patterns:robotic_behavior

[4] https://robmosys.eu/wiki/general_principles:architectural_patterns:component-coordination?s[]=coordination&s[]=service

sensorimotor systems which comes with a set of clearly defined *communication patterns* to define the interaction semantic between different components and that also supports *dynamic wiring* of the data flow between component ports at run-time. The underlying control architecture of *SmartSoft* is three-layered and the sequencer implemented by *SmartTCL* [Ste10, SS11] represents the central element, i.e. a planner is called by the sequencer only in specific application-related situations. According to the contributions to Tier 1 within *RobMoSys* previously mentioned, the model-driven tooling of *SmartSoft* supports, among other things:

- Modeling of *service definitions* (Tier 2) including *communication objects* and generation of corresponding source code
- Modeling of tasks, skills and *coordination service definitions* (Tier 2)
- Modeling of components (Tier 3) and generation of corresponding component source code
- Modeling of systems (Tier 3) including deployments and generation of corresponding deployment source code
- Modeling of behavior (Tier 3) and generation of sequencer-component interactions based on the used skills
- Modeling of task chains (Tier 3) and generation of corresponding source code to perform a compositional performance analysis

The idea of a robotics software ecosystem and its structures are elementary boundary conditions for the method of this work.

2.2 Approaches for Variability Management in Different Domains

2.2.1 General Software-Intensive Domains

This section is about concrete variability management approaches in software-intensive domains that are not robotics but that also have a strong demand of self-management/self-adaptation capabilities to operate in complex environments. They use (combinations of) methods presented in section 2.1 and apply it for their domain considering specific boundary conditions. Hence, we summarize all the following topics and works as variability management in software-intensive systems according to our above definition. A more detailed classification and distinction is not part of this work.

Overview

Autonomic computing [KC03] represents the idea of self-managing systems with the primary goal to reduce the administration effort of human operators. The architectural blueprint in [ibm05] proposes the idea of autonomic managers that continuously perform the steps *Monitor*, *Analyze*, *Plan*, and *Execute* and that contain *Knowledge (MAPE-K)*. The term Self-Adaptive Software refers to systems that are able to change their behavior according to changes in their environment [OGT+99]. A comprehensive survey on approaches in the field of self-adaptive software and a derived taxonomy about this domain is given in [KBR+18]. DSPLs are a popular approach to realize variability management in the domain of autonomic computing and self-adaptive software. As surveys like [GSSC15, BBD17] show, DSPLs have established as an independent collective term representing a vast of different approaches

and its specific boundary conditions to realize variability management for different application areas. Another comprehensive survey and a further resulting taxonomy is given in [ABG+13] about software architecture optimization methods. The purpose of many approaches presented in there is similar to the vision of autonomic computing and self-adaptive software.

An Excerpt of Approaches

Classical SPLs and feature models do not take into account non-functional properties of individual features. Hence, they are not able to determine non-functional properties of product variants and to take them into account as decision criteria. The work in [BTRC05] provides extensions for adding attributes to features and an automated reasoning process based on transformations to a *Constraint Satisfaction Problem (CSP)*. A holistic approach for the consideration of non-functional properties in product derivation of SPLs is provided by the works of Siegmund et.al. [SRK+11a, SRK+11b, SKK+12] including specification, measurement and optimization of non-functional properties with different roles involved. They rely on a special technique to approximate non-functional properties of features based on measuring and comparing a small set of product variants in order to cope with combinatorial explosion of large SPLs. In [SKK+12] they present heuristics to determine feature interactions and their influence to performance in order to improve the prediction accuracy. Feature modeling is also very popular in DSPL. Some approaches [ACL+11, Sal15, PKWB17] apply feature modeling not only for the system but also for the application context. When then defining constraints between both feature models (system and application context) it is possible to derive a suitable context-aware system variant by applying different solution techniques. Sánchez et. al. [SDZ+14, SDPZ+15] describes features with quality attributes and associates the features with software components. They apply a heuristic search to derive appropriate configurations based on the requirements and then transform it to a set of component operations (add/remove/replace/tune). [BGC+08] presents a model-driven approach for developing adaptive component-based systems by associating context states represented in a transition diagram with specific component configurations. [MBJ+09] proposes a general architecture for run-time adaptation based on a set of meta models consisting of DSPL, Context, Reasoning and Architecture.

The works in [WTKD04, KD07] promote the general use of utility functions to achieve self-management for autonomic systems. In [FS09], a DSL is presented that allows to specify adaptation logic of a system. They combine a rule-based and a utility-based approach to profit from the advantages of both approaches. Constraints can be defined between the variants and between context. Variants can be assigned qualitative properties. Rules can be used to define the priority of a property in a certain context situation. A constraint solver is then used to determine valid configurations. Finally, the valid configurations are rated by priorities and impact values of the variants. The *Rainbow* [GSC09] approach is a general methodology for architecture-based self-adaptation of software systems with a strong focus on run-time monitoring. For a specific architecture model, constraints can be defined for important system properties at run-time that are continuously monitored. Utility functions are then used to evaluate different adaptation strategies and the one with the highest utility is executed after the constraint for a monitored property is violated. In [Che08], they introduce some aspects in more detail. They illustrate how to consider uncertainty in the adaptation process and therefore how to calculate expected utility. Furthermore they provide a mathematical formalization of the overall adaptation approach.

The *MADAM* approach [FHS+06, HSSF06, GBE+09] represents an adaptive middleware for mobile applications as well as a model-driven approach for the development of adaptive applications. The middleware monitors user- and system-context and tries to adapt a component-based architec-

ture model in such a way that quality requirements of a user are met. Component implementations are described by non-functional properties. It is allowed to describe context dependencies for certain properties so that they can vary at run-time. A utility function determines the most suitable application variant under consideration of user requirements and preferences as well as resource constraints of the environment. Different application variants can be realized by exchanging component implementations (compositional adaptation) or by parameter adjustments within a component (parameter adaptation). The *MUSIC* approach [HGP+12] extends the concepts of MADAM for a more general application in ubiquitous computing environments.

[ARN12] provides an approach for the composition of webservices considering non-functional concerns. The functional behavior of the desired application is modeled as a workflow of abstract services. For each abstract service there are multiple realizations with different non-functional properties available. The goal is the suitable selection of alternatives for each service in such a way that constraints relating to the whole workflow are fulfilled while maximizing a utility function including user preferences. To consider such global constraints, the non-functional properties of the individual building blocks are composed by specific aggregation functions. The solution method is applied in a distributed way to improve the efficiency. Ghezzi et. al. [GPST13] propose a framework that allows to model software systems using activity diagrams to represent workflows of abstract functionalities. Different implementations with different non-functional properties can be specified at design-time for these abstract functionalities. Furthermore, branches of the forks in the activity diagram can be assigned probability values. The activity diagram is then transformed into a Markov Decision Process (MDP). A selection process adapts the behavior based on the available implementation variants in such a way that specific constraints regarding the properties can be fulfilled taking into account the cumulative reward for each state in the MDP to the final state, a weighting of the requirements and the current execution context.

Leserf et.al. [LdHC15, LdH15, LdSSH19] provide an approach for the optimization of system architectures based on *SysML* [Sys19]. They provide several extensions allowing to model variability, objective functions regarding non-functional properties and constraints. Concrete models are then transformed to constraint satisfaction problems and solved by specific solvers.

A certain number of research works address in particular the challenge of determining NFPs of composed systems in CBSE (composition of NFPs). Crnkovic [CLP05] presents a classification of quality attributes and their composition relationships. Different works [BM04, MMW05, BdCM05, Koz08] address the problem by special modeling and prediction approaches with the focus on performance properties. Staehli and Eliassen [SE04] propose a meta model for QoS semantics of component services. Sentilles [Sen12] provides a holistic approach for the systematic specification and management of non-functional properties for component-based embedded systems. The composition problem of NFPs in CBSE can be seen as an own problem domain. We identified the principle of variability dependencies as one major cause for the NFP composition problem/challenge. The provided DVG method of this work allows to model and resolve variability dependencies and is part of our approach for variability management. Hence, we can state that our variability management approach also addresses the NFP composition problem.

2.2.2 Robotics Domain

Software Product Lines (SPL) for component-based robotic systems are presented in [Ghe13]. The model-based approach allows the definition of domain-specific solutions with possible variability in form of feature diagrams. The different product variants are associated with corresponding architectural configurations at the component level. Different meta models and transformations enable to

generate executable deployments (architectural configurations) for various software frameworks depending on the selected combination of features. The work in [GSB+19] gives a general overview about variability management in robotics. They provide 4 types of variability: Customer Requirements, Environment, Robot Hardware and Middleware and compare ad-hoc mechanisms and systematic SPL approaches from research to address them. The authors conclude that ad-hoc mechanisms are still common practice in industry projects and that dynamic variability should be more addressed in the future.

Sykes et. al. [SHMK08] addresses self-management of autonomous systems in a three-layered architecture [PBJFG+97] by automatic determination of component compositions for actions planned by the goal management layer. In [HSMK09, SHMK10] the method is extended by considering annotated non-functional properties of the components.

Ramaswamy et.al. [RMT14b] identified the need for applying modeling techniques to develop robotic systems. They present a meta model for *solution spaces* that allows to model the execution flow of domain-specific functionalities and also a meta model for non-functional properties in order to evaluate and compare different variants in the solution model. In [RMT14a] they describe a first implementation of their approach based on the Eclipse Modeling Framework. In [RMT15] they show how to map their solution space models to Markov Decision Processes in order to achieve self-adaptation at run-time. Finally, in [RMT20] they list four topics as further research challenges of their work: (i) Non-functional Property Composition, (ii) Runtime Models, (iii) Managing Uncertainty and (iv) Safety Assurance.

[GH15] provides a model-based approach based on feature models with assignable attributes to define and resolve variability of robotic systems at run-time using a rule-based adaptation logic. The works in [ILVS12, LIRVCS13, SLL+15] present a domain-specific language to model run-time adaptation logic for service robots that allows to model variation points, context, adaptation rules and properties. The language is mapped to the constraint modeling language *MiniZinc* ([NSB+07, SFS+14]) for resolution purposes. The determined results are then executed by the task coordination language *SmartTCL* [Ste10, SS11]. In [IRVC13] they combine their language with a formal tool in order to verify and simulate the modeled adaptation logic.

Brugali et. al. [BCMT18] presents an approach for the model-based development of reconfigurable robotic systems in compliance with non-functional requirements. The presented use case addresses safety in navigation and the goal is to verify at design-time which system configurations are able to fulfill specific requirements of an application. The approach uses feature models with annotated properties to represent the variants of a system. These information are then used to determine the overall response time for different system configurations by applying a performance analysis using Queuing Networks [Kle75]. These results are used to check the safety requirements at deployment-time. In [Bru18], they provide a special UML MARTE profile [MAR19] to explicate safety aspects of mobile navigation. In a use case, they illustrate how a constraint solver can be applied to determine the robot velocity in dependence of the safety requirements and further relevant properties.

The work in [VIM+18] contributes meta models to the *RobMoSys* project [rob] for the specification and estimation of QoS metrics as a basis for robot behavior adaptation.

Pirjanian [Pir98] applies the value tree method for the action selection problem in behavior-based robotic systems.

Mansouri presents in [MP16, Man16] the *meta-CSP* approach for hybrid reasoning in robotics which is able to consider multiple concerns of tasks at the same time (task dependency, time, space, resource usage). Stock [SMPH15, Sto16] presents a hierarchical, hybrid planning approach for mobile robot systems that considers both causal, resource and temporal aspects by combining a hierarchical planner with the *meta-CSP* approach [MP16, Man16].

The contribution in [Lot18] provides extensions of the *SmartSoft* [SW99, Sch04] component meta model and a model-based mapping to a performance analyser for predicting performance properties of different execution paths in a composed system at design time. The concept of *cause-effect chains* used there is also relevant for this work but in a more general sense for all kinds of properties.

Part II

Method and Integration/Implementation

Chapter 3

Towards Variability Management in a Robotics Software Ecosystem

In a well advanced ecosystem that offers a large number of reusable building blocks, developing a robotic application for a user/customer can be reduced to the simple process of putting together (composing) existing building blocks in the ideal case. From such a high-level view, variability management in a robotics software ecosystem could be seen as the process of composing building blocks in such a way that they fulfill the application requirements of a user. This problem can exhibit an enormous complexity because of the high diversity in the application domains and their respective requirements as well as the robotic domains, the solutions (offered as building blocks) and their varying properties. In addition, this diversity is not fixed in an ecosystem but open-ended since new application areas may open up and the different robotic domains and their solutions can progress over time. After all, there is also the high diversity of different, dynamic states and conditions in all the complex environments where service robots are finally deployed. Variability management must manage these three dimensions of variability in a robotics software ecosystem (figure 3.1).

3.1 Building Block Models

We need to know what a building block is in order to allow for (i) comparing it with other building blocks (ii) checking how it fits with other building blocks and (iii) matching it with the requirements.

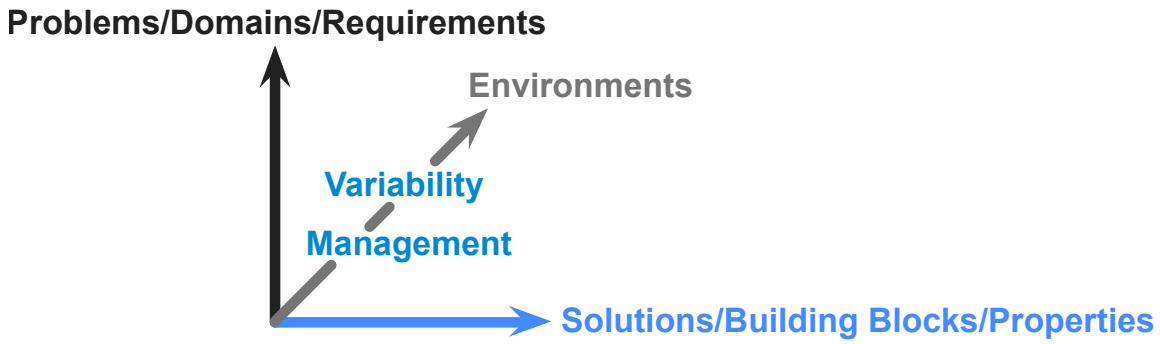


Figure 3.1: The three dimensions of variability in a robotics software ecosystem.

A building block model serves as an abstract representation of a building block and should fulfill the purpose of describing what the associated building block is. An important aspect of the description of a building block is its provided functionality (such as: *This building block is able to detect objects*, *this building block allows a robot to drive from A to B*). Another crucial information of a building block description are additional properties of the provided capability (such as: *The accuracy is average*, *the resources required are moderate*) because a user may not only demand for a specific function but also additional properties of it. Overall, the model of a building block should allow for consistent predictions and what-if analysis with respect to fulfilling requirements to properties.

3.2 Vision of a Holistic Variability Management Workflow

The decisive foundation of variability management in a robotics software ecosystem are the building block models. Figure 3.2 shows a vision of a holistic variability management workflow at tier 3 in a robotics software ecosystem based on the core element of building block models representing developed building blocks finally offered in a marketplace/repository.

In this overview, we distinguish basically between the role of providers and the role of users, i.e. those that develop building blocks and push them to a repository and those that pull suitable building blocks from there to satisfy their application demands. In an idealized variant of variability management in a robotics software ecosystem, these pulls are not done manually by each user but are part of an automated resolution process which processes a user's requirement specification to finally provide a customized system.

3.2.1 Providers (Variability Enrichment)

The providers go independently from others through their individual building block development process to model and implement their building blocks. A provider can be at the same time user of building blocks of other providers if the provider wants to reuse them as part of its provided building block. The building block models of these external building blocks are used to check if they fit to the building block under development and if so they become an integral part of its building block model. Hence, composition of building blocks goes along with composition of their building block models and this new composed building block model should still define consistently what this new composed building block is. Depending on the function granularity of a building block to be developed, the process can be divided in further internal roles. In this case there is also a handing-over and step-by-step extension of the building block model between the internal roles since every role must only focus on its specific responsibilities and competencies.

In general, the development process of building blocks in a robotics software ecosystem by different providers can be considered as a continuous process that is independent of concrete user demands. It is a continuous enrichment of solution variability that enables a robotics software ecosystem to fulfill the vast diversity of potential user requirements.

3.2.2 Users (Variability Reduction)

In the ideal case, a user only needs to specify its application requirements and an automated resolution process is able to provide a customized system satisfying its needs. This is what we classify as *inter-variability* in this work where the associated *inter-resolution* is able to manage variability over all building blocks available in the repository from independent providers. This is the most powerful and ambitious idea of variability management in a robotics software ecosystem because it means

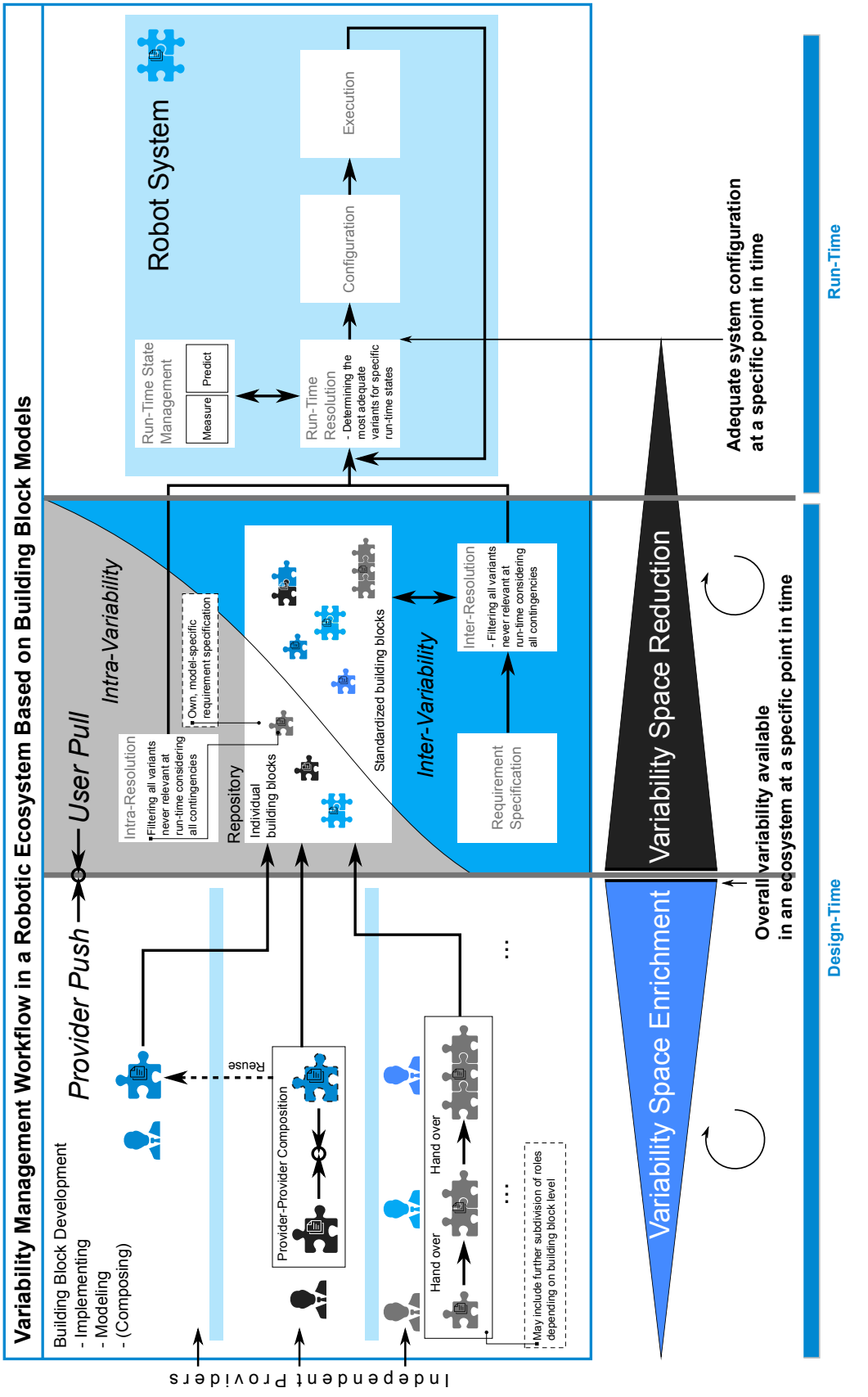


Figure 3.2: Vision of a holistic variability management workflow in a robotics software ecosystem.

that all the available building blocks represent potential variants to fulfill user requirements and are therefore in competition with each other. Realizing (automated) inter-resolution requires that all the building blocks provided by different providers are comparable with each other and with respect to an independent requirement specification. This means that their building block models must be described in a meaningful and commonly agreed way. Achieving that is only possible after elaborate standardization processes have taken place by a community that explicate and prescribe definitions for all the capabilities and properties in the different robotic domains.

In contrast, intra-variability with intra-resolution is the process of managing internal variability of a specific (manually) user selected building block from the repository. Intra-variability of a specific building block from the repository is always a subspace of inter-variability. It is a shortcut approach within the overall vision but depending on the function granularity of the selected building block and the offered variants (flexibility), this can still represent a complex problem. It is assumed that a user is aware of the provided capability of its selected building block and understands that it fits to its (functional) needs.

If we aim to offer the most flexible solution, the theoretical task of intra-/inter-resolution at design-time is to filter intra-/inter-variability that is with respect to the user requirements never relevant at run-time by considering all possible eventualities. The remaining variability is then part of the run-time system and can be reduced continuously in a loop, determining a final system configuration for every respective combination of run-time states occurring at a specific moment in time. If all eventualities were considered at design-time, the robot system would be able to react to any run-time situation in conformance with the user requirements. However, in open-ended worlds in which service robots typically act, such an anticipation is of course not always fully possible.

3.3 Variability Management Based on Building Block Models

An important part for realizing variability management in this work are the building block models which are the basis both for intra- and inter-variability/resolution in the context of our holistic vision. In general, our methodology is designed to support both types of variability. Nevertheless, the use cases we later present in this work are all based on intra-variability with intra-resolution because we do not perform sophisticated domain analysis in this work that comes up with adequate definitions prescribed for the different providers. This is something that needs to be done later by domain experts at tier 2 in a robotics software ecosystem by using the general concepts of this approach defined at tier 1. In the case of intra-variability (in contrast to inter-variability) we can fully ignore the aspect of function/capability matching (this requires a common vocabulary) because here a user selects a specific building block manually. This means that the user understands the function/capability of this building block (e.g. just by reading the name or a short description of the building block) and based on that the user is able to conclude that this building block fits to its function/capability requirements. Hence, the focus of intra-variability is always on additional properties of such manually selected building blocks.

The presented uses cases will also illustrate that describing real building blocks in a robotics software ecosystem can represent a complex modeling process where the process itself can represent a trade-off. Combining that general complexity with our generic consideration of a building block in this work, the challenge of establishing standards for inter-variability becomes even greater. As already argued in figure 1.4 it still needs to be clarified in the future which aspects of building block models as proposed in this work should be defined and prescribed at tier 2 and which not.

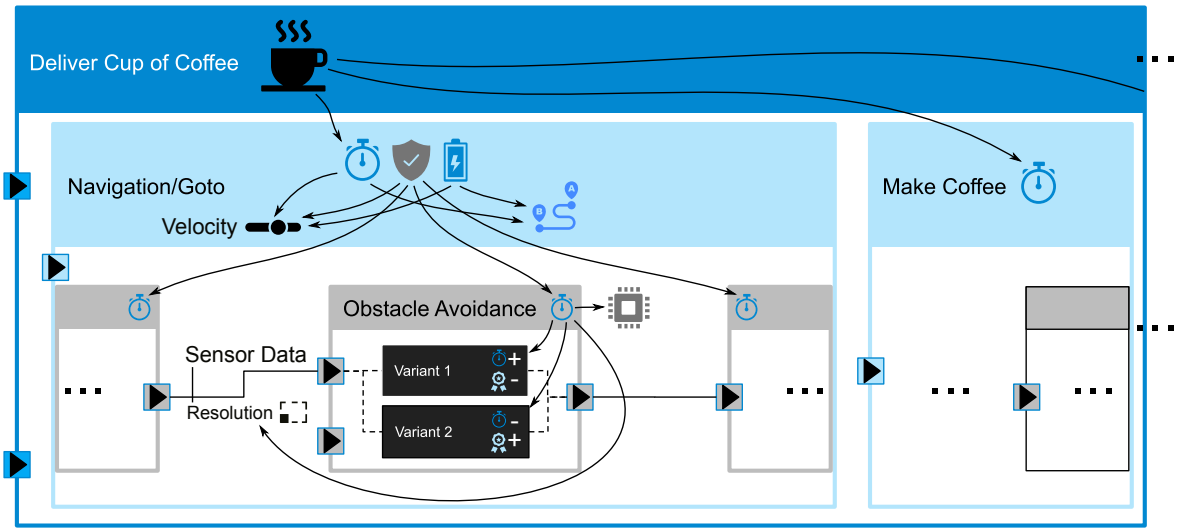


Figure 3.3: Schematic illustration of the explanatory example.

3.3.1 Example-driven Problem Analysis

In the following, we describe an example for intra-variability management of individual building blocks to illustrate what we want to achieve in this work and what are the major challenges. Figure 3.3 provides a visual impression of the example for a better understanding.

Component Level: Obstacle Avoidance

Let us assume that a person wants to implement an algorithm for local obstacle avoidance and offer it in the marketplace of a robotics software ecosystem as a software component so that others can reuse the implemented functionality in their own systems. To ensure port-composability of its component the person can rely on established service definitions to specify (i) which data the component must receive in which way via input ports so that the person's algorithm can be executed properly and also (ii) which processed data the algorithm/component will finally provide to the outside in which way (via an output port). These specifications already represent an important aspect of the components functional model.

Furthermore, the person would also like to specify more properties about the provided component, e.g. the execution time of the implemented algorithm. Now the question is, how should the person describe that in a meaningful way? In the easiest case, the person could think of measuring the execution time and simply annotating the component by the resulting numerical value. It raises the question: What is the use of this? Because on closer inspection, one recognizes that the execution time of the algorithm is never a constant value and can be only described in intervals, e.g. due to scheduling effects, amount of data to be processed (depends on the resolution of the received sensor data), the specific CPU of the user etc.

Furthermore, the implementation inside the component comprises two different algorithms where one of them takes less time but also has less quality and vice versa. Based on that, we can conclude that the description by a simple numerical value without any explication of the dependencies is completely

useless. A description that explicates the concrete states of the dependencies could be at least used to check under which conditions this description is consistent and could be used by all those who meet these state conditions. Finally, a description which is able to put the result in relation to all the different states of the dependencies would be generic applicable and thus ideal.

System Level 1: Navigation/Goto Skill

A system builder wants to reuse the obstacle avoidance component just described in a system realizing a navigation skill that enables a robot to drive from a location A to a location B. This building block also has certain properties such as the timing, energy consumption or safety. These properties also have a lot of dependencies and some of them are linked to properties of building blocks of which this building block is composed of. For example, the braking distance, a safety critical property, depends among other things on the execution times of the involved component tasks (e.g. the execution time of the previously described obstacle avoidance component). Hence, all dependencies impacting the execution time of the obstacle avoidance component will also impact the braking distance property. Similarly, such dependency relations may also exist for the execution times of the other involved components. Furthermore, the braking distance is in conflict with timing because both depend on the selected velocity of the robot. In addition, timing depends on the concrete location A and B. Hence, here also arises the question: How to describe this building block in a meaningful and consistent way such that this description represents added value?

System Level 2: Deliver Cup of Coffee Task

A task should be realized that enables a service robot to make coffee and deliver it to a person. A system builder wants to reuse the navigation skill just described in that system (and thus also the mentioned obstacle avoidance component) such that the robot is able to approach the location of the coffee machine by itself and finally to drive back to the waiting person to serve the coffee. A property of that task could be the fact whether the coffee will be delivered hot enough or not which depends on the interpretation of hot enough of the specific person and again on the timing properties of the skills of which this task is composed of (i.e. (i) approach coffee machine, (ii) make coffee and (iii) approach person).

Conclusion

A requirement of a user who wants to get served a cup of coffee could be: *Deliver a cup of coffee within x_1 minutes after it was ordered and serve it hot enough, i.e. within x_2 minutes after the cup of coffee has been made*. This requirement should be fulfilled by a service robot capable of performing the task deliver cup of coffee. However, the service robot executing the task is also aware about safety restrictions, e.g. it is not allowed to exceed a braking distance of x_3 meters while transporting a hot cup of coffee. Furthermore, the service robot may want to meet both goals while investing as few resources as possible.

If all our building blocks would be described in a meaningful and consistent way with respect to all the existing dependencies and how they relate for different states of these dependencies, a service robot executing the task deliver cup of coffee would be able to determine adequate bindings of its internal execution variants that meet the described requirements. This represents a concrete example for intra-variability management based on building block models.

3.3.2 Overview and Outline of the Methodology

From the introduced example, we can see that describing and resolving the relationships between properties, variants, states, conflicts and requirements of building blocks and also between such entities of different building blocks is the decisive element of realizing variability management. For this matter we present the DVG method.

A Dependency Variability Graph (DVG) system represents a discretized Variability Problem Space (VPS) of a building block describing the dependencies of discretized *Variability Space*s (VSPs) of variability entities (properties, variants, states, conflicts and requirements) by means of certain composition operators. From such a modeled VPS, the optimal binding of active variants can be determined for every possible combination of passive states and the respective requirements (i.e. preferences for conflicts as well as value range constraints for properties). Figure 3.4 illustrates the most important terms and their relationships. Note that for simplicity the different explicated types of variability entities are represented only one-dimensional in this figure although there can for sure exist several independent variability entities of the same type in practice. Therefore a VPS modeled by a DVG can represent an arbitrarily higher dimensional problem space.

The Variability Solution Space (VSS) is a subspace of the VPS that can be determined by resolving a DVG system. It contains all the valid solutions (i.e. combinations of active variants) fulfilling the specified value range constraints for selected properties for every possible combination of passive state dependencies. The VSS also contains the globally optimal combination of active variants for all the possible combination of passive state dependencies which is influenced by the desired preferences to the modeled conflicts between the involved properties.

The modeled dependencies between the variability entities can be considered as *"What Dependencies"* and can be divided into VPS and VSS dependencies. A VPS/VSS dependency is characterized by the fact that changes of the influencing variability entities (where the arrowhead points) and/or the associated composition operators (*"How Dependencies"*) lead to modifications of the VPS/VSS.

In the following we give a short overview about the following chapters with respect to the DVG method and the integration/implementation of the DVG method in a robotics software ecosystem as part of modeling building blocks and realizing variability management.

Method: DVG Basics

In chapter 4 we present the basics of the DVG method (figure 3.5). Starting point is a special form of a hypergraph which is then transformed into a corresponding block-port-connector representation that then forms the basis for everything else that follows. We describe the internal structure of a block and relevant aspects of the *Block-Internal Variability Management* (BIVM) as well as the different basic characteristics of ports. Furthermore, we introduce a small set of basic block patterns each containing a specific *Combinatorial Composition Operator*. A combinatorial composition operator determines how the individual, discrete elements of the different inputs of a block have to be combined to produce an output. Another form of composition operator is the *Value Composition Operator* which defines how the concrete values of a combination, determined by a combinatorial composition operator, have to be combined.

Method: DVG Block-Port Modeling Patterns

We introduce a set of concrete block-port modeling patterns in chapter 5 based on the described foundations. These modeling patterns (figure 3.6) are modeling templates that can be used to compose DVG systems. A DVG system is an important part of a building block model.

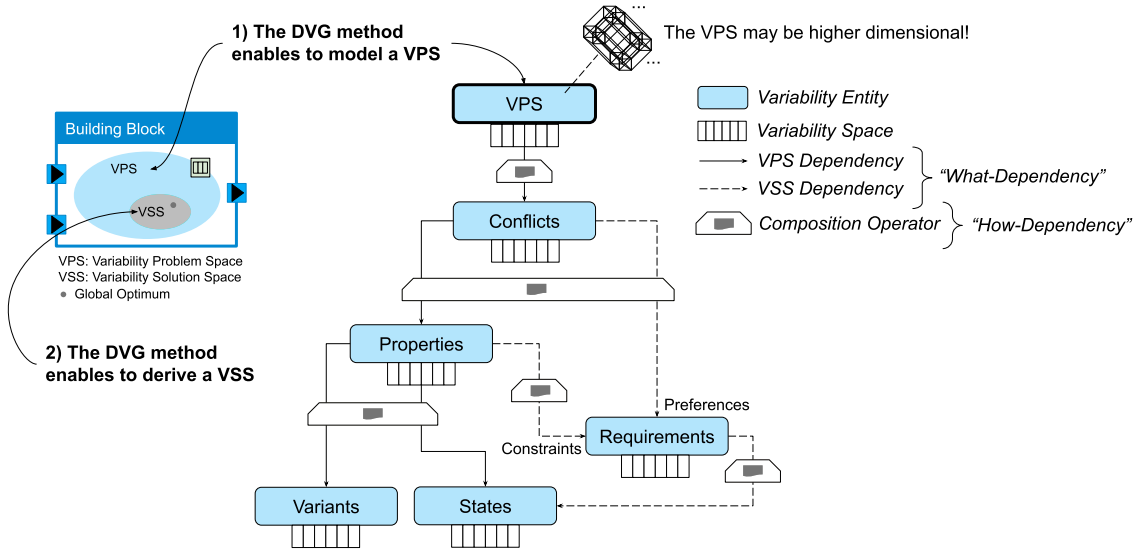


Figure 3.4: Overview of the presented DVG method.

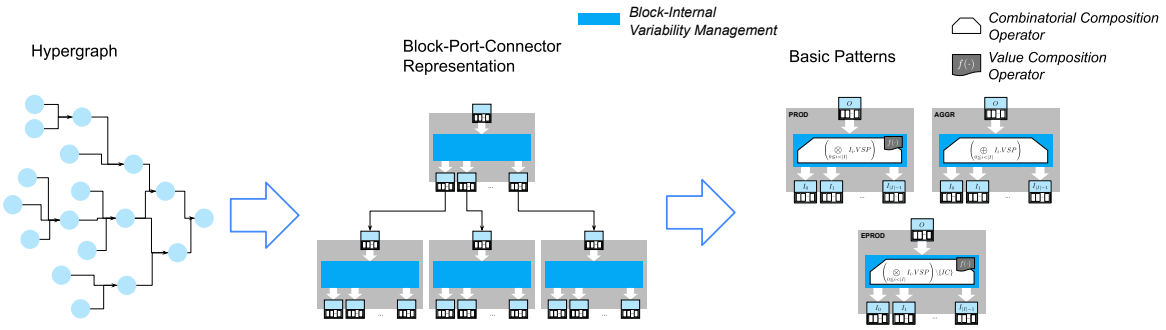


Figure 3.5: Basics of the DVG method: From a hypergraph to a small set of basic patterns based on a block-port-connector representation.

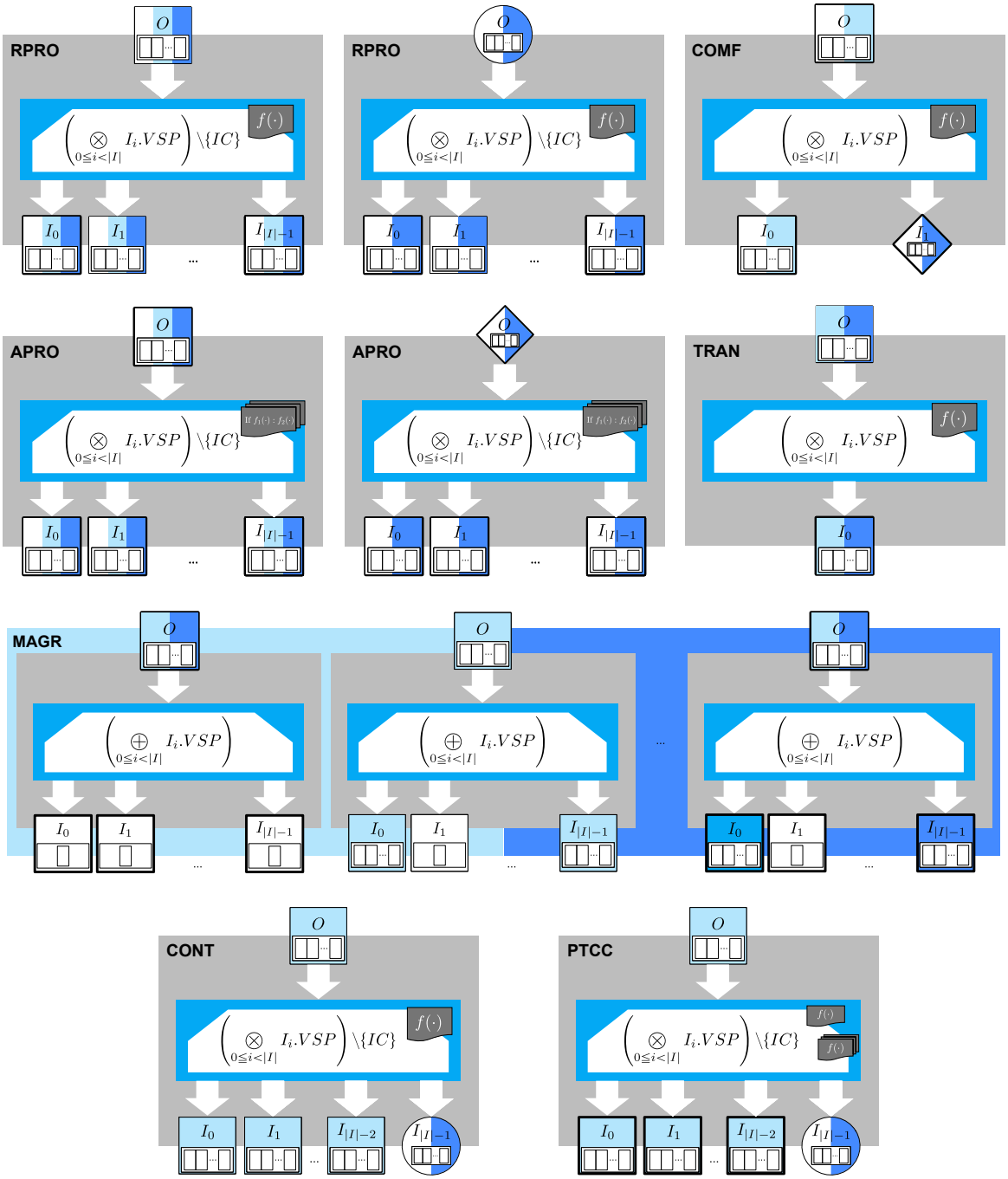


Figure 3.6: The set of block-port modeling patterns determined in this work. Templates to compose DVG systems.

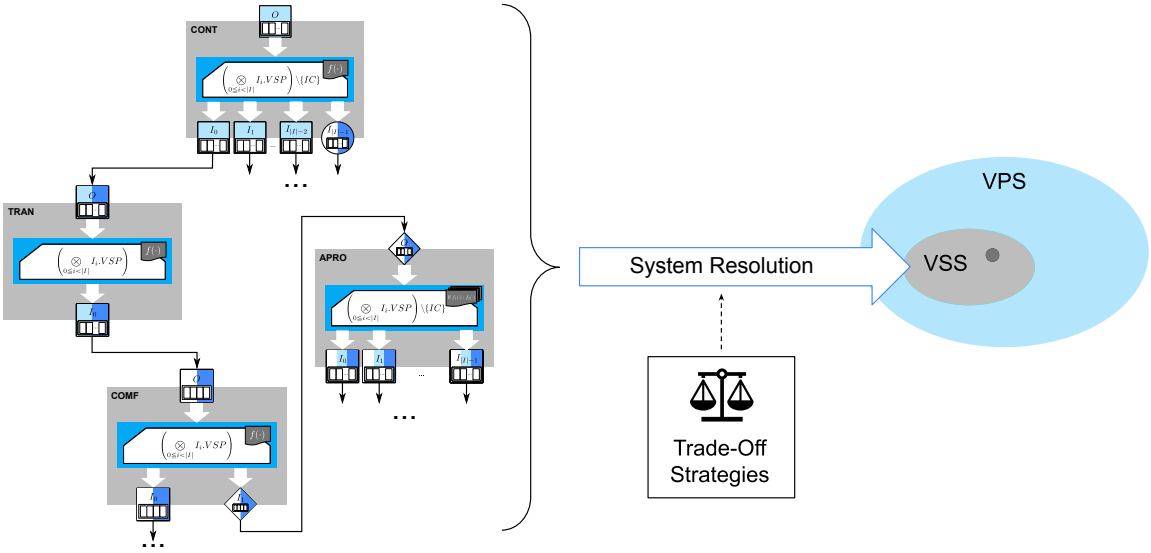


Figure 3.7: DVG systems based on composition of block-port modeling patterns.

Method: DVG Systems

We consider DVG systems, arising through suitable composition of the presented block-port modeling patterns, in chapter 6 in more detail. We analyze the composability between the individual patterns and clarify the overall structure of DVG systems and how this is to be interpreted with respect to the VPS and VSS of a building block. In addition, we introduce how the VSS can be determined from the VPS in different views and we also discuss possible trade-off strategies for the general resolution process of DVG systems (figure 3.7). Finally, we compare DVG systems with the structures of classical value trees and decision trees to illustrate the generalized expressiveness of our DVG approach that is required to model adequate VPSs of building blocks.

Integration/Implementation

We describe the implementation of a blueprint for building block modeling using MDSD in the context of a robotics software ecosystem in chapter 7. This includes a meta model based on EMF Ecore (i) for modeling building blocks but also (ii) for modeling DVG systems which are an essential part of building block models. With the help of textual and graphical DSLs it is then possible to model concrete building blocks and associated DVG systems. A developed code generator is finally able to generate a solver from the DVG system model. The solver realizes the resolution of the modeled DVG system. It determines customized variant bindings of a building block for all possible situations explicated in the model.

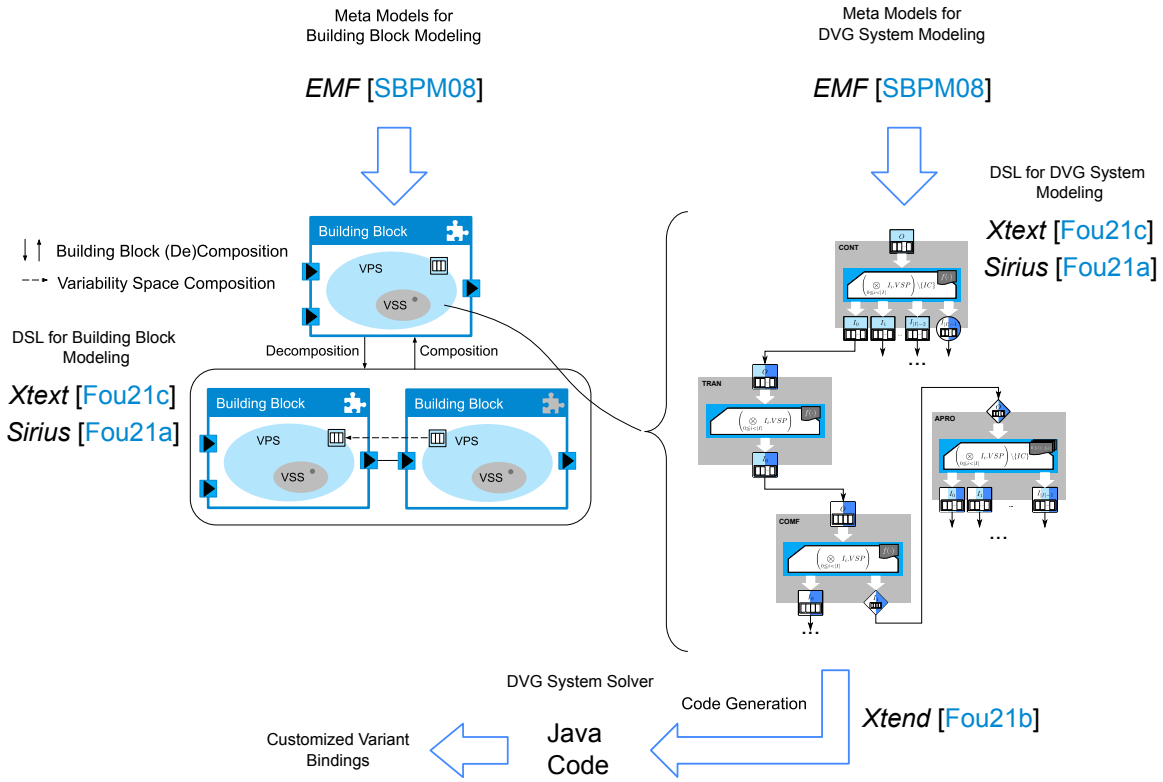


Figure 3.8: The MDE/MDSD implementation of building block modeling and associated DVG system modeling.

Chapter 4

Dependency Variability Graph Basics

In this chapter we introduce the basic principles of the *Dependency Variability Graph* (DVG) method.

4.1 A Special Hypergraph as Foundation

The foundation of every DVG is a directed, simple hypergraph with the following constraints: Every hyperedge must have at least one tail node and exactly one head node. Furthermore, for every node, the out-degree can be greater than or equal to zero while the in-degree must be either zero or one. Simple means there are no loops. Tail nodes of a directed hyperedge are those nodes from where the hyperedge originates while head nodes are those nodes where the arrowhead of the hyperedge points to. Out-degree of a node is the number of outgoing edges part of distinct hyperedges and the in-degree of a node is the number of incoming distinct hyperedges.

Figure 4.1 shows a valid example of such a special hypergraph. We can see that the hyperedge 0 connects the tail nodes 0 and 1 with the head node 3. At the same time, node 3 is also a tail node of the hyperedge 1. The in-degree of the leaf nodes 0, 1 and 2 is 0 while the in-degree of all the other non-leaf nodes 3, 4, 5, 6 and 7 must be always 1. The out-degree of the root node 7 is 0, for node 4 it is 2 and for the rest it is 1.

4.2 From a Hypergraph to a Block-Port-Connector Representation

We apply the principle of modularization to the structure of our special hypergraph by mapping the hypergraph representation to a block-port-connector representation. This step helps to reduce the overall complexity of the approach by introducing clearly separated sub structures of significantly less complexity that can be considered in isolation. The transformation is done as following: Every hyperedge of a hypergraph that connects one or several tail nodes to a head node is considered as a block and the associated nodes represent the ports. More specifically, the tail nodes are the input ports of the block and the head node represents the output port of the block. A connector is introduced between all duplicated ports of different blocks that now appear as output ports in one block and input ports in another block (duplicates which are created for all nodes of our special hypergraph that are at the same time head node and tail node of distinct hyperedges). Note that we then model all connections from the input ports to the output ports, i.e. the arrowhead of a directed edge always points to the output ports. This means that the direction of the arrows is reversed in the block-port-connector representation compared to the original hypergraph representation. Figure 4.2 shows how the hypergraph from figure 4.1 is transformed to a corresponding block-port-connector representation.

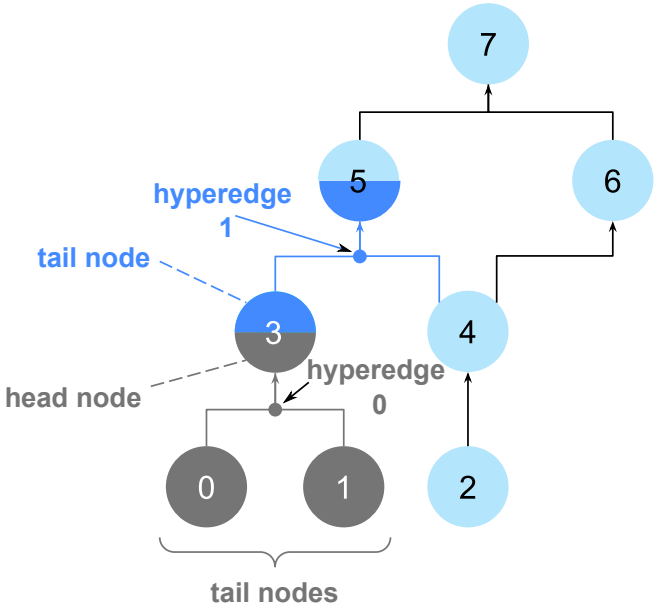


Figure 4.1: Valid example of a special hypergraph structure serving as the foundation of a DVG.

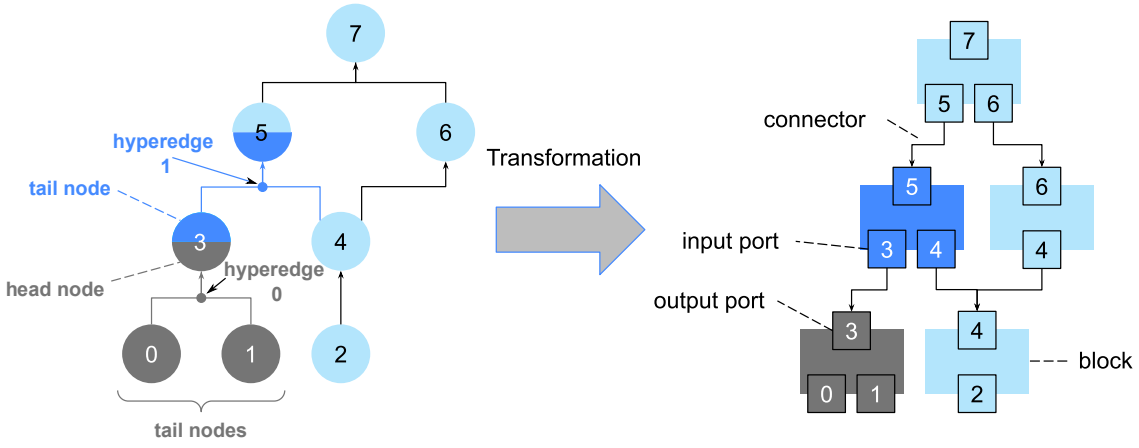


Figure 4.2: Example for the transformation of a hypergraph to a block-port-connector representation.

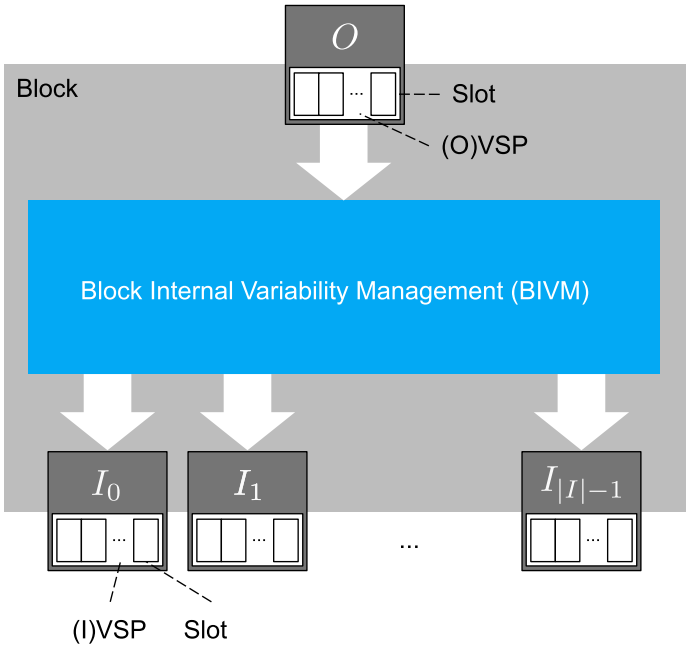


Figure 4.3: Basic internal structure of a block and its ports.

4.3 Blocks and Ports

The block-port-connector representation just introduced is from now on the foundation for everything that follows. In this section, we explain the internal structure and properties of a block and its ports in more detail.

Each port of a block represents a variability entity that consists of a *Variability Space* (VSP) of a specific *Data Object (DO)*. A VSP is a discrete set of *slots*, each holding a concrete *Value Instance (VI)* of the associated DO. The variability relationship between the different slots of a VSP follows a *one out of all* principle meaning that only one exclusive slot of the whole set with its specific VI can occur at a specific moment in time.

The purpose of a block is to handle its *Block-Internal Variability Management* (BIVM) that essentially means processing the *Input Variability Spaces* (IVSPs) of the input ports and creating the *Output Variability Space* (OVSP) of the output port.

Figure 4.3 illustrates the most important elements of the internal structure of a block and its ports.

4.3.1 Block Internal Variability Management

The main purpose of BIVM is to create the OVSP of the output port based on the IVSPs of the input ports. The following key questions arise here:

1. How to combine slots of the different input ports?
2. How to combine the VIs of such combined slots?
3. How to establish and maintain causal links for each newly created slot of an output port to the corresponding combined slots of the input ports?

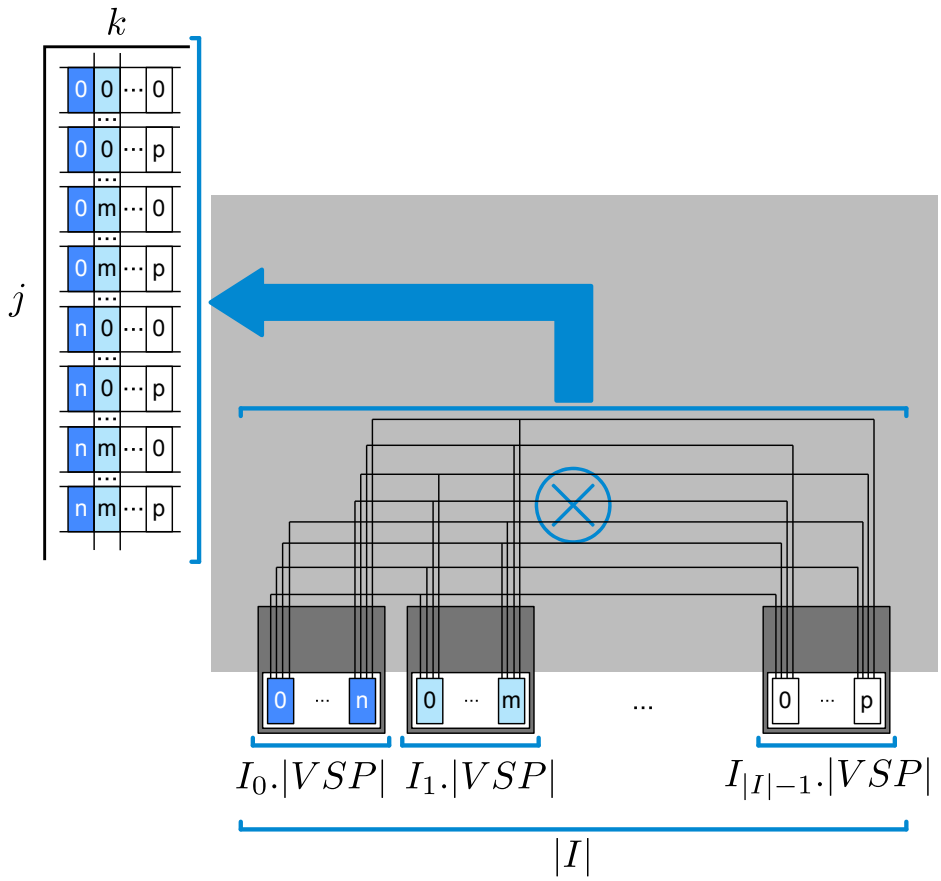


Figure 4.4: The production operator calculates the cartesian product of the IVSPs of a block.

Combinatorial Composition Operators

With respect to the first question, we rely on two basic *combinatorial composition operators* denoted as *production* (symbol: $\otimes$) and *aggregation* (symbol: $\oplus$)

Production Operator The production operator calculates the *cartesian product* between the IVSPs (equation 4.1) where $|I|$ denotes the number of input ports of a block, $|I_i.VSP|$ the size of the VSP of the i-th input port, $|CP|$ the number of resulting combinations of the cartesian product and where $CP_{j,k}$ refers to a slot of the k-th input port of the j-th combination of the created cartesian product. Figure 4.4 visualizes this operation.

$$CP_{j,k} = \left(\bigotimes_{0 \leqq i < |I|} I_i.VSP \right) \text{ where: } 0 \leqq j < \left(\prod_{0 \leqq i < |I|} I_i.|VSP| \right), 0 \leqq k < |I| \quad \text{with: } |CP| = \prod_{0 \leqq i < |I|} I_i.|VSP| \tag{4.1}$$

The production operator guarantees full coverage of potentially achievable result combinations for a block where the different input ports (and thus the represented variability entities) exist separately independently of each other.

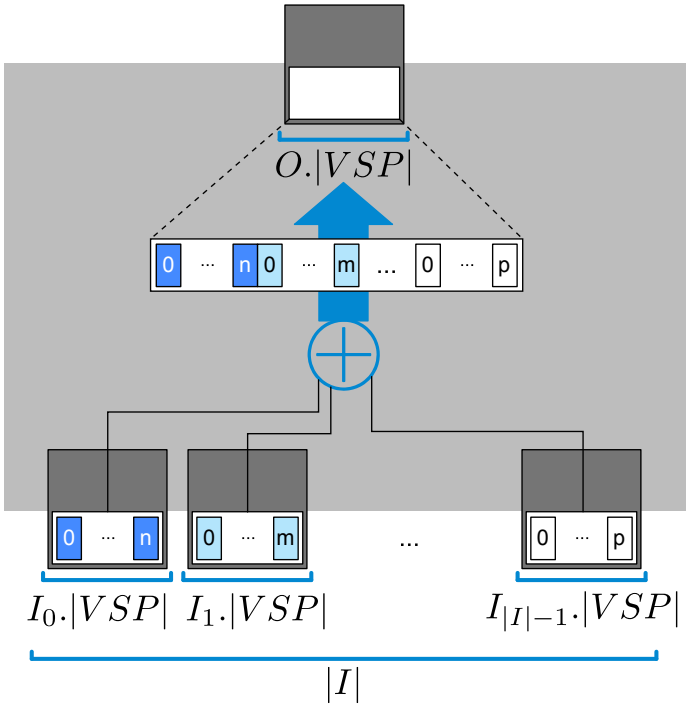


Figure 4.5: The aggregation operator collects the IVSPs of a block.

Aggregation Operator The aggregation operator simply appends the IVSPs sequentially (equation 4.2) where $|O.VSP|$ denotes the size of the resulting OVSP of the output port. Figure 4.5 visualizes this operation.

$$O.VSP = \left(\bigoplus_{0 \leqq i < |I|} I_i.VSP \right) \quad \text{with: } O.|VSP| = \left(\sum_{0 \leqq i < |I|} I_i.|VSP| \right) \tag{4.2}$$

The purpose of this operator is to collect different VSPs achievable via distinct branches. That means that the input ports of a block applying the aggregation operator represent discrete variants. The block can be considered as a fork.

Value Composition Operator

To address the second question, we use the notion of a value composition operator (symbol: $f(\cdot)$) representing an abstract function that operates directly on the VIs of specific slot combinations of the IVSPs to produce new VIs of slots of the OVSP. Figure 4.6 visualizes this operation for combinations created by the production operator where $CP_{j,k}.VI$ refers to the VI of a slot of the k-th input port of the j-th combination of the created cartesian product.

Managing Causal Links

The last concern is handled by organizing the internal structure of slots in a specific manner.

In order to be able to trace back each created slot of a OVSP unambiguously to a specific combination of slots from the IVSP, we create and maintain a *header* within each slot of a VSP that lists $n \times m$

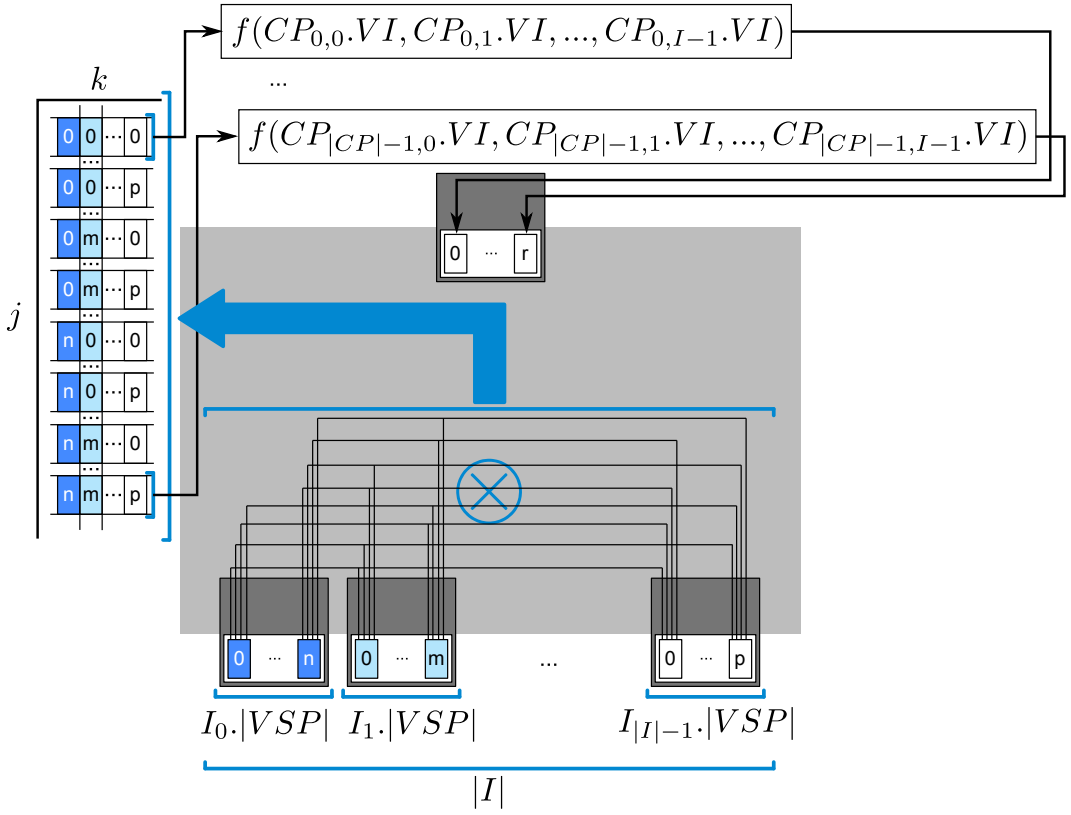


Figure 4.6: Applying the value composition operator to the resulting combinations of the production operator.

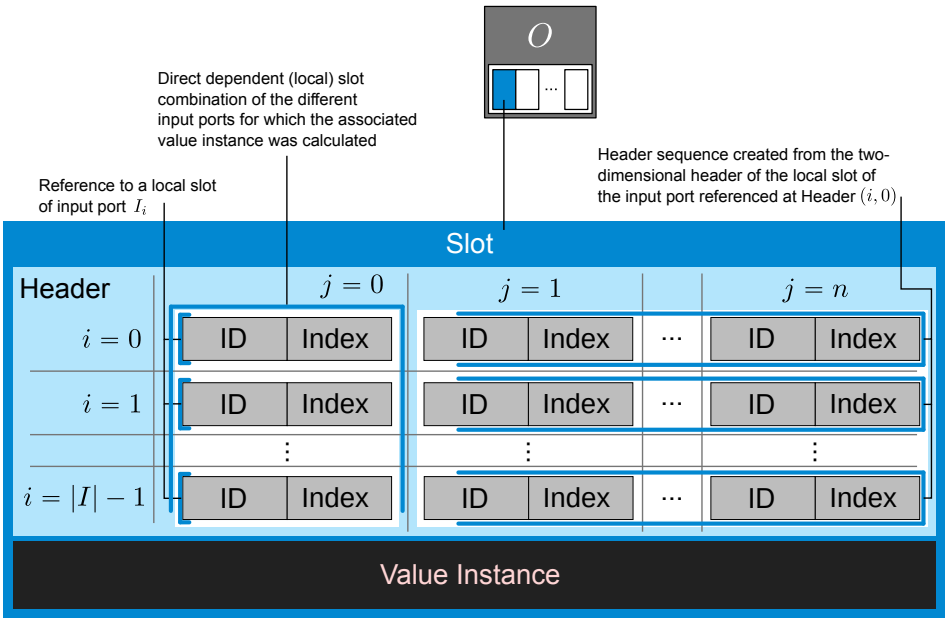


Figure 4.7: The internal structure of a slot. In addition to the VI, each slot maintains a header that captures its dependencies.

pairs of an *id* and an *index*. The id refers to a unique name identifying the corresponding port and the index indicates the corresponding slot index (see figure 4.7 for the internal structure of a slot). If we only capture the direct dependent, local slots, i.e. the slots of the corresponding input ports within each block, the dimension of the header would be $n \times 1$. These relative links are sufficient to lookup the whole dependencies for every slot of every block but to do so we must follow them iteratively all the way down to the leafs of the underlying graph. An alternative is to maintain an absolute list of dependencies for each slot. In that case we additionally append the whole header information of the direct dependent (local) slots (as simple sequences created from their two-dimensional structures) during the BIVM of a block. It follows that a global list of dependencies will always be directly available for each slot of a block after it is processed.

4.3.2 Additional Port Characteristics

We introduce three additional characteristics of ports in the following: (i) *Port Class (PC)*, (ii) *Variability Type (VT)* and (iii) *Representation Type (RT)*. Every port is of a specific class and has a VT and a RT. Figure 4.8 shows the graphical notation for the relevant combinations of PCs, VTs and RTs used in this work in the following.

Port Class

We divide a port into 3 possible classes:

- *Standard Port (SP)*
- *Constraint Port (CP)*
- *Preference Port (PP)*

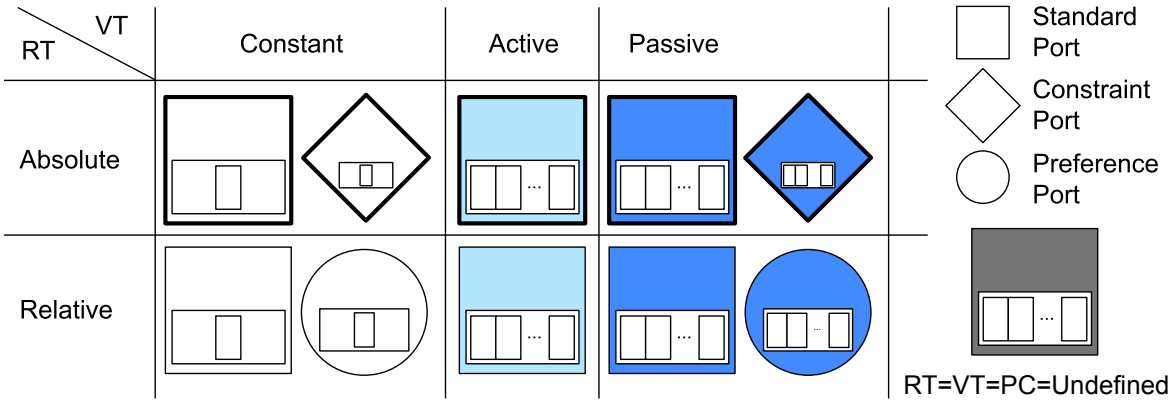


Figure 4.8: Graphical notation for relevant combination of PCs, VTs and RTs.

Different port classes represent different kinds of VSPs with a specific semantic. SPs contain VSPs that represent actual spaces, i.e. spaces defining what can be reached in which situations. They represent VPS dependencies. In contrast, the VSPs of CPs and PPs define a requirement space (constraints and preferences) with respect to the actual space. They represent VSS dependencies.

Variability Type

We distinguish between the following four VTs:

- *Empty (E)*: The port represents a VSP containing $n = 0$ slots
- *Constant (C)*: The port represents a VSP containing $n = 1$ slots
- *Active (A)*: The port represents a VSP containing $n > 1$ slots where 1 out of n VIs are *selectable* at a specific moment in time
- *Passive (P)*: The port represents a VSP with $n > 1$ slots where 1 out of n VIs will *occur* at a specific moment in time

The VT of a port defined in the modeling phase may change as a result of the applied resolution procedure. Therefore we consider the VT of a port P in its *Modeling State* ($P.VT'$), its *Pre-Resolution State* ($P.VT''$) and its *Post-Resolution State* ($P.VT'''$). In general, the size of a VSP of a port that is either active or constant in the modeling state can change after the resolution is finished due to filter operations. That means there is one or several $f(\cdot)$ defined in different blocks realizing a filter operation. In some cases, applying these filter operations can lead to a change of the VT of a port. So can an active VT in the modeling state (i) remain active in the post-resolution state if the filter operation leaves more than 1 slot unfiltered, (ii) reduces to constant if exactly 1 slot remains or (iii) reduces to empty if all slots are filtered. In the same way, constant can remain constant or reduces to empty. The meaning for passive on the other hand is a little different. We consider here an additional pre-resolution state in which passive either remains passive or reduces to constant. The reason for this possible transition depends on the kind of result analysis we want: If passive remains passive we process all $n > 1$ slots of that port in the associated block. This means we perform an analysis in advance, where we check the results for all modeled eventualities of that passive port that might occur. In the other case, we assume that the resolution is triggered at a specific point in time during

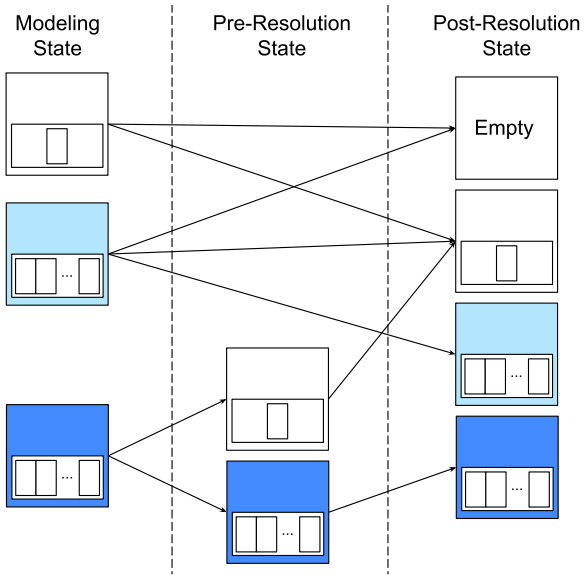


Figure 4.9: Possible transitions for the VT of a standard port between its modeling state, pre-resolution state and post-resolution state.

run-time where a passive port can be reduced to constant, since we consider only a specific snapshot of the passive VSP of that port which occurs at a specific moment in time. Figure 4.9 visualizes the possible transitions of a VT between the different states. Note that empty is not available as modeling state but only as result in the post-resolution state.

Representation Type

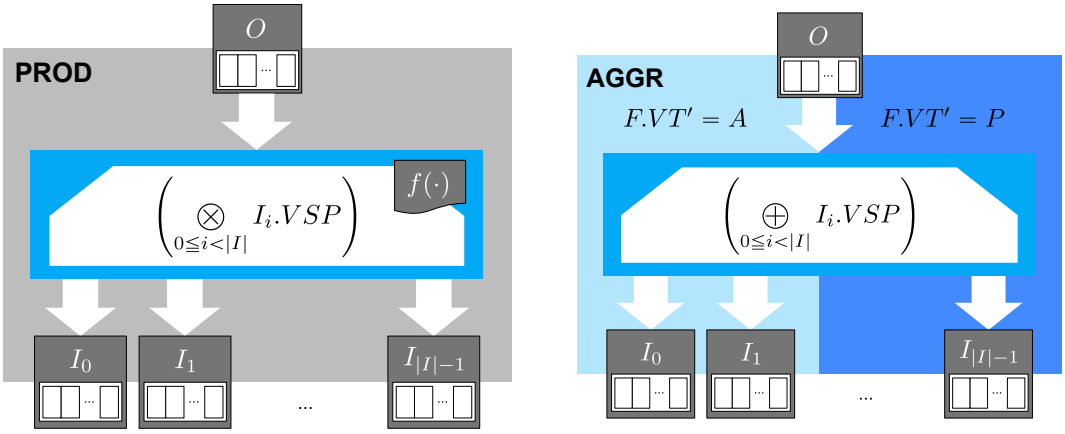
The RT of a port can be either *absolute (A)* or *relative (R)*. An absolute representation means in the standard form of the associated DO. In contrast, relative is a less strong form of representation and just means an arbitrary real value between 0 and 1.

4.3.3 Basic Block-Port Patterns: Production (PROD) and Aggregation (AGGR)

Based on the internal structure of a block and its ports, the concerns of the BIVM and the discussed port characteristics, we present 2 block-port patterns that serve as basic, reusable templates for the contributions of this work: PROD and AGGR.

PROD

Figure 4.10a shows a schematic illustration of the PROD pattern. The number of input ports $|I|$ must be greater than zero. Its BIVM is shown in algorithm 1. The procedure for creating an absolute header row is described in algorithm 3. The equations 4.3 and 4.4 define the relationship between the VT of an output port depending on the VTs of the input ports in the modeling state and the post-resolution state respectively. It states that an output port of a block is considered to the outside as active if at least one input port is active. If in addition one or several passive input ports are also part of this block, it describes how every VI of the active VSP represented by the output port can be achieved in dependence of the corresponding VIs of the other passive input ports according to the defined $f(\cdot)$. In



(a) (b)

Figure 4.10: Schematic illustration of the (a) PROD and (b) AGGR pattern.

Table 4.1: Meaning of the notation used in the different algorithm descriptions.

Notation	Meaning
$I_i.VSP_j.VI$	The VI of the j-th slot of the VSP of the i-th input port
$I_i.VSP_j.H$	The header of the j-th slot of the VSP of the i-th input port
$I_i.N$	The name/unique id of the i-th input port
$M.N$	The name/unique id of a MAGR
$H_{i,j}.ID$	The id of the (id, index) pair at entry (i, j) of the header H
$H_{i,j}.Index$	The index of the (id, index) pair at entry (i, j) of the header H

(a) Concrete notation and its meaning (names of variables occur as is).

Notation	Meaning
$L \leftarrow \varnothing$	Empty initialization of a list structure L of an arbitrary dimension
$L \frown K$	Adds an element K as a new entry to list structure L
(F, S)	Defines a pair of the variables F and S
$P.1$	Accessing the first entry of a pair P
$P.2$	Accessing the second entry of a pair P
$L \backslash K$	Removes entry K from list L

(b) General notation and its meaning (names of variables may change).

other words: We model what will be achieved when we select a specific active variant in dependence of all influencing passive state combinations that might occur. The output port represents all such information after the block is resolved.

Algorithm 1 BIVM of the PROD pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports
▷ Input $f(\cdot)$: A specific value composition operator
▷ Output $O.VSP$: The VSP of the output port O

function PROD($I, f(\cdot)$)
  $T \leftarrow \begin{pmatrix} 0 & 1 & \dots & I_0.|VSP|-1 \\ 0 & 1 & \dots & I_1.|VSP|-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & I_{|I|-1}.|VSP|-1 \end{pmatrix}$
  $CP : \begin{pmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,|I|-1} \\ c_{1,0} & c_{1,1} & \dots & c_{1,|I|-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{|CP|-1,0} & c_{|CP|-1,1} & \dots & c_{|CP|-1,|I|-1} \end{pmatrix} \leftarrow \bigotimes_{0 \leqq i < |I|} T_i$
  for $i \leftarrow 0$ to $< |CP|$ **do**
    $H \leftarrow \varnothing$
    $V \leftarrow \varnothing$
    for $j \leftarrow 0$ to $< |CP_i|$ **do**
      $H \leftarrow H \frown$ CREATEABSOLUTEHEADERROW($I_j.N, CP_{i,j}, I_j.VSP_{CP_{i,j}}.H$)
      $V \leftarrow V \frown I_j.VSP_{CP_{i,j}}.VI$
    end for
    $O.VSP \leftarrow O.VSP \frown (H, f(V))$
  end for
end function

$$O.VT' = \begin{cases} C \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT' = C] \right) = |I| \\ A \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT' = A] \right) > 0 \\ P \text{ else} \end{cases} \tag{4.3}$$

$$O.VT''' = \begin{cases} E \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT''' = E] \right) > 0 \\ C \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT''' = C] \right) = |I| \\ A \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT''' = A] \right) > 0 \\ P \text{ else} \end{cases} \tag{4.4}$$

AGGR

Figure 4.10b shows a schematic illustration of the AGGR pattern. The number of input ports $|I|$ must be greater than one. Its BIVM is shown in algorithm 2. The procedure for creating an absolute header row is described in algorithm 3. For the AGGR pattern applies that the DOs of all the involved ports must be equal (equation 4.5). With respect to RT, all ports within a AGGR must always have the same type (equation 4.6). With respect to the VT, the input ports within a AGGR can be either an arbitrary combination of constant and active or an arbitrary combination of constant and passive, i.e. both active and passive must not occur (equation 4.7). The AGGR pattern is representing a fork and the fork itself also has a VT in the modeling state ($F.VT'$) that must be either active or passive (equation 4.8). Based on that constraints, the equations 4.9 and 4.10 define the relationship between the VT of an output port depending on the VTs of the input ports in the modeling state and the post-resolution state respectively.

Algorithm 2 BIVM of the AGGR pattern. See table 4.1 for some hints about the used notation.

```
▷ Input I: A list of input ports
▷ Output O.VSP: The VSP of the output port O
function AGGR(I)
    for i ← 0 to < |I| do
        for j ← 0 to < I_i.|VSP| do
            H ← ∅
            H ← H ⌢ CreateAbsoluteHeaderRow(I_i.N, j, I_i.VSP_j.H)
            O.VSP ← O.VSP ⌢ (H, I_i.VSP_j.VI)
        end for
    end for
end function
```

$$\forall 1 \leqq i < |I| : O.DO = I_0.DO = I_i.DO \tag{4.5}$$

$$\forall 1 \leqq i < |I| : O.RT = I_0.RT = I_i.RT \tag{4.6}$$

$$\left(\bigcup_{0 \leqq i < |I|} I_i.VT \right) \cap \{A, P\} \neq \{A, P\} \tag{4.7}$$

$$F.VT' \in \{A, P\} \tag{4.8}$$

$$O.VT' = \begin{cases} A \text{ if } F.VT' = A \vee \left(\sum_{0 \leqq i < |I|} [I_i.VT' = A] \right) > 0 \\ \\ P \text{ else} \end{cases} \tag{4.9}$$

$$O.VT''' = \begin{cases} E \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT''' = E] \right) = |I| \\ C \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT''' = E] \right) = |I| - 1 \wedge \left(\sum_{0 \leqq i < |I|} [I_i.VT''' = C] \right) = 1 \\ A \text{ if } \left(\sum_{0 \leqq i < |I|} [I_i.VT''' = A] \right) > 0 \\ P \text{ else} \end{cases} \tag{4.10}$$

Algorithm 3 Procedure for creating an absolute header row for a single slot of an output port based on a certain slot of a specific input port. See table 4.1 for some hints about the used notation.

▷ Input n: The name/id of an input port
▷ Input i: A certain slot index of the input port with name/id n
▷ Input H: A list of list of (id, index) pairs representing the header of the i-th slot of the input port with name/id n
▷ Output R: An absolute header row for a single slot of an output port (a list of (id, index) pairs)

```
1: function CreateAbsoluteHeaderRow(n,i,H)
2:     R ← (n, i)
3:     for all row ∈ H do
4:         for all elem ∈ row do
5:             R ← R ⌢ elem
6:         end for
7:     end for
8:     return R
9: end function
```

4.4 Basic System Aspects

After we know the details of block and port internals, we now come back to the global system view, i.e. several blocks and its ports connected with each other.

4.4.1 Connection Constraints

The following connection constraints for an input port of a block can be derived from the assumptions of our underlying hypergraph:

- An input port of a block can not connect to more than one output port.
- An input port of a block can not connect to its own output port.
- An input port of a block B_1 can not connect to an output port of another block B_2 if one of the input ports of B_2 already connects to the output port of B_1 or if any input port of all blocks $B_3..B_n$ that follow B_2 (i.e. all the blocks whose output ports are connected directly or indirectly (one or more blocks in between) to the input ports of B_2) connects to the output port of B_1.

4.4.2 Split and Merge Situations and the Extended Production Pattern

Input ports of different blocks can be connected to the same output port which means that different blocks depend on the same block. We use the term *Common Port Dependency* (CPD) to denote an output port that is a direct or indirect input for two or more blocks. A *Split and Merge Situation* (SAMS) is given for a block if two or more of its input ports have a CPD, i.e. their dependencies can be traced back to the same output port of a different block. Such a SAMS is either direct (two or more input ports of a block depend on the same output port of another block) or indirect (there is at least one further block between these input ports and the CPD).

If a SAMS occurs for the basic PROD pattern applying the standard production operator, we run into a problem: The different branches of the blocks depending on the CPD produce independent OVSPs containing the VIs of the CPDs VSP. These independent VSPs will finally merge in the block subject to SAMS as different IVSPs. If this block would then apply the production operator to its IVSPs, invalid combinations will be created.

Figure 4.11 shows an example where a PROD block is subject to a SAMS while figure 4.12 illustrates a slot-specific view of the created combinations for that example and indicates which of them are invalid (the blue colored ones). The decisive question is how to handle such situations such that no inconsistencies in form of invalid combinations are processed and forwarded? Basically, there are two solution types:

- A SAMS is prohibited for a PROD in general
- Invalid combinations in a PROD are filtered if a SAMS applies

The former solution would mean to check that every PROD block in a system is not subject to a SAMS. Otherwise the whole system is declared as invalid. A way to check if a SAMS applies by using absolute header information is described in algorithm 4. For the latter solution, we present an extended version of the PROD pattern called EPROD that is able to produce consistent combinations even in presence of SAMS. The pattern checks each combination for validity by using the header information (algorithm 6). A schematic illustration of EPROD is shown in figure 4.13 while its BIVM is described in algorithm 5.

4.4.3 System Resolution

The *System Resolution* (SR) of a DVG system is the process of determining the OVSPs of all its blocks. Since each block can be resolved by applying its corresponding BIVM process, the task of the SR is to trigger these individual BIVM processes in the correct order according to the existing dependency relationships between the blocks. The SR described in algorithm 7 iterates through all the blocks of the system in any order and applies a depth-first search to the current block if it has not yet been resolved. This procedure ensures that the BIVM process of each block is only executed after

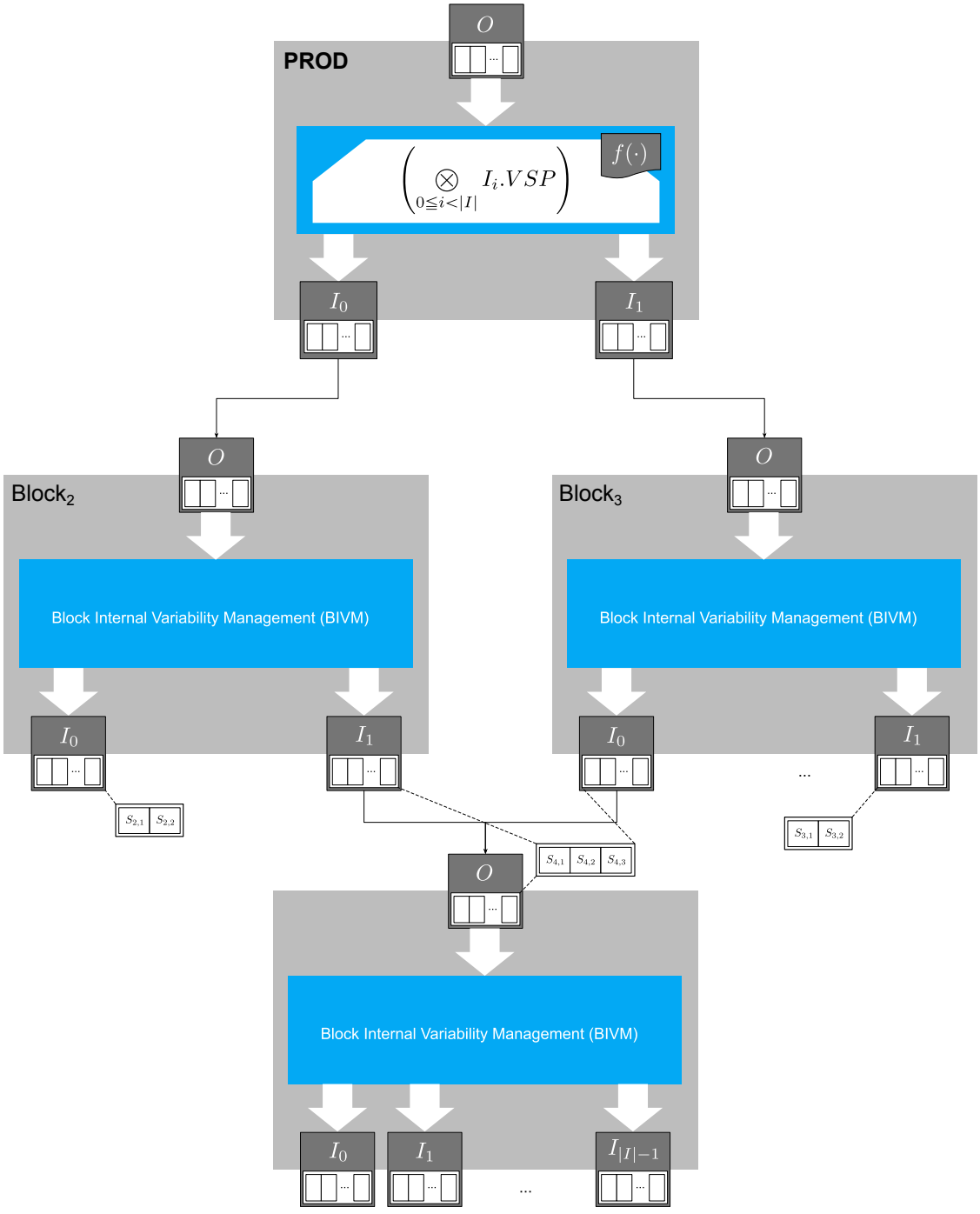


Figure 4.11: Example of a PROD block subject to a SAMS.

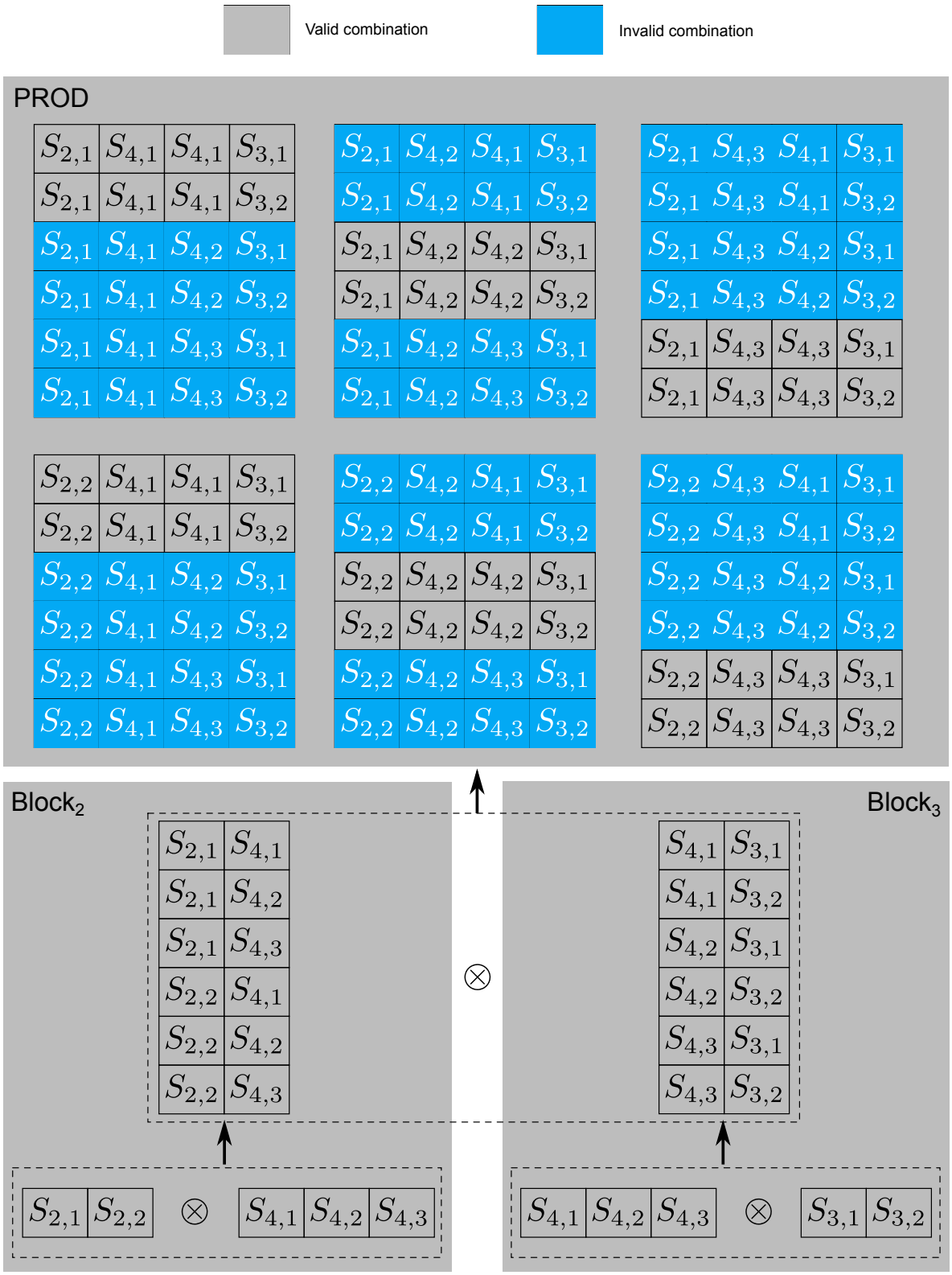


Figure 4.12: Slot-specific view of the produced combinations for the concrete example in figure 4.11. The blue colored combinations are invalid.

Algorithm 4 Algorithm to determine if there is a SAMS for a block. See table 4.1 for some hints about the used notation.

```
    ▷ Input I: A list of input ports
    ▷ Output: Whether there is a SAMS for I or not
 1: function IsSAMS(I)
 2:     H ← ∅
 3:     for i ← 0 to < |I| do
 4:         H ← H ⌢ I_i.VSP_0.H
 5:     end for
 6:     M ← ∅
 7:     for i ← 0 to < |H| do
 8:         for j ← 0 to < |H_i| do
 9:             for k ← 0 to < |H_{i,j}| do
10:                 if H_{i,j,k}.ID ∉ M then
11:                     M ← M ⌢ H_{i,j,k}.ID
12:                 else
13:                     return true
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return false
19: end function
```

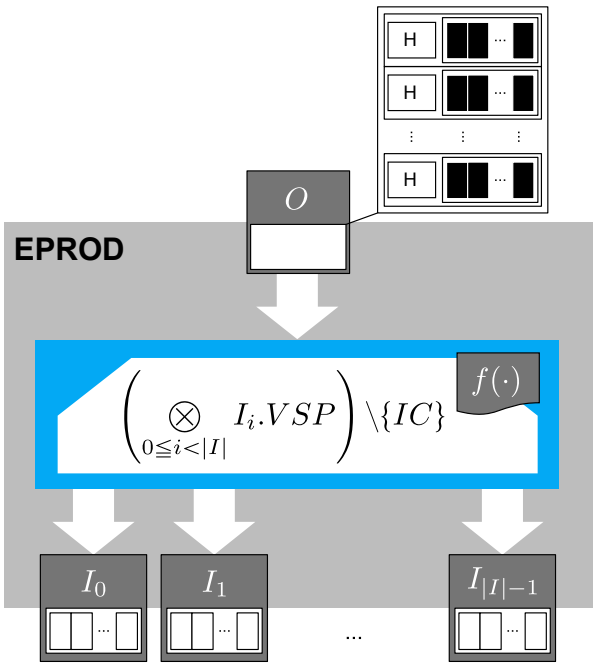


Figure 4.13: Schematic illustration of the extended PROD pattern. $\{IC\}$ means the set of invalid combinations from a SAMS. The output is a list of pairs where the first entry is a header and the second entry a list of VIs.

Algorithm 5 BIVM of the EPROD pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports
▷ Output $O.VSP$: The VSP of the output port O, that is a list of pairs where the first entry is a header and the second entry a list of VIs

1: **function** EPROD(I)
2: $T \leftarrow \begin{pmatrix} 0 & 1 & \dots & I_0.|VSP|-1 \\ 0 & 1 & \dots & I_1.|VSP|-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & I_{|I|-1}.|VSP|-1 \end{pmatrix}$
3: $CP : \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,|I|-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,|I|-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{|CP|-1,0} & c_{|CP|-1,1} & \cdots & c_{|CP|-1,|I|-1} \end{pmatrix} \leftarrow \bigotimes_{0 \leqq i < |I|} T_i$
4: **for** $i \leftarrow 0$ to $< |CP|$ **do**
5: $H \leftarrow \varnothing$
6: $V \leftarrow \varnothing$
7: **for** $j \leftarrow 0$ to $< |CP_i|$ **do**
8: $H \leftarrow H \frown$ CreateAbsoluteHeaderRow($I_j.N, CP_{i,j}, I_j.VSP_{CP_{i,j}}.H$)
9: $V \leftarrow V \frown I_j.VSP_{CP_{i,j}}.VI$
10: **end for**
11: **if** IsValid(H) **then**
12: $O.VSP \leftarrow O.VSP \frown (H, V)$
13: **end if**
14: **end for**
15: **end function**

Algorithm 6 Algorithm to check if a specific slot combination is valid based on its absolute header. See table 4.1 for some hints about the used notation.

▷ Input H: The created absolute header for a specific combination of slots from the input ports of a block
▷ Output: Whether the combination represented by H is valid or not

```
1:  function IsValid(H)
2:      for i ← 0 to < |H| do
3:          for j ← 0 to < |H_i| do
4:              if H_{i,j}.ID ∉ M then
5:                  M ← (H_{i,j}.ID, H_{i,j}.Index)
6:              else
7:                  if H_{i,j}.Index ≠ M_{H_{i,j}.ID}.Index then
8:                      return false
9:                  end if
10:             end if
11:         end for
12:     end for
13:     return true
14: end function
```

all the BIVM processes of its child blocks were already processed. The given algorithm guarantees completeness, meaning that the VPS described by a DVG system is fully traversed by determining the whole OVSP of every block of the system.

Algorithm 7 The SR of a DVG system that guarantees completeness.

▷ Input S: A DVG system
▷ IsResolved(B) returns if block B has already been resolved or not
▷ Before start, we assume $\forall B_{i=0..|S|-1} \in S$: IsResolved($B_i$) = false
▷ The function HasDependency($B.I_i$) checks if the i-th input port of block B references the output port of another block $K \neq B$
▷ The function GetDependency($B.I_i.R$) returns the block K that contains the output port which is referenced by the i-th input port of block B
▷ BIVM(B) calls the BIVM process of block B
▷ IsResolved(B, true) sets the resolved state of block B to true

```
1:  function SR(S)
2:      for all B ∈ S do
3:          if ¬IsResolved(B) then
4:              ResolveNext(B)
5:          end if
6:      end for
7:  end function
8:
9:  function ResolveNext(B)
10:     for i ← 0 to < B.|I| do
11:         if HasDependency(B.I_i) then
12:             K ← GetDependency(B.I_i.R)
13:             if ¬IsResolved(K) then
14:                 ResolveNext(K)
15:             end if
16:         end if
17:     end for
18:     BIVM(B)
19:     IsResolved(B, true)
20: end function
```

Chapter 5

Dependency Variability Graph Block-Port Modeling Patterns

In this chapter we present a set of concrete block-port modeling patterns which are based on the fundamentals and the basic block-port patterns from the previous chapter. The presented patterns in the following serve as concrete modeling templates to compose DVG systems that describe essential aspects of building blocks.

Note that we consider SP as the default port class. Hence, we implicitly assume in our following descriptions about the patterns that ports have to be of that class except it is not explicitly stated otherwise.

5.1 Relative Production (RPRO)

The *Relative Production* (RPRO) pattern is a specialization of PROD with the purpose of producing a OVSP with a relative RT by applying a specific $f(\cdot)$ to a combination of IVSPs with arbitrary RTs (equation 5.1). SAMS are allowed for this pattern. Hence, the EPROD is applied to filter invalid combinations that may occur. There are no constraints regarding the VT of the input ports, i.e. there can be any combination of constant, active and passive VTs if the output port is of the class SP (equation 5.2). Figure 5.1a and algorithm 8 show the relevant details of RPRO.

$$\begin{aligned} \forall 0 \leqq i < |I| : I_i.RT &\in \{R, A\} \\ O.RT &\in \{R\} \end{aligned} \tag{5.1}$$

$$\forall 0 \leqq i < |I| : I_i.VT' \in \begin{cases} \{C, A, P\} \text{ if } O.PC = SP \\ \{C, P\} \text{ if } O.PC = PP \end{cases} \tag{5.2}$$

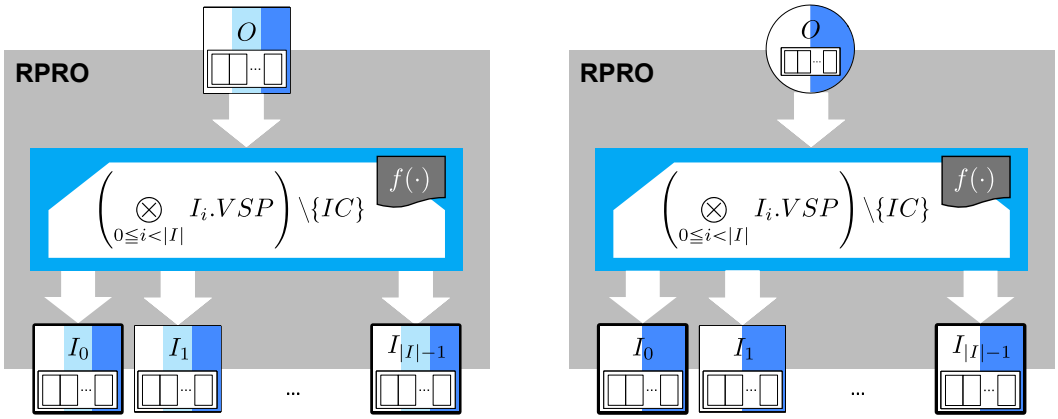
5.1.1 RPRO with Preference Output

A further usage of RPRO is to provide a preference specification based on its inputs. In that case, the output port of this RPRO pattern variant is of class PP and the VT of the output port is restricted to be either constant or passive. It follows (based on equation 4.3) that the input ports must be either constant or passive as well (equation 5.2). Figure 5.1b shows a schematic illustration of that pattern variant.

Algorithm 8 BIVM of the RPRO pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports
▷ Input $f(\cdot)$: A specific value composition operator
▷ Output $O.VSP$: The VSP of the output port O

function RPRO($I, f(\cdot)$)
  s ← false
  if IsSAMS(I) **then**
    s ← true
  end if
  $T \leftarrow \begin{pmatrix} 0 & 1 & \dots & I_0.|VSP|-1 \\ 0 & 1 & \dots & I_1.|VSP|-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & I_{|I|-1}.|VSP|-1 \end{pmatrix}$
  $CP : \begin{pmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,|I|-1} \\ c_{1,0} & c_{1,1} & \dots & c_{1,|I|-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{|CP|-1,0} & c_{|CP|-1,1} & \dots & c_{|CP|-1,|I|-1} \end{pmatrix} \leftarrow \bigotimes_{0 \leqq i < |I|} T_i$
  for $i \leftarrow 0$ to $< |CP|$ **do**
    $H \leftarrow \varnothing$
    $V \leftarrow \varnothing$
    for $j \leftarrow 0$ to $< |CP_i|$ **do**
      $H \leftarrow H \frown$ CreateAbsoluteHeaderRow($I_j.N, CP_{i,j}, I_j.VSP_{CP_{i,j}}.H$)
      $V \leftarrow V \frown I_j.VSP_{CP_{i,j}}.VI$
    end for
    if ¬s **then**
      $O.VSP \leftarrow O.VSP \frown (H, f(V))$
    else
      if IsValid(H) **then**
        $O.VSP \leftarrow O.VSP \frown (H, f(V))$
      end if
    end if
  end for
end function



(a) (b)

Figure 5.1: The RPRO pattern (a) with a SP as output and (b) with a PP as output.

5.2 Comparison Filter (COMF)

The *Comparison Filter* (COMF) pattern is a specialization of PROD and filters slots of an IVSP that are not valid regarding a specific constraint space. Hence, it has exactly 2 input ports where the first one always represents the VSP to be filtered and the second one the constraint space defined via a CP (equation 5.3). The DO of both input ports and the output port must match (equation 5.4) and their RT must be absolute (equation 5.5). The VT of the first input port must be either constant or active because there are no reasons to filter passive VSPs while the second CP can be only constant or passive (equation 5.6). Furthermore, both input ports must be completely independent of each other, i.e. no SAMS is allowed. Figure 5.2 and algorithm 9 show the relevant details of COMF.

$$|I| = 2 \text{ where } I_1.PC = CP \tag{5.3}$$

$$I_0.DO = I_1.DO = O.DO \tag{5.4}$$

$$I_0.RT = I_1.RT = O.RT \in \{A\} \tag{5.5}$$

$$\begin{aligned} I_0.VT' &\in \{C, A\} \\ I_1.VT' &\in \{C, P\} \end{aligned} \tag{5.6}$$

5.3 Absolute Production (APRO)

The *Absolute Production* (APRO) pattern produces an OVSP with an absolute RT by applying a specific $f(\cdot)$ to different IVSPs with an absolute RT (equation 5.7). SAMS are allowed to occur and will be filtered by an internal EPROD. There are no constraints regarding the VT of the input ports if the output port is of the class SP (equation 5.8).

From an outside view, the APRO is same to the RPRO with the only difference of the RT constraints. However, the APRO can have a more complex internal structure. This is because it offers the option to model a set of different *descriptions* where each description must contain a *core* block

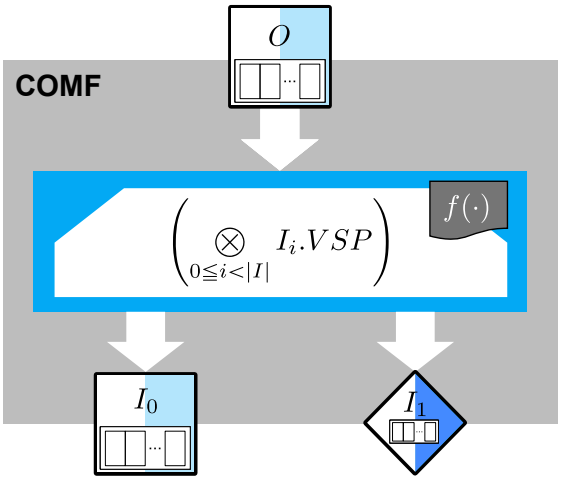


Figure 5.2: The COMF pattern.

Algorithm 9 BIVM of the COMF pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports
▷ Input $f(\cdot)$: A specific value composition operator defining a comparison function
▷ Output $O.VSP$: The VSP of the output port O

function COMF($I, f(\cdot)$)
  if IsSAMS(I) **then**
    return Error: Invalid structure
  end if
  for $i \leftarrow 0$ to $< I_0.|VSP|$ **do**
    for $j \leftarrow 0$ to $< I_1.|VSP|$ **do**
      $H \leftarrow \varnothing$
      if $f(I_0.VSP_i.VI, I_1.VSP_j.VI)$ **then**
        $H \leftarrow H \frown$ CreateAbsoluteHeaderRow($I_0.N, i, I_0.VSP_i.H$)
        $H \leftarrow H \frown$ CreateAbsoluteHeaderRow($I_1.N, j, I_1.VSP_j.H$)
        $O.VSP \leftarrow O.VSP \frown (H, I_0.VSP_i.VI)$
      end if
    end for
  end for
end function

whose BIVM works similar to the BIVM of RPRO (see algorithm 8). A description can optionally have a set of *preconditions*. All of them must be fulfilled in order to provide a non-empty output in the associated core block. If there are more than one description in a APRO, each description must have a precondition and the overall number of preconditions per description must be equal (see equation 5.9 where $|D|$ denotes the number of descriptions and $D_i.|P|$ the number of preconditions of the i-th description). The purpose of the different descriptions and its preconditions is to have a trade-off variant for describing complex input-output relationships where an accurate $f(\cdot)$ that is generic over all possible combinations of the different inputs is not known yet. That means, different preconditions of different descriptions with different $f(\cdot)$ state that the output is only consistent/accurate if the inputs fulfill specific conditions. Figure 5.3a provides a schematic illustration of the outer view of the APRO pattern while 5.4 describes the internal structure in more detail.

$$\begin{aligned} \forall 0 \leqq i < |I| : I_i.RT &\in \{A\} \\ O.RT &\in \{A\} \end{aligned} \tag{5.7}$$

$$\forall 0 \leqq i < |I| : I_i.VT' \in \begin{cases} \{C, A, P\} \text{ if } O.PC = SP \\ \{C, P\} \text{ if } O.PC = CP \end{cases} \tag{5.8}$$

$$\forall 1 \leqq i < |D| : D_0.|P| > 0 \wedge D_0.|P| = D_i.|P| \tag{5.9}$$

5.3.1 Precondition

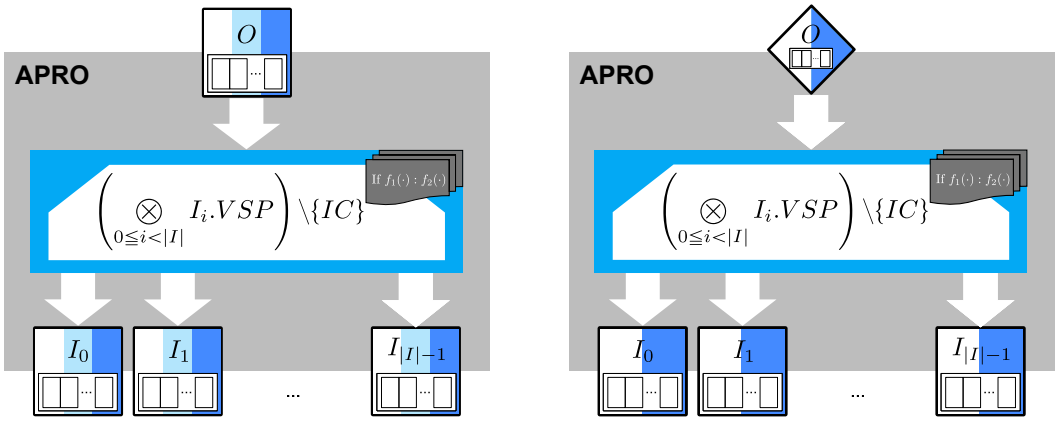
A special COMF with different VT constraints (equation 5.10) is used to realize a precondition in a description of an APRO. The first input port of a precondition is always connected to one of the outer input ports of the APRO. The second input (CP) has no dependencies to the outside and represents only a constant comparison value (see equation 5.11 where $D_i.P_j.I_1.R$ denotes the connection reference R of the first input port I of the j-th precondition of the i-th description). Note, that preconditions for the different descriptions must be designed in such a way that same input combinations for the set of preconditions must not be valid in more than one description.

$$\begin{aligned} I_0.VT' &\in \{C, A, P\} \\ I_1.VT' &\in \{C\} \end{aligned} \tag{5.10}$$

$$\forall 0 \leqq i < |D|, \forall 0 \leqq j < D_i.|P| : D_i.P_j.I_1.R = \varnothing \tag{5.11}$$

5.3.2 Core

The BIVM of the core block of a description is exactly the same as for the RPRO shown in algorithm 8 if there is only one description and no precondition. In that case the input ports of the inner core are connected directly to all outer input ports of the APRO. If preconditions are used, the core is only connected to a subset of the input ports of the APRO directly while the remaining ones are linked to the core indirectly via the output ports of the preconditions. The BIVM of the core will then consider all its IVSPs when creating the header of a resulting slot of its OVSP but does not take into account the inputs from the preconditions when applying the corresponding $f(\cdot)$. Note, that the output port of the core of a description will be empty if at least one of its preconditions is not satisfied. The following additional constraints are defined:



(a) (b)

Figure 5.3: The APRO pattern (a) with a SP as output and (b) with a CP as output.

- The set of the outer input ports of APRO referenced directly by the input ports of the inner core must match between all cores of the different descriptions (see equation 5.12 where $D_i.C.I.R$ denotes the set of port connection references R of an input port set I of the core C of the i-th description D)

- The same outer input port of APRO can not be referenced directly from an inner input port of a core and an inner input port of a precondition at the same time in the same description (equation 5.13)

$$\forall 1 \leqq i < |D| : D_0.C.I.R = D_i.C.I.R \tag{5.12}$$

$$\forall 0 \leqq i < |D|, \forall 0 \leqq j < D_i.C.|I|, \forall 0 \leqq k < D_i.|P| : D_i.C.I_j.R \neq D_i.P_k.I_0.R \tag{5.13}$$

5.3.3 Final Output

If there is only one description, the output port of the APRO is the output port of the core from that one description. Otherwise, a internal AGGR is applied to aggregate the output ports of the cores of the different descriptions. This aggregated output port then represents the final output of the APRO. Based on that, the constraint given in equation 5.14 must always hold. Then, the VT of the final output port in the post-resolution state of a APRO can be formally derived by equation 5.15 where $A.I_i$ denotes the i-th input port of the AGGR block in the APRO.

$$\begin{aligned} &\forall 1 \leqq i < |D| : \\ &\quad O.DO = D_0.C.O.DO = D_i.C.O.DO \wedge \\ &\quad O.RT = D_0.C.O.RT = D_i.C.O.RT \wedge D_0.C.O.VT' = D_i.C.O.VT' \end{aligned} \tag{5.14}$$

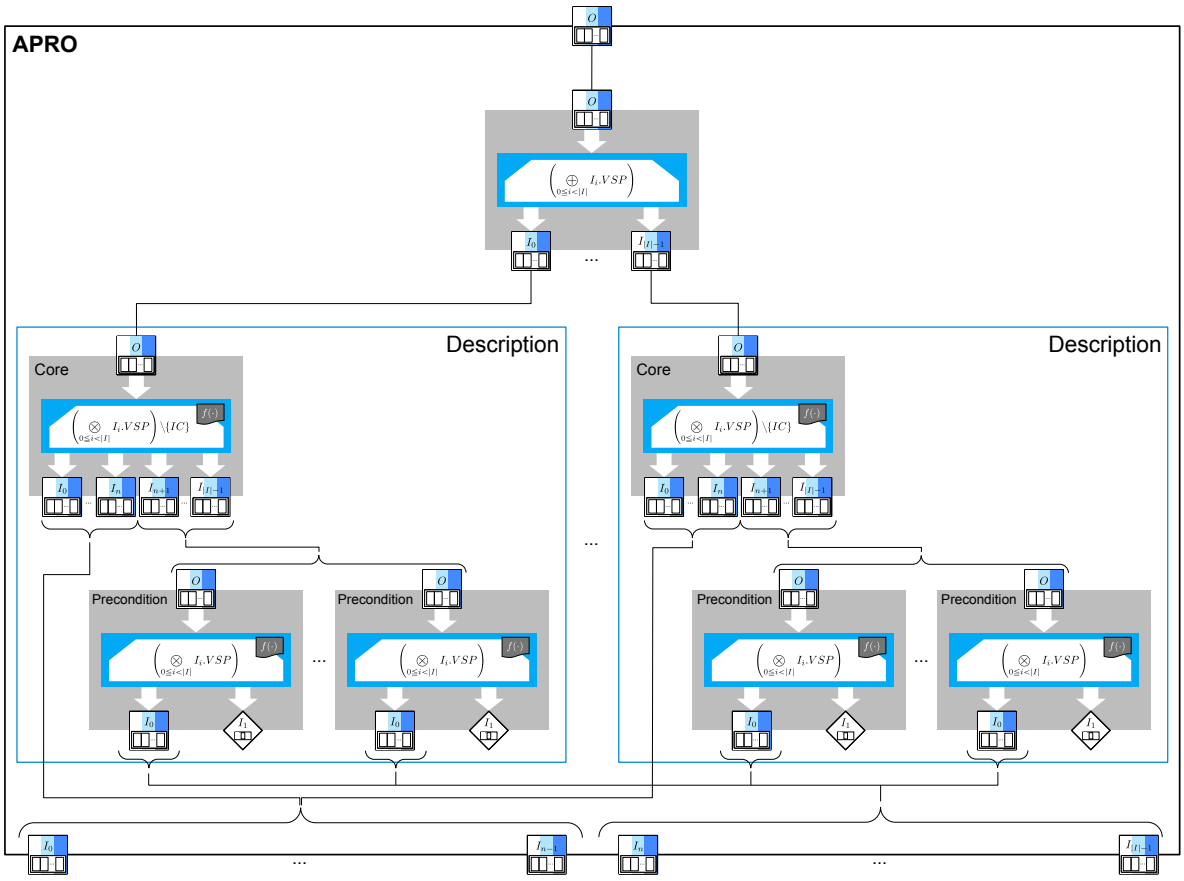


Figure 5.4: The internal structure of the APRO pattern.

$$O.VT''' = \begin{cases} E \text{ if } \left(\sum_{0 \leqq i < A.|I|} [A.I_i.VT''' = E] \right) = A.|I| \\ C \text{ if } \left(\sum_{0 \leqq i < A.|I|} [A.I_i.VT''' = E] \right) = A.|I| - 1 \wedge \left(\sum_{0 \leqq i < A.|I|} [A.I_i.VT''' = C] \right) = 1 \\ D_0.C.O.VT' \text{ else} \end{cases} \tag{5.15}$$

5.3.4 APRO with Constraint Output

Similar to RPRO providing a VSP of preferences, the APRO pattern can be used to provide a VSP of resulting constraints in dependence of its inputs. In that case the output port of this APRO pattern variant is of class CP and the VT of the output port is restricted to be either constant or passive. It follows (based on equation 4.3) that the input ports must be either constant or passive as well (equation 5.8). Figure 5.3b shows a schematic illustration of that pattern variant.

5.4 Multiple Aggregation (MAGR)

The *Multiple Aggregation* (MAGR) pattern is an extension of AGGR because it can contain one or several AGGRs (equation 5.16). The main purpose of this pattern is to model a common fork that represents different VSP combinations of distinct DOs.

Each AGGR A within a MAGR M must have at least 2 input ports (equation 5.17). Furthermore, each A within a M must represent a different DO (equation 5.18). The order of the input ports of the different As in a M defines the relationship regarding the common fork as following: The VSP of the j-th input port of the i-th A is directly coupled with the j-th input ports of all the other As. That means, if the j-th variant is selected/happens for this common fork, we can achieve the VSPs of each j-th input port of every A of a M. Hence, all As must have the same number of input ports (equation 5.19). This relationship is also captured in the header of each slot to ensure that other patterns referencing different output ports of the same M are able to identify and filter invalid combinations. According to 4.6, all ports within a A must always have the same RT for every A in a M and can be either absolute or relative (equation 5.20). According to 4.7, the input ports within an A can be either an arbitrary combination of constant and active or an arbitrary combination of constant and passive, i.e. both active and passive must not occur for every A in a M (equation 5.21). The determination of the VT of an output port based on the VT of the input ports is same as for the basic AGGR (see equation 4.9 and 4.10). Figure 5.5 and algorithm 10 show the relevant details of MAGR.

$$M.|A| > 0 \tag{5.16}$$

$$\forall 0 \leqq i < M.|A| : M.A_i.|I| > 1 \tag{5.17}$$

$$\forall 0 \leqq i < M.|A|, \forall 0 \leqq j < M.|A| \text{ where } i \neq j : M.A_i.O.DO \neq M.A_j.O.DO \tag{5.18}$$

$$\forall 1 \leqq i < M.|A| : M.A_0.|I| = M.A_i.|I| \tag{5.19}$$

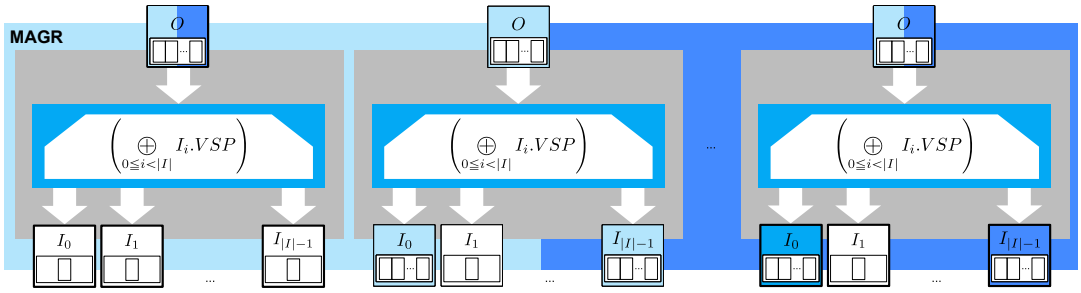


Figure 5.5: The MAGR pattern.

$$\begin{aligned} \forall 0 \leqq i < M.|A|, \forall 1 \leqq j < M.A_i.|I| : \\ M.A_i.O.RT = M.A_i.I_0.RT = M.A_i.I_j.RT \\ \in \{A, R\} \end{aligned} \tag{5.20}$$

$$\forall 0 \leqq i < M.|A| : \left(\bigcup_{0 \leqq j < M.A_i.|I|} M.A_i.I_j.VT \in \{C, A, P\} \right) \cap \{A, P\} \neq \{A, P\} \tag{5.21}$$

Algorithm 10 BIVM of the MAGR pattern. See table 4.1 for some hints about the used notation.

▷ Input A: A list of a list of input ports where $A_i.I_j$ refers to the j-th input port of the i-th AGGR
▷ Output $A_{[0..|A|-1]}.O.VSP$: The VSP of the output port O of every AGGR A

1: **function** MAGR(A)
2: **for** $i \leftarrow 0$ to $< |A|$ **do**
3: **for** $j \leftarrow 0$ to $< A_i.|I|$ **do**
4: $fid \leftarrow (M.N, j)$
5: **for** $k \leftarrow 0$ to $< A_i.I_j.|VSP|$ **do**
6: $H \leftarrow \varnothing$
7: $H \leftarrow H \frown fid$
8: $H \leftarrow H \frown$ CREATEABSOLUTEHEADERROW($A_i.I_j.N, k, A_i.I_j.VSP_k.H$)
9: $A_i.O.VSP \leftarrow A_i.O.VSP \frown (H, A_i.I_j.VSP_k.VI)$
10: **end for**
11: **end for**
12: **end for**
13: **end function**

5.5 Transformation (TRAN)

The *Transformation* (TRAN) pattern is a specialization of PROD that maps the absolute IVSP of exactly one input port ($|I| = 1$) to a relative OVSP (equation 5.22) by applying a specific normalization function. The VT of the input port must be either active or passive (equation 5.23). Figure 5.6 and

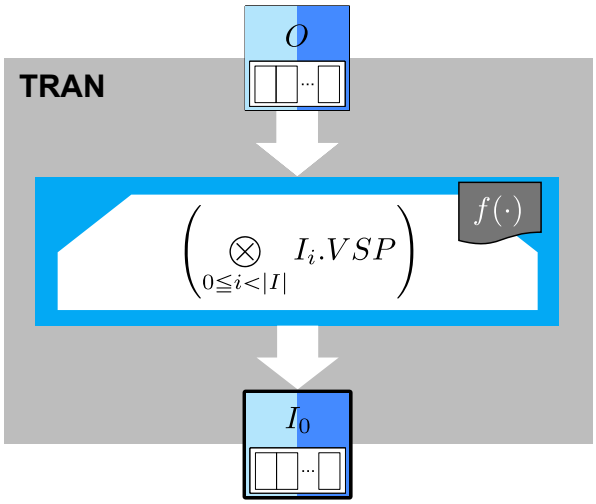


Figure 5.6: The TRAN pattern.

algorithm 11 show the relevant details of TRAN.

$$\begin{aligned} I_0.RT &\in \{A\} \\ O.RT &\in \{R\} \end{aligned} \tag{5.22}$$

$$I_0.VT' \in \{A, P\} \tag{5.23}$$

Algorithm 11 BIVM of the TRAN pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports
▷ Input $f(\cdot)$: A specific value composition operator defining a normalization function
▷ Input min: Optional minimum value constant for the normalization
▷ Input max: Optional maximum value constant for the normalization
▷ Output $O.VSP$: The VSP of the output port O

1: **function** TRAN($I, f(\cdot), min, max$)
2: **if** $min = \varnothing$ **then**
3: $min \leftarrow \min(I_0.VSP)$
4: **end if**
5: **if** $max = \varnothing$ **then**
6: $max \leftarrow \max(I_0.VSP)$
7: **end if**
8: **for** $i \leftarrow 0$ to $< I_0.|VSP|$ **do**
9: $H \leftarrow$ CREATEABSOLUTEHEADERROW($I_0.N, i, I_0.VSP_i.H$)
10: $v \leftarrow I_0.VSP_i.VI$
11: $O.VSP \leftarrow O.VSP \frown (H, f(v, min, max))$
12: **end for**
13: **end function**

5.6 Contradiction (CONT)

The *Contradiction* (CONT) pattern is a specialization of PROD that models a conflict between its input ports and produces a relative OVSP by applying a specific rating operator taking into account a preference specification for these input ports. Therefore it requires at least 2 input ports representing the conflict and an additional, last input port of the class PP representing a preference space i.e. a VSP of preference specifications (equation 5.24). The DOs of all the conflict input ports and the output port must be distinct from each other (equation 5.25). Both all input ports and the output port must have a relative RT (equation 5.26) because the output represents a normalized utility VSP and the inputs must be represented uniformly (normalized) in order to apply a rating function in a meaningful way. Furthermore, deriving utility VSPs based on the preference space of a PP makes only sense for active VTs of the input ports in conflict with each other (equation 5.27). In contrast, the VT of the PP can be either constant or passive (equation 5.28). A CONT is only consistent if a SAMS exists between the input ports in conflict with each other because a conflict can not exist at all if these input ports have no CPD which would mean they are completely independent. Hence the inequality given in equation 5.29 must always hold for a CONT. For that reason the pattern uses the EPROD to filter invalid combinations which must inevitably occur in this block. Figure 5.7 and algorithm 12 show the relevant details of CONT.

$$|I| > 2 \text{ where } I_{|I|-1}.PC = PP \tag{5.24}$$

$$\forall 0 \leqq i < |I| - 1, \forall 0 \leqq j < |I| - 1 \text{ where } i \neq j : I_i.DO \neq I_j.DO \neq O.DO \tag{5.25}$$

$$\begin{aligned} \forall 0 \leqq i < |I| - 1 : I_i.RT &\in \{R\} \\ O.RT &\in \{R\} \end{aligned} \tag{5.26}$$

$$\forall 0 \leqq i < |I| - 1 : I_i.VT' \in \{A\} \tag{5.27}$$

$$I_{|I|-1}.VT' \in \{C, P\} \tag{5.28}$$

$$O.|VSP| < \left(\prod_{0 \leqq i < |I|} I_i.|VSP| \right) \tag{5.29}$$

5.7 Pareto Transformation Contradiction Composite (PTCC)

The *Pareto Transformation Contradiction Composite* (PTCC) pattern provides an utility OVSP for conflicting IVSP combinations that are on the *Pareto Frontier*, i.e. that are non-dominated. The equations 5.24, 5.25, 5.27, 5.28 and 5.29 from the CONT also apply for the PTCC. For the same reasons as for CONT, PTCC must use the EPROD. The only difference with respect to CONT is with the RT of the conflict input ports that must be absolute instead of relative (equation 5.30) here. The main internal steps of the pattern are (i) to perform a $|I|$-1-dimensional pareto filtering process of the value combinations from the conflict input ports, (ii) $|I|$-1 times application of normalization functions to normalize the remaining, unfiltered absolute slots from the conflict input ports and finally

Algorithm 12 BIVM of the CONT pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports where the last input port is of type PP
▷ Input $f(\cdot)$: A specific value composition operator defining a rating function
▷ Output $O.VSP$: The VSP of the output port O

1: **function** CONT($I, f(\cdot)$)
2: **if** $\neg$IsSAMS(I) **then**
3: **return** Error: Invalid structure
4: **else**
5: $$T \leftarrow \begin{pmatrix} 0 & 1 & \dots & I_0.|VSP| - 1 \\ 0 & 1 & \dots & I_1.|VSP| - 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1 & \dots & I_{|I|-1}.|VSP| - 1 \end{pmatrix}$$
6: $$CP : \begin{pmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,|I|-1} \\ c_{1,0} & c_{1,1} & \dots & c_{1,|I|-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{|CP|-1,0} & c_{|CP|-1,1} & \dots & c_{|CP|-1,|I|-1} \end{pmatrix} \leftarrow \bigotimes_{0 \leqq i < |I|} T_i$$
7: **for** $i \leftarrow 0$ to $< |CP|$ **do**
8: $H \leftarrow \varnothing$
9: $V \leftarrow \varnothing$
10: $PV \leftarrow \varnothing$
11: **for** $j \leftarrow 0$ to $< |CP_i|$ **do**
12: $H \leftarrow H \frown$ CreateAbsoluteHeaderRow($I_j.N, CP_{i,j}, I_j.VSP_{CP_{i,j}}.H$)
13: **if** $j < |CP_i| - 1$ **then**
14: $V \leftarrow V \frown I_j.VSP_{CP_{i,j}}.VI$
15: **else**
16: $PV \leftarrow I_j.VSP_{CP_{i,j}}.VI$
17: **end if**
18: **end for**
19: **if** IsValid(H) **then**
20: $O.VSP \leftarrow O.VSP \frown (H, f(V, PV))$
21: **end if**
22: **end for**
23: **end if**
24: **end function**

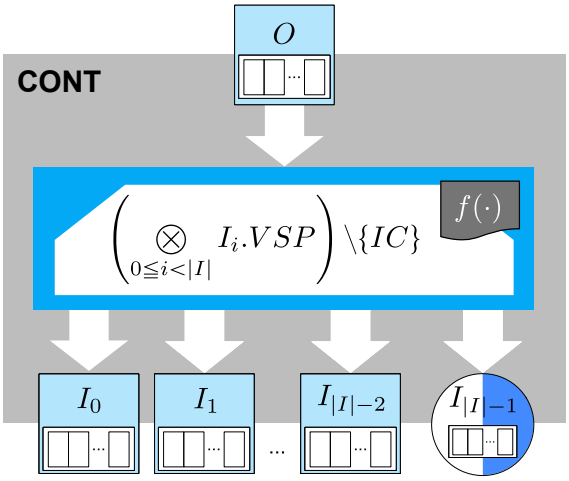


Figure 5.7: The CONT pattern.

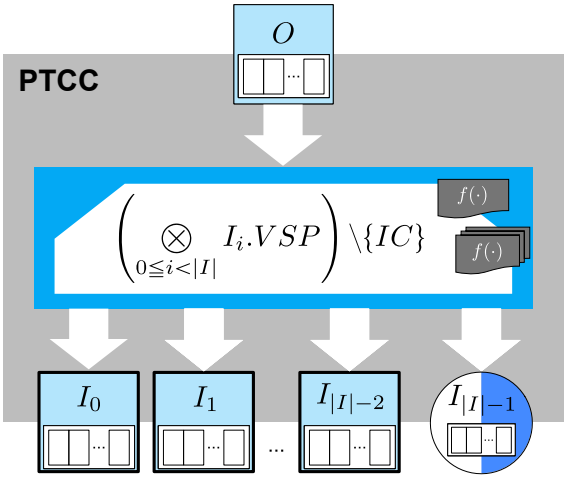


Figure 5.8: The PTCC pattern.

(iii) the calculation of the resulting utility OVSP by applying a specific rating function to the unfiltered, normalized value combinations from the conflict input ports in combination with the preference space of the PP. Figure 5.8 and algorithm 13 show the relevant details of the PTCC. Algorithm 14 describes the used IsDom function.

$$\begin{aligned} \forall 0 \leqq i < |I| - 1 : I_i.RT \in \{A\} \\ O.RT \in \{R\} \end{aligned} \tag{5.30}$$

Algorithm 13 BIVM of the PTCC pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports where the last input port is of type PP.
▷ Input T: A list of specific value composition operators defining normalization functions.
▷ Input r: A specific value composition operator defining a rating function.
▷ Output $O.VSP$: The VSP of the output port O.

function PTCC(I, T, r)
  if ¬IsSAM(I) **then**
    return Error: Invalid structure
  else
    $V \leftarrow$ EPROD(I) ▷ See figure 4.13 and algorithm 5
    for $i \leftarrow 0$ to $< |I| - 1$ **do**
      $M \leftarrow M \frown$ IsToMaximize(I_i) ▷ If i-th input wants to be maximized or not
    end for
    for $i \leftarrow 0$ to $< |V|$ **do**
      $D \leftarrow D \frown$ false
    end for
    for $i \leftarrow 0$ to $< |V|$ **do** ▷ V: A list of pairs of a header and a list of VIs
      for $j \leftarrow 0$ to $< |V|$ **do**
        if $i \neq j \wedge \neg D_i$ **then**
          if IsDom($V_i.2 \backslash V_i.2_{|V_i.2|-1}, V_j.2 \backslash V_j.2_{|V_j.2|-1}, M$) **then**
            $D_i \leftarrow$ true ▷ i-th comb is dominated by another comb $j \neq i$
          end if
        end if
      end for
    end for
    for $i \leftarrow 0$ to $< |D|$ **do**
      if $\neg D_i$ **then**
        $P \leftarrow P \frown V_i$ ▷ P: The valid, non-dominated (nd) combs (pareto frontier)
      end if
    end for
    for $i \leftarrow 0$ to $< |I| - 1$ **do**
      $S \leftarrow \varnothing$
      for $j \leftarrow 0$ to $< |P|$ **do**
        $S \leftarrow S \frown P_j.2_i$ ▷ S: List of VIs of slots of i-th input port of j-th valid, nd comb
      end for
      $R \leftarrow R \frown T_i(S)$ ▷ $T_i(S)$ returns list of normalized values of applying T_i to S
    end for
    for $i \leftarrow 0$ to $< |P|$ **do**
      $L \leftarrow \varnothing$
      for $j \leftarrow 0$ to $< |R|$ **do**
        $L \leftarrow L \frown R_{j,i}$ ▷ L: List of normalized VIs of i-th valid, nd comb
      end for
      $v \leftarrow r(L, I_{|I|-1}.VSP_{P_i.1_{|I|-1,0}.Index}.VI)$ ▷ $P_i.1$ is a header
      $O.VSP \leftarrow O.VSP \frown (P_i.1, v)$
    end for
  end if
end function

Algorithm 14 Algorithm to check if F is dominated by S.

▷ Input F: A list of VIs of a valid combination
▷ Input S: A list of VIs of a valid combination where $S \neq F$ and $|S| = |F|$
▷ Input M: A list of boolean values where M_i indicates if the i-th value in F and S wants to be maximized or not (minimized)

```
1:  function IsDom(F, S, M)
2:      for i ← 0 to < |F| do
3:          if M_i then
4:              if F_i > S_i then
5:                  return false
6:              end if
7:          else
8:              if F_i < S_i then
9:                  return false
10:             end if
11:         end if
12:     end for
13:     return true
14: end function
```

Chapter 6

Dependency Variability Graph Systems

6.1 Composition Relationships

A DVG system can be constructed by connecting the ports of different modeling patterns described in the previous chapter in a certain way. In the following we will provide an overview about their composition relationships by listing for every pattern if it is composable with any other pattern or not. A pattern A is composable with pattern B if A can depend on B, i.e. if an input port of pattern A can be connected with an output port of pattern B. Note that we restrict our analysis here to the SPs of the modeling patterns because in general different port classes are not composable with each other. Furthermore, we treat every pattern as an independent structure which means that all input ports define an explicit DO that they want to represent. A general rule that must then always be fulfilled is that a specific input port can be connected to a specific output port only if both represent the same DO. Note, that this constraint is not always mentioned separately in the following explanations but applies in principle. Table 6.1 provides an overview of the conducted comparison. We use the following notation:

- ✓: A is always composable with B (if their DOs matches)
- ✓(VT|RT|SAMS): A is composable with B if B fulfills specific constraints regarding VT, RT or SAMS
- ✗: A is never composable with B
- ✗!: A is basically composable with B. However this composition seems unnecessary and is therefore prohibited explicitly.

6.1.1 Comparison Filter (COMF)

COMF

I_0 of COMF A is composable with another COMF B if both their input ports of class CP ($B.I_1$ and $A.I_1$) are independent of each other. It can be used to model a filtering chain regarding the same DO by applying different comparison functions. This can make sense in some situations e.g. to model a range filter for numerical values with lower and upper bound thresholds and corresponding comparison functions in A and B.

Table 6.1: Composition Relationships between the modeling patterns considering their SPs: Is pattern A composable with pattern B?

A \ B	COMF	RPRO	APRO	MAGR	TRAN	CONT	PTCC
COMF	✓(SAMS)	✗	✓(VT)	✓(VT)	✗	✗	✗
RPRO	✓	✓	✓	✓	✓	✗!	✗!
APRO	✓	✗	✓	✓(RT)	✗	✗	✗
MAGR	✓	✓	✓	✓(VT,RT)	✓	✗!	✗!
TRAN	✓(VT)	✗	✓(VT)	✓	✗	✗	✗
CONT	✗	✓(VT)	✗	✓(VT)	✓(VT)	✓	✓
PTCC	✓(VT)	✗	✓(VT)	✓(VT)	✗	✗	✗

RPRO, TRAN, CONT, PTCC

I_0 of COMF A is never composable with a RPRO, TRAN, CONT or PTCC B because $A.I_0$ expects an absolute RT but B provides a relative one.

APRO, MAGR

I_0 of a COMF A is composable with APRO or MAGR B if the VT of B is constant or active.

6.1.2 Relative Production (RPRO)

COMF, RPRO, APRO, MAGR, TRAN

An input port of a RPRO A is always composable with a COMF, RPRO, APRO, MAGR or TRAN B.

CONT, PTCC

An input port A of a RPRO is basically composable with a CONT or PTCC B. However, this composition relationship seems unnecessary. Therefore, we prohibit it explicitly.

6.1.3 Absolute Production (APRO)

COMF, APRO

An input port of a APRO A is always composable with a COMF or APRO B.

RPRO, TRAN, CONT, PTCC

An input port of a APRO A is never composable with a RPRO, TRAN, CONT or PTCC B because A expects an absolute RT but B provides a relative one.

MAGR

An input port of a APRO A is composable with a MAGR B if its RT is absolute.

6.1.4 Multiple Aggregation (MAGR)

COMF, APRO, RPRO, TRAN

An input port of a MAGR A is always composable with a COMF, APRO, RPRO or TRAN B.

MAGR

An input port of a MAGR A is composable with another MAGR B if their VT and RT matches and can be used to represent nested forks.

CONT, PTCC

An input port A of a MAGR is basically composable with a CONT or PTCC B. However, this composition relationship seems unnecessary. Therefore, we prohibit it explicitly.

6.1.5 Transformation (TRAN)

COMF

An input port of a TRAN A is composable with a COMF B if the VT of B is active.

RPRO, TRAN, CONT, PTCC

An input port of a TRAN A is never composable with a RPRO, TRAN, CONT or PTCC B because A expects an absolute RT but B provides a relative one.

APRO

An input port of a TRAN A is composable with a APRO B if the VT of B is either active or passive.

MAGR

An input port of a TRAN A is always composable with MAGR B.

6.1.6 Contradiction (CONT)

COMF, APRO

An input port $I_{0..|I-1|}$ of a CONT A is never composable with a COMF or APRO B because A expects a relative RT but B provides an absolute one.

RPRO, TRAN, MAGR

An input port $I_{0..|I-1|}$ of a CONT A is composable with a RPRO, TRAN or MAGR B if the VT of B is active.

CONT, PTCC

An input port $I_{0..|I-1|}$ of a CONT A is always composable with a CONT or PTCC B.

6.1.7 Pareto Transformation Contradiction Composite (PTCC)

COMF, APRO, MAGR

An input port $I_{0..|I-1|}$ of a PTCC A is composable with a COMF, APRO or MAGR B if the VT of B is active.

RPRO, TRAN, CONT, PTCC

An input port $I_{0..|I-1|}$ of a PTCC A is never composable with a RPRO, TRAN, CONT or PTCC B because A expects an absolute RT but B provides an absolute one.

6.2 Further System Aspects

6.2.1 Complete and Partial DVGs

We distinguish between *complete* and *partial* DVGs. A complete DVG has exactly 1 root block pattern that is either of type CONT or PTCC. A root block pattern is characterized by an output port that is not connected to any input port of any other block pattern in the DVG. With that constraint, we declare conflict resolution to the ultimate and crucial goal of a complete DVG. Since, both CONT and PTCC implement the extended PROD, it is guaranteed that their output ports will always contain all the relevant and valid combinations. This constraint is also in line with table 6.1 where we show that CONT and PTCC are the two block patterns with the fewest possible patterns that can depend on them: Both CONT and PTCC can be referenced only by a CONT (see columns of CONT and PTCC) which implies that the root block pattern must be CONT or PTCC and if the latter is the case the DVG can not contain a CONT at all. In contrast, a partial DVG has no special additional constraints and must only conform to the composition relationships described in the previous section.

6.2.2 Source Spaces

We use the term *Configuration Source Space* (CONSS) to refer to a set of specific entities in a DVG that indicate the real source of active influence. The following entities represent CONSSs:

- Every input port of a block pattern with an active VT that has no connection to an output port.
- Every MAGR with an active VT of the fork.

In the same way, the term *State Source Space* (STASS) refers to a set of specific entities in a DVG that indicate the real source of passive influence. The following entities represent STASSs:

- Every input port of a block pattern with a passive VT that has no connection to an output port.
- Every MAGR with a passive VT of the fork.

6.2.3 Resolution Views

In section 4.3.2, we described the possible transition from a passive port to a constant port between the modeling state and the pre-resolution state. This was argumented by the kind of result analysis or view we want to have when performing the resolution at different points in time. In general, we

divide between a *full view* and a *constant view* on the resolution and this is related to the treatment of the STASS set in the pre-resolution state.

A full view means a passive port in the STASS remains passive and we resolve the associated block for all the possible outcomes of that port (its whole VSP). Likewise we resolve all possible branches of the passive forks of the MAGR patterns in the STASS. In contrast, constant view means a passive port in the STASS reduces to constant and we resolve the associated block for exactly one possible outcome of that port (one specific slot from its VSP). Likewise we resolve exactly one specific branch of the passive forks of the MAGR patterns in the STASS.

6.2.4 Determining and Interpreting the VSS

We already described the SR of DVG systems at the basic block level in chapter 4.4.3. This procedure can now be applied to more concrete DVG systems to determine the VSS of its modeled VPS. In such systems each block is one of the concrete modeling patterns with its BIVM as described in chapter 5.

In a complete DVG, the root block pattern R is either of type CONT or PTCC both providing an OVSP with achievable utility values for all available variants and possible states after the SR is executed. The usefulness of utility values in their range $[0, 1]$ is proportional to their value, i.e. 0 is least useful and 1 is most useful. Hence, identifying the slot $\hat{S}$ with the highest VI of the OVSP of the root block pattern in a complete DVG (equation 6.1) represents the most useful combination of variant bindings (a specific binding of the CONSS set) for the most favorable combination of states that can occur (a specific binding of the STASS set). The DVG is satisfiable if there is at least one slot in the OVSP of R in the post-resolution state, i.e. if the VT is not empty (equation 6.2). In contrast, the resolution of a complete DVG is unsatisfiable if the root block pattern $R \in \{CONT, PTCC\}$ has an output port with an empty VT in its post-resolution state ($R.O.VT''' = E$). This means that there is not at least one combination of variant bindings (a specific binding of the CONSS set) that satisfies the defined constraints via the COMF patterns in a DVG system even not for the most favorable combination of states that can occur (a specific binding of the STASS set). Hence, COMF patterns with their specific constraint parameterizations are always important and prioritized requirements in a DVG system and thus an important mean to reduce the solution space (VSS).

$$\hat{S} = R.O.VSP_{\left(\underset{0 \leqq i < R.O.|VSP|}{\arg\max} R.O.VSP_i.VI\right)} \tag{6.1}$$

$$\text{Satisfiable} = \begin{cases} \text{True if } R.O.VT''' \neq E \\ \text{False else} \end{cases} \tag{6.2}$$

The meaning of *satisfiable* in equation 6.2 is clear in cases where the STASS set is empty for a DVG system or if we want to consider the constant view on the STASS in the pre-resolution state:

1. The DVG is *always satisfiable* for the applied COMF patterns and their specific constraint parameterizations (for a specific binding of the STASS).

2. The DVG is *never satisfiable* for the applied COMF patterns and their specific constraint parameterizations (for a specific binding of the STASS).

However, if a STASS set exists in a DVG system and we want to consider the full view on the STASS in the pre-resolution state, equation 6.2 has to be interpreted differently:

1. The DVG system is at least *partially satisfiable* for the applied COMF patterns and their specific constraint parameterizations, i.e. satisfiable for at least one favorable state combination that may occur based on the whole set of eventualities that can occur
2. The DVG system is never satisfiable for the applied COMF patterns and their specific constraint parameterizations, i.e. it is for all eventualities that can occur not satisfiable.

In that case, we can not conclude based on equation 6.2 if a DVG system is always satisfiable which means satisfiable for all eventualities that can potentially occur. This distinction requires more detailed investigations that are described in the following section.

The Lookup Principle

Lookup in a DVG system is possible if the abstract form described in equation 6.3 is given where A represents a combination of value bindings of the CONSS set, LUT a Lookup Table and I_P a set of indices of a combination of value bindings of the STASS set occurring at a specific moment in time. We can derive this form based on the information given in the $R.O.VSP$ of a DVG system in the post-resolution state. This form has 2 substantial advantages:

- We get more insights, i.e. are able to conclude if the DVG system is always satisfiable or only partially satisfiable. In the latter case we can also check for which combinations this applies and for which not.
- Significant time reduction in case of repeated online usage at specific points in time (different constant views) by retrieving the current results from a lookup table based on a complete pre-calculation for all eventualities performed at design-time. Thus we do not need to execute the process always again at run-time for all the frequent value changes in the passive states.

$$A = LUT_{I_P} \tag{6.3}$$

Let us suppose that P is the STASS set of a DVG, then the LUT is $|P|$-dimensional. Each dimension is associated with one element of this set and the size $|P_i|$ of the i-th dimension is either given by the size of the respective VSP or the number of branches of the fork depending on if P_i is a passive port or a MAGR representing a passive fork. For every possible index combination of P we collect the corresponding slots from $R.O.VSP$ that refer to this index combination, determine the slot with the maximum value and store it in the LUT at this specific index combination. $\hat{S}_{i_0 \in [0..|P_0|[,..,i_{|P|-1} \in [0..|P_{|P|-1}|[}$ is then defined as the most appropriate slot (with respect to value bindings of the CONSS set) for a value combination of P represented by the indices $i_0, .., i_{|P|-1}$ (equation 6.4). Equation 6.5 then states if the DVG is always satisfiable (AS) or not where $\varnothing$ means that an entry in the lookup table is undefined. Equation 6.6 illustrates the relationship of the lookup table with respect to $\hat{S}$ as defined in equation 6.1. Hence, if a STASS set exists in a DVG, $\hat{S}$ can be interpreted as the most useful configuration binding of the CONSS set in the most favorable state combination that can occur in the STASS set.

$$\hat{S}_{i_0 \in [0..|P_0|[,..,i_{|P|-1} \in [0..|P_{|P|-1}|[} = LUT_{i_0,..,i_{|P|-1}} \tag{6.4}$$

$$\text{AS} = \begin{cases} \text{False if } \forall 0 \leqq i_0 < |P_0|, .., \forall 0 \leqq i_{|P|-1} < |P_{|P|-1}| : \exists i_0, .., i_{|P|-1} LUT_{i_0,..,i_{|P|-1}} = \varnothing \\ \text{True else} \end{cases} \tag{6.5}$$

$$\begin{aligned}\hat{S} &= \underset{0 \leqq i_0 < |P_0|,..,0 \leqq i_{|P|-1} < |P_{|P|-1}|}{\arg\max} LUT_{i_0,..,i_{|P|-1}} \\ &= R.O.VSP_{\left(\underset{0 \leqq i < R.O.|VSP|}{\arg\max} R.O.VSP_i.VI\right)}\end{aligned} \tag{6.6}$$

6.2.5 Resolution Trade-Off Strategies

In this section we consider the general conflict between *completeness* and *time* of the global SR. A DVG system is resolved completely if all possible and valid combinations modeled in a DVG system were considered. Applying the standard SR and the BIVMs of the modeling patterns as described so far always ensures completeness. Note, that the interpretation of completeness always depends on the discretization accuracy of the different VSPs defined in a DVG system. If completeness is given, $\hat{S}$ and $\hat{S}_{i_0 \in [0..|P_0|[,..,i_{|P|-1} \in [0..|P_{|P|-1}|[}$ represent the *global maximum utility* that can be achieved for a specific discretization accuracy and a specific constant view. However, completeness always comes with high costs in time. Hence, in the following we present two possible approaches that can reduce the time.

Iterative Random Sampling with Local Upper Bounds

Instead of resolving the DVG system one time completely, this strategy resolves the DVG system $n \geqq 1$ times but considers only a partial set of the total combinations in each of these n resolution steps. In order to accomplish that, a local upper bound $u \in \mathbb{N}$ is introduced that defines the total number of slots per input port that are processed and combined in one block. This is mainly relevant for all the patterns that are specializations of PROD/EPROD. For $|I|$ input ports the number of local combinations $|C|$ is guaranteed to be $|C| \leqq u^{|I|}$ instead of $|C| = \prod_{0 \leqq i < |I|} I_i.|VSP|$. In the case that $\forall 0 \leqq i < |I| : I_i.|VSP| \leqq u$, the strategy has no effect and leads to the same result as the standard procedure. The concrete selection of the u slots from the different input ports is then determined by random. Algorithm 15 describes how the BIVM of PROD can be very easily extended to implement this principle.

The advantage of this strategy is that it can save a lot of time if we are only interested in determining a satisfiable solution. However, it is not guaranteed that the found solution will represent the global maximum utility. We can further consider different subvariants of this resolution strategy:

- The resolution procedure is aborted after step $i \in [0..n-1]$ if at least one satisfiable/always satisfiable solution was found in step i. This is the fastest variant but has the lowest probability that the solution with its local maximum utility approaches the global maximum utility.
- All n resolution steps are executed. Hence, with the parameter n we can impact the trade-off between completeness/global maximum utility and time according to our needs.

Furthermore we can differ between stateless and stateful resolution steps. A stateless resolution step is in contrast to a stateful resolution step fully unaware about the considered combinations of the previous resolution steps. While this is easier to implement, there is always a possibility of combination redundancy between the different steps - an effect that is not wanted. In the case of stateful resolution steps we can theoretically prevent combination redundancy because we always know which combinations were already considered and which not. However, it is not obvious how to sample a redundancy-free combination based on that knowledge combined with the general principle of randomness and how much time overhead this creates.

Note, that with respect to a full view we are typically interested in always satisfiable solutions. Therefore it makes less sense to define upper bounds for entities with passive VTs and should be rather limited to entities with active VTs.

Algorithm 15 BIVM of the PROD pattern with a local upper bound u and random sampling. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports
▷ Input $f(\cdot)$: A specific value composition operator
▷ Input u: A natural number representing the upper bound of combinations to consider
▷ Output $O.VSP$: The VSP of the output port O
▷ RANDOMGENERATOR(n, a, b): Returns a list of n unique values in the range $[a, b]$

1: **function** PROD$(I, f(\cdot), u)$
2: $R \leftarrow \varnothing$
3: **for** $i \leftarrow 0$ to $< |I|$ **do**
4: $R \leftarrow R \frown$ RANDOMGENERATOR$(\min(I_i.|VSP|, u) - 1, 0, I_i.|VSP| - 1)$
5: **end for**
6: $T \leftarrow \begin{pmatrix} R_{0,0} & R_{0,1} & \cdots & |R_{0,\min(I_0.|VSP|,u)-1}| \\ R_{1,0} & R_{1,1} & \cdots & |R_{1,\min(I_1.|VSP|,u)-1}| \\ \vdots & \vdots & \ddots & \vdots \\ R_{|R|-1,0} & R_{|R|-1,1} & \cdots & |R_{|R|-1,\min(I_{|I|-1}.|VSP|,u)-1}| \end{pmatrix}$
7: $CP : \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,|I|-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,|I|-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{|CP|-1,0} & c_{|CP|-1,1} & \cdots & c_{|CP|-1,|I|-1} \end{pmatrix} \leftarrow \bigotimes_{0 \leqq i < |I|} T_i$
8: **for** $i \leftarrow 0$ to $< |CP|$ **do**
9: $H \leftarrow \varnothing$
10: $V \leftarrow \varnothing$
11: **for** $j \leftarrow 0$ to $< |CP_i|$ **do**
12: $H \leftarrow H \frown$ CREATEABSOLUTEHEADERROW$(I_j.N, CP_{i,j}, I_j.VSP_{CP_{i,j}}.H)$
13: $V \leftarrow V \frown I_j.VSP_{CP_{i,j}}.VI$
14: **end for**
15: $O.VSP \leftarrow O.VSP \frown (H, f(V))$
16: **end for**
17: **end function**

Equivalence Filtering

Depending on its implementation equivalence filtering is either the process of removing slots with duplicate VIs from its VSP or the general prevention of adding such duplicates. This step can be advisable in certain situations because it can reduce the time significantly while not impacting the completeness at all. The reason for this is that local VI duplicates will always add more combinations to consider in a DVG system but in certain situations it is excluded that these local VI duplicates will lead to different global VIs, i.e. they do not impact the global SR at all. More concretely: Duplicate VIs from the OVSP of a pattern that is based on PROD or EPROD can not differ globally if none of

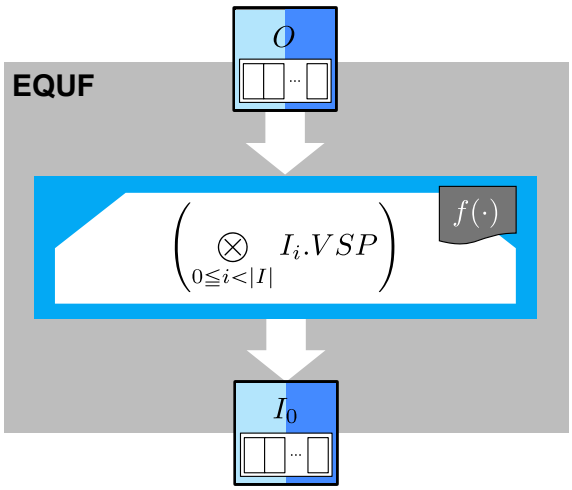


Figure 6.1: The EQUF pattern. Relative RTs for the input port and the output port are also possible.

its input ports depend (directly or indirectly) on an output port of a MAGR that has more than one AGGR. In that case it is allowed to filter duplicates if time wants to be optimized. The disadvantage is that we lose the header information of these filtered combinations in R because their VIs are then represented by exactly one representative. It depends on the particular situation and application if this is a problem or not.

Equivalence checks depend on the concrete DO. For example, if real numbers are involved, we can define an accuracy threshold that defines how far two VIs are allowed to differ from each other and still being considered as equivalent. The higher the threshold the higher the probability that duplicates are filtered to reduce the time but at the cost of accuracy and vice versa. Note that it depends heavily on the concrete problem if duplicates for a specific threshold are likely to occur or not. If the chance to detect duplicates is very low, the additional time overhead introduced by equivalence filtering could even increase the overall time in the end.

Because equivalence filtering is only allowed in special situations and not always desired, we offer the EQUF as an independent, additional pattern. It can be treated as a specialization of PROD and its RT and VT constraints are described in equation 6.7 and 6.8. Figure 6.1 and algorithm 16 show the relevant details of EQUF.

$$\begin{aligned} I_0.RT &\in \{A, R\} \\ O.RT &= I_0.RT \end{aligned} \tag{6.7}$$

$$I_0.VT' \in \{A, P\} \tag{6.8}$$

6.3 Comparison With Other Structures

In the following we compare the structure of DVG systems with the structures of two known methods for modeling decision problems (value trees and decision trees). We show that by means of a suitable composition of specific patterns, both the structure of classical value trees and the structure of classical decision trees can be reproduced. The DVG approach thus corresponds to a generalization of these structures and also allows to combine these structures. In the third part of this work we will then

Algorithm 16 BIVM of the EQUF pattern. See table 4.1 for some hints about the used notation.

▷ Input I: A list of input ports
▷ Input $f(\cdot)$: A special value composition operator defining an equivalence check function
▷ Output $O.VSP$: The VSP of the output port O

```
1: function EQUF(I, f(·))
2:     for i ← 0 to < I_0.|VSP| do
3:         if ¬CONTAINS(O.VSP, I_0.VSP_i.VI, f(·)) then
4:             H ← CREATEABSOLUTEHEADERROW(I_0.N, i, I_0.VSP_i.H)
5:             O.VSP ← O.VSP ⌢ (H, I_0.VSP_i.VI)
6:         end if
7:     end for
8: end function
```

see based on concrete examples of real building blocks, that these more complex and generalized structures are necessarily required in order to model adequate VPSs of these building blocks.

6.3.1 Modeling Value Trees

In this section we show that value trees can be formalized as DVG systems. A value tree has certain simplifying assumptions that reduce the overall complexity of the problem space that can be modeled.

Meta Model

Figure 6.2 shows our meta model of a value tree: A *ValueTree* consists of one *GoalLayer* with one *Goal* which is an instance of a *Block*. The *Goal* depends on at least one *CriterionBlock* that is either a *Criterion* or an *Attribute* which are placed in the first *CriterionLayer*. A *Criterion* in the first *CriterionLayer* depends on other *Criteria* or *Attributes* of the second *CriterionLayer* and so on for any desired number of *CriterionLayers*. An *Attribute* depends on at least two *Alternatives* (a *Block* instance) that must be contained in exactly one *Alternative Layer*. There is a *Value* (numerical representation *value*) for every combination of the *Block* instances *Attribute* and *Alternative*. A *Goal* and a *Criterion* must have a *CriterionOperator* that implements a specific rating method, e.g. *Weighted Sum Model* (*WSM*). In contrast, an *Attribute* must have an *AttributeOperator* which is used to normalize the *Values* from the different *Alternatives*.

Example Figure 6.3 shows an abstract example model of a value tree with 2 *CriterionLayers*, 5 *Attributes* and 4 *Alternatives* resulting in 5·4 = 20 *Value* combinations.

Analysis The following mapping exists between a value tree and a DVG system:

- *GoalLayer* (Level 0): A root block of type CONT
- *CriterionLayer* (Level 1 to n-2): Any number of CONTs
- *CriterionLayer* (Level n-1): The penultimate layer contains of m TRAN patterns calculating normalized values for m *Attributes*
- *AlternativeLayer* (Level n): The leaf block is of type MAGR (active fork VT) with m AGGRs representing m *Attributes* each with k input ports indicating the overall number of *Alternatives*

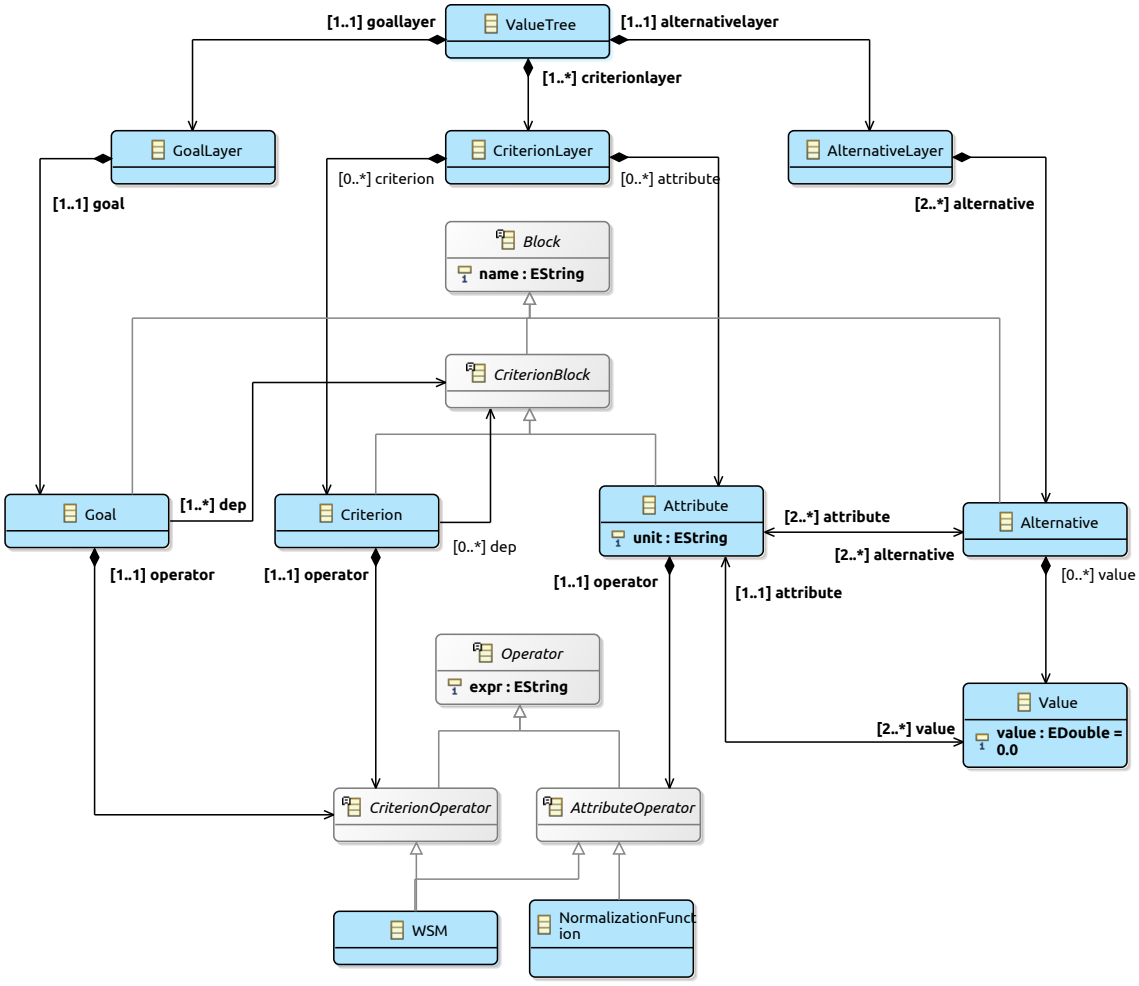


Figure 6.2: The meta model of a value tree.

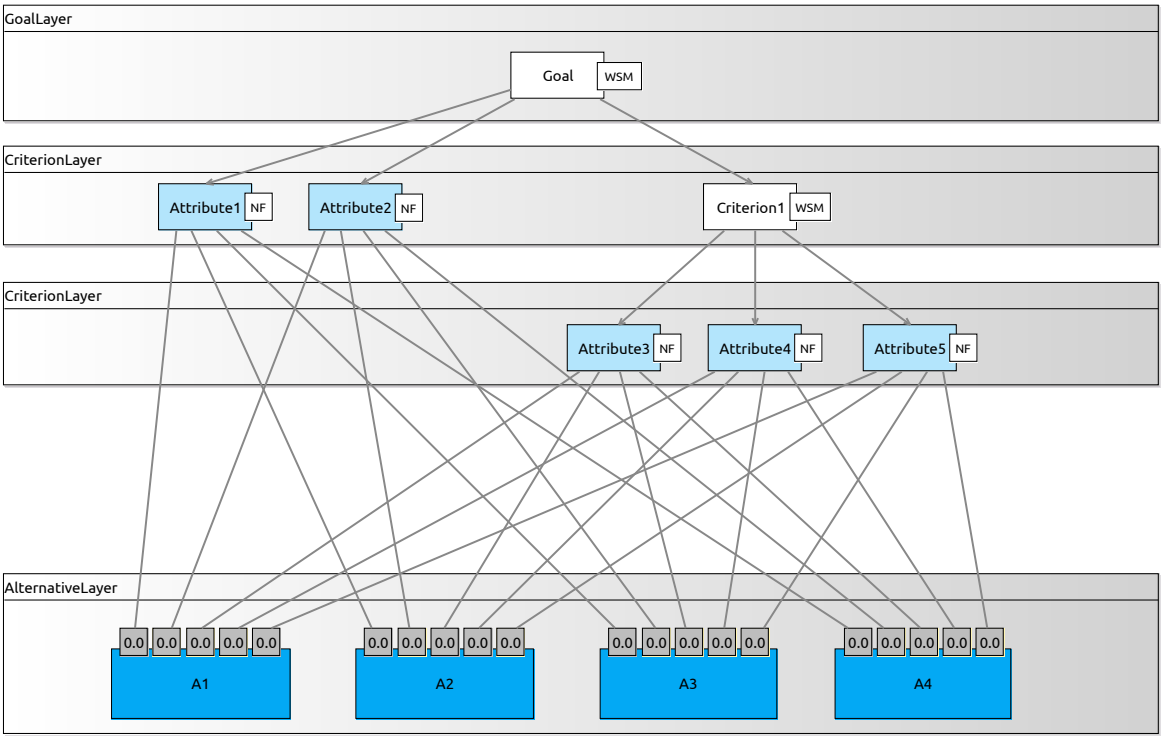


Figure 6.3: An abstract example model of a value tree based on the meta model in figure 6.2.

Figure 6.4 shows the corresponding DVG system for the value tree example model from figure 6.3. As we can see, the problem space that can be described by a standard value tree is quite simple because there is only one element in the CONSS (one active MAGR representing the *AlternativeLayer*).

6.3.2 Modeling Decision Trees

Figure 6.5 illustrates an example of a decision tree from [Cle96]. In figure 6.6 we show the same example modeled by a special partial DVG system (a composition of MAGR patterns).

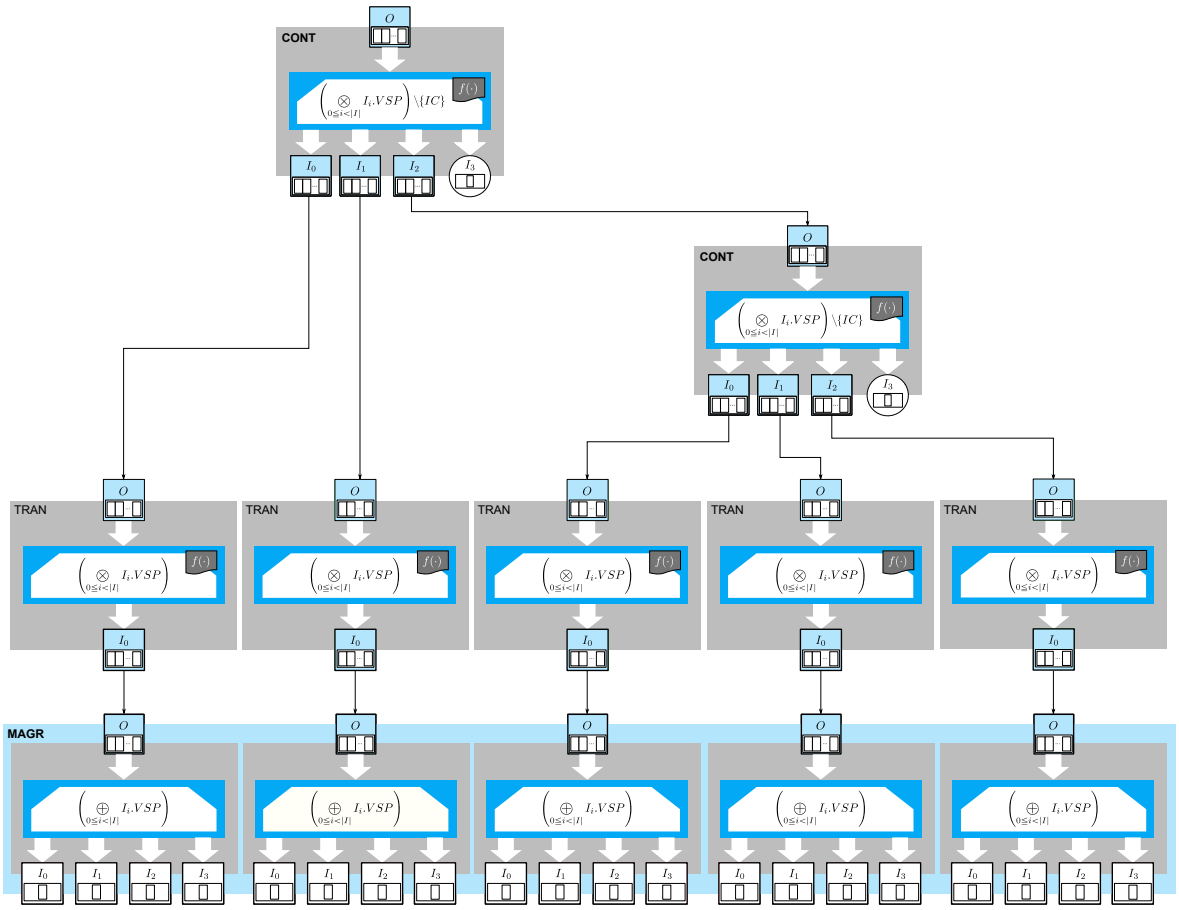


Figure 6.4: DVG system representation for the value tree example model in figure 6.3.

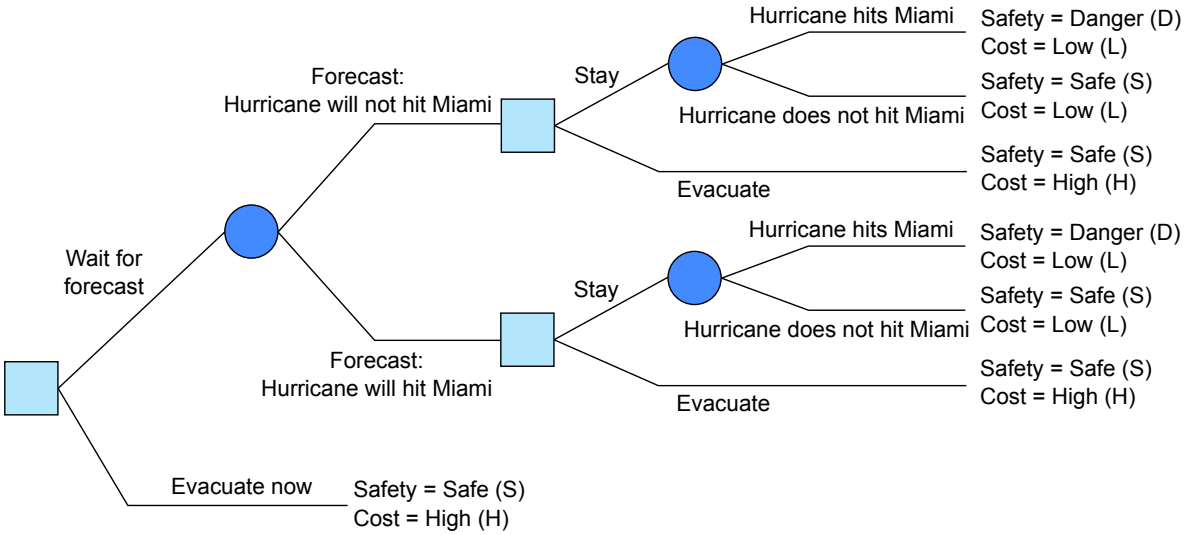


Figure 6.5: Own illustration of a decision tree example from [Cle96].

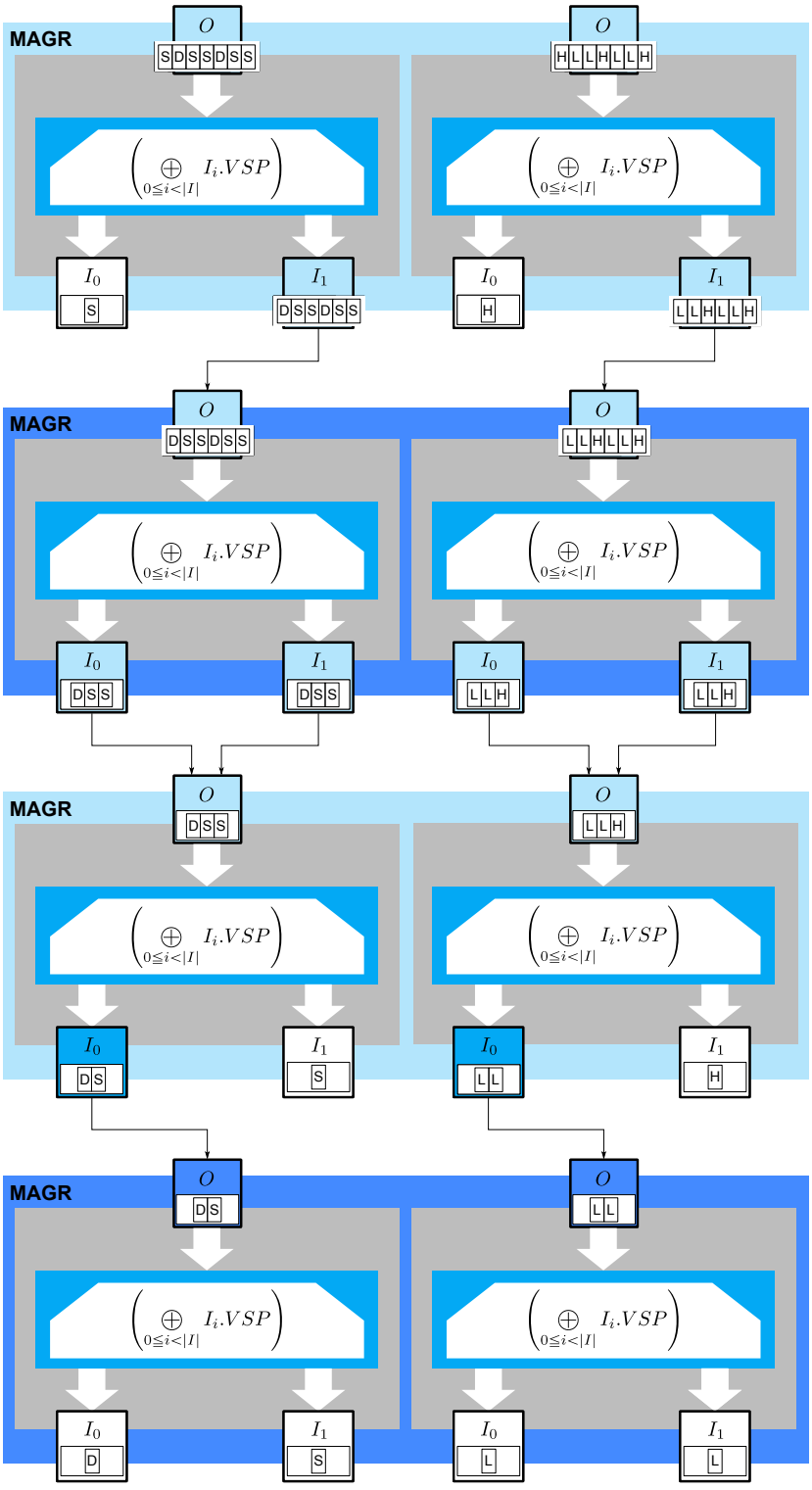


Figure 6.6: The decision tree example from figure 6.5 modeled as a special partial DVG system (a composition of MAGR patterns).

Chapter 7

Integration/Implementation

In this chapter we describe an implementation for a general method of building block modeling using MDSD where DVG system modeling is an important part of it. A building block is a central entity in the context of a robotic software ecosystem and can be considered in a general sense as a specific robotic function of any level of abstraction. A building block in a robotics software ecosystem can be provided by someone and can re-used by others to compose it with other building blocks to new robotic systems. A building block in a robotics software ecosystem should define what it is so that an inspector can understand if it fits to its need. A building block usually comes with different forms of internal variability heavily impacting the definition of what it is. Therefore, a building block can be basically treated as a variability building block and should therefore explicate (i) what variability entities it consists and (ii) how they relate. Both aspects are defined in the corresponding building block model of a building block.

In the next sections we propose meta models divided into different parts that serve the common purpose of defining the blueprint for building block modeling. The meta models are described with *EMF Ecore* [SBPM08]. It is a special class diagram that allows to define meta models within *Eclipse*. For each meta model we either designed a textual or a graphical DSL. Textual DSLs are modeled with *Xtext* [Fou21c] while *Sirius* [Fou21a] is used for the graphical DSLs. The code generator is implemented with *Xtend* [Fou21b] and is able to derive a solver (a java program) for DVG system models which are an essential part of building block models. Figure 7.1 provides an overview of the implementation.

Note that all names which refer to entities of a meta model are always shown in italics in the following text.

7.1 Data Object Definition and VSP Initialization

In the context of a robotics software ecosystem we have specific needs for defining the underlying data structure of the VSP of a port. The meta model in figure 7.2 describes how DOs are defined. This is strongly inspired by the concept of a *Communication Object* [SLLS16, Sta18] used in SmartSoft to define the data structure of a *Service Definition*.

A *DataObjectRepo* is the container that stores a collection of *DataObjectDef*s. We distinguish between *UnboundedDataObjectDef* and *BoundedDataObjectDef*. The latter directly initializes a VSP for the DO to explicate general domain restrictions with respect to its VIs. The meta model for initializing VSPs is shown in figure 7.3. Such a repository of reusable DOs can be defined by domain experts (tier 2) in a robotics software ecosystem to establish a common vocabulary of understanding

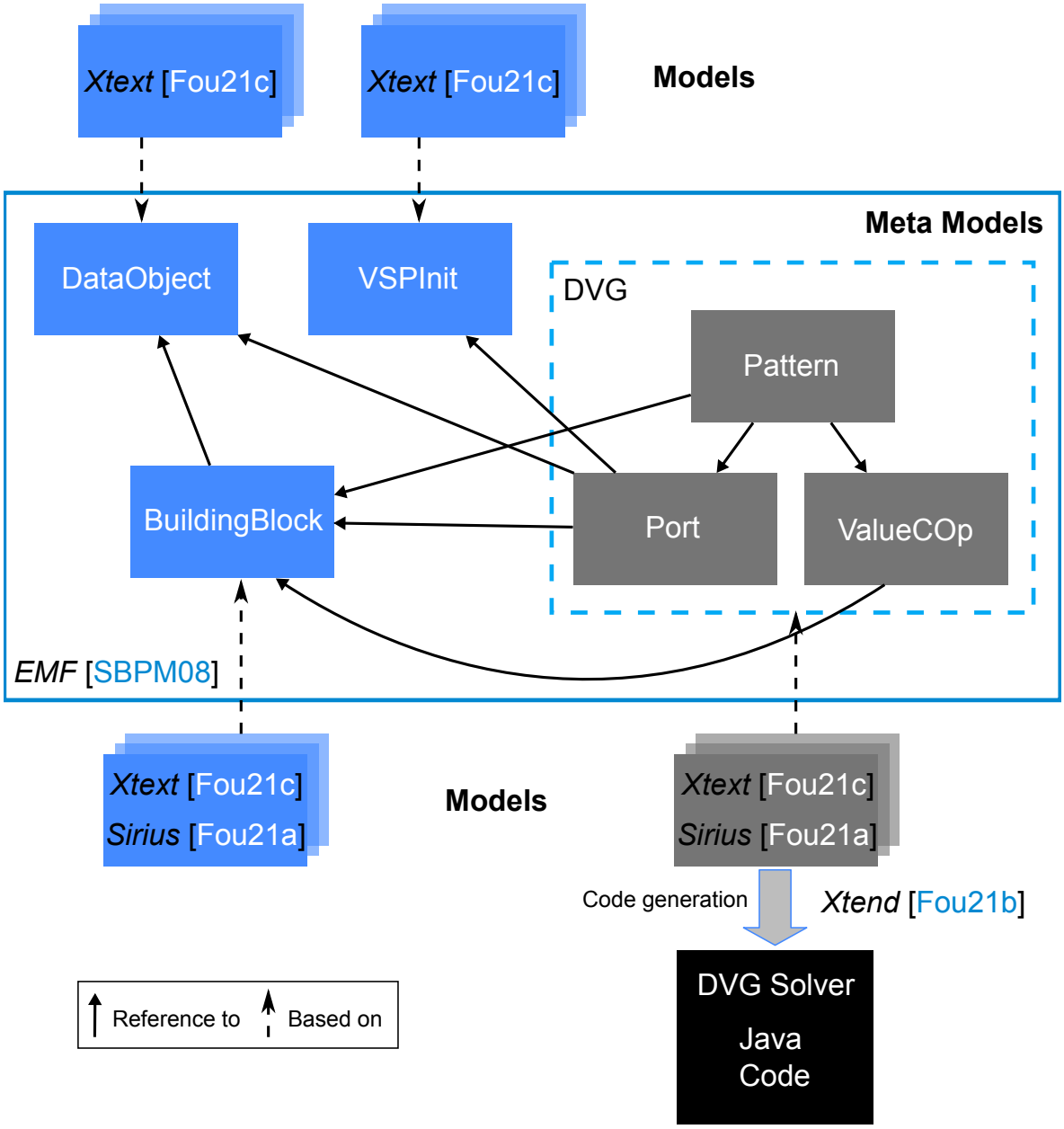


Figure 7.1: Overview of the MDSD/MDE approach of this work for modeling and resolving building blocks.

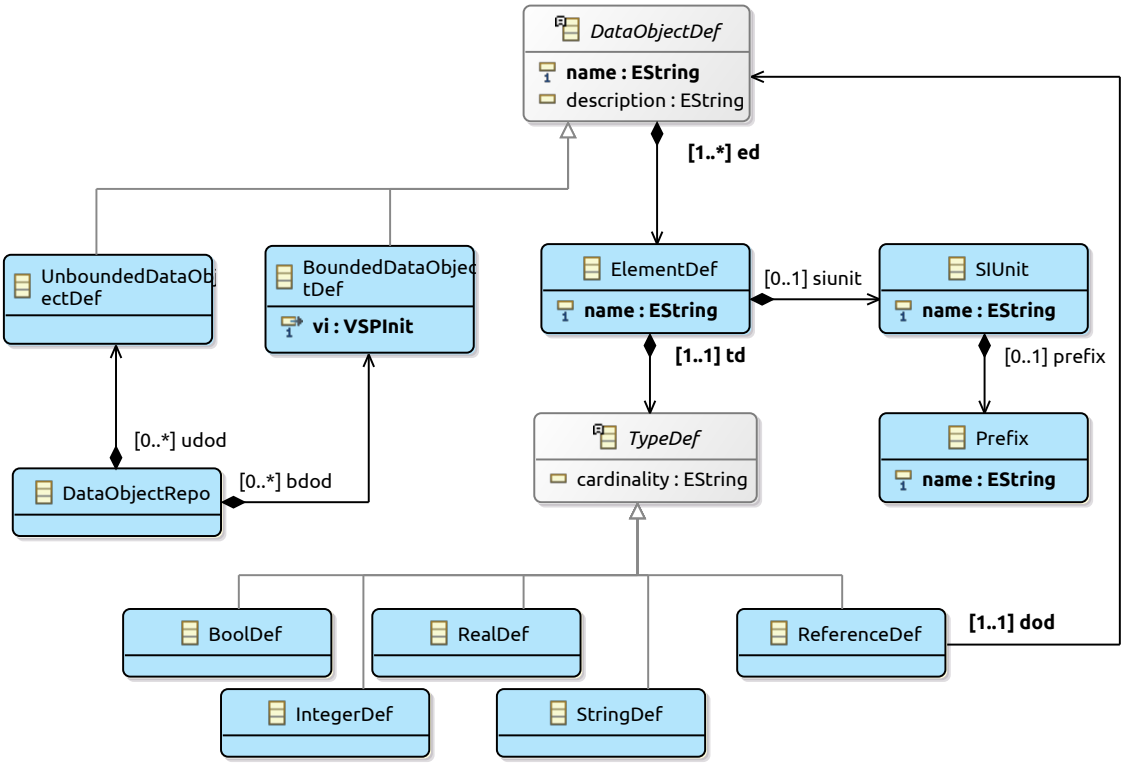


Figure 7.2: The *DataObjectDef* meta model.

between the providers and users at tier 3.

7.2 Building Block

The *BuildingBlockDescription* meta model shown in figure 7.4 deals with (i) describing what variability entities are present and thus characteristic for a building block. We can divide the meta model in 3 parts:

- *VariabilityEntitySet*
- *Port*
- *Decomposition*

Elements of all 3 parts contain variability entities which means that these elements derive from the abstract class *VariabilityEntity*. A *VariabilityEntity* is identified by a *name* and must either reference an existing *DataObjectDef* or define one on its own (XOR constraint between *dor* and *doc*). In the following, we will explain the 3 parts of the meta model in more detail.

7.2.1 VariabilityEntitySet

A *BuildingBlock* can have a *VariabilityEntitySet* that can contain a *PropertySet*, a *ParameterSet*, a *ContextSet* and/or a *PortElementSet* which in turn are collections of the *VariabilityEntity* instances

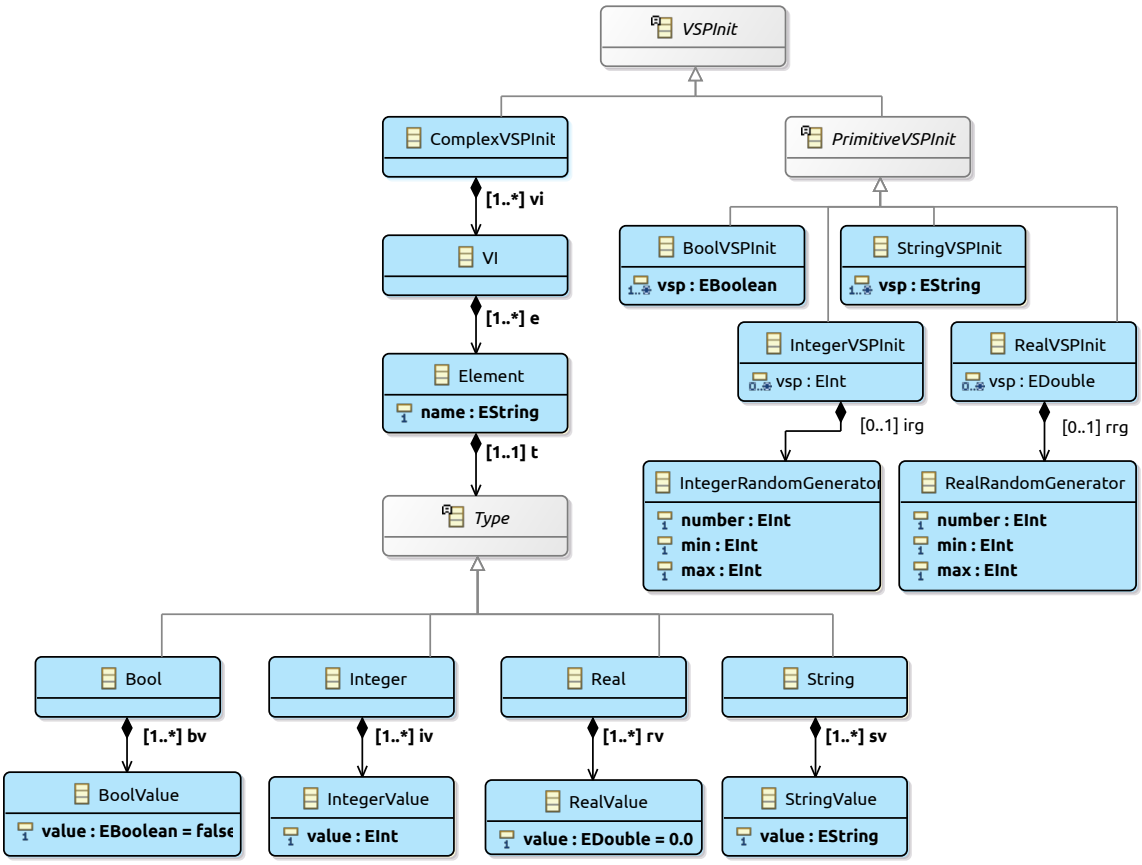


Figure 7.3: The *VSPInit* meta model.

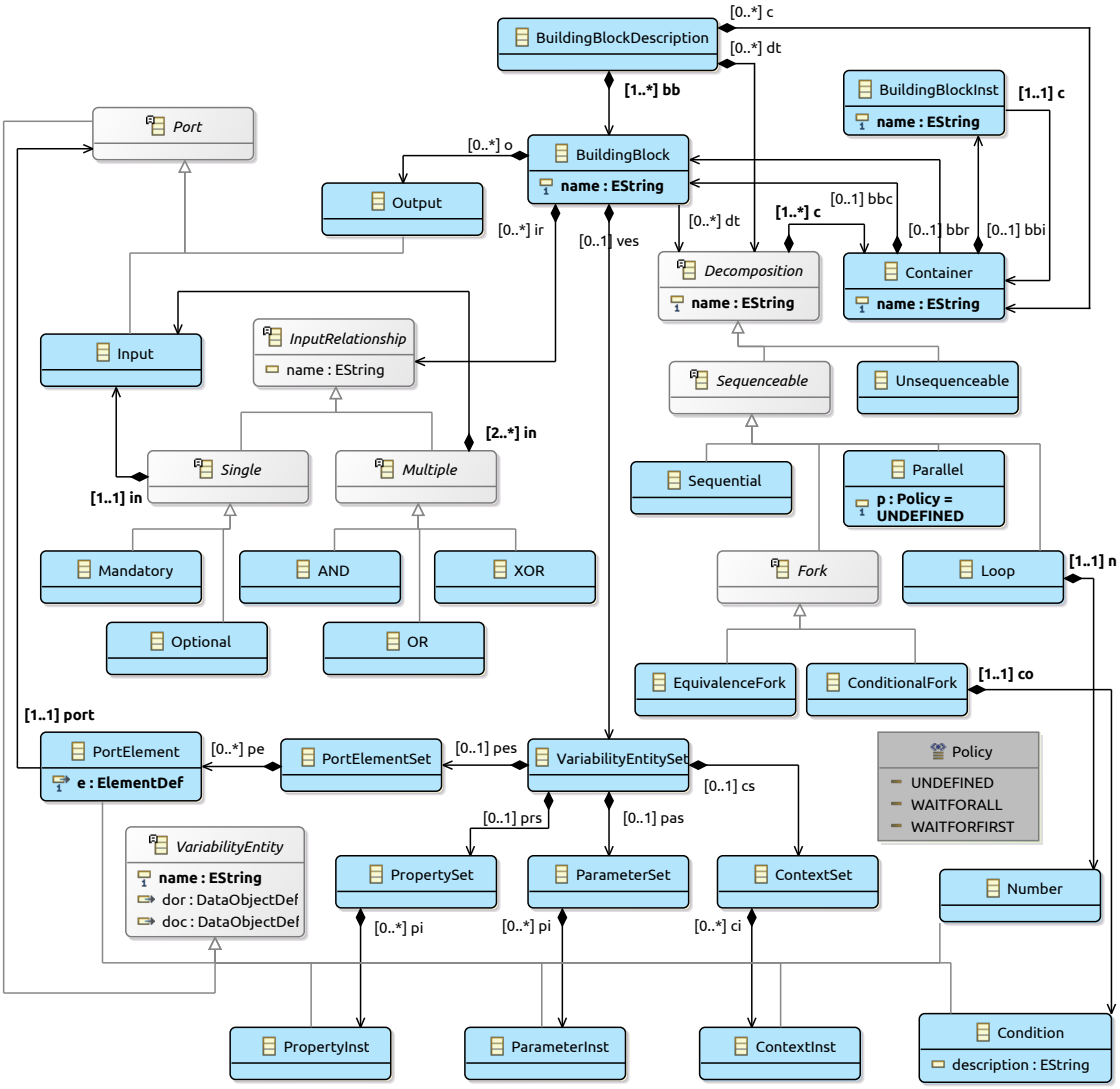


Figure 7.4: The *BuildingBlockDescription* meta model.

PropertyInst, *ParameterInst*, *ContextInst* and *PortElementInst*. A *Property* represents an abstract characteristic of the provided function of a *BuildingBlock* while a *Parameter* maps to a concrete configuration point in the system awaiting a specific variant binding. The type *Context* can be used to define a relevant *VariabilityEntity* of the *BuildingBlock* that does not directly describe a *Property* of the provided function but a relevant, external influencing factor. A *PortElement* is defined to declare a specific *ElementDef* of the corresponding *DataObjectDef* of a specific *Port* of the *BuildingBlock* as a *VariabilityEntity*.

7.2.2 Port

The *Input* and *Output* of a *BuildingBlock* are *Port*s which define what information the *BuildingBlock* receives and processes and what it provides to the outside based on the processed *Input*s. A *Port* is basically a *VariabilityEntity* and therefore has a *DataObjectDef*. A *BuildingBlock* can have different numbers of *Input*s and *Output*s. All *Input*s must be part of a *InputRelationship* and the assignment of *Input*s to *InputRelationship*s allows to get a basic understanding about the internal variability of the *BuildingBlock* related to the overall combinations of processing an *Input* or not (figure 7.5). This also serves as a first indicator for the function granularity and/or degree of flexibility of the *BuildingBlock*. We use the following instances of *InputRelationship*s accordingly to the typical relationships used in feature models:

- *Mandatory (I)*: This single *Input* is required and must be processed in any case to provide the *Output*
- *Optional (I)*: This single *Input* can but does not have to be processed necessarily in order to provide the *Output*
- *AND ($I_1, .., I_n$)*: All *Input*s are required and must be processed in any case to provide the *Output*
- *OR ($I_1, .., I_n$)*: At least 1 *Input* is required and must be processed in any case to provide the *Output*
- *XOR ($I_1, .., I_n$)*: Exactly 1 *Input* is required and must be processed in any case to provide the *Output*

7.2.3 Decomposition

Decomposition is the process of refining a *BuildingBlock* by a special *Decomposition* type to *BuildingBlocks* of a more detailed abstraction level. Hence, if a *BuildingBlock* can not be described accurately enough at its current abstraction level, *Decomposition* allows to expand any desired details of the internals. We distinguish between the two main *Decomposition* types (i) *Sequenceable* and (ii) *Unsequenceable*. The former one is further divided into the following sub types:

- *Sequential ($B_1, .., B_n$)*: The building blocks $B_1, .., B_n$ are executed one after the other in the given sequence
- *Parallel ($B_1, .., B_n$)*: The building blocks $B_1, .., B_n$ are executed simultaneous
- *Loop ($B_1, .., B_n, m$)*: The building blocks $B_1, .., B_n$ are executed m times in the given sequence

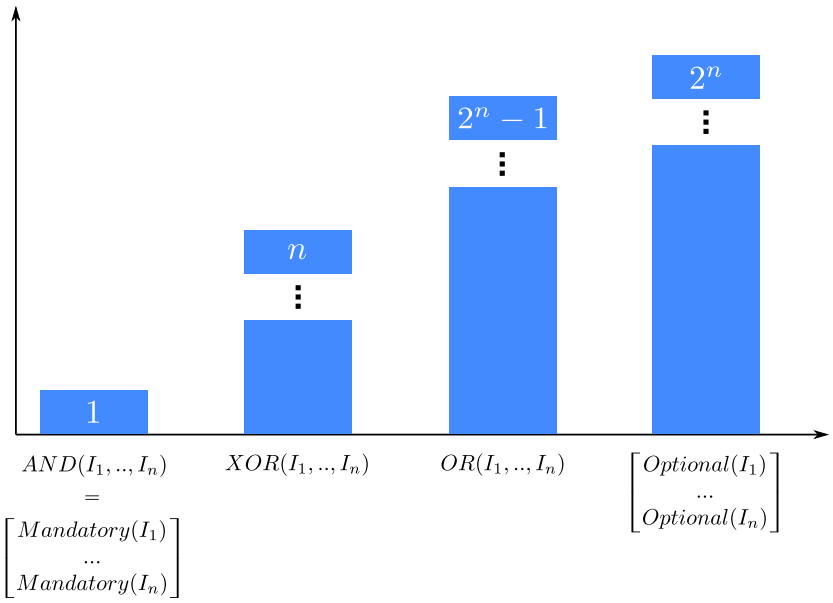


Figure 7.5: A schematic illustration about the overall number of variants related to the processing of *Input*s depending on different *InputRelationship*s.

- *ConditionalFork* $(B_1, .., B_n)$: The execution of B_i with $i \in [1, n]$ that fulfills a given condition
- *EquivalenceFork* $(B_1, .., B_n)$: The execution of B_i with $i \in [1, n]$ that is preferred over the others

The *Number* n of a *Loop* represents a potential VSP over an integer in the range $[0, +\infty]$ and is therefore treated as a *VariabilityEntity*. The same applies for the *Condition* of a *ConditionalFork*: Different VIs of the VSP defined over the *DataObjectDef* of a *Condition* will trigger the execution of a specific *BuildingBlock* from the corresponding *ConditionalFork*. The condition-based execution of a *BuildingBlock* is always mandatory for realizing the desired *Output* of the parent *BuildingBlock* and will lead to undefined states otherwise. This is one difference to the *EquivalenceFork*, where the term equivalence means achieving equivalent results exclusively with respect to the desired *Output* of the parent *BuildingBlock* for all possible *BuildingBock* variants of the *EquivalenceFork*. However, this equivalence will not always apply with respect to a *Property*. Hence, a *EquivalenceFork* is a purposefully added variation point to increase the diversity of ways to achieve the parent *Output* with varying *Property* results (increase of flexibility with respect to non-functional properties).

The latter type *Unsequenceable* $(B_1, .., B_n)$ expresses that the given sequence $B_1, ..B_n$ can not be enumerated at design-time because of high variability dependencies. This means that in theory the actual sequence could be any combination of the *Sequenceable* sub types. Using *Unsequenceable* is different to not decomposing the *BuildingBlock* at all because we are still able to further decompose the corresponding *BuildingBlock*s and to explicate their internals. Therefore, *Unsequenceable* indicates that the concerned *BuildingBlock*s, the exact sequence of them and all the modeled details of their childs are indeed relevant with respect to the desired decision-making but that the explication of all variants with respect to the sequence order is simply not possible by hand in this model.

The *Container* class is used to contain a *BuildingBlock* within a *Decomposition* structure. There are 3 exclusive semantics of containing a *BuildingBlock*:

- The *Container* contains a *BuildingBlock* (composition relationship *bbc*)

- The *Container* references a *BuildingBlock* (reference relationship *bbr*)
- The *Container* contains a *BuildingBlockInst* (composition relationship *bbi*)

The first one indicates that the corresponding *BuildingBlock* is an own implementation of the provider and is therefore described (and decomposed) directly in this model. In contrast, the second one refers to an already existing *BuildingBlock* of another model that the provider wants to reuse at this place because it matches to the required function there. Hence, the *Container* refers to the corresponding *BuildingBlock* described in another model and makes this structure available in the referencing model. The last variant (containing a *BuildingBlockInst*) is used to refer to an already existing *Container* in the current model in order to prevent the redundant modeling of a contained or referenced *BuildingBlock* that occurs more than once in the overall decomposition structure. It is just a way to reduce the modeling effort because it indicates that the referencing *Container* has to be substituted with the whole *BuildingBlock* structure (including all its childs) contained in the referenced *Container*.

7.3 DVG

The *DVG* meta model deals with (ii) describing how variability entities of a building block relate to each other. It is an implementation of the DVG block-port modeling patterns in Ecore and divided into 3 parts:

- *Pattern*
- *Port*
- *ValueCOp*

The *Pattern* part is shown in figure 7.6 and defines that a *DVG* is a composition of different *Pattern*s. The concrete *Pattern* classes reference elements from the *Port* part and the *ValueCOp* part shown in figure 7.7 and 7.8. The *ValueCOp* part is an implementation of the value composition operator ($f(\cdot)$) principle.

Each *DVG* must reference a specific *BuildingBlock* to indicate to which it belongs. All *Port* classes that are instances of *AbstractOutputPort* contained by a *DVG* must reference a *VariabilityEntity* of the *BuildingBlock* that this *DVG* references in order to declare what *VariabilityEntity* and therefore what *DataObjectDef* this *AbstractOutputPort* represents. In general, we always distinguish between SP, CP (**CPort*) and PP (**PPort*) for corresponding instances of *AbstractOutputPort*, *AbstractInputPort* as well as *AbstractInitPort* in the meta model. **PPort* must be further divided based on their concrete rating models because they are assumed to come with their own, specific preference specifications. At the moment, we have only implemented the *Weighted Sum Model (WSM)*. Accordingly, we have defined the concrete classes *OutputWSMPort*, *InputWSMPort* and *InitWSMPort*. *AbstractInitPort*s are special *AbstractOutputPort*s that are used to initialize a VSP and are contained by the *INIT* pattern. Hence, all leaf patterns of this *DVG* implementation are of type *INIT*.

The difference between each of *InputPort* and *ExtInputPort*, *InputCPort* and *ExtInputCPort* and *InputWSMPort* and *ExtInputWSMPort* is that the latters are used to define a dependency interface to the outside, i.e. these specializations of *AbstractInputPort*s (*ExtInput*Port*) need to be connected with matching *AbstractOutputPort*(s) which are defined in a different *DVG* model. All these *ExtInput*Port*s

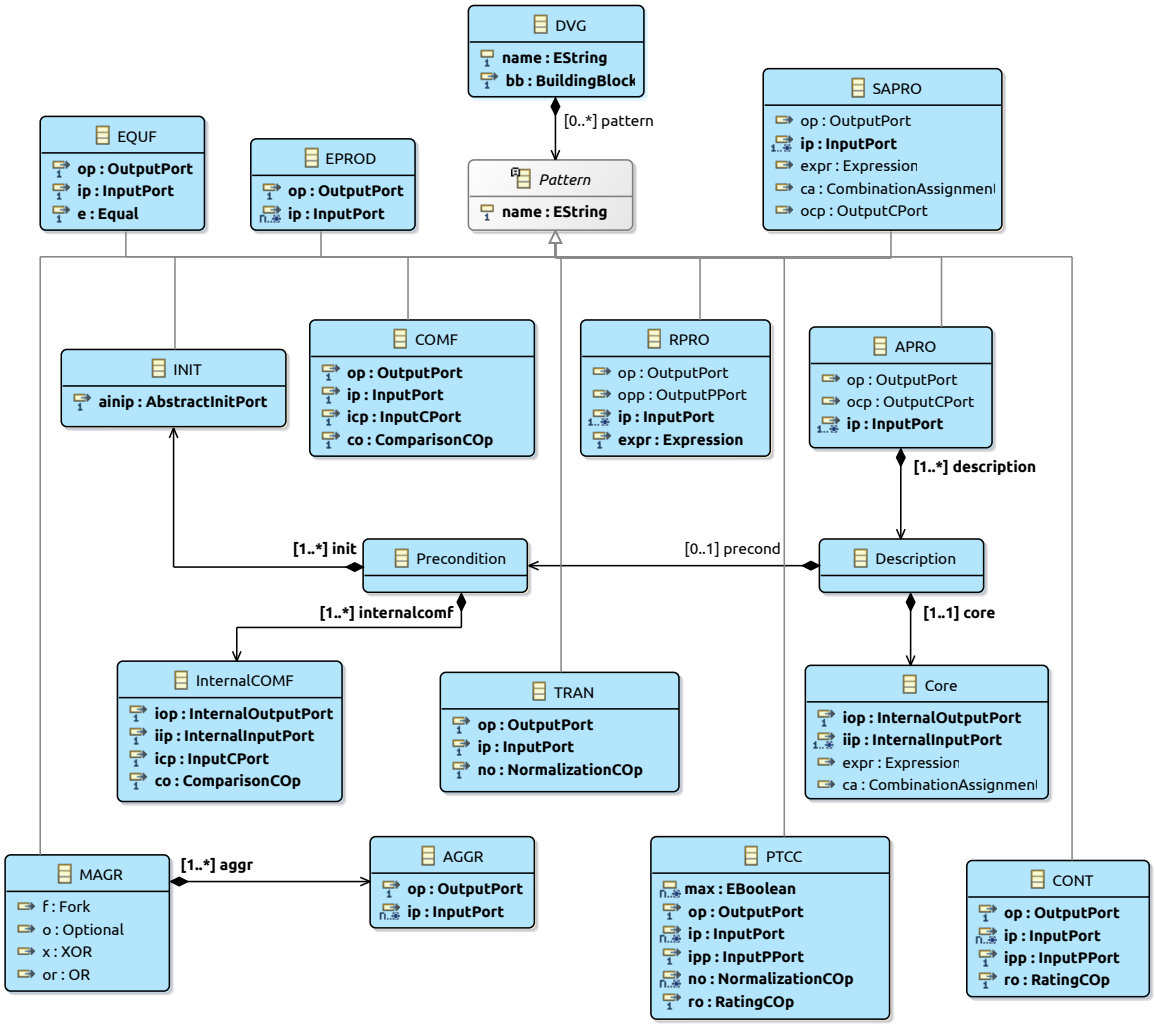


Figure 7.6: *Pattern* part of the *DVG* meta model.

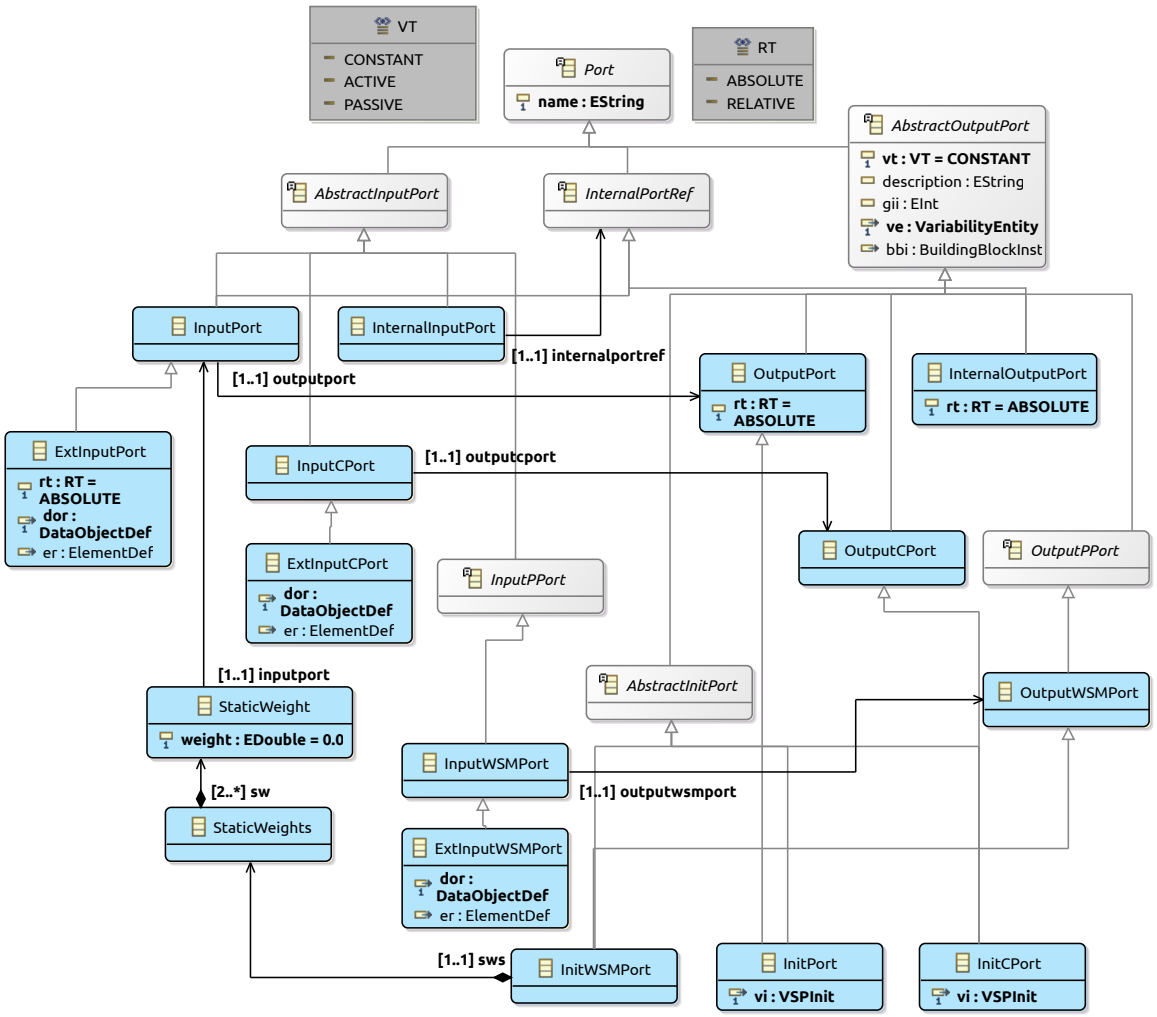


Figure 7.7: *Port* part of the *DVG* meta model.

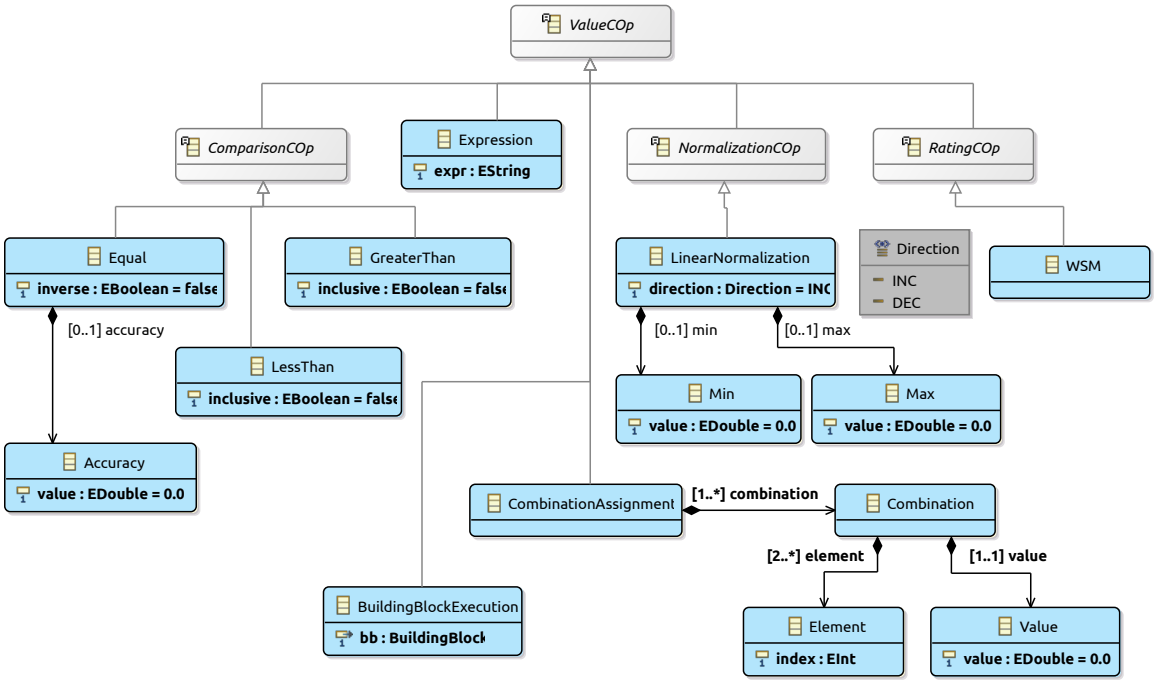


Figure 7.8: *ValueCOp* part of the *DVG* meta model.

must reference a *DataObjectDef* from the *DataObjectRepo* and optionally a corresponding *ElementDef* of that *DataObjectDef*. In addition a *RT* has to be defined. These information serve as the declaration of what exactly must be received from the outside (from other models created by different people) in order to realize a consistent composition with respect to the data types. All the *ExtInput*Port*s can also be connected locally with corresponding *AbstractInitPort*s which are then initialized with test data in order to perform a "what if analysis". But when composing a new *BuildingBlock*, all *ExtInput*Port*s of involved *BuildingBlock*s must be connected with matching *AbstractOutputPort*(s) of *DVG*s from other *BuildingBlock*s.

In the following we shortly describe the different *Pattern* instances of the meta model.

7.3.1 Initialization (INIT)

The *INIT* pattern is an implementation-specific structure containing exactly 1 *AbstractInitPort* to initialize either a SP (*InitPort*), CP (*InitCPort*) or PP (currently only *InitWSMPort* for the supported WSM). For that, *InitPort* and *InitCPort* must contain one *VSPInit* (figure 7.3) to assign VIs accordingly to the *DataObjectDef* of the *VariabilityEntity* that this *AbstractInitPort* instance references. In contrast, *InitWSMPort* just requires to have *StaticWeights* defined.

Note that the *INIT* pattern is implementation-specific and was not mentioned in the theoretical descriptions of the previous chapters. However, the relationship is simple as following: Every input port of the blocks of a DVG system with no connection to an output port in the theoretical consideration must now have a matching output port of a *INIT* pattern to which it is connected.

7.3.2 Extended Production (EPROD)

We defined the EPROD as an independent *Pattern* although it is already implemented internally in the other relevant *Pattern*s. However, the independent structure is sometimes useful for example to express a logical AND constraint between its input ports (no input ports must be empty in order to provide a non-empty output) where no value composition operator can be applied to the produced combinations. This *EPROD* implementation is designed to provide the VSP of its first input port as output. The *EPROD* must have one *OutputPort* and at least two *InputPort*s.

7.3.3 Comparison Filter (COMF)

The COMF realization must have one *OutputPort*, one *InputPort*, one *InputCPort* and one *ComparisonCOp*. A *ComparisonCOp* is a special class of a *ValueCOp*. The following concrete functions are currently defined. $\forall 0 \leqq i < I_0.|VSP|, \forall 0 \leqq j < I_1.|VSP|$:

- $I_0.VSP_i.VI == I_1.VSP_j.VI$ when using *Equal* with *inverse* = false
- $I_0.VSP_i.VI \neq I_1.VSP_j.VI$ when using *Equal* with *inverse* = true
- $I_0.VSP_i.VI < I_1.VSP_j.VI$ when using *LessThan* with *inclusive* = false
- $I_0.VSP_i.VI \leqq I_1.VSP_j.VI$ when using *LessThan* with *inclusive* = true
- $I_0.VSP_i.VI > I_1.VSP_j.VI$ when using *GreaterThan inclusive* = false
- $I_0.VSP_i.VI \geqq I_1.VSP_j.VI$ when using *GreaterThan inclusive* = true

Note, that the functions are only defined for primitive *VariabilityEntiti*es. A *VariabilityEntity* is primitive if:

- a *DataObjectDef* is associated that has only one *ElementDef* and the *cardinality* of its *TypeDef* is not set or one (not set means one)
- a *ElementDef* of a *DataObjectDef* is associated and the *cardinality* of its *TypeDef* is not set or one (not set means one)

Furthermore, *LessThan* and *GreaterThan* are only defined for primitive *VariabilityEntiti*es where the instance of *TypeDef* is either *IntegerDef* or *RealDef*. The *Equal* class further allows to define an *Accuracy* for comparing *RealDef*s by defining the attribute *value*. In that case equality and inequality checks are defined by equation 7.1 and 7.2, respectively.

$$(I_0.VSP_i.VI == I_1.VSP_j.VI) = \begin{cases} True \text{ if } |I_0.VSP_i.VI - I_1.VSP_j.VI| < value \\ False \text{ else} \end{cases} \tag{7.1}$$

$$(I_0.VSP_i.VI \neq I_1.VSP_j.VI) = \begin{cases} True \text{ if } |I_0.VSP_i.VI - I_1.VSP_j.VI| > value \\ False \text{ else} \end{cases} \tag{7.2}$$

7.3.4 Relative Production (RPRO)

The RPRO realization must have at least one *InputPort* and either one *OutputPort* or one *OutputPPort* (XOR constraint between *op* and *opp*). Hence, *RPRO* can be also applied to provide a dynamic preference specification depending on other *VariabilityEntit*ies of the system. Finally, the *RPRO* class requires one *Expression* that is an instance of a *ValueCOp* and which is defined by a String *expr*. In that *expr* we can write code in the solver language in order to express the functional relationship between the *OutputPort* or *OutputPPort*, respectively and the *InputPort*s. In order to reference the corresponding VSPs of the different *InputPort*s in an *expr* we must use the *name* attribute of these *InputPort*s that must be additionally enveloped by $ characters. This is required in order to be able to parse these names during the code generation and replacing it with code that accesses corresponding VIs of the associated VSPs.

7.3.5 Absolute Production (APRO)

The APRO realization must have at least one *InputPort* and either one *OutputPort* or one *OutputCPort* (XOR constraint between *op* and *ocp*). Hence, *APRO* can be also applied to provide a dynamic comparison space depending on other *VariabilityEntit*ies of the system. We use the special classes *InternalInputPort*, *InternalOutputPort* and *InternalCOMF* for internal purposes to realize the desired behavior of *APRO*. The *INIT* Pattern is used to initialize the constant comparison value for the *icp* of an *InternalCOMF* in a *Precondition*. In that context, *ainip* is restricted to the *InitCPort* class. The *Core* of a *Description* must either have one *Expression* or one *CombinationAssignment* (XOR constraint between *expr* and *ca*). A *CombinationAssignment* is also an instance of a *ValueCOp* and is realized by enumerating *Combination*s and assigning a *Value* via the *value* attribute where the i-th *Element* of a *Combination* refers to the i-th *InputPort* and the *index* attribute of a *Element* indicates the corresponding slot of an *InputPort*.

7.3.6 Simple Absolute Production (SAPRO)

Simple Absolute Production (*SAPRO*) realizes a APRO without *Description*s and *Precondition*s in order to have a simpler modeling structure for use cases where no different *Description*s and *Precondition*s are required. The XOR constraints between *op* and *ocp* as well as between *expr* and *ca* also apply here.

7.3.7 Multiple Aggregation (MAGR)

The MAGR realization must have at least one *AGGR* where each *AGGR* must have one *OutputPort* and at least two *InputPort*s. Furthermore, the *MAGR* class must reference either a *Fork*, an *Optional*, a *XOR* or an *OR* class from the referenced *BuildingBlock* (XOR constraint between *f*, *o*, *x* and *or*) to indicate of which specific *Fork* or *InputRelationship* this *MAGR* represents the different branches. The VT of the *MAGR* is then derived depending on the reference:

- The VT of *MAGR* is active if the reference is one of the *InputRelationship*s (*Optional*, *XOR* or *OR*).
- The VT of *MAGR* is active if the reference is an *EquivalenceFork*.
- The VT of *MAGR* is passive if the reference is a *ConditionalFork*.

7.3.8 Transformation (TRAN)

The TRAN realization must have one *OutputPort*, one *InputPort* and one *NormalizationCOp*. A *NormalizationCOp* is an instance of a *ValueCOp* and is only applicable to primitive *VariabilityEntit*ies with the concrete *TypeDef* instances *IntegerDef* or *RealDef* and must provide a resulting real value in the range $[0, 1]$. We defined the concrete class *LinearNormalization* as one example of a *NormalizationCOp*. *LinearNormalization* has the attribute *Direction* that can be increasing (*INC*) or decreasing (*DEC*) and means that the resulting values either increase or decrease with increasing values in the input. Furthermore, a *LinearNormalization* can have optionally a *Min value* and a *Max value*. Depending on the applied combination, the resulting output value $O.VSP_i.VI$ of an input value $I_0.VSP_i.VI$ is then defined by equation 7.3 where $\varnothing$ denotes if *Max* or *Min* is undefined.

$$O.VSP_i.VI = \begin{cases} \frac{\max(I_0.VSP) - I_0.VSP_i.VI}{max(I_0.VSP) - min(I_0.VSP)} \text{ if } Min = \varnothing \wedge Max = \varnothing \\ \frac{\max(I_0.VSP) - I_0.VSP_i.VI}{max(I_0.VSP) - Min.value} \text{ if } Min \neq \varnothing \wedge Max = \varnothing \\ \frac{Max.value - I_0.VSP_i.VI}{Max.value - min(I_0.VSP)} \text{ if } Min = \varnothing \wedge Max \neq \varnothing \\ \frac{Max.value - I_0.VSP_i.VI}{Max.value - Min.value} \text{ if } Min \neq \varnothing \wedge Max \neq \varnothing \end{cases} \tag{7.3}$$

$$O.VSP_i.VI = \begin{cases} 1 - O.VSP_i.VI \text{ if } direction = INC \\ O.VSP_i.VI \text{ if } direction = DEC \end{cases}$$

The meta model can be extended by many other useful *NormalizationCOp*s for simple reuse. As an alternative, we can specify any desired normalization function also in a *RPRO* pattern by specifying a corresponding *Expression* in the syntax of the solver language.

7.3.9 Contradiction (CONT)

The CONT realization must have one *OutputPort*, at least two *InputPort*s, one *InputPPort* and one *RatingCOp*. A *RatingCOp* is an instance of a *ValueCOp* and is only applicable to primitive *VariabilityEntit*ies with the concrete *TypeDef* instances *IntegerDef* or *RealDef* and must provide a resulting real value in the range $[0, 1]$. We defined the concrete class *WSM* as one example of a *RatingCOp*. *WSM* is the implementation of the *Weighted Sum Model* as defined in equation 7.4 that describes how to calculate the i-th output value ($O.VSP_i.VI$) where w_j defines a weight in the range $[0, 1]$ for the j-th *InputPort* and where $I_j.VSP_{k \in [0, I_j.|VSP|[}.VI$ refers to the VI of a specific slot index k from the VSP of the j-th *InputPort*. The corresponding PP for this concrete method was already introduced (*InputWSMPort*). We always need to ensure that the concrete *ipp* defined for a *CONT* is in line with the concrete *ro* applied. For example: If the *CONT* defines *InputWSMPort* as *ipp*, the *ro* needs to be *WSM* so that the applied method is compatible with the received preference specification.

$$O.VSP_i.VI = \sum_{0 \leqq j < |I|} w_j \cdot I_j.VSP_{k \in [0, I_j.|VSP|[}.VI \tag{7.4}$$

7.3.10 Pareto Transformation Contradiction Composite (PTCC)

The PTCC realization must have one *OutputPort*, $n > 1$ *InputPort*s and *NormalizationCOp*s, a boolean list *max* with n entries, one *InputPPort* and one *RatingCOp*. The i-th element of *max* indicates if the i-th *InputPort* wants to be maximized or not which is an important information for the

internal Pareto-Filter. The i-th *no* is applied internally to the not filtered, remaining slots of the VSP of the i-th *InputPort*. Same as for the *CONT* pattern, we always need to ensure that the concrete *ipp* defined for a *PTCC* is in line with the concrete *ro* applied.

7.3.11 Equivalence Filter (EQUF)

The EQUF realization has one *OutputPort*, one *InputPort* and one *ComparisonCOp* of instance *Equal*. *Equal* is currently only defined for primitive *VariabilityEntit*ies.

7.4 Variability Entities of Multiple Building Block Instances

Multiple *BuildingBlock* instances occur if the same *BuildingBlock* appears more than one time in the overall *Decomposition* structure which means it is planned to be executed several times. In the *BuildingBlockDescription* meta model (figure 7.4) we use the class *BuildingBlockInst* to refer to different instances of the same *BuildingBlock*. In addition, all *BuildingBlock*s that are part of a *Loop Decomposition* and all its childs are basically always subject to multiple instances if the *Number* is larger than 1. A general question is how to handle the *VariabilityEntit*ies of multiple *BuildingBlock* instances in a *DVG* model? Should we consider the *VariabilityEntit*ies of different instances of a *BuildingBlock* separately of each other or not?

A *Decomposition* structure defines a plan of actions in form of *BuildingBlock*s that later at runtime will be executed at a certain time in a certain place. These *BuildingBlock*s and therefore some of its *VariabilityEntit*ies can be influenced by these general temporal and spatial states. From the perspective of a *DVG*, this means that the resulting possible VSP of a corresponding passive port at a specific point in time can heavily depend on when and where the associated *BuildingBlock* is executed. Hence, in some situations it might make sense to restrict the VSP of a passive port already at design-time depending on the knowledge we have about the execution order of different *BuildingBlock* instances. In that case, modeling separate *VariabilityEntity* instances is useful. From the perspective of an active variant for example in form of a *Parameter* of a specific *BuildingBlock*, it would be not possible to bind different values of this *Parameter* for different *BuildingBlock* instances if this *VariabilityEntity* is not modeled separately in the *DVG* model.

In general, it depends on the specific application and the intended complexity of the associated *DVG* model if it is required to model independent ports for the same *VariabilityEntit*ies of different *BuildingBlock* instances. In the *Port* part of the *DVG* meta model we offer the possibility to express that an instance of an *AbstractOutputPort* represents the *VariabilityEntity* of a specific *BuildingBlock* instance by either referencing a *BuildingBlockInst* or by specifying a *gii (Global Instance Index)*.

7.5 Towards Standardization and Automated Matching

The ideal conception of a robotics software ecosystem offers an automated process that determines matching (composed) building blocks from its repository according to the requirements of a user (inter-variability management). A decisive prerequisite to accomplish that is a commonly agreed vocabulary for the specification of requirements (what is required?) and for the description of building blocks (what is provided?). Tier 2 in a robotics software ecosystem is responsible for defining such agreements and requires elaborate domain engineering from domain experts. The presented concept of the *DataObjectRepo* is one concrete example that can be used at tier 2.

However, also a *BuildingBlockDescription* and a *DVG* system model can be used as tier 2 structures. In that case, we can treat a *BuildingBlockDescription* as an abstract definition from a domain expert and every provider of a concrete *BuildingBlock* must adhere to these abstract definitions. In the same way, we can consider a *DVG* system model as abstract. An abstract *DVG* system model is always associated with a specific *BuildingBlock* from an abstract *BuildingBlockDescription* and describes a whole *DVG* system but without initializing VSPs at the leafs (no *INIT* patterns). These initialization values are then supplied by concrete *BuildingBlock*s provided at tier 3. This approach restricts the implementation freedom at tier 3 significantly, greatly reduces the effort for providers to describe their *BuildingBlock*s consistently and is the key for allowing automated resolution even in the case of inter-variability.

However, it is questionable if such standardizations can be established for every kind of building blocks even those representing complex robotic tasks or applications. One may also argue that this is actually not desired at all because the description of a building block should be treated as a decisive quality feature serving as a business model for its provider. The more effort is invested to come up with complex *DVG* system models, the more powerful, flexible and attractive a building block becomes to potential users and a provider should be rewarded according to the invested effort. Maybe a mixture of these variants could be a solution: Standardization takes place only for certain basic building blocks and more complex building blocks can reuse these basic building block. In addition the more complex buliding blocks could offer further individual and competing characteristics with respect to the description. For now, this discussion is of less practical relevance, because the idea of a robotics software ecosystem is still young, standardization efforts have not taken place and can only evolve over time by a community.

Part III

Applications and Conclusion

Chapter 8

Building Block Use Cases

In this chapter we illustrate different use cases related to building blocks of different abstraction levels that can exist in a robotics software ecosystem and that need to be modeled consistently by its provider in such a way that the model contributes an added value for somebody else (a user or another provider) who wants to reuse it meaningfully for its own purpose.

The building blocks that we describe in the following are model-based abstractions of real existing *SmartSoft* components and systems which are used widely in different real world service robotic applications. The models that we propose aim to be adequate and do not claim to be complete and/or most accurate. For the former one, this means that not every kind of variability that exists in the real component/system will be also necessarily explicated in the corresponding model. Consistency is ensured here by assuming that every not explicated variability in the model will have some constant binding in the real component/system which remains internal and is unchangeable. The latter one means that the model does not necessarily represent the relationships and results as accurate as possible. Nevertheless, the models still have an added value and by just investing more effort in analyzing the real components/systems, the accuracy of the associated models can be easily improved by using the provided tools based on the general method of this work.

Figure 8.1 shows three in the following described building block examples (CDL, Navigation and OrderPicking) for which we model VPSs and generate VSSs. The figure also illustrates the respective abstraction level of these building blocks in a robotics software ecosystem (Component, system of components/Skill, System of skills/Task) and their composition relationships. So is CDL part of Navigation and Navigation is part of OrderPicking. In case of Navigation, we show the simple reuse of the modeled VPS of CDL as part of the VPS of Navigation. In contrast, for OrderPicking we show that it is also possible to reuse only specific parts of the VPSs of the inner building blocks in order to construct models according to the desired level of detail. Finally, the last use case SharedOrderPicking is a special multi-robot collaboration scenario in which two service robot resources are involved that execute two different tasks. It can be treated as a system of tasks that have to be executed by different service robots. The service robots interact with each other via special synchronization mechanisms at the behavior level during the execution in order to fulfill the collaboration successfully.

Note that in the following text, each word in the font `name` indicates an element with this name in the presented models. Italic words still refer to corresponding elements of the meta models.

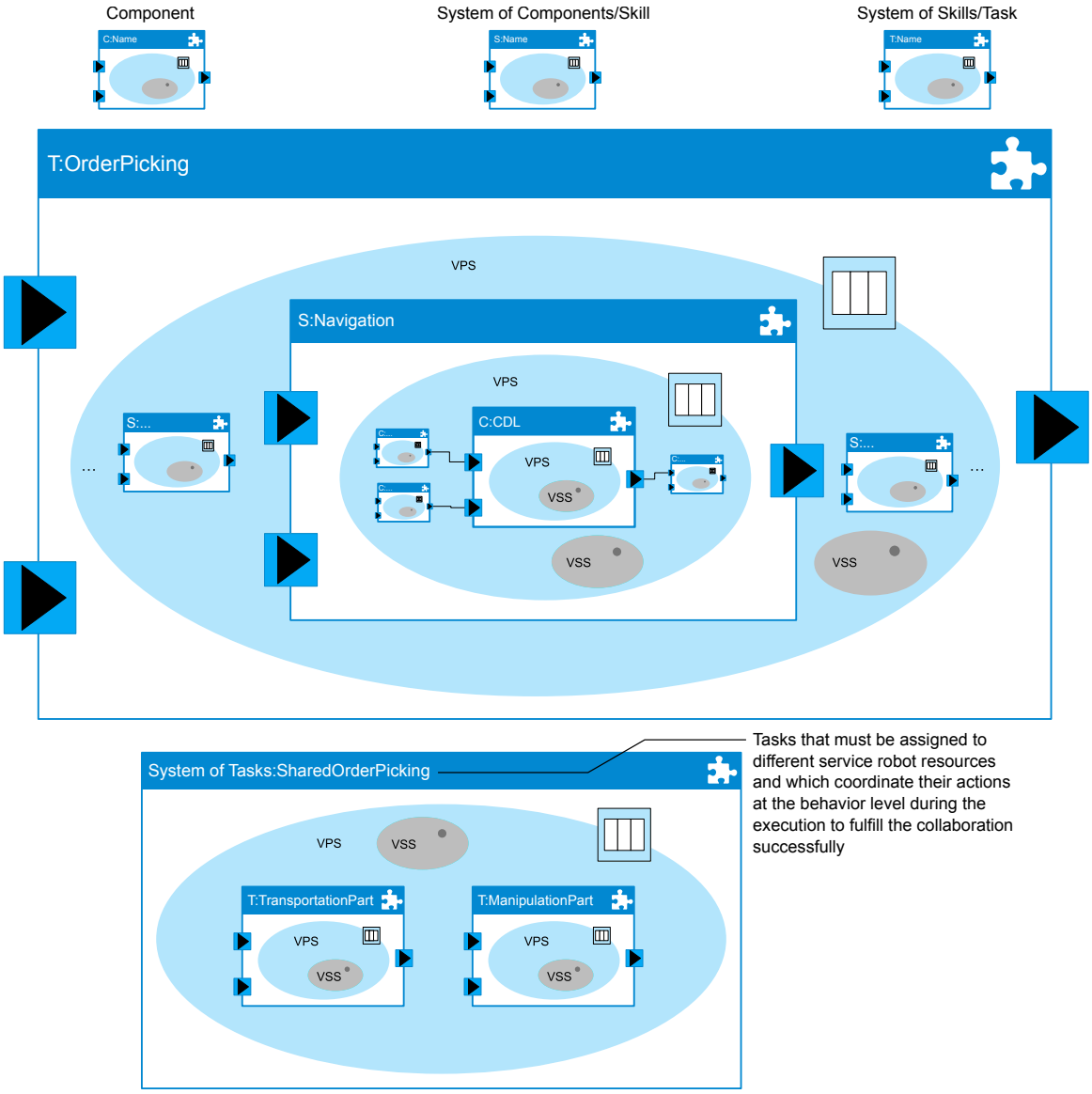


Figure 8.1: The composition relationships of the different building block examples (CDL, Navigation, OrderPicking) described in this chapter and their associated abstraction level in a robotics software ecosystem (Component, System of components/Skill, System of skills/Task). SharedOrderPicking is a multi-robot collaboration scenario that can be treated as a system of tasks which must be assigned to different service robot resources.

8.1 Component: CDL

The *Curvature Distance Lookup (CDL)* approach [Sch98] is a method for reactive obstacle avoidance under kinematic, dynamic and shape constraints. It is based on the *Dynamic Window Approach (DWA)* [FBT97] that operates on a discretized two-dimensional velocity space to determine a suitable local curvature (a translational (v) and rotational (ω) velocity combination) for a specific local state of the robot and its environment while considering its dynamic constraints. In contrast to the standard DWA method, the CDL approach shifts the distance calculation function that provides the remaining arc length between the robot and its local obstacle map to an offline part in order to reduce the execution time of the algorithm and thus the computational load at run-time. The standard CDL approach is defined for two-dimensional kinematics including a forward translational velocity v_x and a rotational velocity ω. The work in [BS15] provides a generalization of the approach with respect to a three-dimensional kinematic (v_x, v_y, ω) in order to also support omnidirectional driving.

We can formalize the offline part of the generalized CDL approach as a DVG system (figure 8.2) to define the dependency relationship between the curvature distance property, the discretized velocity space and the discretized obstacle map. The overall velocity space is defined by the three independent velocity components v_x, v_y and ω. Each component can be considered as a individual VSP of type active because the online part of the algorithm will later at run-time choose an appropriate velocity triple from a specific subspace of the combined velocity components which is always restricted by the current dynamic window. Furthermore, the two-dimensional obstacle map can be interpreted as a passive VSP because it will exhibit a highly dynamic obstacle occupancy state at run-time for which we assume that we have no knowledge about its transition depending on our actions. Enumerating VIs for such an obstacle map in a DVG system model is of course not really meaningful because the combinatorics is much to large. However, in that particular use case we are interested in the curvature distance of all velocity combinations to all potential occupied cells of the obstacle map. Hence, we can simply represent the VSP of our obstacle map with one VI in which all cells are occupied. Hence, we can describe the obstacle map as a constant port. The decisive information of a velocity combination regarding the distance calculation is the location of the *ICC (Instantaneous Center of Curvature)*, i.e. the point in a two-dimensional cartesian space around which the robot rotates when driving constantly with a specific velocity combination. We can determine the ICC from a velocity triple (v_x, v_y, ω) by equation 8.1.

$$ICC(v_x, v_y, \omega) = \begin{pmatrix} \frac{\sqrt{v_x^2 + v_y^2}}{\omega} \cdot sin(atan2(v_y, v_x)) \\ \frac{\sqrt{v_x^2 + v_y^2}}{\omega} \cdot cos(atan2(v_y, v_x)) \end{pmatrix} \tag{8.1}$$

If we perform equivalence filtering after applying equation 8.1 to the cartesian product of the three independent velocity VSPs, we can reduce the overall number of combinations to consider in the subsequent distance calculation step significantly which saves time and space. The combination reduction principle of curvatures/ICCs is actually an implicit domain characteristic of the CDL approach to make it feasible. Hence, for this concrete CDL example, we can show that our general procedure of equivalence filtering is directly able to reproduce this special implicit domain characteristic when the problem is formalized as a DVG system.

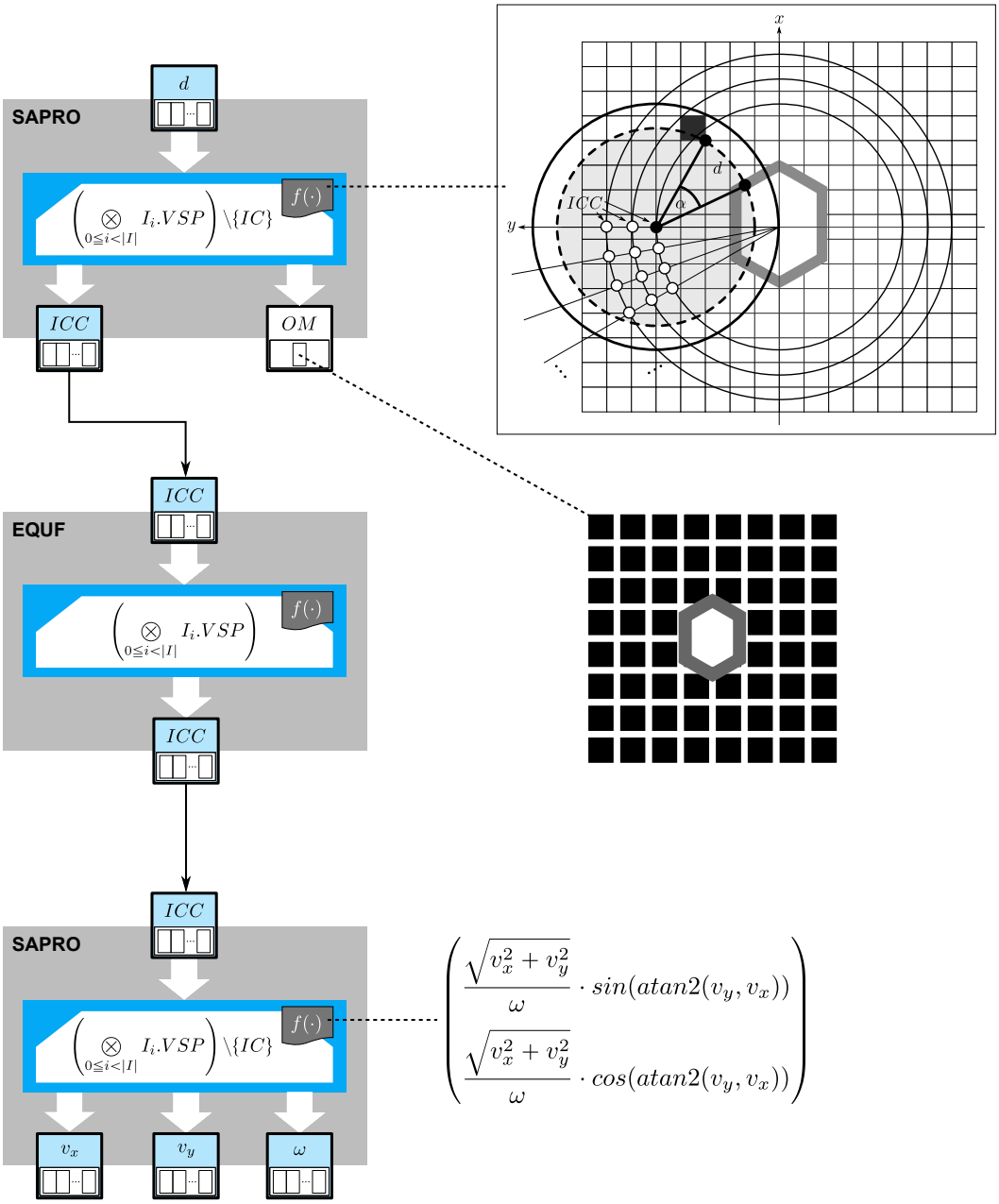


Figure 8.2: The offline part of the CDL approach described as a DVG.

8.1.1 The Building Block Model

The CDL component shown in figure 8.3 implements the online part of the CDL approach. We assume that this component supports exclusively the two-dimensional variant (2 DOFs). Then the procedure of the algorithm can be described as follows:

1) Definition of the dynamic window based on the current state of the robot and its dynamic constraints.

2) Extracting the relevant curvatures from the dynamic windows as representatives of the possible velocity combinations.

3) Mapping the current laser scan into the local obstacle map and determining the minimum distance for each curvature in the dynamic window by using the lookup table from the offline part.

4) Evaluation of all admissible velocity combinations from the dynamic window by applying an objective function.

5) Provide the highest rated velocity combination to the outside.

The `CDL` building block represents the abstract summary of all relevant *VariabilityEntit*ies that matter in this use case (see table 8.1 for a short description). We decompose `CDL` into the three more fine-grained building blocks `Init`, `LaserScanMappingAndMinDistDet` and `DetVelComb` via the *Sequential Decomposition* type. These three building blocks are model based representatives of the steps 1) and 2) (`Init`), 3) (`LaserScanMappingAndMinDistDet`) and finally 4) and 5) (`DetVelComb`) of the CDL online part as described previously. `LaserScanMappingAndMinDistDet` is further decomposed into `CellMappingAndMinDistCalc` to explicate the mapping process for one ray of the whole scan regarding one cell of the obstacle map (therefore *Loop decomposition*). In addition, `DetVelComb` is decomposed via *EquivalenceFork* into the building blocks `StrategyA` and `StrategyB` representing two functional equivalent alternatives to determine a curvature.

8.1.2 The DVG Model

The associated DVG system model of `CDL` is shown in figure 8.4. In this model we have one complete DVG and 4 independent partial DVGs.

Partial DVGs

The four partial DVGs are responsible to restrict the internal component variables `MaxTraVel`, `MaxRotVel`, `MaxTraAcc` and `MaxRotAcc` that characterize the dynamic window depending on external capabilities of concrete hardware devices. Note that we assume that only forward motions are allowed, so we have to consider only the maximum limits (positive signs). In all 4 cases we define the final value as the minimum of the external capability value and an internal constant value. In the model we represented the external inputs as passive with 2 example values in order to simulate potential composition requests from the outside for this local view. The defined minimum function with respect to `MaxTraVel` and `MaxRotVel` is essential because our corresponding internal constant values represent the limits of our velocity space from which we derived the set of relevant curvatures in the offline part. Hence, if we would not restrict external values of specific hardware here we could

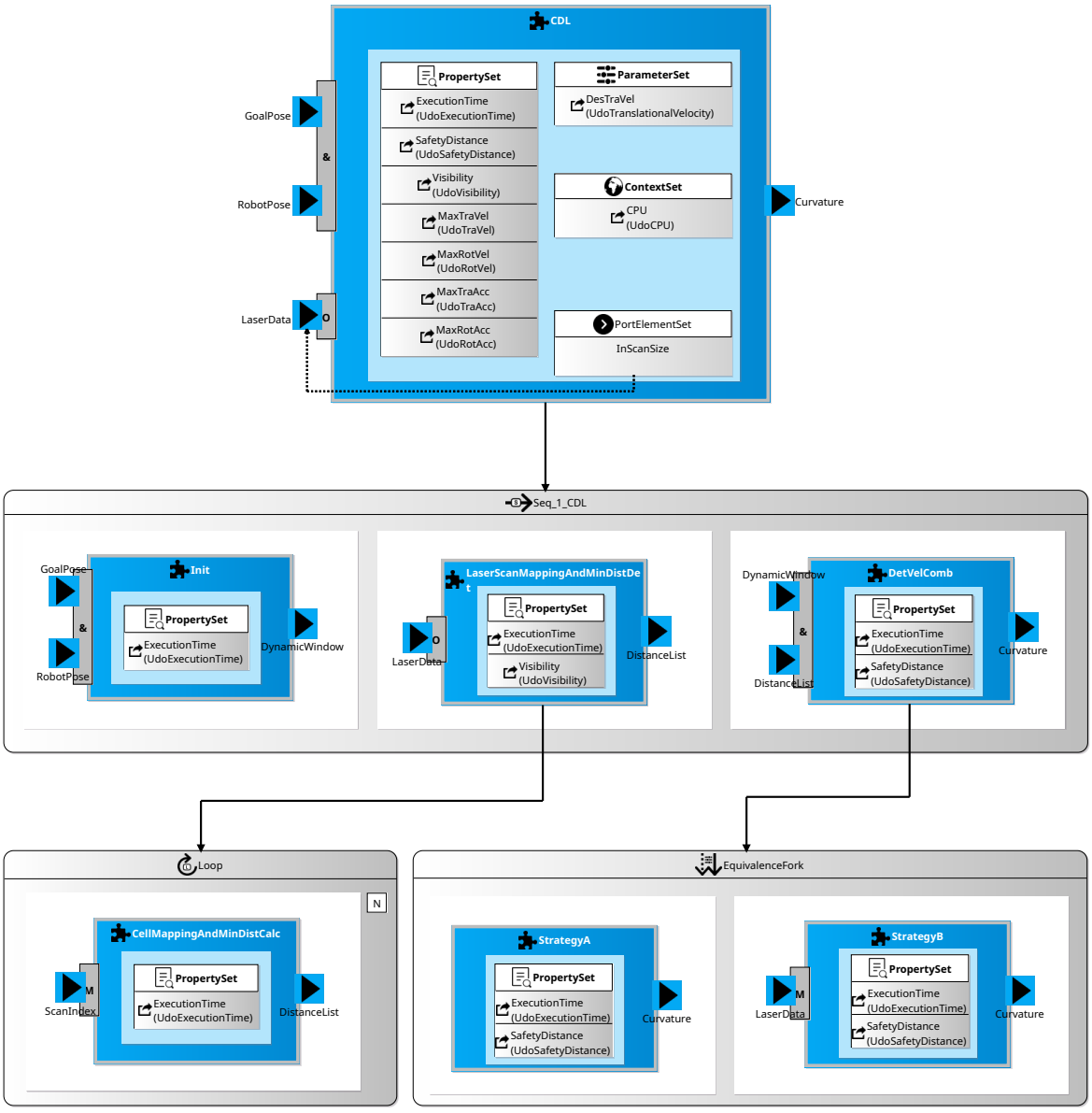


Figure 8.3: The building block model of the CDL component.

Table 8.1: Meaning of the *VariabilityEntit*ies modeled in the `CDL` building block in sense of an absolute representation of the associated *DataObjectDef*. This does not mean that each *VariabilityEntity* in the model will be also represented in absolute form.

Name	**DataObjectDef**	**Unit**	**Description**
`ExecutionTime`	Real	μs	The execution time of a building block
`SafetyDistance`	Real	mm	The minimum lateral distance of a Curvature to an obstacle cell
`Visibility`	Real	$\%$	Percentage of actually present obstacle cells correctly recognized and mapped
`MaxTraVel`	Real	mm/s	Positive translational velocity limit of the discretized velocity space
`MaxRotVel`	Real	$^\circ/s$	Positive rotational velocity limit of the discretized velocity space
`MaxTraAcc`	Real	mm/s^2	Translational acceleration limit of the dynamic window
`MaxRotAcc`	Real	$^\circ/s^2$	Rotational acceleration limit of the dynamic window
`DesTraVel`	Real	mm/s	Desired translational velocity during drive
`InScanSize`	Integer	-	The overall number of rays of a single laser scan
`CPU`	String	-	The name of a specific CPU model

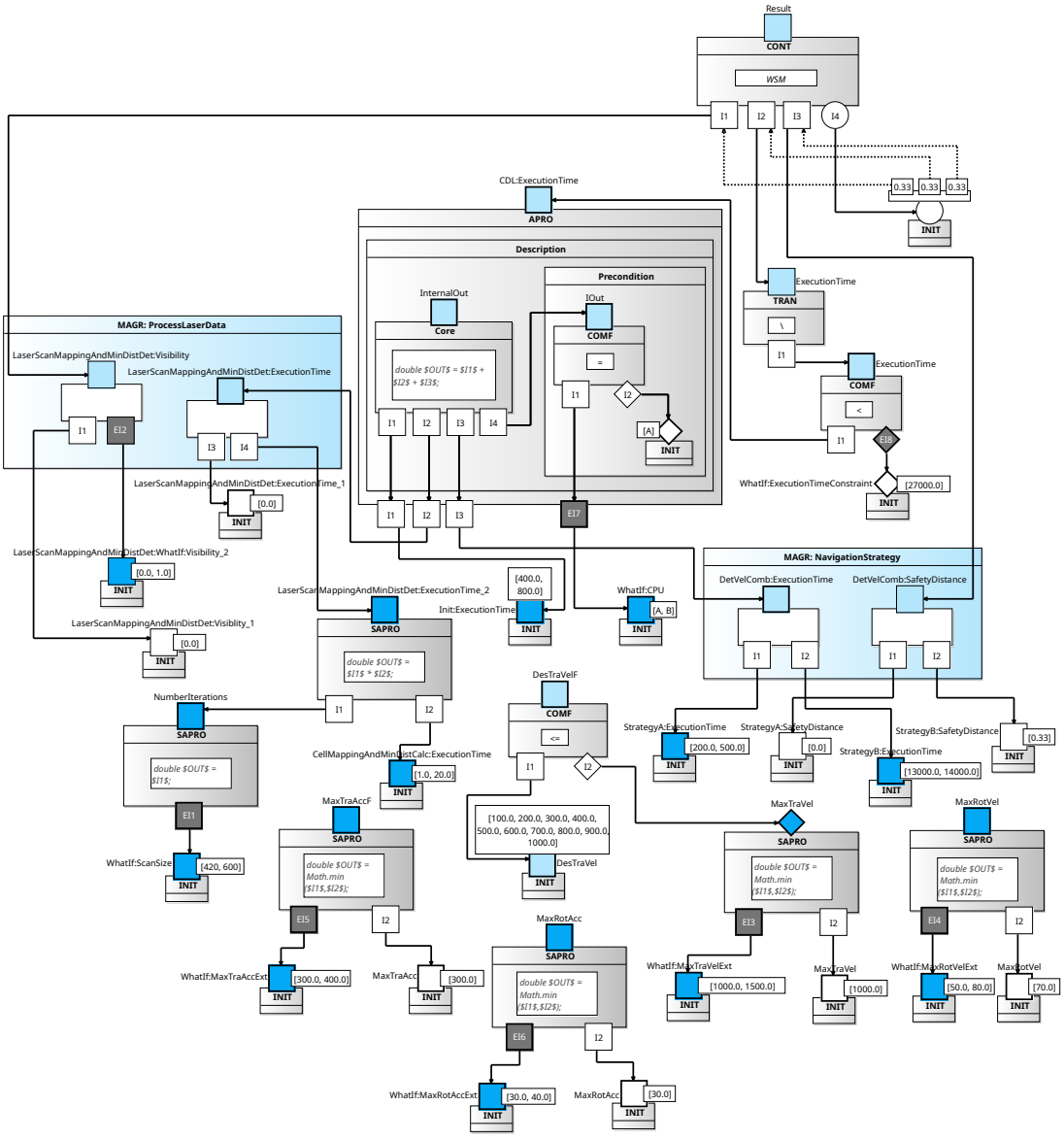


Figure 8.4: The DVG system model of the CDL building block from figure 8.3.

run into an inconsistent situation where our lookup table can not provide distance values for some velocity combinations. The restricted `MaxTraVel` result is modeled by a passive *OutputCPort*. The OVSP is then used to filter the active VSP of `DesTraVel` which we assigned an initial VSP of size `10` between the values `100.0` and `1000.0` with a discretization accuracy of `100.0`. This makes sense because it should be not allowed to define desired velocities that can never be reached in real.

The argumentation for `MaxTraAcc` and `MaxRotAcc` is different: Both values define the current size of the dynamic window per each time step which means that they also define the maneuverability of a robot. Since these parameters are in contrast to `MaxTraVel` and `MaxRotVel` not subject to the inconsistent situation described before it seems unclear at the first sight why we also restrict external capabilities here. The reason is that the *Property* `ExecutionTime` of `DetVelComb` depends on these two values because we need to evaluate more velocity combinations within one time step if the dynamic window is larger. In our case we assume that the `ExecutionTime` was only tested and described for our two internal constant values. Therefore, this composition description ensures accuracy while not exhausting the full potential capabilities of external composition requesters. The exact opposite can be accomplished if we would always override the internal values by the external ones. A third further alternative could be to describe the `ExecutionTime` of `DetVelComb` by APRO with a *Precondition* that demands for specific `MaxTraAcc` and `MaxRotAcc` values under which the `ExecutionTime` values are accurate. In that case the accuracy becomes an inherent part of the overall composition validity check.

Complete DVG

The complete *DVG* is used to evaluate our active variants. The `CDL` building block in figure 8.3 offers 2·2 = 4 active variants because of the *EquivalenceFork* where we can choose `StrategyA` or `StrategyB` and the optional `LaserData` *Input* that we can choose to process or not.

Init The `Init` building block is described regarding the `ExecutionTime` (absolute representation). It is defined as a passive port in order to specify a best-case and worst-case interval since precise constant descriptions are not meaningful enough in this case. Note, that this also applies for all other ports describing the `ExecutionTime` property in the following.

LaserScanMappingAndMinDistDet This building block is described regarding `ExecutionTime` (absolute representation) and `Visiblity` (relative representation) and is the representative building block for the active variant of processing `LaserData` (this building block is executed) or not (this building block is not executed). Hence, we need to describe `ExecutionTime` and `Visibility` in dependence of these two variants by using a MAGR `ProcessLaserData`.

Not Processing LaserData If we do not process `LaserData` we know that the `Visibility` will be `0.0` (relative representation) because our obstacle map will assume that every cell is free (we assume no history) which is very likely not the true case. This is of course a very risky variant but has the advantage that the `ExecutionTime` of this building block will be `0.0` (absolute representation) since it is not executed at all. Both *VariabilityEntit*ies are described by constant ports because the values are certain and have no external dependencies.

Processing LaserData This does not apply for the opposite case where we process `LaserData` and thus execute `LaserScanMappingAndMinDistDet`. With respect to `Visibility`, we assume that it depends completely on the outside. Hence, we define an *ExtInputPort* and model a passive port that simulates 2 potential "what if" input values (a worst-case (`0.0`) and a best-case (`1.0`)). The `ExecutionTime` is defined as the product of the `ExecutionTime` of `CellMappingAndMinDistCalc` and the number of loop iterations (`NumberIterations`) which depends on the size of the `LaserScan` (`InScanSize`). We model an *ExtInputPort* and again define a passive port (`ScanSize`) for a "what if" analysis using the values `420` and `600`.

DetVelComb This building block is described regarding `ExecutionTime` (absolute representation) and `SafetyDistance` (relative representation) and is the representative building block for the active variants `StrategyA` and `StrategyB`. Hence, we need to describe `ExecutionTime` and `SafetyDistance` in dependence of these two variants by using a MAGR `NavigationStrategy`.

StrategyA `StrategyA` applies a standard objective function that evaluates velocity combinations regarding `DesTraVel` and the angle deviation of the current `RobotPose` to the next `GoalPose` but does not consider `SafetyDistance` at all. Hence, we assign a constant value of `0.0` to this variant. The advantage is that the `ExecutionTime` interval is lower than for `StrategyB`.

StrategyB `StrategyB` considers the `SafetyDistance` when determining a curvature but that requires additional costly online calculations that increase the `ExecutionTime` a lot in contrast to `StrategyA`. For `SafetyDistance` we assign a constant, relative value of `0.33` because this is the weighting of the internal objective function for `SafetyDistance` with respect to the other considered criteria `DesTraVel` and angle deviation.

CDL The `ExecutionTime` of `CDL` (`CDL:ExecutionTime`) is defined as the sum of the `ExecutionTime` of `Init`, `LaserScanMappingAndMinDistDet` and `DetVelComb` because of their *Sequential Decomposition*. We model this description by using APRO containing a *Precondition* that demands for a specific `CPU` model in order to guarantee accuracy of the explicated `ExecutionTime` values in μs for which we assume that those have been measured during execution while the explicated `CPU` model (`A`) was used. Hence, this description design greatly reduces the overall composability of the building block (will match only for users with the same `CPU` model) to guarantee accuracy. If a less restrictive description is wanted, more analysis is required to generalize the `CPU` dependency.

We also modeled a COMF that allows to define and take into account a maximum value constraint from the outside for the `ExecutionTime` of `CDL`. The remaining values are transformed to a relative representation and are then used as input for the final CONT that evaluates `ExecutionTime`, `Visiblity` and `SafetyDistance`.

Results The size of the CONSS of the DVG system model of `CDL` is 2 with 2 variant bindings each:

- `ProcessLaserData` = `true` or `false`
- `NavigationStrategy` = `StrategyA` or `StrategyB`

Hence, the total combined variant space is 4. The DVG system model can now be tested with different configurations extensively to derive important insights about the resulting properties, specific relationships between variability entities and the most adequate variant bindings for all possible contingencies and situations. For example:

- Based on the configuration in figure 8.4 the most useful overall system state is reached for the execution variant `{true A}` if all passive ports reach their modeled best-case.
- A change in the preference weighting from `{0.33, 0.33, 0.33}` to `{0.25, 0.25, 0.5}` makes the execution variant `{true B}` to the most useful overall system state given that all passive ports reach their modeled best-case.
- Given that preference weighting is `{a > 0.0, b = 0.0, c = 1-a}`, the most useful overall system state is always `{true B}` independent of our exact preference values for `a` and `c`. This shows the independence between `Visibility` and `SafetyDistance` that are only in conflict with `ExecutionTime`.
- If we have the worst-case situation for `Visibility` the most adequate execution variant will always be `{false x}`. This makes sense because if it is not possible to see anything due to external reasons, the processing makes also no sense since we can not improve the `Visibility` value (the value of not processing is also `0.0`). Therefore the robot will at least try to save resources in such a situation.
- For an overall `ExecutionTime` constraint of `27` ms all possible variants are satisfiable even if all passive ports reach their modeled worst-case.

8.2 Skill: Navigation

This navigation skill example is at the system level because it is a composition of different components defining a specific navigation architecture that realizes the skill. It allows a mobile robot to drive from its current location to a desired target location. This use case builds on the previous CDL example: The CDL component alone is not sufficient to realize a remarkable capability at the behavior level because it is only one building block of a more complex navigation system. There are a lot of other components required to realize a navigation skill, for example a component that determines the current location of the robot or a component planning a global path to the target location based on intermediate waypoints that are then approached successively by the CDL component. Therefore, we assume that the provider of this navigation building block will reuse a set of components already available in a repository which realize required sub functionalities. The provider is then in charge to describe the whole system meaningfully with respect to other potential users based on the building block models of the building blocks the provider reuses and its own extensions.

8.2.1 The Building Block Model

The `Navigation` building block model is shown in figure 8.5. `Navigation` represents *VariabilityEntit*ies, i.e. *Propert*ies and *Context* that are of relevance while the robot is driving to the target location. An overview about all explicated *VariabilityEntit*ies in this building block that are not already described in table 8.1 can be found in table 8.2.

The second level ReactiveNavigationLoop introduces a specific task chain [Lot18] of the system and one of its important resulting properties (WCRT). From a technical viewpoint all leaf building blocks that then follow in the subsequent *Decomposition* are tasks of a component (in a computing sense, also known as threads) so high frequency processes performing calculations. Therefore, the third level that explicates concrete tasks of this chain from different components is represented by an *Unsequenceable Decomposition* because due to the high frequency nature of component tasks we are never able to specify all possible execution sequences that can potentially occur in reality. The first and the last container in this *Decomposition* reuses and therefore references the BaseStateTask and the VelCmdTask from the MobileBase building block shown in figure 8.6a. This building block describes a component that implements the driver of a specific hardware Model and thus enables the technical capabilities of physical locomotion for this specific device. The corresponding DVG system model of the MobileBase is shown in figure 8.6b. As we can see, the model explicates information about the physical dynamic limitations of the device which is of relevance for the CDL building block (see table 8.1), but also describes information regarding the ExecutionTime of the BaseStateTask and the VelCmdTask that run independent of each other (*Parallel Decomposition*).

Furthermore, the DVG system model of MobileBase defines a COMF that uses the supported Models of the implemented driver represented by its names as the comparison set (second input port). In this example, we assume that the building block only supports the Model with the name A. The first input port is of type ExtInputPort and receives desired Models of other users in the ecosystem as part of their requirement specification. For example: This building block is worthless for a user that is not in possession of Model A but of B. Hence, this fact should be captured in the matching process for this building block in order to consider such potential possession constraints expressed by users as part of their requirement specifications.

The SensorProcessing building block introduces variability with respect to the provision of LaserData and therefore represents the aggregated results of the properties ExecutionTime, Accuracy, Dimension, Visibility and ScanSize based on the child models. Hence, it follows a *EquivalenceFork Decomposition* with two building block branches. The first alternative references the Laser building block shown in figure 8.7 where the associated component task implements the driver for a specific sensor hardware device of the Model A. The properties HFOV and Resolution are characteristics of that sensor and based on that we can determine the ScanSize of the LaserData as shown later in the DVG system model. The second alternative CameraProcessing represents a way to produce LaserData based on a camera device. For that, it decomposes into a sequence of two tasks of different components: The first reuses and references the Camera building block described in figure 8.8. Just like in the case of Laser and MobileBase it represents the driver implementing a real camera device of a Model A. The second building block LaserFromRGBD represents a task of a component that receives RGBDData from Camera and transforms it to LaserData. This building block is implemented by the provider of this Navigation building block in order to realize the variation point with respect to the provision of LaserData.

Back in the third *Decomposition* layer, the third container there indicates the reuse of the CDL building block already described in the previous section (figures 8.3 and 8.4) by referencing it.

8.2.2 The DVG Model

The DVG system model of the Navigation building block is shown in figure 8.9. It contains references to the DVG system models of the building blocks Laser, Camera, MobileBase and

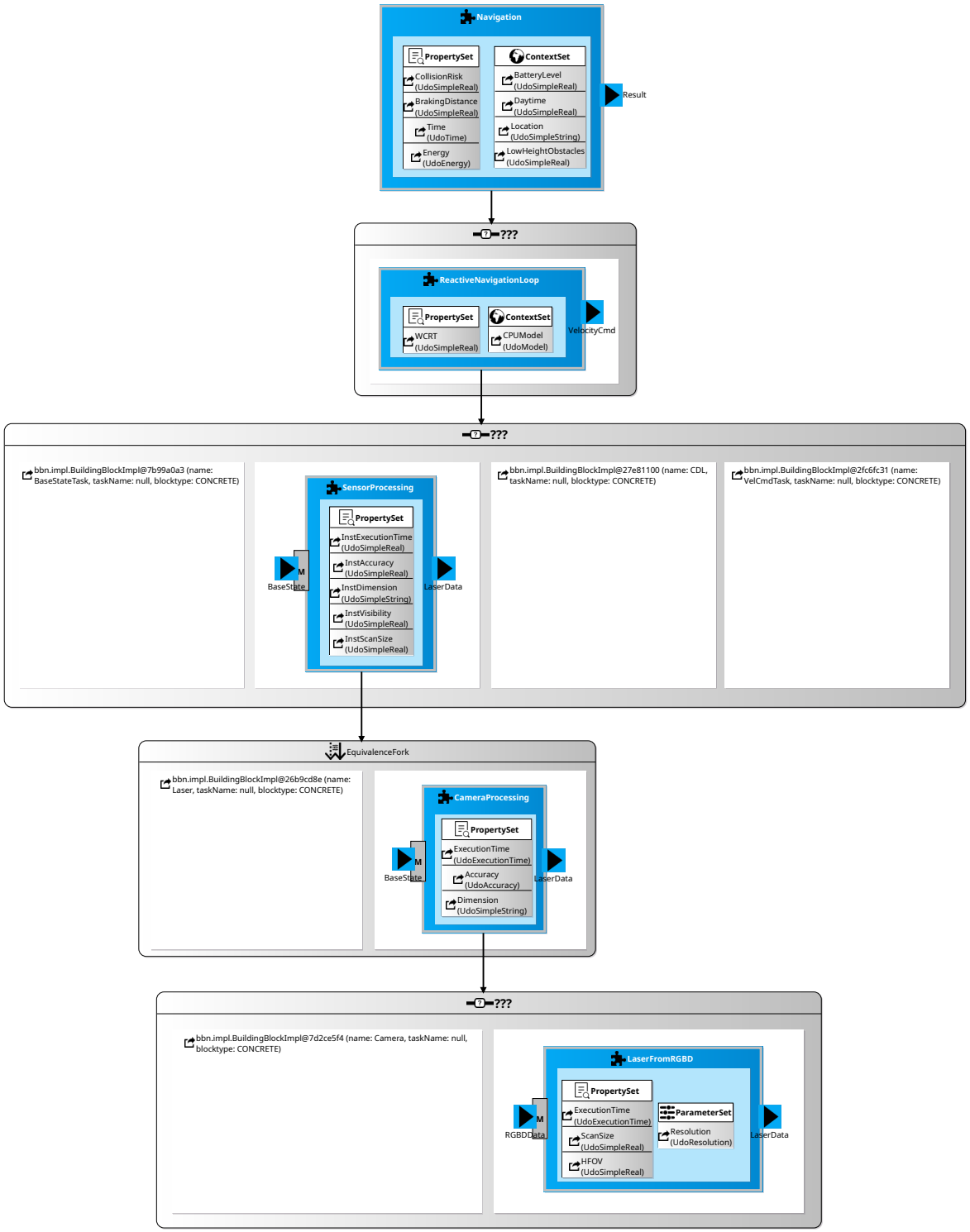
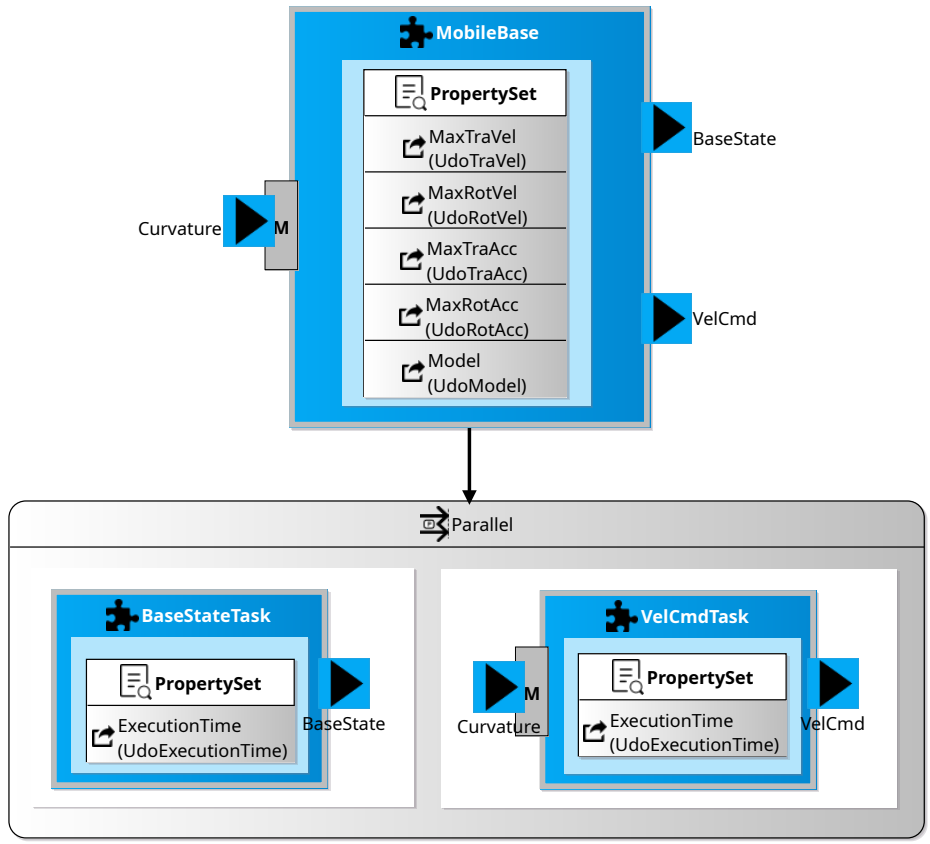


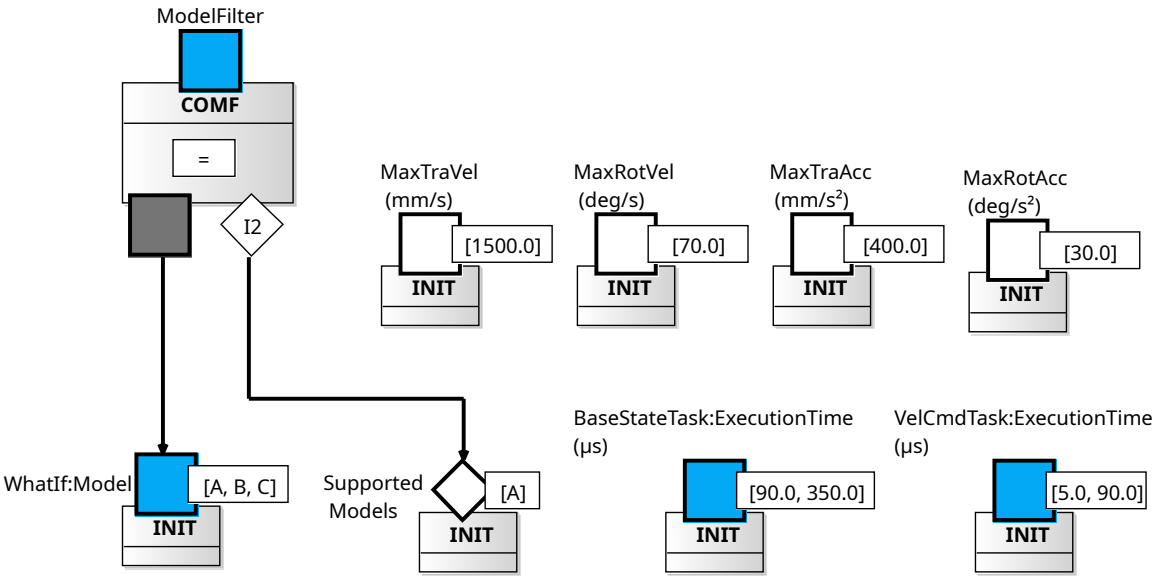
Figure 8.5: The building block model of the `Navigation` example.

Table 8.2: Meaning of the *VariabilityEntit*ies modeled in the `Navigation` building block in sense of an absolute representation of the associated *DataObjectDef* that are not yet listed in table 8.1. This does not mean that each *VariabilityEntity* in the model will be also represented in absolute form.

Name	**DataObjectDef**	**Unit**	**Description**
`Time`	Real	s	The remaining time duration of the drive
`Energy`	Real	As	The energy consumption of the navigation system
`CollisionRisk`	Real	$\%$	Likelihood of colliding with a static or dynamic obstacle
`BrakingDistance`	Real	mm	The distance still traveled after a stop command
`BatteryLevel`	Real	$\%$	The current battery level of the robot
`Daytime`	Real	$HH:MM$	The current daytime
`Location`	$\{x, y, \theta\}$	$\{mm, mm, ^\circ\}$	The current location of the robot
`LowHeightObstacles`	Real	$\%$	Percentage of obstacles in an environment with a height less than the height of the mounted sensor
`WCRT`	Real	s	The worst case response time of a chain of component tasks
`Model`	String	-	A name that identifies a specific hardware device uniquely
`Dimension`	String	-	The dimension of a sensor (2D or 3D)
`Accuracy`	Real	mm	The deviation between distance values provided by a sensor and the actual true values
`HFOV`	Real	$^\circ$	The horizontal field of view of a sensor device
`Resolution`	Real	$^\circ$	The angular distance between the single rays of a laser scan

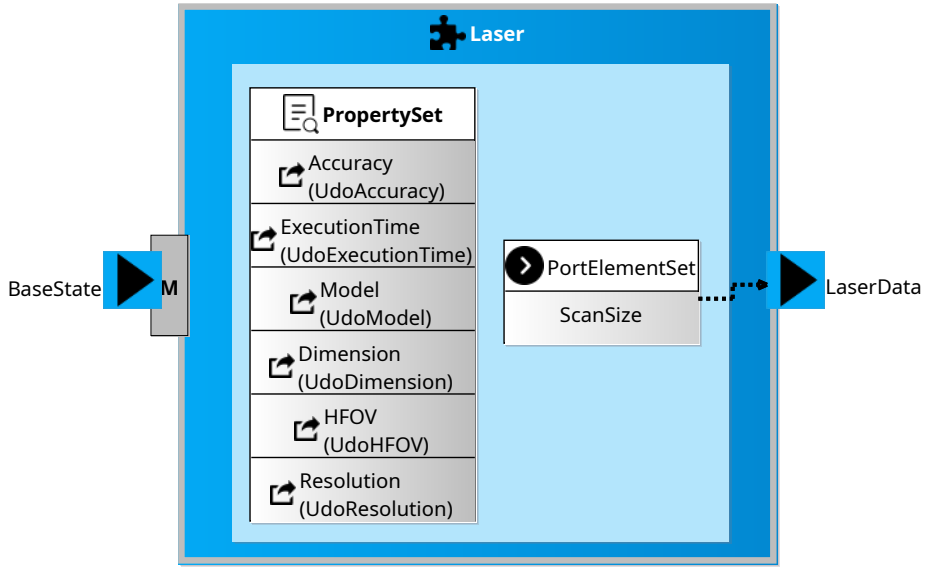


(a)

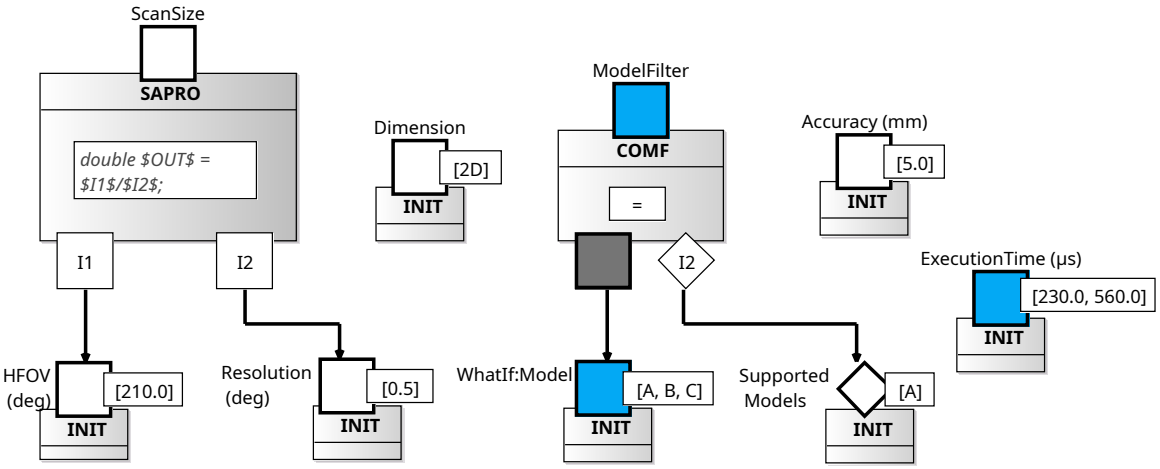


(b)

Figure 8.6: (a) The building block model of the `MobileBase`. (b) The associated DVG system model.

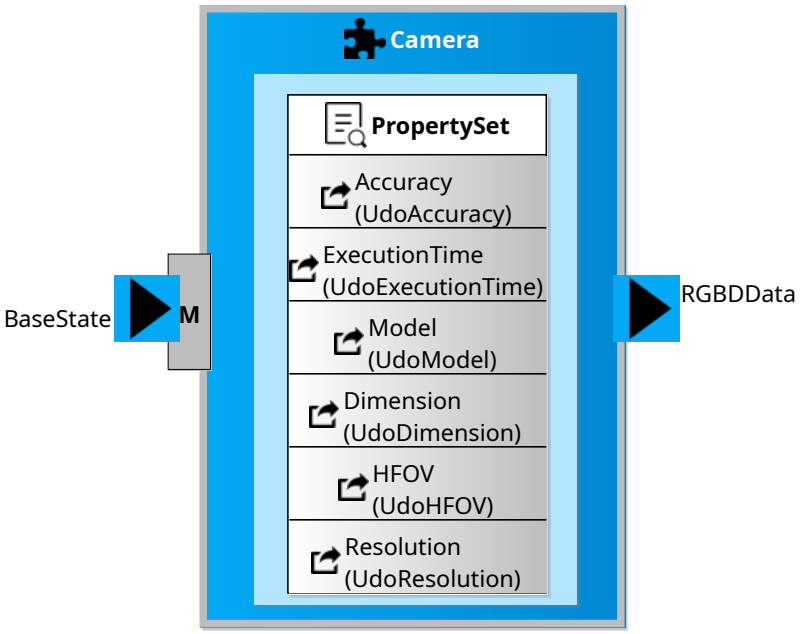


(a)

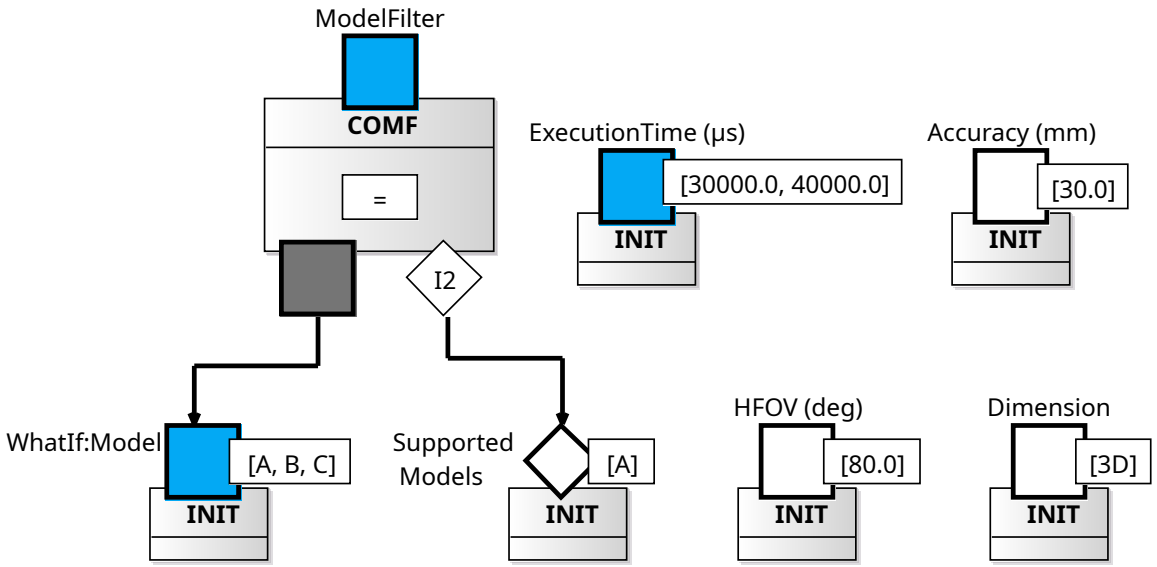


(b)

Figure 8.7: (a) The building block model of the `Laser`. (b) The associated DVG system model.



(a)



(b)

Figure 8.8: (a) The building block model of the `Camera`. (b) The associated DVG system model.

CDL and also to the requirement specification of the end user.

SensorProcessing

The MAGR SensorProcessing is used to aggregate relevant properties of the sensor variants. As we can see, the ScanSize of the first variant (laser) is determined by connecting the *ExtInputPort* with the output port (ScanSize) of the Laser DVG system model (figure 8.7b). The second input port is connected to a *SAPRO* that determines the ScanSize of the second variant (camera) based on the horizontal field of view of the camera (port HFOV in figure 8.8b) and the Resolution parameter of the LaserFromRGBD building block that is assumed to be constant (LaserFromRGBD:Resolution with value 0.5).

The Accuracy and Dimension properties in the second and third AGGR of the MAGR are both referenced from the corresponding ports in the Laser DVG system model (Accuracy and Dimension, figure 8.7b) and the Camera DVG system model (Accuracy and Dimension, figure 8.8b).

Finally, the ExecutionTime for the laser variant is received from the corresponding port of the Laser DVG system model (figure 8.7b) while for the camera it is modeled as the sum of the ExecutionTime of the Camera (figure 8.8b) and the ExecutionTime of the LaserFromRGBD building block.

The aggregated Dimension is used as input for a RPRO that determines the Visibility based on a simple utility function: The Visibility is 1.0 if the Dimension of the corresponding sensor is 3D. If not, then the value depends on LowHeightObstacles which must be specified and provided by the end user of the Navigation building block (see later). The aggregated Accuracy is further processed by a TRAN applying a decreasing linear normalization function.

Wiring CDL

The provider of the Navigation building block is in charge to wire the *ExtInputPort*s of the DVG system model of the reused CDL building block with corresponding matching output ports of the DVG system model of the Navigation building block and other reused DVG system models:

- *ExtInputPort* EI1 is connected to the SensorProcessing:ScanSize port in the Navigation DVG system model.
- *ExtInputPort* EI2 is connected to the Visibility port in the Navigation DVG system model.
- *ExtInputPort* EI3 is connected to the MaxTraVel port in the MobileBase DVG system model.
- *ExtInputPort* EI4 is connected to the MaxRotVel port in the MobileBase DVG system model.
- *ExtInputPort* EI5 is connected to the MaxTraAcc port in the MobileBase DVG system model.
- *ExtInputPort* EI6 is connected to the MaxRotAcc port in the MobileBase DVG system model.

- *ExtInputPort* `EI7` is connected to the `CPU` port in the requirement specification model as part of the `Navigation` building block (see figure 8.10).

Note that we did not connect *ExtInputPort* `EI8` to any requirement specification from the end user that would allow to define a maximum time limit for the execution time of the `CDL` building block. The reason is that this is not necessary in the context of `Navigation` because we have a higher level property `BrakingDistance` which is influenced by this execution time. Therefore we offer a constraint specification directly for this higher level property in order to simplify the requirement specification process for an end user.

WCRT and Braking Distance

The `WCRT` (*Worst Case Response Time*) of a task chain is very complex to determine due to a lot of dependencies that have to be considered such as the execution times of the involved tasks and their activation semantics, the specific CPU and also the kind of scheduling strategy used [Lot18]. These are all *VariabilityEntit*ies in the context of this work. However, due to these complexities, specialized solvers must be applied in order to derive reliable `WCRT` values. The contributions in [Lot18] which establish an interface between modeled task chains in a robotics software ecosystem and such a specialized solver are not yet an integrated part of the DVG method. Hence, we are currently not able to apply external solvers for performance aspects directly as a further specialization of a *ValueCOp* because it requires some additional implementation effort. For that reason, we describe the `WCRT` property in this example by using *CombinationAssignment* and thus assume that the provider analyzed all relevant paths in the task chain externally and then inserted the resulting values manually in this DVG system model of the `Navigation` building block. Input to the APRO that applies the *CombinationAssignment* for producing the `WCRT` output port are:

1. `BaseStateTask:ExecutionTime`
2. `SensorProcessing:ExecutionTime`
3. `CDL:ExecutionTime`
4. `VelCmdTask:ExecutionTime`

All these ports representing `ExecutionTimes` specify best-case and worst-case values that can be directly used as inputs for the corresponding tasks in the external task chain model that is linked to a performance analyzer. The *CombinationAssignment* where we insert manually the results from the performance analyzer contains 8 combinations because of the size 3 of the CONSS (`SensorProcessing` = `Laser` or `Camera`, `ProcessLaserData` = `true` or `false`, `NavigationStrategy` = `StrategyA` or `StrategyB`) each with 2 possible variants (2^3). For complexity reasons, we do not explicate any further variability dependencies of a performance analysis here and assume that they are just constant and not available for configuration. One exception is the `CPU` (represented as a string) that we use as a precondition for the description of this APRO.

The `BrakingDistance` property depends on the `WCRT`, the acceleration deceleration and the current translational velocity of the robot. The acceleration deceleration value can be received from the `MaxTraAccF` output port and the current translational velocity can be approximated with the `DesTraVelF` output port that represents the filtered `DesTraVel` parameter (both can be found in the DVG system model of `CDL`). The specified formula in the *SAPRO* defines the relationship between `BrakingDistance` and these three variability dependencies.

A COMF is then modeled with respect to the BrakingDistance in order to consider value restrictions for this property from potential end users. Finally, the BrakingDistance is normalized in a TRAN.

Conflicts

The first modeled conflict with the output port CollisionRisk depends on LaserScanMappingAndMinDistDet:Visibility (CDL), Accuracy, DetVelComb:SafetyDistance (CDL) and BrakingDistance. The CONT expects a preference specification from the end user.

The overall utility value of the DVG system model is finally captured in the Result output port of the topmost CONT. This CONT depends on the CollisionRisk, Time and Energy. In this example, both Time and Energy have a relative representation and the modeled dependency is the translational velocity of the robot (DesTraVelF of CDL). Time is given by an increasing linear utility function (a TRAN is used to accomplish that) while Energy is modeled via an exponential function specified in the *Expression* of a RPRO where the utility values decrease with increasing DesTraVelF input values. The topmost CONT with the Result output port finally expects a preference specification from the end user to determine the overall result.

Finally we use a EPROD that receives Result as first input and further the filtered results from the matching process of the hardware models (what is supported by the building blocks and what is possessed by users). This expresses that the determined Result is only valid if all hardware model constraints of building blocks representing a hardware device (Laser, Camera and MobileBase) are fulfilled with respect to the hardware possession of a user.

End User Context and Requirement Specification

Figure 8.10 shows a context and requirements specification for the Navigation building block that a potential end user of this system could define:

- Model demands for Laser (LaserModel), Camera (CameraModel) and MobileBase (MobileBaseModel) to specify what hardware devices the user possesses. These output ports serve as input for the corresponding COMFs in the DVG system models of Laser, Camera and MobileBase respectively. These results are then forwarded to the EPROD in the DVG system model of Navigation.
- The specific CPU model that the user employs in his robot system. The DVG system model of Navigation references this output port in the precondition of the APRO that provides the WCRT in order to guarantee accuracy of the given values.
- LowHeightObstacles (see the meaning of LowHeightObstacles in table 8.2) with dependencies to SymbolicLocations in the environment of the end user. The DVG system model of Navigation references this output port to determine Visibility.
- A dynamic maximum value constraint for the BrakingDistance (BDConstraint) varying with the current Daytime finally considered in the DVG system model of Navigation.
- A static preference specification CollisionRiskPref for the CollisionRisk conflict in the DVG system model of Navigation.

- A dynamic preference specification DyPr for the overall conflict in the DVG system model of Navigation depending on the current BatteryLevel of the robot.

Results

The size of the CONSS of the DVG system model of the Navigation building block is 4:

- ProcessLaserData = true or false (from CDL)
- NavigationStrategy = StrategyA or StrategyB (from CDL)
- SensorProcessing = Laser or Camera
- DesTraVel = [100.0, 1000.0] (mm/s) with a discretization step size of 100.0 (from CDL)

We can see that the size of the VSP of the first three elements is 2 and for the last one 10 because the CDL internal MaxTraVel is set to 1000.0 and we defined a discretization step size of 100.0. Hence, our total combined configuration space for which we want to determine the most adequate configuration variant is $2^3 \cdot 10 = 80$. If the end user would not restrict the BrakingDistance property at all, Result would contain 42240 possible system states that can occur and its calculated utility values. For the given example specification that is even more restrictive at noon (e.g. safety reasons because the environment is assumed to be more crowded by people at this daytime) the overall number of valid system states is reduced to 35552. We can now check if the DVG system model is always satisfiable or not and inspect what is the most adequate combined configuration variant for each possible system state modeled. Finally, we can store these results in a lookup table so that a mobile robot executing the Navigation building block can access these information to configure itself at run-time adequately according to the specified requirements and the current situation.

8.3 Task: Order Picking

The order picking example is an even more complex system that involves skills of different robotic domains. The general task represented by this building block is to collect n objects (NumberObjects) of the same type of a specific order From a specific location and deliver it To a specific location.

8.3.1 The Building Block Model

Figure 8.11 shows the building block model and its *Decomposition*: The topmost building block OrderPicking defines the property Time and the context Capacity referring to the overall number of objects the used robot can transport (we assume the robot carries a container where the objects can be placed). A description about all the relevant *VariabilityEntit*ies used in this example can be found in table 8.3. OrderPicking decomposes into a *Loop* containing the building block sequence GotoLocation, DetectObjects, ManipulateObjects and GotoLocation_2. The first GotoLocation approaches the GoalPosition of the location where the desired objects to collect are stored. DetectObjects analyzes the target scene to provide detected ObjectPoses. ManipulateObjects further decomposes into a *Loop* containing a PickAndPlace building block which grasps an object from one pose (F) in ObjectPoses and places it to a target pose (T) (we assume a fixed place on the robot where the container is located). Finally, GotoLocation_2 approaches the GoalPosition where the objects have to be delivered to.

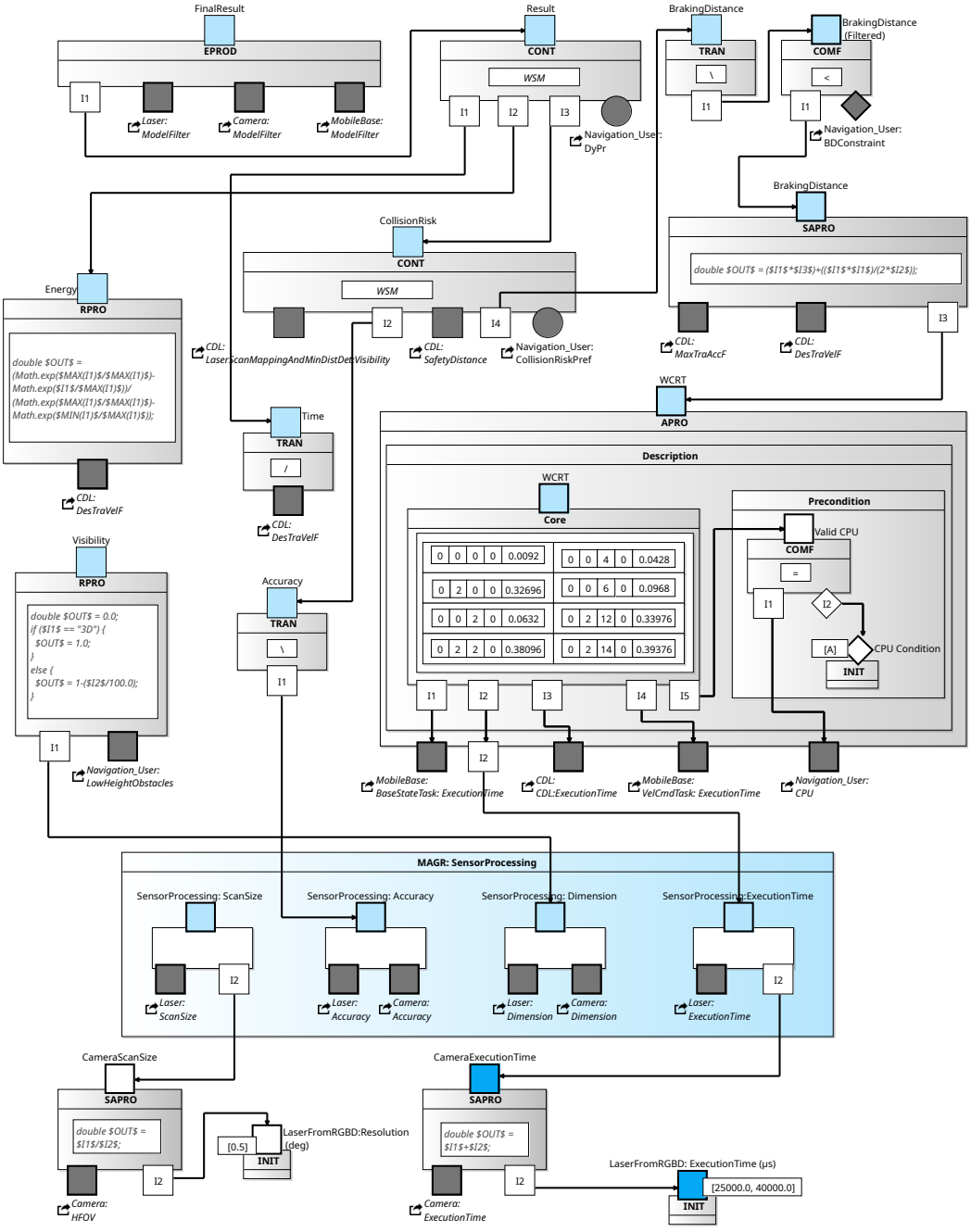


Figure 8.9: The DVG system model of the `Navigation` building block with dependencies/references to Laser (8.7b), Camera (8.8b), MobileBase (8.6b), CDL (8.4) and additional user specifications (8.10).

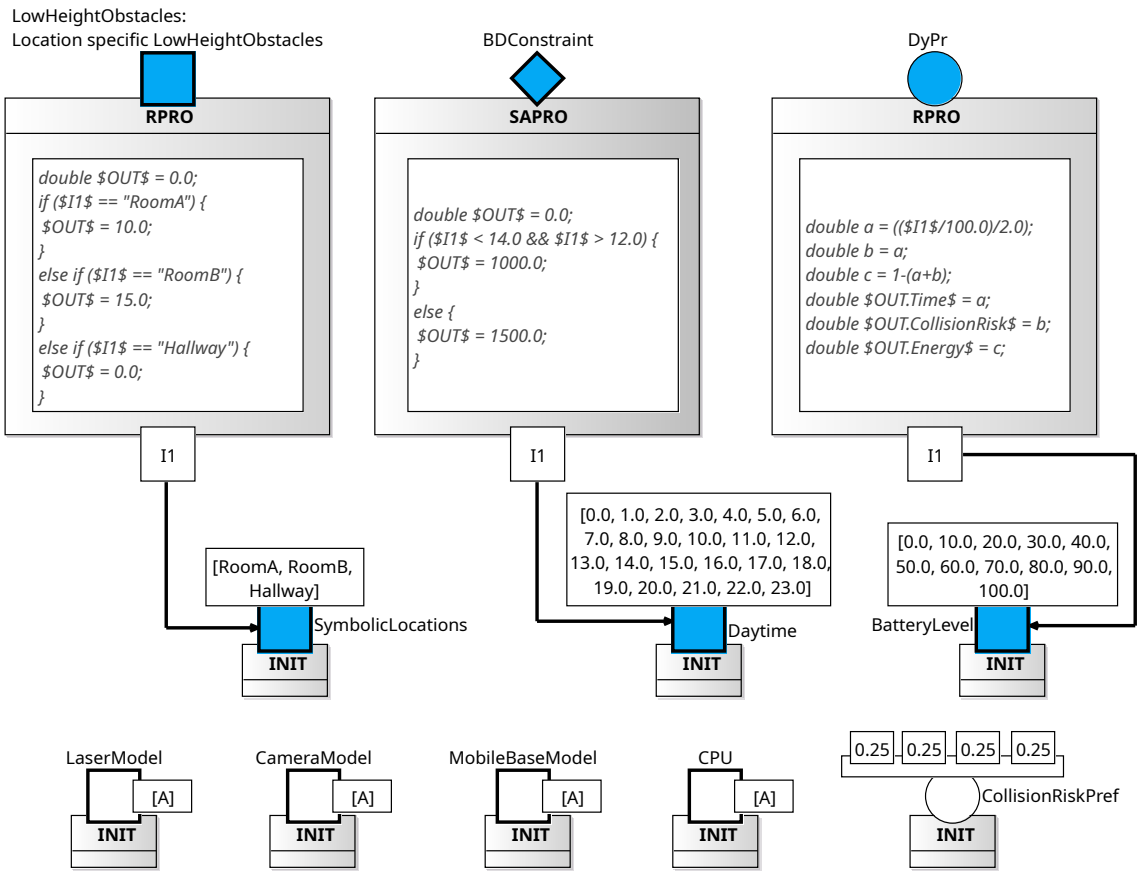


Figure 8.10: Example for a context and requirement specification of a potential end user of the `Navigation` building block.

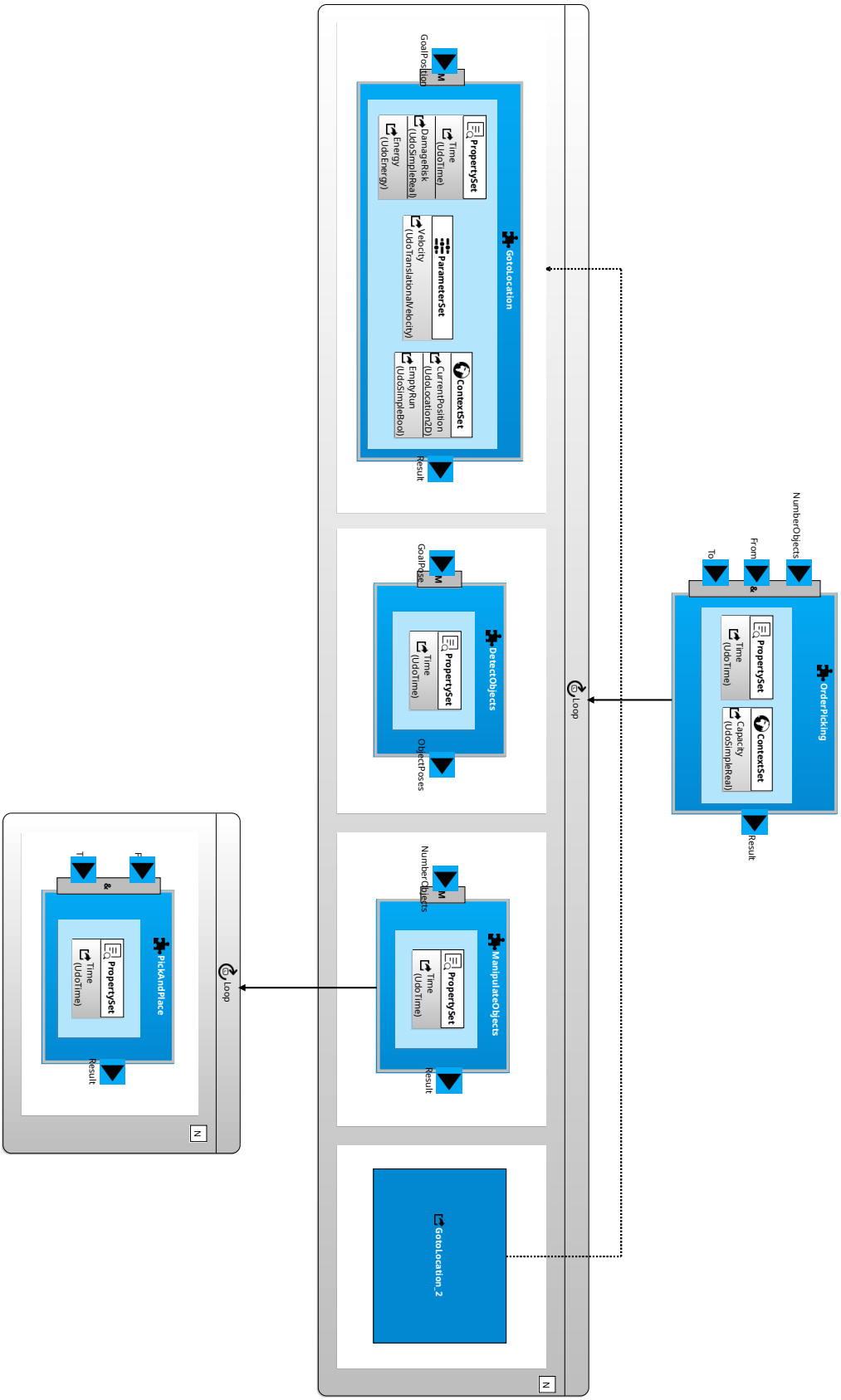


Figure 8.11: The building block model of the `OrderPicking` example.

Table 8.3: Meaning of the *VariabilityEntit*ies modeled in the `OrderPicking` building block in sense of an absolute representation of the associated *DataObjectDef*. This does not mean that each *VariabilityEntity* in the model will be also represented in absolute form.

Name	**DataObjectDef**	**Unit**	**Description**
`NumberObjects`	Integer	Piece	The number of objects to collect
`Time`	Real	s	The required time duration of a building block
`Capacity`	Integer	Piece	The maximum number of objects that fit into the container carried by the robot
`GoalPosition`	$\{x, y, \theta\}$	$\{mm, mm, ^\circ\}$	The target location of a GotoLocation building block
`CurrentPosition`	$\{x, y, \theta\}$	$\{mm, mm, ^\circ\}$	The current location of the robot when a GotoLocation building block is executed
`DamageRisk`	Real	$\%$	The risk of damaging carried objects
`Velocity`	Real	mm/s	The desired translational velocity during the execution of a GotoLocation building block
`EmptyRun`	Bool	-	Whether the container is empty or not during the execution of a GotoLocation building block
`Energy`	Real	As	The energy consumption of a Goto building block

8.3.2 The DVG Model

Figure 8.12 shows the DVG system model of `OrderPicking`.

Time Composition Model

One major part of this example is the description of a simple, optimistic `Time` composition model (absolute representation) for the `OrderPicking` building block (t_o) which is also shown in equations 8.2 - 8.6. For the used abbreviations, please see table 8.4.

Table 8.4: Abbreviations used in the equations and their meaning.

Abbreviation	Meaning
n	`NumberObjects`
c	`Capacity`
t_{g0}	`Time` of `GotoLocation`
t_{g1}	`Time` of `GotoLocation_2`
t_d	`Time` of `DetectObjects`
t_m	`Time` of `ManipulateObjects`
t_{pap}	`Time` of `PickAndPlace`
$x_{gp_{g_i}}$	x of `GoalPosition` of g_i
$y_{gp_{g_i}}$	y of `GoalPosition` of g_i
$x_{cp_{g_i}}$	x of `CurrentPosition` of g_i
$y_{cp_{g_i}}$	y of `CurrentPosition` of g_i
v_{g_i}	`Velocity` of g_i

$$t_o = \begin{cases} n \cdot c^{-1} \cdot (t_{g_0} + t_d + t_m + t_{g_1}) & \text{if } n \bmod c = 0 \\ \lceil n \cdot c^{-1} \rceil \cdot (t_{g_0} + t_d + t_m + t_{g_1}) - ((\min(n, c) - (n \bmod c)) \cdot t_{pap}) & \text{else} \end{cases} \tag{8.2}$$

$$t_{g_i} = \frac{\sqrt{(x_{gp_{g_i}} - x_{cp_{g_i}})^2 + (y_{gp_{g_i}} - y_{cp_{g_i}})^2}}{v_{g_i}} \tag{8.3}$$

$$t_d = 11.0 \tag{8.4}$$

$$t_m = \min(n, c) \cdot t_{pap} \tag{8.5}$$

$$t_{pap} = 16.0 \tag{8.6}$$

If the number of desired objects n is larger than the capacity c, more than one instance of the building blocks of the second level (*Loop*) *Decomposition* must be executed to fulfill the task. In this example we assume the following: All objects of an order can be found at the same location (`From`) and must be delivered to the same location (`To`). Hence, the `StartPosition` *sp* and `CurrentPosition` *cp* of `GotoLocation` g_0 and `GotoLocation_2` g_1 will not

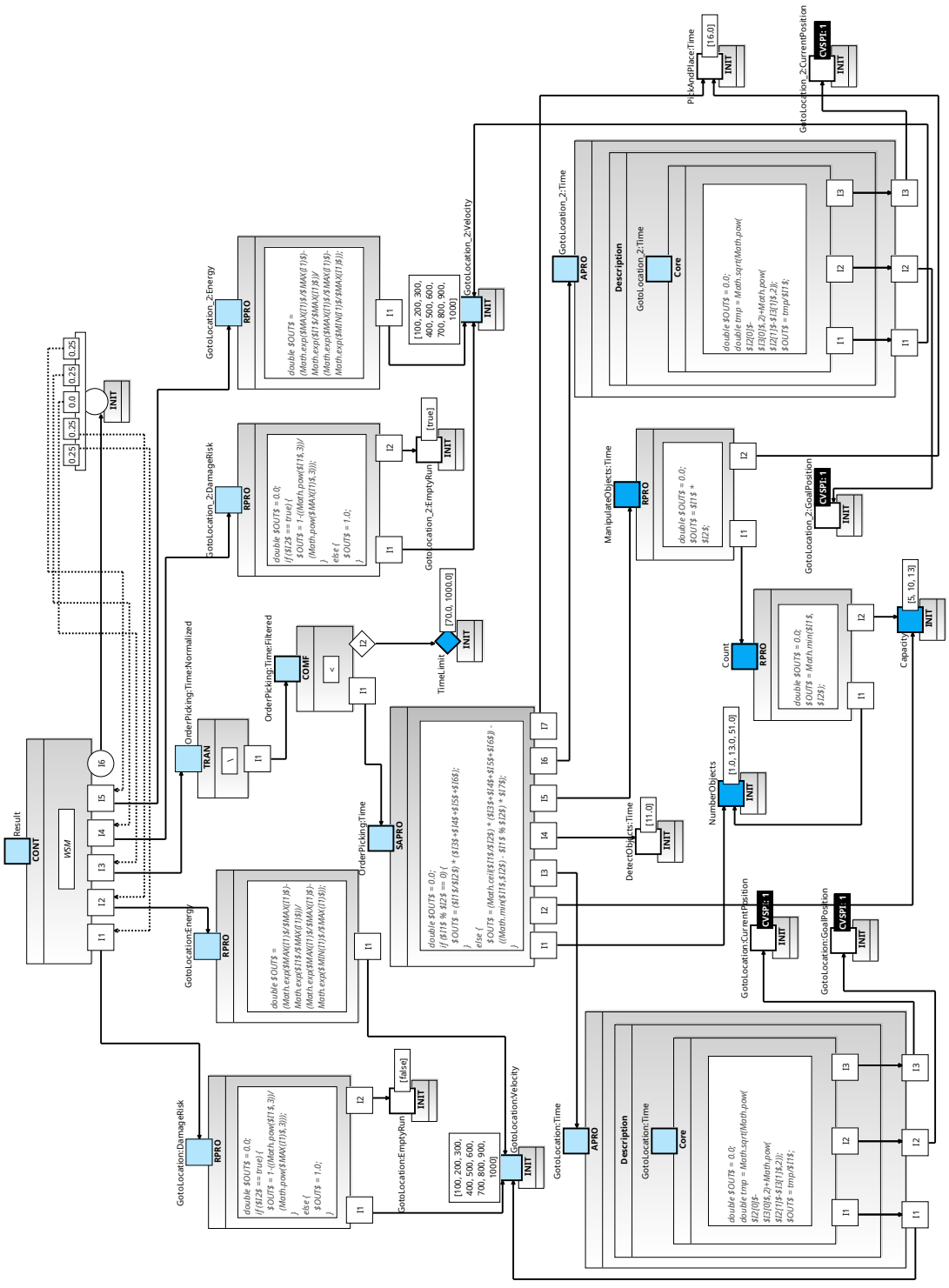


Figure 8.12: The DVG model of the `OrderPicking` building block (VIs of complex DOs (`CVSPI`) are not displayed in the graphical model).

change between different loop instances i (equation 8.7 and 8.8). Therefore, we do not need to model separate port instances in the DVG system model representing different VIs for the `GotoLocation`/`GotoLocation_2`-specific *VariabilityEntit*ies for each i-th loop execution. Note that this model also assumes that the current position of the robot when the order is placed is `To`.

$$\forall 0 \leqq i < \lceil n \cdot c^{-1} \rceil : (cp_{g_0^i} = To) \wedge (gp_{g_0^i} = From) \tag{8.7}$$

$$\forall 0 \leqq i < \lceil n \cdot c^{-1} \rceil : (cp_{g_1^i} = From) \wedge (gp_{g_1^i} = To) \tag{8.8}$$

Damage Risk

The DVG system model also describes the `DamageRisk` for the transported objects of both Goto building blocks. We defined a specific *Expression* describing a decreasing `DamageRisk` utility value (relative representation) with an increasing `Velocity` of the robot. But this applies only if the robot is currently transporting objects, i.e. if `EmptyRun` is `false`. If `EmptyRun` is `true`, the `DamageRisk` of transported objects is not present at all and therefore the `DamageRisk` utility value is set to `1.0`. The possible VIs for the VSP of `EmptyRun` is basically restricted to two values due to its boolean type. An isolated view of a `GotoLocation` building block without any further context would require to consider both potential value occurrences for a full analysis. However, in this specific application context of this building block, we always know that `EmptyDrive = true` will apply for `GotoLocation` while `EmptyDrive = false` will apply for `GotoLocation_2`. Therefore we can directly reduce the VSP of `EmptyRun` for both `GotoLocation` building block instances to a constant value.

Results

In this example the two `Velocity` parameters of the two `GotoLocation` building blocks represent the elements of the CONSSs for which adequate variant bindings have to be determined before the execution of the `OrderPicking` building block. The model allows the specification of a maximum `Time` limit for the overall `OrderPicking` building block as the prioritized requirement. We can then perform a multidimensional analysis including varying values for `Velocity`, constraints of `Time`, `NumberObjects`, `Capacity` and different `From`/`To` locations in order to determine satisfiable and adequate combinations. The general decision-making behavior of this example can be generally described as follows: It will be tried to save `Energy` and reduce the `DamageRisk` as good as possible based on the available `Time` limit, i.e. it will be searched for the minimum `Velocity` value bindings that allow to fulfill the `Time` limit. In the distribution process of these values to the two `GotoLocation` building blocks, higher `Velocity` value bindings will be basically preferred for the first instance (`GotoLocation`) because of the not present `DamageRisk` there.

Note, that the presented `Time` estimate model of the `GotoLocation` building blocks is quite simple and optimistic and therefore does not guarantee that an execution of `OrderPicking` with the determined `Velocity` value bindings will also fulfill the specified `Time` limit. However, accuracy is not the focus of this use case and not of this work in general. The decisive argument here is: The more effort is invested by a community participating in a robotics software ecosystem to determine better building block models (i.e. for example variability dependencies and better *ValueCOp*s), the better the results of a modeled DVG system when these more precise building block models are easily included by reuse.

In [BS20] we provide a similar use case based on a more accurate Time model for the GotoLocation building blocks but where only one common Velocity port in the DVG system model is considered for both GotoLocation building block instances. The work also describes a more complex Time composition model for the overall OrderPicking building block. Note that the OrderPicking use case presented here can be easily linked with the previous Navigation building block by substituting the GotoLocation building block models with the Navigation building block model. The result would be an even more powerful DVG system model because Navigation explicates a larger CONSS. The meaning of Velocity described here as an isolated abstraction can be directly substituted with DesTraVel of the CDL building block. However, for complexity reasons we presented this use case independently to focus specially on the planning aspect of this example (assigning values to a CONSS before the execution of the associated building blocks) and also to show that DVG system models can be described at different abstraction levels according to the desired purpose.

8.4 Collaborative Task: Shared Order Picking

In the last use case, we illustrate a collaboration of two service robots executing an order picking task because one alone does not have all the capabilities to execute the whole task. A general representation of the workflow is given in figure 8.13 where we can see two task nets (Transportation and Manipulation) based on two systems executed on two different service robots (Robotino and Larry, see figure 8.14).

8.4.1 The Building Block Model

A corresponding representation of the general workflow from figure 8.13 based on our building block meta model is shown in figure 8.15. Each building block has the property Time assigned representing the estimated time duration of the associated task. Note that each building block is either part of Transportation or part of Manipulation which is described in the following. The same service robot can not be assigned to building blocks part of both Transportation and Manipulation. Transportation can only be executed by Robotino and Manipulation can only be executed by Larry.

SharedOrderPicking is a *Parallel* decomposition into CollectBox and ApproachAndDetect (policy: WAITFORALL). CollectBox is part of Transportation and ApproachAndDetect is part of Manipulation.

The task of CollectBox is to fetch an empty transport container in which the ordered objects will be later placed and delivered. It is a *Sequential* decomposition of GotoLocation, DockToStation, LoadFromStation and UndockFromStation. The CurrentPosition of GotoLocation is the current location of Robotino (assigned to Transportation) and the GoalPosition is the location of the corresponding station (PickUpStation) from where the empty transport container can be fetched. Before we can LoadFromStation, we need to execute DockToStation. Finally UndockFromStation must be executed.

ApproachAndDetect is a *Sequential* decomposition of GotoLocation_2 and DetectObjects. The CurrentPosition of GotoLocation_2 is the current location of Larry (assigned to Manipulation) and the GoalPosition is the location from where the ordered objects should be picked (PickingPlace). DetectObjects determines the object poses of the objects to be picked. Note that we assume that the number of detected objects is always at least as

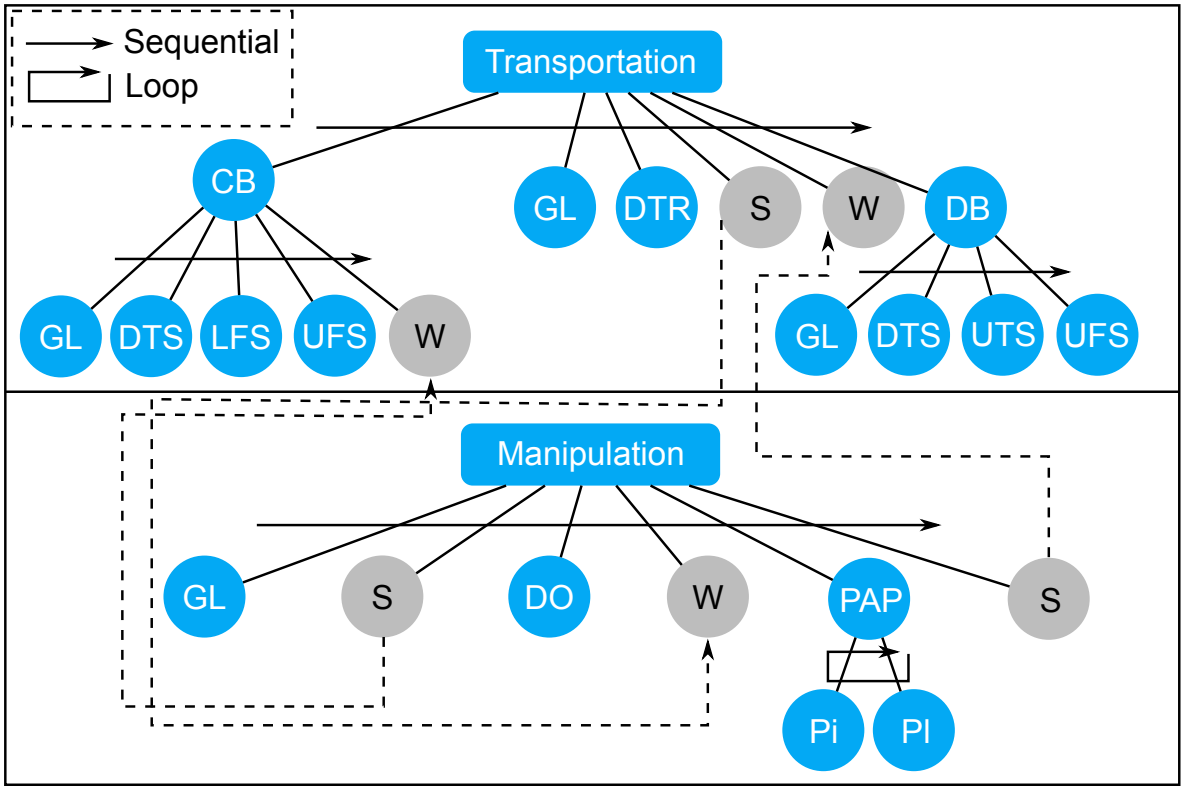


Figure 8.13: General representation of the workflow based on two task nets (Transportation and Manipulation) executed on two different service robots (Robotino and Larry) with special synchronization actions W (Wait) and S (Signal) between the two task nets. CB = CollectBox, GL = GotoLocation, DTS = DockToStation, LFS = LoadFromStation, UFS = UndockFromStation, DTR = DockToRobot, DB = DeliverBox, UTS = UnloadToStation, DO = DetectObjects, PAP = PickAndPlace, Pi = Pick, Pl = Place.



Figure 8.14: The service robots available in this example. Larry (left) is able to handle the Manipulation task, while Robotino is able to handle the Transportation task (Image: Courtesy Forschungsgruppe Servicerobotik).

high as the number of ordered objects.

After the parallel execution of `CollectBox` and `ApproachAndDetect`, `GotoLocation_3` follows. It is part of Transportation and means that Robotino that just fetched the empty transport container from the `PickUpStation` now approaches the `PickingPlace` location where Larry (assigned to the Manipulation part) is already located. The `CurrentPosition` of `GotoLocation_3` is therefore `PickUpStation` and the `GoalPosition` is `PickingPlace`. After the `PickingPlace` is reached, Robotino must `DockToRobot`, i.e. dock to Larry. `PickAndPlace` must then be executed by Larry in order to `Pick` the detect objects and `Place` it in the empty transport container located on Robotino. After all ordered objects are placed in the transport container, `DeliverBox` must be executed.

`DeliverBox` is part of Transportation (executed by Robotino) and is a *Sequential* decomposition of `GotoLocation_4`, `DockToStation`, `UnloadToStation` and `UndockFromStation`. `GotoLocation_4` approaches a station (`DeliveryStation`) where the transport container can be delivered to. The `CurrentPosition` of `GotoLocation_4` is `PickingPlace` and the `GoalPosition` is `DeliveryStation`. Before `UnloadToStation`, `DockToStation` must be executed. Finally, `UndockFromStation` must be executed.

8.4.2 The DVG Model

The equations for the composition of the overall time of `SharedOrderPicking` ($t_{SharedOrderPicking}$) are shown in equations 8.9-8.18. As in the previous example, we use the optimistic time estimation model in form of the straight line distance for the `GotoLocation` building blocks. The DVG model realizing that time composition model is shown in figure 8.16.

$$t_{SharedOrderPicking} = \min(t_{CollectBox}, t_{ApproachAndDetect}) + t_{GotoLocation_3} + t_{DockToRobot} + t_{PickAndPlace} + t_{DeliverBox} \tag{8.9}$$

$$t_{CollectBox} = t_{GotoLocation} + t_{DockToStation} + t_{LoadFromStation} + t_{UndockFromStation} \tag{8.10}$$

$$t_{PickAndPlace} = NumberObjects \cdot (t_{Pick} + t_{Place}) \tag{8.11}$$

$$t_{DeliverBox} = t_{GotoLocation_4} + t_{DockToStation} + t_{UnloadToStation} + t_{UndockFromStation} \tag{8.12}$$

$$t_{ApproachAndDetect} = t_{GotoLocation_2} + t_{DetectObjects} \tag{8.13}$$

$$cp_{GotoLocation} = \text{Current location of Robotino} \tag{8.14}$$

$$cp_{GotoLocation_2} = \text{Current location of Larry} \tag{8.15}$$

$$gp_{GotoLocation} = cp_{GotoLocation_3} = \text{Location of } \texttt{PickUpStation} \tag{8.16}$$

$$gp_{GotoLocation_2} = gp_{GotoLocation_3} = cp_{GotoLocation_4} = \text{Location of } \texttt{PickingPlace} \tag{8.17}$$

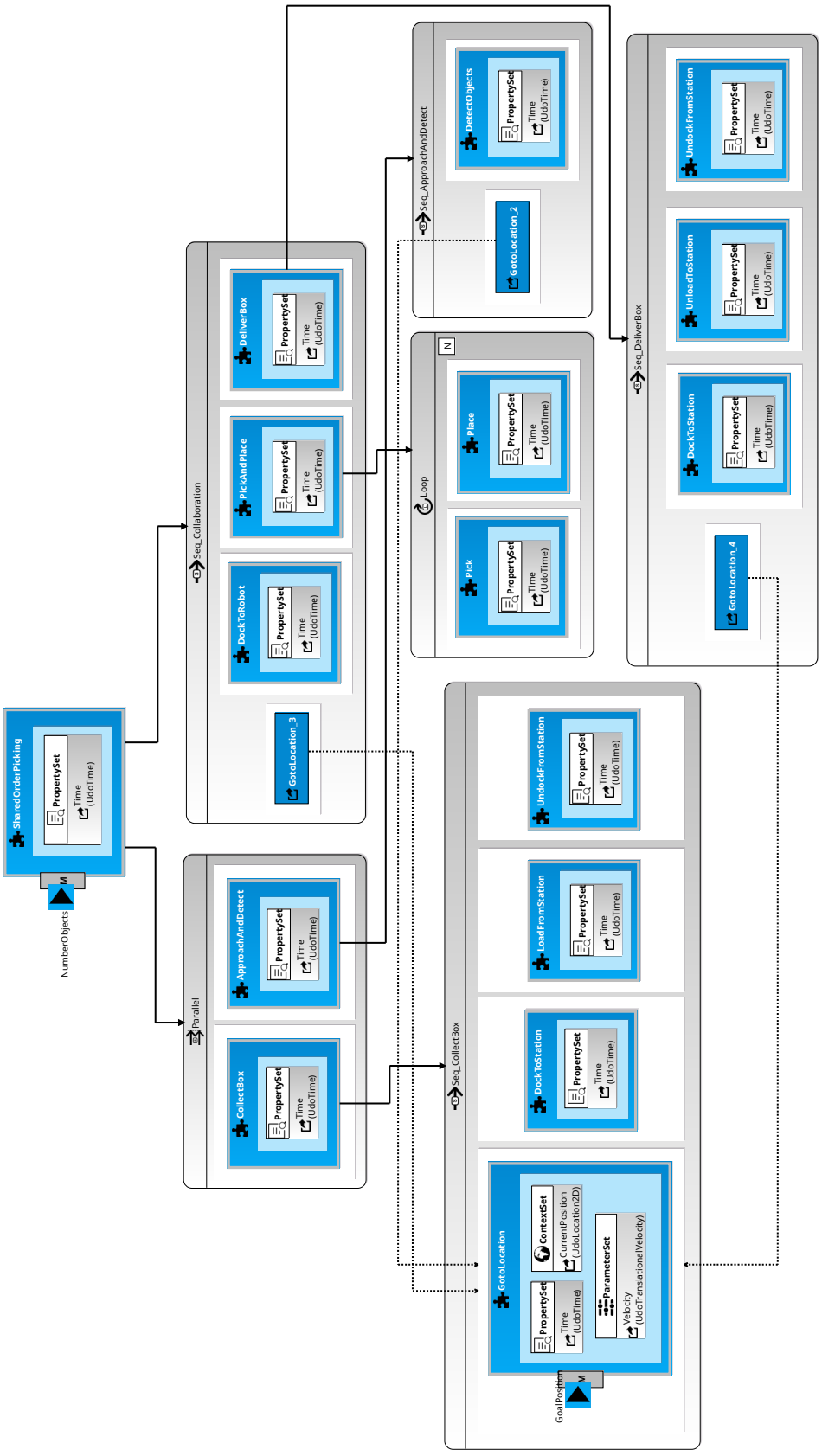


Figure 8.15: The building block model of `SharedOrderPicking`.

$$gp_{GotoLocation_4} = \text{Location of } \texttt{DeliveryStation} \tag{8.18}$$

The model offers two possible applications as shown in the following.

Application 1: Compliance with Time Limit Depending on Velocity Variability Primary goal specified in this application of the model is again the compliance with a specific time limit in dependence of the velocity to be selected for the different `GotoLocation` building block instances. For Robotino we consider 12 possible velocities that must be selected for three independent `GotoLocation` building block instances while for Larry we consider 10 possible velocities that must be selected for one `GotoLocation` building block instance. Therefore, we have to consider $12^3 \cdot 10$ velocity combinations overall.

In this example, we do not model energy explicitly. However, we determine the combination with the lowest velocity values able to fulfill the time requirement by selecting from the valid combinations (< time limit) the one with the highest time. Thus, energy is considered implicitly.

Application 2: Minimizing Time Depending on Location Variability With a few small changes, the example can be used for another purpose: Let us suppose that there are several `PickUpStation`(s) and/or `PickingPlace`(s) and/or `DeliveryStation`(s) with different locations and the service robot can choose which to approach. This means that the locations are not modeled as constant but as active. We are then interested in the combination of a `PickUpStation`, `PickingPlace` and `DeliveryStation` in dependence of the current position of the respective service robot that minimizes the overall time. In that case, the active velocity considered for each `GotoLocation` building block instance can be reduced to constant with the respective maximum velocity of the service robot as the only value because others are out of question if it is about minimizing time.

Experiment Figures 8.17-8.19 show excerpts of the real-world execution of `SharedOrderPicking` with Robotino and Larry. Using application 1 of the described model, it is possible to automatically parameterize the desired velocities for the four different `GotoLocation` building block instances by taking into account the time requirement, the current positions of the service robots when a order is placed and the locations of `PickUpStation`, `PickingPlace` and `DeliveryStation`. All what the user has to do is to specify the time limit. The experiment illustrates the significant simplification of the development process of service robotic applications that are now also able to meet non-functional requirements (of course always in relation to the quality of the model which depends on the invested effort in the modeling process).

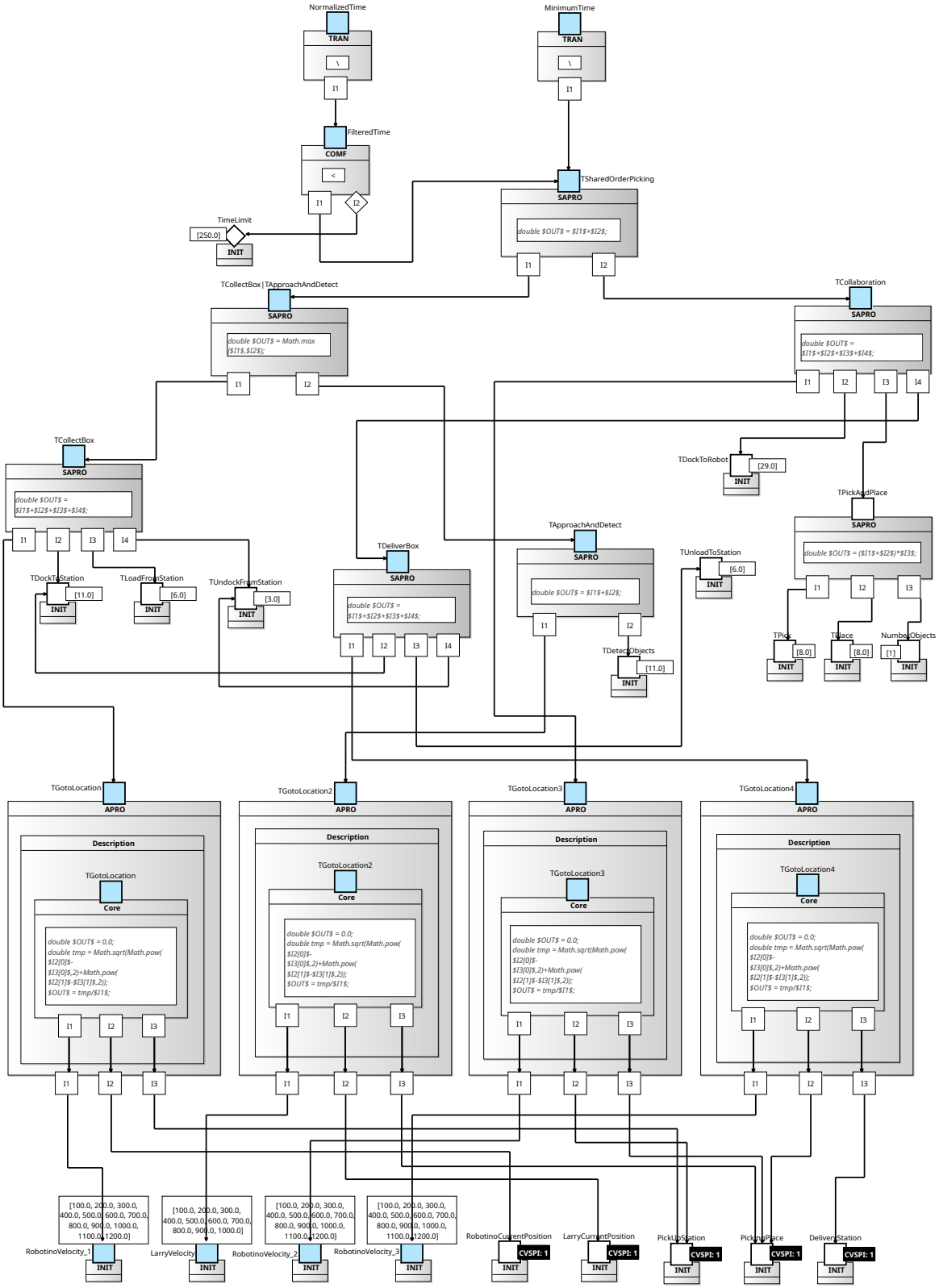


Figure 8.16: The DVG model of `SharedOrderPicking`.



Figure 8.17: Robotino approaching `PickingPlace` (execution of `GotoLocation_3`). The velocity parameter of that building block was determined automatically before the execution of `SharedOrderPicking` started in order to meet a specified time limit while not performing better than required (to save energy).



Figure 8.18: Execution of `PickAndPlace` by Larry.

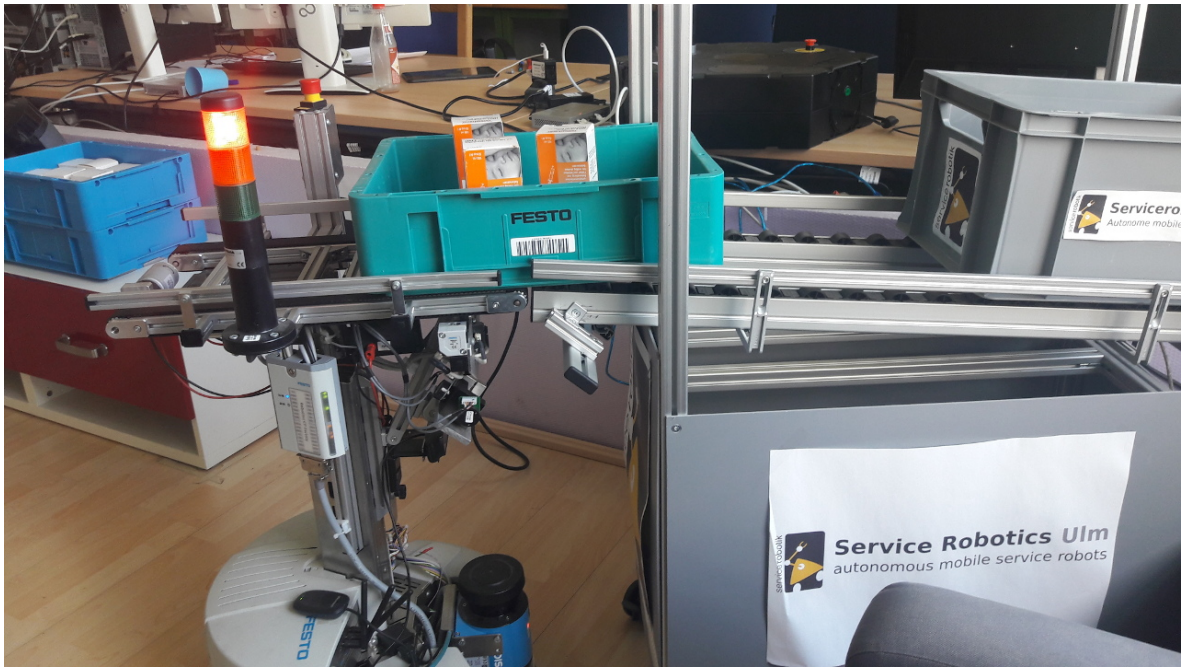


Figure 8.19: Robotino executing `UnloadToStation`.

Chapter 9

Conclusion

The presented DVG method as a formal approach for variability management in a robotics software ecosystem by modeling, composing and resolving VSPs of building blocks is strongly influenced by real robotic building block use cases and the question how to represent adequate models of these building blocks. These models must allow for composition of their properties and the derivation of corresponding customized and at least adequate variant bindings. The concept of VSPs combined with the basic principle of their causality (dependencies) described in form of special graphs is the required foundation to address the intended generality of the approach. To manage the complexity of this variability management approach (both modeling and resolution) and to support separation of roles, we rely on modularization based on the block-port-connector principle. A set of derived block patterns handling their internal variability management can be easily composed via their ports in certain, carefully selected combinations to extend the overall VPS gradually. While it was clear from the very beginning that more complex VPSs can not be optimally resolved efficiently anyway, this was no argument to not come up with such a generic method at all. The DVG method just offers the possibility to explicate every detail but does not impose it. Hence, it can be used at any abstraction level according to the desired needs. The solution procedure that is presented in this work to solve a VPS is in its standard form a brute force approach. Of course, this speaks all the less in favor of efficient solvability but the realization of this approach is the decisive key to get a comprehensive understanding about the overall general problem because there is an undeniable, theoretical truth in this approach: If we know for every possible state combination of a world about the resulting properties of our actions and their utility with respect to the current requirements, we can select appropriate actions at every specific moment in time for a specific state combination that currently occurs. Based on that brute force approach as basis it was then possible to derive additional strategies for reducing the time complexity: (i) the lookup principle at the cost of space complexity, (ii) iterative random sampling with local upper bounds at the cost of completeness and (iii) equivalence filtering that has no drawbacks if applied in allowed and appropriate situations. Furthermore, the decomposition granularity of building blocks, the explication granularity of variability entities and the discretization accuracy of their VSPs are always suitable customization options to make a solution feasible while it remains useful and still has added value. The brute force approach is also the ideal choice for a holistic compositional analysis because we get presented the whole overview about all possible combinations and can therefore derive the effects of adding/removing/exchanging/re-configuring building blocks, adding further world states and requirement states, etc.

The implementation of the DVG method in form of a meta model in combination with a meta model for modeling building blocks represents the blueprint for building block modeling in the context

of a robotics software ecosystem. The provided DSLs based on these meta models can be used to create model instances of building blocks (to model the decomposition structure of building blocks and their variability entities) and their DVG systems (to model the dependencies and relationships between the VSPs of the variability entities) and to compose different DVG system models of different building blocks to support reusability. An implemented code generator allows to produce a solver of the DVG system models that is able to determine customized variant bindings for all kinds of modeled situations and requirements.

9.1 Future Work

There is a lot of interesting work to do in the future. Hence, some relevant points and food for thought are described in the following. In general, the contributed methodology is assumed to be generic enough in its structural aspects so that all topics described in the following can be considered as extensions that can be integrated and implemented with more or less effort but without breaking existing fundamental structural aspects.

9.1.1 Extended Production (EPROD)

The EPROD is not the smartest way with respect to the resolution time to filter invalid combinations which arise due to a SAMS in a DVG system but at least it ensures completeness. It is not clear whether there is a more clever approach to filter invalid combinations without the need to check every combination locally again or whether that is just the price to pay for our modularization into independent blocks. Nevertheless, the general structure of DVG systems based on the arrangements of the presented block-port modeling patterns and their general meaning is independent of possible other, more sophisticated system resolution procedures that may be added in the future.

9.1.2 Complex Data Objects

We assume that for complex data objects (containing more than one *Element*), all the *Element*s must be present (AND relationship). We should consider the application of other relationships here and how they can be integrated into the existing method.

9.1.3 Meta Properties

This work shows that modeling real robotic building blocks can be a challenging task. Hence, the process of modeling can represent a trade-off in itself. This leads to the consideration that models should actually have some kind of meta properties assigned to make this trade-off more explicit. Possible meta properties associated with output ports of a DVG model could be for example:

- The RT of an output port since absolute representations are more meaningful than relative representations.
- Accuracy of an output port since absolute representations alone do not guarantee consistency. This would require some ground truth.
- Time of resolving an output port.
- Generality of an output port defined by the number of preconditions.

No clarity about such meta properties makes inter-variability management so challenging. For that reason, defining abstract DVG models at tier 2 in a robotics software ecosystem can make sense to enable comparability between different building block providers.

9.1.4 Improving Planning Capabilities

In the last use case we illustrated a form of planning, i.e. we determined the bindings of a CONSSs for associated building blocks before their execution. We need to improve that mechanism for cases in which the exact order of building block executions is not known beforehand (*Unsequenceable* decomposition). Such situations can not be modeled at design-time and we need to come up with an adequate method to do that at run-time in an iterative way after a lot of variability has been already reduced. In case of planning we also have to deal with predicting states of passive variability in a dynamic environment for building blocks that are executed in the future. We need to find out how such prediction methods can be realized and combined with the existing method.

9.1.5 Probability

A passive VSP represents a set of possible states in a dynamic environment that can occur. These states are not always equally likely to occur. However, this is our current implicit assumption in our implementation for all passive ports (discrete uniform distribution). Hence, it should be checked how different probability distributions can be considered by the existing method.

9.1.6 Causality Cycles

So far we have a clear distinction into active and passive variability in the DVG method and we always describe active variability (CONSS) in dependence of passive variability (STASS). However, from a theoretical causality perspective, there may be situations where the bindings for some elements in the STASS occurring at some points in the future $t + i$ can depend on the choices over the CONSS we make at the respective current point in time t. At the moment, we do not support such cases. An interesting approach could be to try to model such dependencies in another extended model and to link both in order to model causality cycles between active and passive variability.

9.1.7 Integration in SmartSoft

Our presented meta models and the corresponding DSLs for modeling concrete building blocks and DVG systems are only prototype implementations to support the formal contributions of this work. They are not designed to be applied directly in an existing, matured model-driven robotics framework such as the *SmartSoft* world [1] although we used the current functionality of *SmartSoft* as the basis of what is already there and what still needs to be addressed in such an environment. In that sense, the building block models can be considered as further abstractions of component and system models in *SmartSoft*. That is for example the reason, why we do not need to wire ports of building blocks that represent components in our building block model, because we know that this is already done in a corresponding system model already existing in *SmartSoft*. However, the full-fledged technical integration into *SmartSoft* still has to be done in the future. Furthermore, our presented DSLs must be most likely further narrowed down in their complexity and tailored to more fine-grained roles in the development process of a building block. In that context, specific levels of building blocks (function,

[1] https://wiki.servicerobotik-ulm.de/smartmdsd-toolchain:start

component task, component, system/skill, system/task) may become more explicit again if special characteristics are identified that can reasonably justify such an explication and distinction. All that depends on a thorough analysis of different aspects and will take some time.

Furthermore, the run-time resolution process of the DVG method must be combined with *SmartTCL* [2] as the central element of run-time execution coordination to finally configure the determined variant bindings in the corresponding components and systems.

9.1.8 Systematic Support of Multi-Robot Tasks

With multi-robot tasks we mean that either multiple robot resources and/or multiple tasks are available as variabilities that need to be matched against each other. Applying the method from this work at such a level requires some extensions because we need to make sure that for specific task requirements, valid combinations of robot resources based on their set of provided capabilities are determined and assigned to the different tasks. Furthermore, we need independence between a task requirement model and the potential solution models and a dynamic integration of them in the resolution process. In [BS22] we describe a use case with multiple robot resources that need to be assigned to an order picking task which must be executed by a single robot or by a collaboration of two robots (it combines in a certain way use case 8.3 and 8.4). For that we extended the DVG part of the overall method to check the capabilities of the robots and to manage the allocation combinatorics according to the task model consistently. Based on the experiences made, we restructured some of these extensions and moved some of them to the building block part of the overall method in order to better support composition and separation of roles and to simplify the modeling process. The video https://youtu.be/U0qWJbcsI6s gives an impression of what is possible now based on these restructured extensions with respect to multi-robot tasks.

9.2 Final Remark

The general aim of a robotics software ecosystem is to significantly reduce the efforts/costs of implementing tailored service robotic applications. By this, the use of robotics technology becomes economically feasible/affordable for many people. This work contributes a general method for variability management in a robotics software ecosystem to address this general aim. The formal DVG method and the implementation/integration using MDSD by considering important principles of a robotics software ecosystem (composition of building blocks and separation of roles) are important contributions with respect to the sketched idealized, high-level vision of an advanced robotics software ecosystem: Determining automatically customized and flexible service robotic systems/applications by a suitable composition and configuration of available building blocks.

[2] https://wiki.servicerobotik-ulm.de/about-smartsoft:robotic-behavior:smarttcl

Bibliography

[ABG+13] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.

[ACL+11] M. Acher, P. Collet, P. Lahire, S. Moisan, and J. Rigault. Modeling variability from requirements to runtime. In *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 77–86, April 2011.

[ARN12] Mohammad Alrifai, Thomas Risse, and Wolfgang Nejdl. A hybrid approach for efficient web service composition with end-to-end qos constraints. *ACM Trans. Web*, 6(2):7:1–7:31, June 2012.

[BAH19] Pascal Bercher, Ron Alford, and Daniel Höller. A survey on hierarchical planning - one abstract idea, many concrete realizations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 6267–6275. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[Bar99] Roman Barták. Constraint programming: In pursuit of the holy grail. *Proceedings of WDS99 (invited lecture)*, 01 1999.

[Bat05] Don Batory. Feature models, grammars, and propositional formulas. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, pages 7–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BBD17] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. Dynamic software product line engineering: A reference framework. *International Journal of Software Engineering and Knowledge Engineering*, 27:191–234, 03 2017.

[BBF+16] Timo Blender, Thiemo Buchner, Benjamin Fernandez, Benno Pichlmaier, and Christian Schlegel. Managing a mobile agricultural robot swarm for a seeding task. In *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, pages 6879–6886, 2016.

[BC05] Felix Bachmann and Paul Clements. Variability in software product lines. Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.

[BCMT18] D. Brugali, R. Capilla, R. Mirandola, and C. Trubiani. Model-Based Development of QoS-Aware Reconfigurable Autonomous Robotic Systems. In *2018 Second IEEE International Conference on Robotic Computing (IRC)*, pages 129–136, Jan 2018.

[BdCM05] E. Bondarev, P. de With, M. Chaudron, and J. Muskens. Modelling of input-parameter dependency for performance predictions of component-based embedded systems. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 36–43, Aug 2005.

[Ber69] Ludwig Von Bertalanffy. *General System Theory: Foundations, Development, Applications*. George Braziller, 1969.

[BGC+08] Nelly Bencomo, Paul Grace, Flores-Cortes Carlos, Danny Hughes, and Gordon S. Blair. Genie: supporting the model driven development of reflective, component-based adaptive systems. pages 811–814, 01 2008.

[BKP15] Mieczysław Borysiewicz, Karol Kowal, and Sławomir Potempski. An application of the value tree analysis methodology within the integrated risk informed decision making for the nuclear facilities. *Reliability Engineering & System Safety*, 139:113–119, 2015.

[BM04] Antonia Bertolino and Raffaela Mirandola. Cb-spe tool: Putting component-based performance engineering into practice. In Ivica vic, Judith A. Stafford, Heinz W. Schmidt, and Kurt Wallnau, editors, *Component-Based Software Engineering*, pages 233–248, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BPS99] Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European journal of operational research*, 119(3):557–581, 1999.

[Bro86] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1):14–23, 1986.

[Bru18] D. Brugali. Modeling and Analysis of Safety Requirements in Robot Navigation with an Extension of UML MARTE. In *2018 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, pages 439–444, Aug 2018.

[BS15] T. Blender and C. Schlegel. Motion control for omni-drive servicerobots under kinematic, dynamic and shape constraints. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, 2015.

[BS20] T. Blender and C. Schlegel. Implementing resource adequate service robot behavior by systematic management of non-functional properties: An intralogistics use case. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 659–666, 2020.

[BS22] Timo Blender and Christian Schlegel. Dynamic allocation of service robot resources to an order picking task considering functional and non-functional properties. In *2022 IEEE/ACM 4th International Workshop on Robotics Software Engineering (RoSE)*, pages 25–32, 2022.

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, Berlin, Heidelberg, 2005. Springer-Verlag.

[Che08] Shang-Wen Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, USA, 2008. AAI3305807.

[Cle96] Robert T. Clemen. *Making Hard Decisions: An Introduction to Decision Analysis*. Duxbury Press, 1996.

[CLP05] Ivica Crnkovic, Magnus Larsson, and Otto Preiss. Concerning predictability in dependable component-based systems: Classification of quality attributes. In *Architecting Dependable Systems III*, pages 257–278. Springer, 2005.

[DKS+12] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, SIMPAR'12, pages 149–160, Berlin, Heidelberg, 2012. Springer-Verlag.

[euR15] euRobotics aisbl. Robotics 2020 Multi-Annual Roadmap (MAR) For Robotics in Europe Horizon 2020 Call ICT-2016 (ICT-25 & ICT-26). Release B 03/12/2015, 2015.

[FBT97] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine*, 4(1):23–33, 1997.

[FGH06] Peter Feiler, David Gluch, and John Hudak. The architecture analysis & design language (aadl): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[FHS+06] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, March 2006.

[Fir89] Robert James Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, USA, 1989. AAI9010653.

[Fou21a] Eclipse Foundation. Sirius website. URL: http://www.eclipse.org/sirius/ (visited: April 26, 2021).

[Fou21b] Eclipse Foundation. Xtend website. URL: http://www.eclipse.org/xtend/ (visited: April 26, 2021).

[Fou21c] Eclipse Foundation. Xtext website. URL: http://www.eclipse.org/Xtext/ (visited: April 26, 2021).

[FS09] Franck Fleurey and Arnor Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, pages 606–621, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[GA15] Ilche Georgievski and Marco Aiello. Htn planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156, 2015.

[Gat91] Erann Gat. *Reliable goal-directed reactive control of autonomous mobile robots*. PhD thesis, Virginia Tech, 1991.

[GBE+09] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav. A comprehensive solution for application-level adaptation. *Softw. Pract. Exper.*, 39(4):385–422, March 2009.

[GH15] L. Gherardi and N. Hochgeschwender. RRA: Models and Tools for Robotics Run-time Adaptation. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1777–1784, Sep. 2015.

[Ghe13] Luca Gherardi. *Variability Modeling and Resolution in Component-based Robotics Systems*. PhD thesis, 04 2013.

[GPST13] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 33–42, May 2013.

[GSB+19] Sergio García, Daniel Strüber, Davide Brugali, Alessandro Di Fava, Philipp Schillinger, Patrizio Pelliccione, and Thorsten Berger. Variability modeling of service robots: Experiences and challenges. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[GSC09] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. *Software Architecture-Based Self-Adaptation*, pages 31–55. Springer US, Boston, MA, 2009.

[GSG01] Janne Gustafsson, Ahti Salo, and Tommi Gustafsson. Prime decisions: An interactive tool for value tree analysis. In Murat Köksalan and Stanley Zionts, editors, *Multiple Criteria Decision Making in the New Millennium*, pages 165–176, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[GSSC15] G. Guedes, C. Silva, M. Soares, and J. Castro. Variability management in dynamic software product lines: A systematic mapping. In *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software*, pages 90–99, Sep. 2015.

[HGP+12] S. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G.A. Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *Journal of Systems and Software*, 85(12):2840 – 2859, 2012. Self-Adaptive Systems.

[HHPS08] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[HSMK09] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A case study in goal-driven architectural adaptation. In *Software Engineering for Self-Adaptive Systems*, pages 109–127. Springer, 2009.

[HSSF06] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *10th International Software Product Line Conference (SPLC'06)*, pages 10 pp.–150, 2006.

[ibm05] An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.

[ILVS12] Juan Francisco Inglés-Romero, Alex Lotz, Cristina Vicente-Chicote, and Christian Schlegel. Dealing with Run-Time Variability in Service Robotics: Towards a DSL for Non-Functional Properties. In *3rd International Workshop on Domain-Specific Languages and models for Robotic systems (DSLRob-12), Proceedings of SIMPAR 2012 Workshop*, Tsukuba, Japan, November 2012.

[IRVC13] Juan F. Inglés-Romero and Cristina Vicente-Chicote. Towards a formal approach for prototyping and verifying self-adaptive systems. In Xavier Franch and Pnina Soffer, editors, *Advanced Information Systems Engineering Workshops*, pages 432–446, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[KBR+18] Christian Krupitzer, Martin Breitbach, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems (extended version), 2018.

[KC03] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan 2003.

[KCH+90] K. C. Kang, S. Cohen, J. A. Hess, William E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. 1990.

[KD07] J. O. Kephart and R. Das. Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[Kle75] Leonard Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, USA, 1975.

[Koz08] Heiko Koziolek. *Parameter dependencies for reusable performance specifications of software components*. PhD thesis, 2008.

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32–32, 1992.

[LdH15] P. Leserf, P. de Saqui-Sannes, and J. Hugues. Multi domain optimization with sysml modeling. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, 2015.

[LdHC15] P. Leserf, P. de Saqui-Sannes, J. Hugues, and K. Chaaban. Sysml modeling for embedded systems design optimization: A case study. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 449–457, 2015.

[LdSSH19] Patrick Leserf, Pierre de Saqui-Sannes, and Jérôme Hugues. Trade-off analysis for sysml models using decision points and csps. *Software and Systems Modeling*, 18(6):3265–3281, 2019.

[LIRVCS13] Alex Lotz, Juan F. Inglés-Romero, Cristina Vicente-Chicote, and Christian Schlegel. Managing run-time variability in robotics software by modeling functional and non-functional behavior. In Selmin Nurcan, Henderik A. Proper, Pnina Soffer, John Krogstie, Rainer Schmidt, Terry Halpin, and Ilia Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 441–455, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Lot18] Alex Lotz. *Managing Non-Functional Communication Aspects in the Entire Life-Cycle of a Component-Based Robotic Software System*. Dissertation, Technische Universität München, München, 2018.

[Man16] Masoumeh Mansouri. *A Constraint-Based Approach for Hybrid Reasoning in Robotics*. PhD thesis, Örebro University, School of Science and Technology, 2016.

[MAR19] MARTE. UML Profile for MARTE™: Modeling and Analysis of Real-Time Embedded Systems, Version 1.2. OMG Document formal/19-04-01 (http://www.omg.org/spec/MARTE/1.2/), 2019.

[MBJ+09] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[MH99] Jyri Mustajoki and Raimo P. Hämäläinen. Web-hipre - a java applet for ahp and value tree analysis. In *5th International Symposium on the Analytic Hierarchy Process (ISAHP'99*, pages 12–14, 1999.

[MH00] Jyri Mustajoki and R.P. Hämäläinen. Web-hipre: Global decision support by value tree and ahp analysis. *INFOR*, 38(3):208–220, 2000.

[MMW05] Marcus Meyerhöfer and Klaus Meyer-Wegener. Estimating non-functional properties of component-based software based on resource consumption. *Electr. Notes Theor. Comput. Sci.*, 114:25–45, 01 2005.

[MP16] Masoumeh Mansouri and Federico Pecora. A robot sets a table: a case for hybrid reasoning with different types of knowledge. *Journal of Experimental & Theoretical Artificial Intelligence*, 28(5):801–821, 2016.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[OGT+99] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. F. Johnson, N. Medvidovic, A. Quilici, David S. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intell. Syst.*, 14:54–62, 1999.

[PBJFG+97] R Peter Bonasso, R James Firby, Erann Gat, David Kortenkamp, David P Miller, and Mark G Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3):237–256, 1997.

[Pir97] Paolo Pirjanian. *An overview of system architectures for action selection in mobile robotics*. LIA Report. Aalborg University, Laboratory of Image Analysis, 1997.

[Pir98] Paolo Pirjanian. *Multiple objective action selection & behavior fusion using voting*. PhD thesis, 1998.

[PKWB17] M. Pfannemueller, C. Krupitzer, M. Weckesser, and C. Becker. A dynamic software product line approach for adaptation planning in autonomic computing systems. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 247–254, July 2017.

[RMT14a] A. Ramaswamy, B. Monsuez, and A. Tapus. Saferobots: A model-driven framework for developing robotic systems. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1517–1524, Sep. 2014.

[RMT14b] Arun Kumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. Solution Space Modeling for Robotic Systems. *Journal for Software Engineering Robotics (JOSER)*, 5(1):89–96, 2014.

[RMT15] A. Ramaswamy, B. Monsuez, and A. Tapus. Model-driven self-adaptation of robotics software using probabilistic approach. In *2015 European Conference on Mobile Robots (ECMR)*, pages 1–6, Sep. 2015.

[RMT20] Arunkumar Ramaswamy, Bruno Monsuez, and Adriana Tapus. *Formal Specification of Robotic Architectures for Experimental Robotics*, pages 15–37. Springer International Publishing, Cham, 2020.

[rob] RobMoSys EU H2020 Project. (2017-2020). RobMoSys: Composable Models and Software for Robtics Systems - Towards an EU Digital Industrial Platform for Robotics.

[Saa77] Thomas L Saaty. A scaling method for priorities in hierarchical structures. *Journal of Mathematical Psychology*, 15(3):234–281, 1977.

[Saa90] Thomas L. Saaty. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1):9–26, 1990. Desicion making by the analytic hierarchy process: Theory and applications.

[Sal15] Karsten Saller. *Model-Based Runtime Adaptation of Resource Constrained Devices*. PhD thesis, Technische Universität, Darmstadt, January 2015.

[SBPM08] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley, 2nd edition, 2008.

[Sch98] C. Schlegel. Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot. In *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)*, volume 1, pages 594–599 vol.1, 1998.

[Sch04] Christian Schlegel. *Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach*. Dissertation, University of Ulm, Ulm, 2004.

[SDPZ+15] L. Emiliano Sánchez, J. Andres Diaz-Pace, Alejandro Zunino, Sabine Moisan, and Jean-Paul Rigault. An approach based on feature models and quality criteria for adapting component-based systems. *Journal of Software Engineering Research and Development*, 3(1):10, Jun 2015.

[SDZ+14] L. E. Sánchez, J. A. Diaz-Pace, A. Zunino, S. Moisan, and J. Rigault. An approach for managing quality attributes at runtime using feature models. In *2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse*, pages 11–20, Sep. 2014.

[SE04] Richard Staehli and Frank Eliassen. Compositional quality of service semantics. *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 62, 2004.

[Sen12] Séverine Sentilles. *Managing Extra-Functional Properties in Component-Based Development of Embedded Systems*. PhD thesis, Mälardalen University, Västerås, Sweden, June 2012.

[SFS+14] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc challenge 2008-2013. *AI Magazine*, 35(2):55–60, Jun. 2014.

[SHB+16] Enea Scioni, Nico Huebel, Sebastian Blumenthal, Azamat Shakhimardanov, Markus Klotzbuecher, Hugo Garcia, and Herman Bruyninckx. Hierarchical hypergraph for knowledge-centric robot systems: a composable structural meta model and its domain specific language npc4. *JOSER: Journal of Software Engineering for Robotics*, 7(11):55–74, 2016.

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In Betty H. C. Cheng, Rogério de Lemos, David Garlan, Holger Giese, Marin Litoiu, Jeff Magee, Hausi A. Müller, and Richard N. Taylor, editors, *2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2008, Leipzig, Germany, May 12-13, 2008*, pages 1–8. ACM, 2008.

[SHMK10] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 431–438, 2010.

[SKK+12] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 167–177, 2012.

[SLL+15] Christian Schlegel, Alex Lotz, Matthias Lutz, Dennis Stampfer, Juan Inglés-Romero, and Cristina Vicente-Chicote. Model-driven software systems engineering in robotics: Covering the complete life-cycle of a robot. *it - Information Technology*, 57:85–98, 03 2015.

[SLLS16] Dennis Stampfer, Alex Lotz, Matthias Lutz, and Christian 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)*, 7:3–19, July 2016.

[SLLS21] Christian Schlegel, Alex Lotz, Matthias Lutz, and Dennis Stampfer. *Composition, Separation of Roles and Model-Driven Approaches as Enabler of a Robotics Software Ecosystem*, pages 53–108. Springer International Publishing, Cham, 2021.

[SMPH15] Sebastian Stock, Masoumeh Mansouri, Federico Pecora, and Joachim Hertzberg. Hierarchical hybrid planning in a mobile service robot. In Steffen Hölldobler, Rafael Peñaloza, and Sebastian Rudolph, editors, *KI 2015: Advances in Artificial Intelligence*, pages 309–315, Cham, 2015. Springer International Publishing.

[SRK+11a] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *2011 15th International Software Product Line Conference*, pages 160–169, 2011.

[SRK+11b] Norbert Siegmund, M. Rosenmüller, Martin Kuhlemann, Christian Kästner, S. Apel, and G. Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20:487–517, 2011.

[SS11] A. Steck and C. Schlegel. Managing Execution Variants in Task Coordination by Exploiting Design-Time Models at Run-Time. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, USA, September 2011.

[Sta18] Dennis Stampfer. *Contributions to System Composition using a System Design Process driven by Service Definitions for Service Robotics*. Dissertation, Technische Universität München, München, 2018.

[Ste10] Andreas Steck. Conditional reactive task execution in a three layer architecture for service robots. Master's thesis, University of Applied Sciences Ulm, 2010.

[Sto16] Sebastian Stock. *Hierarchische hybride Planung für mobile Roboter*. PhD thesis, University of Osnabrück, Germany, 2016.

[SVC06] Thomas Stahl, Markus Völter, and Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006.

[SW99] C. Schlegel and R. Worz. The software framework smartsoft for implementing sensorimotor systems. In *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No.99CH36289)*, volume 3, pages 1610–1616 vol.3, 1999.

[Sys19] SysML. OMG Systems Modeling Language (OMG SysML™), Version 1.6. OMG Document formal/19-11-01 (http://www.omg.org/spec/SysML/1.6/), 2019.

[THv18] Byomkesh Talukder, Keith W. Hipel, and Gary W. vanLoon. Using multi-criteria decision analysis for assessing sustainability of agricultural systems. *Sustainable Development*, 26(6):781–799, 2018.

[Tsa95] Edward Tsang. *Foundations of constraint satisfaction*. Academic Press, London, 1995.

[TSSR98] Evangelos Triantaphyllou, B. Shu, S. Sanchez, and Thomas Ray. *Multi-criteria decision making: An operations research approach*, volume 15, pages 175–186. 02 1998.

[VH13] Mark Velasquez and Patrick Hester. An analysis of multi-criteria decision making methods. *International Journal of Operations Research*, 10:56–66, 05 2013.

[VIM+18] Cristina Vicente-Chicote, Juan F. Inglés-Romero, Jesús Martínez, Dennis Stampfer, Alex Lotz, Matthias Lutz, and Christian Schlegel. A component-based and model-driven approach to deal with non-functional properties through global qos metrics. In Regina Hebig and Thorsten Berger, editors, *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC,*

MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018, volume 2245 of *CEUR Workshop Proceedings*, pages 40–45. CEUR-WS.org, 2018.

[Wik21] RobMoSys Wiki. Block-port-connector. URL: https://robmosys.eu/wiki/modeling:principles:block-port-connector (visited: November 23, 2021).

[WTKD04] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 70–77, 2004.

[ZSS18] Qingyun Zhu, Purvi Shah, and Joseph Sarkis. Addition by subtraction: Integrating product deletion with lean and sustainable supply chain management. *International Journal of Production Economics*, 205:201–214, 2018.

[ZSWD98] Stelios H. Zanakis, Anthony Solomon, Nicole Wishart, and Sandipa Dublish. Multi-attribute decision making: A simulation comparison of select methods. *European Journal of Operational Research*, 107(3):507–529, 1998.

[ZV17] Anna Zafeiris and Tamás Vicsek. *Why we live in hierarchies: a quantitative treatise*. Springer, 2017.