

Open Research Online

The Open University's repository of research publications and other research outputs

Addressing Resource Variability Through Resource-Driven Adaptation

Thesis

How to cite:

Akiki, Paul (2023). Addressing Resource Variability Through Resource-Driven Adaptation. PhD thesis The Open University.

For guidance on citations see [FAQs](#).

© 2023 Paul Akiki



<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Version: Version of Record

Link(s) to article on publisher's website:

<http://dx.doi.org/doi:10.21954/ou.ro.000159f5>

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data [policy](#) on reuse of materials please consult the policies page.

oro.open.ac.uk

School of Computing and Communications
Faculty of Science, Technology, Engineering, and Mathematics
The Open University

Addressing Resource Variability Through Resource-Driven Adaptation

Paul A. Akiki

**A thesis submitted in fulfilment of the requirements for the
degree of
Doctor of Philosophy in Computing**

**United Kingdom
March 2023**

To my family:

My father Antoine and my mother Randa

My brother Pierre and my sister Patricia

Abstract

Software systems execute tasks that depend on different types of resources. However, the variability of resources may interfere with the ability of software systems to execute important tasks. Resource variability can occur due to several reasons including unexpected hardware failures, excess workloads, or lack of materials. For example, in automated warehouses, malfunctioning robots could delay product deliveries causing customer dissatisfaction and, therefore, reducing an enterprise's sales. Moreover, the unavailability of medical materials hinders the ability of hospitals to perform medically-critical operations causing loss of life. In this thesis, we propose to address the problem of resource variability through resource-driven adaptation, using task models as input for adaptation decisions. The thesis presents the following contributions:

- **SPARK**: a framework for performing proactive and reactive resource-driven adaptation based on multiple task-related criteria. The framework supports different types of depletable and reusable resources that could face variability. SPARK assists with four types of adaptation, namely: (i) execution of a similar task that requires fewer resources, (ii) substitution of resources by alternative ones, (iii) execution of tasks in a different order, and (iv) cancellation of the execution of tasks.
- **SERIES**: a task modelling notation and editor tool that enables software practitioners to create task models that serve as input for SPARK. SERIES supports the representation of task priorities, task variants, task execution types, resource types, and properties representing users' feedback.

SPARK was evaluated in terms of the percentage of executed critical task requests, the average criticality of the executed task requests in comparison to the non-executed ones, overhead, and scalability through two case studies concerned with a medicine consumption system and a manufacturing system. The results of the evaluation showed that SPARK increased the number of executed critical task requests during

resource variability. Additionally, the results showed that the time it takes to prepare and apply adaptation plans does not add significant overhead that hinders the ability of software systems to execute tasks in a tolerable waiting time. Furthermore, SPARK was shown to be scalable since the abovementioned time increases polynomially relative to the input size (number of tasks and task variants).

SERIES was evaluated through a user study with twenty software practitioners. The results showed that software practitioners performed very well when explaining and creating task models using SERIES. These results were reflected in the task modelling activities that the participants performed as well as in their positive feedback regarding the usability of SERIES and the clarity of its semantic constructs.

Overall, we conclude that the research presented in the thesis contributes to addressing resource variability through resource-driven adaptation. We also provide suggestions for future work that can extend this research.

Author's Declaration

The work presented in this thesis is an original contribution of the author. Parts of this work were published in the following papers.

Book Chapter

P. Akiki, A. Zisman, and A. Bennaceur. **Modelling Software Tasks for Supporting Resource-driven Adaptation**. LNBIP Series, Springer (*under review*)
Chapter 7 - (Akiki, Zisman and Bennaceur, *under review*)

Conference Proceedings

P. Akiki, A. Zisman, and A. Bennaceur. **SERIES: A Task Modelling Notation for Resource-driven Adaptation**. Proceedings of the 24th International Conference on Enterprise Information Systems (ICEIS), SCITEPRESS - Science and Technology Publications, 2022
Chapter 5 - (Akiki, Zisman and Bennaceur, 2022)

Workshop Proceedings

P. Akiki, A. Zisman, and A. Bennaceur. **Work With What You've Got: An Approach for Resource-Driven Adaptation**. International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), IEEE, 2021
Chapter 6 - (Akiki, Zisman and Bennaceur, 2021)

Doctoral Symposium

P. Akiki. **Towards an approach for resource-driven adaptation**. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, 2021
Chapter 4 - (Akiki, 2021)

Acknowledgements

First, I would like to thank my supervisors Prof. Andrea Zisman and Dr Amel Bennaceur for their support and guidance throughout my PhD journey. I am very grateful for their advice and comments, which helped in improving this work.

I would like to thank the SEAD/SPARE research group for their feedback on my work. Moreover, I am grateful to Dr Tamara Lopez, Dr Irum Rauf, Dr Min Zhang, and Dr Vikram Mehta for their valuable feedback that improved the design of my user study. Furthermore, I am grateful to my mini-viva examiners Prof. Yijun Yu and Dr Patrick Wong for their comments.

I would like to thank Prof. Marian Petre and Dr Daniel Gooch for managing the postgraduate forum sessions, which provided support for PhD students in their journey. I am also grateful to all the people who provided feedback on my work during my presentations at the CRC student conferences, in particular Prof. Arosha Bandara, Prof. Marian Petre, and Dr Daniel Gooch.

I would like to thank my third-party monitor Prof. Janet van der Linden for her support during my PhD journey.

I would like to thank everyone who dedicated time to participate in my user study. Your contribution was vital for the completion of this work.

I would like to extend my gratitude to my family, who have encouraged and supported me during my PhD journey.

Finally, I would like to thank The Open University for granting me the funding and opportunity to pursue a PhD degree.

Table of Contents.

ABSTRACT	V
AUTHOR'S DECLARATION	VII
ACKNOWLEDGEMENTS	IX
TABLE OF CONTENTS.....	XI
LIST OF FIGURES.....	XVII
LIST OF TABLES	XXI
LIST OF EQUATIONS	XXIII
LIST OF CODE LISTINGS	XXV
LIST OF ABBREVIATIONS	XXVII
GLOSSARY.....	XXIX

1 Introduction 1

1.1 Problem and motivation	1
1.2 Motivating examples.....	4
1.2.1 A warehouse system.....	4
1.2.2 A manufacturing system	6
1.2.3 An Enterprise Resource Planning (ERP) system	6
1.3 Scope of the thesis.....	7
1.4 Research Design	8
1.4.1 Research questions, contributions, and evaluations	8
1.4.2 Developing the proposed contributions.....	11
1.4.3 Research methods	11
1.5 Thesis organisation.....	13

2 Background 15

2.1 Introduction	15
2.2 Classification of resource types	15
2.3 Tasks and their required resource types.....	17
2.4 Self-adaptive systems	18
2.4.1 Autonomic manager	18
2.4.2 SAS questions and dimensions	20
2.4.3 Self, context, and resources in self-adaptive systems	22
2.5 Resource-driven adaptive systems	22

2.6 Chapter summary.....	23
3 Literature Review	25
3.1 Introduction	25
3.2 Resource-driven adaptation approaches	26
3.2.1 Brownout	26
3.2.2 Task-based	31
3.2.3 Scheduling.....	33
3.2.4 Code-based	35
3.2.5 Policy-based.....	37
3.2.6 Architecture-based	38
3.2.7 Query-based	39
3.2.8 Dynamic software-product-line.....	40
3.2.9 Other approaches	40
3.3 Task modelling	42
3.3.1 Task modelling notations.....	42
3.3.1.1 Representation of task models.....	43
3.3.1.2 Operators and task types.....	43
3.3.1.3 Task modelling for resource-driven adaptation	44
3.3.2 Feature modelling notations.....	45
3.4 Summary of critical analysis.....	47
3.5 Filling the gaps.....	50
4 Overview of the Work.....	53
4.1 Introduction	53
4.2 Stakeholders	54
4.3 Adaptation components	57
4.3.1 Proactive.....	58
4.3.2 Reactive	58
4.3.3 The benefit of combining proactive and reactive adaptation	59
4.4 Task models	60
4.5 Integrating resource-driven adaptation in a software system	60
4.6 Chapter summary.....	62
5 SERIES: A Task Modelling Notation for Resource-Driven Adaptation.....	63
5.1 Introduction	63
5.2 Meta-model of SERIES.....	65

5.2.1 Constructs that SERIES incorporates from CTT	65
5.2.2 Abstract task	65
5.2.2.1 Description	66
5.2.2.2 Execution type.....	66
5.2.2.3 Feedback properties	66
5.2.2.4 Resource types and their assignment to tasks	67
5.2.2.5 Categories of resource types and tasks.....	68
5.2.2.6 Parameters	69
5.2.3 Application task	69
5.2.3.1 Priorities.....	69
5.2.3.2 Service method	70
5.2.4 Application task variant	70
5.2.4.1 Parameter conditions	70
5.2.4.2 Resource intensiveness	71
5.2.4.3 Roles	71
5.2.4.4 Substitutability	72
5.3 Example task model from an automated warehouse system	72
5.3.1 Abstract task: Prepare Order.....	74
5.3.1.1 Description	74
5.3.1.2 Execution type.....	74
5.3.1.3 Parameters	74
5.3.1.4 Resource types.....	75
5.3.1.5 Feedback properties	75
5.3.2 Application tasks.....	75
5.3.2.1 Application task 1: Locate items in the warehouse.....	76
5.3.2.2 Application task 2: Pack items in a box.....	76
5.3.2.3 Application task 3: Decorate box	76
5.3.2.4 Order of the application tasks	76
5.3.3 Application task variants for “pack items in a box”	77
5.3.3.1 Application task variant 1: Pack randomly.....	77
5.3.3.2 Application task variant 2: Pack by item type	78
5.3.4 Application task variants for “decorate box”	78
5.3.4.1 Application task variant 1: Decorate with Premium decoration	78
5.3.4.2 Application task variant 2: Decorate with Regular decoration...	79
5.4 Supporting tool of SERIES	80
5.4.1 Panels: Task model explorer, visual task model, and properties.....	80
5.4.2 Actions: Creating, loading, and saving task models.....	82
5.4.3 Adding, modifying, and removing tasks	84
5.5 Graphical representation of SERIES task models.....	85
5.5.1 Representation of tasks and task variants	85
5.5.2 Representation of relationships	85
5.5.3 Different levels of detail	86
5.5.4 The layout of task models	86
5.6 Mapping of concepts from SERIES to SPARK.....	87

5.7 Chapter summary.....	88
6 SPARK: A Framework for Resource-Driven Adaptation	89
6.1 Introduction	89
6.2 Proactive adaptation components	92
6.2.1 Task execution monitor.....	92
6.2.2 Task execution forecaster	92
6.2.3 Task prioritisation	93
6.2.3.1 Task priority calculator	93
6.2.3.2 Task priority calculator: example.....	95
6.2.3.3 Task priority adjuster.....	95
6.2.3.4 Task priority adjuster: example	97
6.2.4 Adaptation type selector	98
6.2.4.1 Supported adaptation types.....	98
6.2.4.2 Adaptation type selection	100
6.2.4.3 Changing when a task is executed (CWTE)	100
6.2.4.4 Sacrificing functionality (SF).....	101
6.2.4.5 Sacrificing quality (SQ).....	101
6.2.4.6 Financing a task's execution (FET)	102
6.2.5 Adaptation type selector: example.....	103
6.3 Reactive adaptation components.....	103
6.3.1 Resource type state monitor	103
6.3.2 Resource type state analyser	104
6.3.3 Task execution manager	105
6.3.4 Task execution allocator	106
6.3.5 Task execution allocator: example.....	109
6.3.6 Feedback elicitor and provider	110
6.4 Implementation of adaptation components	111
6.5 Chapter summary.....	114
7 Task Modelling Notation Evaluation (SERIES).....	115
7.1 Introduction	115
7.2 Assessment of SERIES using existing recommendations	116
7.2.1 Cognitive Dimensions Framework	117
7.2.2 Physics of Notations.....	118
7.2.3 Further evaluation	119
7.3 A study to evaluate SERIES with software practitioners	120
7.3.1 Participants.....	120
7.3.2 Design of the study.....	122
7.3.2.1 Hypothesis.....	123

7.3.2.2 Environment and data collection.....	123
7.3.2.3 Explaining and creating task models	124
7.3.2.4 Feedback questionnaire	127
7.3.3 Processing and presenting the data.....	128
7.3.3.1 Scoring participants' explanations and created task models....	128
7.3.3.2 Quoting the participants and classifying their comments	129
7.3.4 Results of participants' explanation and creation of task models...	130
7.3.4.1 Results of the explanation of task models	131
7.3.4.2 Results of the creating of task models	132
7.3.4.3 Result comparison.....	133
7.3.5 Results of participants' feedback on clarity of semantic constructs	133
7.3.6 Results of participants' feedback on usability	136
7.3.6.1 Ratings for ease of use	136
7.3.6.2 Selected product reaction cards	138
7.3.7 Results: participants' final comments	143
7.3.8 Discussion of the results	144
7.3.9 Threats to validity.....	146
7.4 Chapter summary.....	146

8 Framework for Resource-Driven Adaptation Evaluation (SPARK)147

8.1 Introduction	147
8.2 Preliminary evaluation of SPARK	148
8.2.1 Evaluating feasibility with an automated warehouse simulation ..	148
8.2.1.1 Design of the simulation	149
8.2.1.2 Results.....	150
8.2.2 Evaluation of overhead and scalability.....	151
8.2.2.1 Proactive adaptation planning	151
8.2.2.2 Reactive identification of tasks and task variants	152
8.2.3 Summary of the results.....	153
8.2.4 Threats to validity.....	153
8.3 Evaluating SPARK with two case studies	154
8.3.1 Datasets.....	154
8.3.2 Design of the case studies	155
8.3.2.1 Metrics and cases	156
8.3.2.2 Simulation tool and adaptation types	158
8.3.2.3 Hypothesis.....	159
8.3.3 Case study 1: Medicine consumption system	159
8.3.3.1 Metric 1: Percentage of executed critical task requests.....	159
8.3.3.2 Metric 2: Average criticality of executed tasks	162
8.3.3.3 Metrics 3 and 4: Overhead and scalability	164
8.3.3.4 Summary of case study 1	166
8.3.4 Case study 2: Manufacturing system	167

8.3.4.1 Metric 1: Percentage of executed critical task requests.....	167
8.3.4.2 Metric 2: Average criticality of executed tasks	169
8.3.4.3 Metrics 3 and 4: Overhead and scalability.....	171
8.3.4.4 Summary of case study 2.....	172
8.3.5 Comparison between the two case studies	173
8.3.6 Threats to validity.....	174
8.4 Evaluating intrusiveness	174
8.5 Chapter summary.....	175
9 Conclusions and Future Work	177
9.1 Conclusions	177
9.2 Limitations	180
9.2.1 Using SPARK in different types of software systems	181
9.2.2 Using SERIES with other frameworks.....	181
9.2.3 Integrating SPARK into applications.....	181
9.2.4 Providing additional analysis in SPARK.....	181
9.3 Future work	182
9.3.1 Extending SERIES and its supporting tool.....	182
9.3.2 Integrating SPARK with real-world software systems.....	182
9.3.3 Exploring other techniques.....	183
9.4 Final thoughts	185
BIBLIOGRAPHY.....	187
APPENDICES	201
A Artefacts from the evaluation of SERIES	203
A.1 Manufacturing Domain: Requirements and Task Models.....	203
A.2 Surveillance Domain: Requirements and Task Models	205
A.3 Questionnaire.....	208
B Artefacts from the evaluation of SPARK.....	213
B.1 Case Study 1: Medicine Consumption System.....	213
B.2 Case Study 2: Manufacturing System.....	215

List of Figures

Figure 1.1 – A resource-driven adaptive system.....	2
Figure 1.2 – Enterprise systems as motivating examples.....	5
Figure 1.3 – Mapping research questions to chapters.....	9
Figure 1.4 – Proposed solution for addressing resource variability	10
Figure 2.1 – MAPE-K control loop	19
Figure 3.1 – Example of UI and tasks.....	27
Figure 3.2 – Example of a component and tasks	29
Figure 3.3 – Characteristics of proposed work (a) and their self-* properties (b) ..	52
Figure 4.1 – An overview of the proposed work.....	54
Figure 4.2 – The proposed adaptation components based on MAPE-K	56
Figure 4.3 – Integrating resource-driven adaptation in a software system.....	61
Figure 5.1 – Meta-model of SERIES represented as a class diagram.....	64
Figure 5.2 – A task model example from an automated warehouse system	73
Figure 5.3 – Tool for creating and modifying task models using SERIES.....	79
Figure 5.4 – Data entry windows for properties that represent sets of values..	80
Figure 5.5 – Supporting tool of SERIES – setup data tab.....	81
Figure 5.6 – Data entry windows for setup data properties.....	81
Figure 5.7 – Opening and saving task models from and to a database.....	82
Figure 5.8 – Supporting tool of SERIES – configuration tab	84
Figure 6.1 – Logging and prioritisation of tasks.....	90
Figure 6.2 – SPARK proactive and reactive adaptation components	91

Figure 6.3 – Activity diagram of the decision-logic for task execution	105
Figure 6.4 – Method M1 steps.....	107
Figure 6.5 – Method M2 steps.....	108
Figure 6.6 – Plan creation	110
Figure 6.7 – Task priority calculation overview	111
Figure 7.1 – Overview of the evaluation of SERIES.....	116
Figure 7.2 – Experience of the participants in the software industry	121
Figure 7.3 – Countries where the participants are working	121
Figure 7.4 – Experience of the participants with visual modelling notations .	121
Figure 7.5 – Where the participants used visual modelling notations	121
Figure 7.6 – Overview of the study’s activities.....	122
Figure 7.7 – Example: task model that the participants explained.....	124
Figure 7.8 – Example: requirements for creating a task model.....	125
Figure 7.9 – Example: task model that the participants created.....	126
Figure 7.10 – Time taken by the participants to explain the task models	130
Figure 7.11 – Time taken by the participants to create the task models	130
Figure 7.12 – Scores on the participants’ explanation of task models	131
Figure 7.13 – Scores on the participants’ creation of task models	131
Figure 7.14 – Participants’ feedback on the semantic constructs of SERIES..	134
Figure 7.15 – Participants’ feedback on the ease of use of SERIES.....	135
Figure 7.16 – PRCs selected by the participants to describe SERIES	137
Figure 7.17 – Percentage of comments in each theme	144
Figure 8.1 – Overview of the evaluation of SPARK.....	148
Figure 8.2 – Automated warehouse example.....	149
Figure 8.3 – Prepare order task	149

Figure 8.4 – Automated warehouse simulation software	149
Figure 8.5 – Customer order preparation during resource variability.....	150
Figure 8.6 – Running time scalability of proactive adaptation planning.....	151
Figure 8.7 – Overhead and scalability evaluation without caching	152
Figure 8.8 – Overhead and scalability evaluation with caching	153
Figure 8.9 – Evaluation case studies with subcases.....	156
Figure 8.10 – Case Study 1: percentages of executed critical task requests....	160
Figure 8.11 – Case Study 1: executed task requests that are more critical.....	163
Figure 8.12 – Case Study 1: proactive adaptation planning running time.....	164
Figure 8.13 – Case Study 1: identification of tasks/variants running time	165
Figure 8.14 – Case Study 1: task execution allocation running time.....	166
Figure 8.15 – Case Study 2: percentages of executed critical task requests....	167
Figure 8.16 – Case Study 2: executed task requests are more or less critical .	170
Figure 8.17 – Case Study 2: proactive adaptation planning running time.....	171
Figure 8.18 – Case Study 2: identification of tasks/variants running time	172
Figure 8.19 – Case Study 2: task execution allocation running time.....	173
Figure A.1 – Task model for explanation (manufacturing)	203
Figure A.2 – Requirements to create a task model (manufacturing)	204
Figure A.3 – Expected task model based on requirements (manufacturing)...	205
Figure A.4 – Task model for explanation (surveillance).....	205
Figure A.5 – Requirements to create a task model (surveillance).....	206
Figure A.6 – Expected task model based on requirements (surveillance)	207

List of Tables

Table 2.1 – Examples of resource types and their groups	17
Table 2.2 – Examples of tasks and their primary required resources	18
Table 2.3 – Questions and dimensions of self-adaptive systems	20
Table 2.4 – Examples of software systems (un)affected by resource variability	23
Table 3.1 – Summary of existing resource-driven adaptation approaches	47
Table 3.2 – Summary of existing task modelling notations	49
Table 5.1 – Mapping concepts from SERIES meta-model to SPARK components	87
Table 6.1 – Initial priority calculation example.....	95
Table 6.2 – Adjusted priority calculation example	98
Table 6.3 – Cost of adaptation calculation example	102
Table 6.4 – Example data showing how M1 and M2 calculate NATE	109
Table 8.1 – Characteristics of the datasets used in the evaluation.....	155
Table 8.2 – Case Study 1: the case where 20% of tasks (variants) are critical	161
Table 8.3 – Case Study 2: the case where 20% of tasks (variants) are critical	168
Table B.1 – Case Study 1: the case where 40% of tasks (variants) are critical	213
Table B.2 – Case Study 1: the case where 60% of tasks (variants) are critical	214
Table B.3 – Case Study 1: the case where 80% of tasks (variants) are critical	214
Table B.4 – Case Study 2: the case where 40% of tasks (variants) are critical	215
Table B.5 – Case Study 2: the case where 60% of tasks (variants) are critical	216
Table B.6 – Case Study 2: the case where 80% of tasks (variants) are critical	216

List of Equations

Equation 6.1 – Priority value range	94
Equation 6.2 – Priority value classification	94
Equation 6.3 – Threshold calculation	94
Equation 6.4 – Threshold forecasted task execution priority calculation	94
Equation 6.5 – Initial task priority calculation	94
Equation 6.6 – Cost function for deprioritising a task	96
Equation 6.7 – Initial epsilon value calculation	96
Equation 6.8 – Priority adjustment calculation.....	96
Equation 6.9 – Input value range	100
Equation 6.10 – Cost function for adaptation.....	100

List of Code Listings

Listing 5.1 – JSON representation of the “Prepare Order” abstract task	83
Listing 6.1 – Priority calculation source code (excerpt)	112

List of Abbreviations

MAPE-K	Monitor-Analyse-Plan-Execute-Knowledge
RAS	Resource-driven Adaptive System
SAS	Self-Adaptive System
SERIES	task modelling notation for Resource-driven adaptation in software Systems
SPARK	resource-driven adaptation framework

Glossary

The following list includes definitions for key terms, which are frequently used throughout this thesis:

- **Enterprise** is an organisation (e.g., a business like a retail store).
- **Enterprise System** is a software system that manages the activities of enterprises. Examples of an enterprise system include an automated warehouse system, manufacturing system, and enterprise resource planning (ERP) system.
- **Resource** is an entity that is needed to carry out a task.
- **Resource-driven Adaptive System** is a type of self-adaptive system, where the trigger for adaptation is the variability of resources.
- **Self-Adaptive System** is a system that can adapt itself automatically based on changes in its context.
- **Task** represents an activity in a software system.
- **Abstract Task** involves complex actions and is broken down into a sequence of child (sub) tasks.
- **Application Task** is executed by the software system without user interaction.
- **Application Task Variant** is a special case of an application task and is needed to (1) avoid treating all executions of an application task in the same way when adapting and (2) identify how to execute an application task with fewer resources.
- **Resource Intensiveness** indicates the level of resource consumption of an application task variant for a type of resource.

1

Introduction

This thesis addresses resource variability, which prevents software systems from executing their tasks. This chapter motivates the research using examples of multiple types of software systems. It explains the research objectives, presents the research questions, and summarises the contribution.

1.1 Problem and motivation

Existing software systems rely on resources to execute tasks. For example, using robots in automated warehouses, medical materials in hospital management systems, energy in electricity grids, or ingredients in food production systems. The availability of resources, however, can vary due to several reasons such as unexpected hardware failures, excess workloads, or lack of materials. For example, product deliveries could be delayed due to robots malfunctioning in automated warehouses, medical operations may be cancelled due to unavailable materials and theatres in hospitals, prices of electricity may be increased due to high demand, or the food supply chain could be disrupted due to high demand of food products and lack of ingredients.

Adaptation can help software systems to deal with resource variability (Adelstein *et al.*, 2005). Resource-driven Adaptive Systems (RASs) are a type of self-adaptive system (SAS) in which changes in the system are driven by resource variability (Christi, Groce and Wellman, 2019). This means in RASs, the unavailability or scarcity of resources to carry out a task can trigger an adaptation that permits the software system to keep executing tasks. An autonomic manager oversees the adaptation process for a software

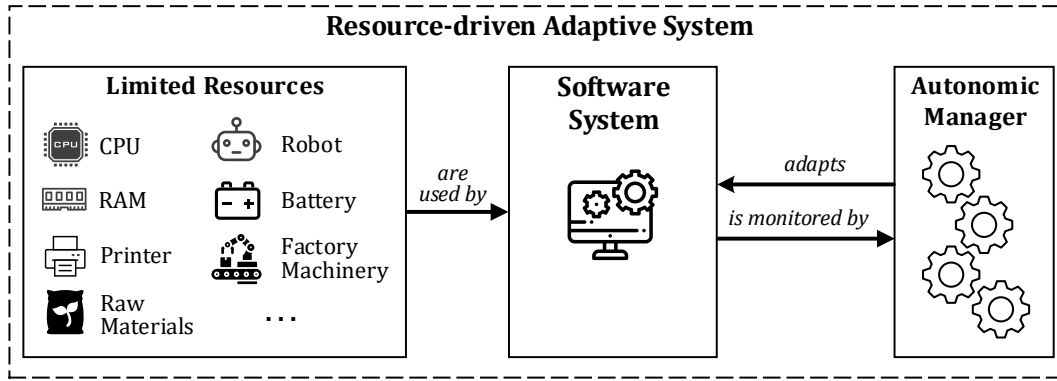


Figure 1.1 – A resource-driven adaptive system

system by making adaptation decisions during resource variability. Figure 1.1 depicts the element of a RAS.

Existing work on resource-driven adaptation focus on single *types of adaptation* that involve disabling optional components (Klein, Maggio, *et al.*, 2014; Xu and Buyya, 2019), reducing the data returned by a query (Gotz *et al.*, 2015; Viswanathan, Jindal and Karanasos, 2018), changing system configurations through policies (Efstratiou *et al.*, 2002; Keeney and Cahill, 2003), or reducing source code that consumes a lot of computational resources (Christi, Groce and Gopinath, 2017; Christi and Groce, 2018). *By supporting multiple types of adaptation, software systems become more versatile in addressing resource variability.* For example, if an optional component needs an unavailable resource, then it is possible to disable this component. Otherwise, if the component is not optional the software system should seek another type of adaptation.

Additionally, existing resource-driven adaptation approaches focus on a single type of *reusable* resource like CPU (Maggio, Klein and Arzén, 2014; Sun, Cai and Loparo, 2019), RAM (Huber *et al.*, 2017; Christi and Groce, 2018), battery (Pascual, Pinto and Fuentes, 2015; Yan *et al.*, 2019), and bandwidth (Sousa *et al.*, 2006; Papakos, Capra and Rosenblum, 2010), or even on *depletable* resource types such as food ingredients (Bennaceur *et al.*, 2019). However, software systems rely on multiple types of depletable and reusable resource types that are impacted by variability. For example, automated warehouse systems rely on resource types like robots, boxes, and bubble wrap for packing products to be shipped. As indicated by Xu and Buyya (2019), *supporting multiple types of resources would make resource-driven adaptation approaches more comprehensive.* This means the adaptation approach would be applicable to multiple types of resources rather than a specific type of resource.

Furthermore, *the consideration of tasks in resource-driven adaptation provides granularity in the adaptation decision-making* because there could be differences in (i) the priorities that specify the level of importance of each task, (ii) the applicability of an adaptation type to a task, (iii) the types of resource that are used by a task, and (iv) the resource consumption of task variants that represent different versions of a task. Instead of considering some software components to be always optional (Klein, Maggio, *et al.*, 2014; Xu and Buyya, 2019) or using configurations that apply to an entire software system (Efstratiou *et al.*, 2002; Papakos, Capra and Rosenblum, 2010), it is possible to make more granular adaptation decisions based on tasks. For example, the task of displaying product recommendations in an online retail store system can be optional when it is not initiated by users with privileged roles (e.g., VIP customers). This would reduce the consumption of computational resources during resource variability while keeping the task available where it is most needed.

When considering tasks in resource-driven adaptation, it is not sufficient to support single-user scenarios as is done in some approaches that target mobile devices (Sousa *et al.*, 2006; Rigole *et al.*, 2007) because software systems have multiple users who are initiating tasks that are competing for resources. For example, multi-user enterprise systems differ from single-user mobile apps because the adaptation must consider the perspectives of end-users and corporate management regarding which tasks are more important during resource variability. Additionally, sharing resources among tasks for a limited time via a leasing mechanism (Perttunen, Jurmu and Riekkki, 2007) is not sufficient because there are tasks that require using resources until completion (e.g., medical operations make use of medical equipment until completion).

Moreover, *software systems should perform resource-driven adaptation at runtime* because information like the importance of a task or the choice of a type of adaptation is unknown at design time. Hence, existing approaches that modify the source code of software systems (Christi and Groce, 2018; Christi, Groce and Wellman, 2019) are not suitable for performing resource-driven adaptation at runtime. Furthermore, these approaches are limited to software systems that are written in particular programming languages because annotations are added to the code as is done by Christi *et al.* (2017) with Java programs. On the other hand, a resource-driven adaptation approach would be technology independent if such annotations are represented at a higher level of abstraction than source code. Resource-driven adaptation approaches would be applicable to multiple resource variability scenarios if

they support several types of software systems. Some existing approaches focus on types of software systems such as power consumption in unmanned aerial vehicles (Yan et al., 2019) and data exchange between robots (Gotz et al. 2015), which narrows their scope of applicability.

1.2 Motivating examples

In order to motivate, illustrate, and evaluate the work in this thesis, I consider three examples of enterprise systems, namely: *warehouse systems*, *manufacturing systems*, and *Enterprise Resource Planning (ERP) systems*. Enterprise systems are software systems that manage the activities of enterprises (Oz, 2009). What follows is an explanation of why enterprise systems are relevant examples for the research presented in this thesis.

As shown by the examples in Figure 1.2, enterprise systems make use of tasks that require limited resources such as robots, boxes, raw materials, factory machinery, RAM and CPU. Hence, enterprise systems are affected by resource variability. Furthermore, enterprise systems make use of tasks that differ among enterprises (Lucas, Xu and Babaian, 2013). So, it is not possible to define a software system's behaviour at development time in a single way that accommodates the needs of all enterprises based on the availability of resources. Therefore, software systems should perform resource-driven adaptation at runtime while prioritising tasks so limited resources would remain available for the tasks that need them most, namely the critical tasks. The prioritisation should consider multiple criteria, which include the task's parameter values, resource consumption, initiation time during the day, user's role, historical data about usage frequency, and domain-related criticality (e.g., it could be more critical for a warehouse to ensure timely delivery of products for VIP customers). The following sections discuss the usefulness of resource-driven adaptation with the abovementioned prioritisation criteria and the adaptation types mentioned in Section 1.1 in the context of the motivating examples presented in Figure 1.2.

1.2.1 A warehouse system

Consider that a retail store has a warehouse that is automated by robots. The retail store receives customer orders throughout the day. Robots perform order preparation tasks by retrieving from the warehouse the products corresponding to customer

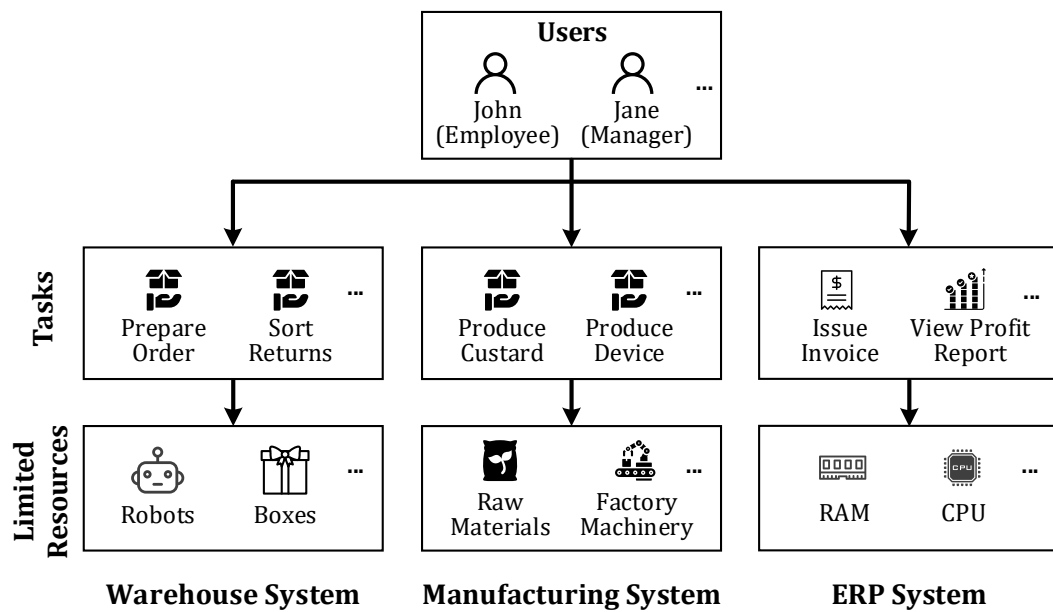


Figure 1.2 – Enterprise systems as motivating examples

orders and packing them in boxes ready for delivery. Hence, robots are essential resources for executing key warehouse activities. However, robots can temporarily go out of service due to unexpected errors or due to the need for recharging, thereby delaying order fulfilment and causing financial losses. Software systems could avoid these negative implications of resource variability through adaptation as explained below.

The preparation of customer orders for shipping is a critical task, but robots also work on tasks like sorting returned products, which could be less important in some cases (e.g., during specific times of the day). Hence, it is possible to substitute inoperative order preparation robots with robots that are working on less important tasks. In this way, the order preparation stays on track until the inoperative robots are back in service (e.g., after repair). This demonstrates situations of task prioritisation, resource substitution, and execution of tasks in a different order (leaving the sorting of returned products for another time).

Another possibility is to alter the robots' behaviour by changing the way they pack products. Assume that the robots can pack products in a box using two ways: (i) placing similar products next to each other in a box (e.g., trousers and shirts in separate piles), or (ii) placing products randomly in a box. These two ways are variants of the same "pack products" task. The first variant provides a better presentation for

the customer, while the second one executes faster because robots do not have to arrange the products. It is possible to keep order preparations on track, by using the first variant for the orders of Very-Important-Person (VIP) customers and the second one for other orders. Hence, in this case, tasks are changed to similar ones by allowing products to be packed but in a random way. This example shows two variants of a task (pack products) and two types of resources (robots and boxes).

1.2.2 A manufacturing system

A manufacturing system controls the production of goods at a factory. One example is food manufacturing where a factory produces custard using resource types that include food ingredients and manufacturing machinery. In some cases, the required food ingredients are not available and should be substituted with others. For example, custard powder could be substituted with eggs and cornflour. In other cases, it may not be possible to manufacture the same product using the available resources. In such cases, the alternative is to change the behaviour of the manufacturing process to either produce less or to continue the production of some items at another time. Moreover, some tasks may be more critical than others and adaptation would help the system to keep limited resources available for the critical tasks that need them most.

1.2.3 An Enterprise Resource Planning (ERP) system

Consider an ERP system that is facing resource variability on one of its servers due to hardware failure. In this case, tasks would take longer to execute due to having less hardware capacity to serve the same number of task requests. The system would maintain a good response time by performing adaptation (e.g., executing task variants that consume fewer hardware resources). This is useful even if the failure will be repaired in a short time. Otherwise, critical tasks could be impeded causing harm to people and losses to enterprises. For example, if the ERP was used at a hospital, e.g., ERP for healthcare (2022), patients might not be given the necessary treatment on time due to delayed processing of paperwork (e.g., retrieval of medical records). In another example, if the ERP was used at an import/export company, e.g., Blue Link (2022), delayed paperwork could lead to customer dissatisfaction and ultimately leads to profit loss. What follows are examples of tasks, from ERP systems, which are common among many types of enterprises.

Consider that the task of issuing an invoice for a customer during peak time is more critical for an enterprise than the task of viewing a profit report. Also, consider that the task of viewing a profit report is CPU intensive and can execute later during the day. If tasks are executed on a first-come-first-serve basis, non-critical tasks like viewing a profit report could impede critical tasks like invoicing. Consider that the task of viewing a profit report has a summarised variant that is less CPU-intensive. Also, consider that there is a secondary server that hosts an older version of the data that the profit report requires. The software system could adapt by executing the summarised variant of the report; generating the report on the secondary server; queuing the report for the users to view it later outside peak work time. The software system needs to consider multiple adaptation possibilities because some might not be applicable. For example, if either the summarised report or the older data from the secondary server were insufficient for the user then the report would be queued for later.

Moreover, consider that the abovementioned task of viewing a profit report is issued by a manager for a limited year range. In this case, the enterprise could consider the task of viewing a profit report to be critical because it is issued by a user who has a privileged role (manager) and with parameter values (year range) that reduce the data and make it less CPU intensive (i.e., consumes fewer resources). Hence, the software system should consider these variations otherwise tasks like viewing a profit report would always be less important than other tasks like issuing an invoice.

1.3 Scope of the thesis

The scope of the work in this thesis is concerned with run-time adaptation of software systems due to variation in available resources. More specifically, the work focuses on supporting (i) different types of adaptation activities due to variability of resources; (ii) adaptation due to various tasks that the system needs to fulfil; (iii) different types of resources and considering multiple resources at the same time; and (iv) a proactive and reactive approach for adapting software systems.

Considering the abovementioned scope, I conducted a literature review and gap analysis of the existing work on resource-driven adaptation. This showed that the existing work on resource-driven adaptation does not (i) consider tasks and the various types of tasks to provide granularity in the resource-driven adaptation

process, (ii) support multiple types of resources to make the resource-driven adaptation applicable to multiple types of software systems, and (iii) support multiple types of adaptation to have alternatives in case a type of adaptation is not applicable.

To address the problem of resource variability, a framework is needed to support resource-driven adaptation in software systems and fill the abovementioned gaps (i)-(iii) in existing works. Furthermore, since this framework should consider tasks, it requires input data about the software system's tasks. Hence, a task modelling notation is needed to define this input. I explored existing task modelling notations to check if they can support resource-driven adaptation and found that they are missing useful characteristics in this regard. Hence, new task modelling notation is also needed to complement the resource-driven adaptation framework.

As discussed in Section 1.2, this thesis uses motivating examples from enterprise systems, like warehouse management systems and manufacturing systems, because they rely on multiple types of resource to execute tasks and are affected by resource variability. Nonetheless, other types of software systems that are affected by resource variability could also benefit from this work.

1.4 Research Design

This section introduces the research questions and maps them to the contribution and evaluation chapters as shown in Figure 1.3. It also provides an overview of the research methods that this thesis uses.

1.4.1 Research questions, contributions, and evaluations

The objective of this thesis is to address the research problem presented in Section 1.1. This problem is captured by a broad research question (**RQ**), which is broken down into two sub-questions, **RQ1** and **RQ2**, as follows:

RQ: *“How can software systems address resource variability through resource-driven adaptation?”*

- **RQ1:** *“How to model tasks of software that require the use of resources?”*
- **RQ2:** *“How and when software systems adapt to enable the execution of critical tasks when resources are limited?”*

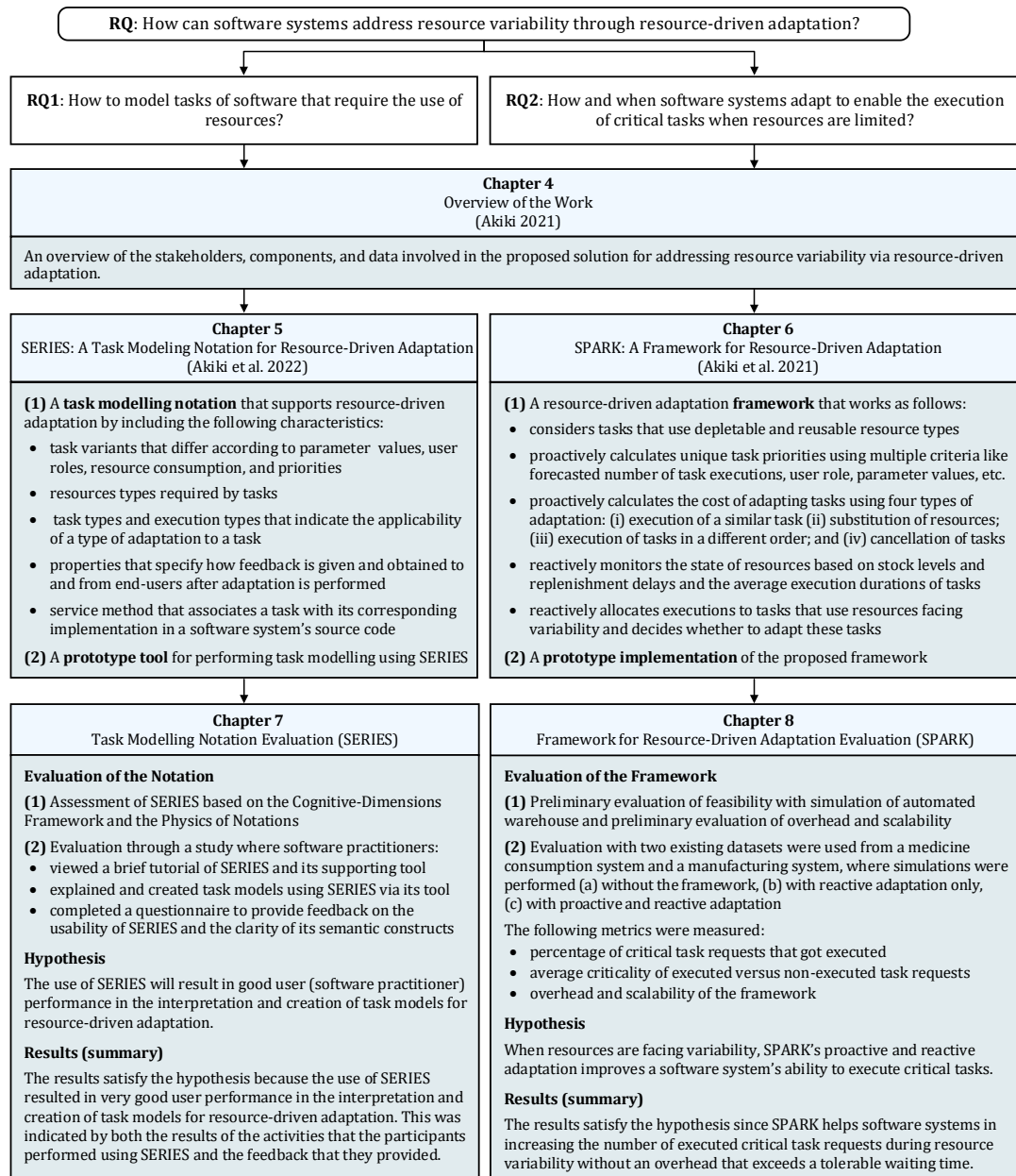


Figure 1.3 – Mapping research questions to chapters

The answers to RQ1 and RQ2 constitute the two research contributions (RCs) of the proposed solution, namely devising a task modelling notation (RC1) and a framework (RC2) that supports resource-driven adaptation to address the issue of resource variability. Figure 1.4 provides a high-level overview of the proposed solution. A software system has tasks that are initiated by users and require resources to execute. These tasks and resources are represented as task models via the proposed task modelling notation. The task models serve as input for the proposed framework, which performs adaptation during resource variability.

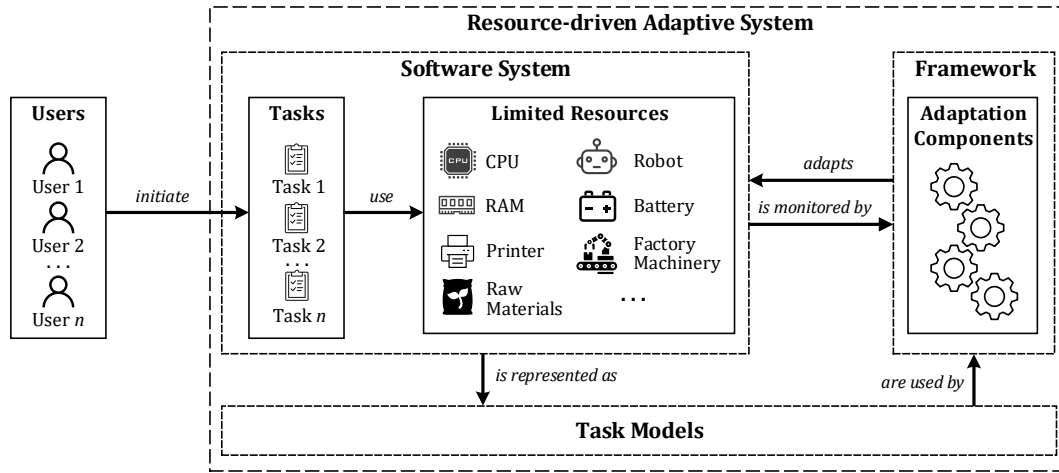


Figure 1.4 – Proposed solution for addressing resource variability

As shown in Figure 1.4, a **framework** is needed to realise the components of the autonomic manager and to support resource-driven adaptation in multiple types of resource-dependent software systems. This framework makes adaptation decisions that enable a software system to keep executing (critical) tasks during resource variability. A critical task has high importance for its domain and is more privileged in accessing the resources it needs in comparison to non-critical tasks. The consideration of tasks in resource-driven adaptation provides granularity in adaptation decision-making. Hence, a resource-driven adaptation framework requires input data that describes the software system's tasks and their properties (e.g., required resources). This input enables the framework to decide on (i) the importance of tasks, (ii) whether adaptation is required for a task, (iii) which type(s) of adaptation (are) applicable for a task, and (iv) how to perform a type of adaptation.

A **task modelling notation** is needed to create task models that represent the abovementioned input. Such notation facilitates the representation of task models hierarchically using a graphical syntax. Figure 1.3 shows a summary of the abovementioned research questions and the contributions of the work with respect to the research questions and specific chapters that describe the work.

Chapter 4 presents an overview of my work. The overview depicts the involved stakeholders and data, in addition to the proposed adaptation components that are based on the MAPE-K control loop (Kephart and Chess, 2003).

Chapter 5 presents a **task modelling notation** (RC1) for resource-driven adaptation called SERIES. This notation offers characteristics that are useful for resource-driven adaptation but are missing from existing task modelling notations

(Limbourg and Vanderdonckt, 2004; Guerrero-García, González-Calleros and Vanderdonckt, 2012; Martinie *et al.*, 2019). In brief, these characteristics include task variants, resource types, task execution types, end-user feedback properties, and service methods (refer to Figure 1.3).

Chapter 6 presents a proactive and reactive **framework** (RC2) for resource-driven adaptation called SPARK. The framework uses historical data to proactively prepare an adaptation plan. This plan is used during reactive decisions to execute adaptation, due to variability in resources. In comparison to existing work on resource-driven adaptation, the novelty aspects of the framework are concerned with support for: (i) tasks that use multiple types of depletable and reusable resources, (ii) four types of adaptation, and (iii) unique prioritisation of tasks using multiple criteria that include task priorities, time of day, user role, task parameters, and forecasted task executions (refer to Figure 1.3).

Chapters 7 and 8 present an evaluation of the task modelling notation SERIES and the resource-driven adaptation framework SPARK, respectively. The following section explains the research methods used in the evaluation of the work.

1.4.2 Developing the proposed contributions

I used meta-modelling and prototyping in this thesis to develop the proposed notation and framework. Meta-modelling involves creating a set of concepts and the relations between them, whereby this set of concepts (meta-model) is used to represent models (Allemang and Hendler, 2011). I used meta-modelling to describe the concepts of SERIES that represent task models.

Moreover, prototyping involves creating a prototype software either to get feedback from users or to assess the feasibility of designing a system (Pressman, 2010, p. 43,75). I developed prototypes of SPARK and the supporting tool for SERIES. I used these prototypes as part of the evaluation of SERIES and SPARK, which involved the research methods explained next.

1.4.3 Research methods

Several research methods have been used in software engineering research (Shaw, 2002). To evaluate the usability of the proposed notation, I conducted a user study, which is a type of controlled experiment that tests a hypothesis by measuring the

effect of a dependent variable (i.e., user performance) on an independent variable (i.e., interpretation and creation of task models) (Easterbrook *et al.*, 2008). Hence, the user study allows us to test whether SERIES is usable by the intended population, namely software practitioners, to interpret and create task models. Furthermore, since it is a type of controlled experiment, the user study enables us to control what users do and to control influences that might impact their performance (Preece, Rogers and Sharp, 2015, pp. 474–475). For example, all the participants were asked to use SERIES for activities that have the same level of difficulty. The user study measured the ability of software practitioners to explain and create SERIES task models. It also measured how the participants perceive the usability of SERIES and the clarity of its semantic constructs. Furthermore, the task modelling notation was assessed based on two paradigms, namely the Cognitive Dimensions Framework (Green and Petre, 1996) and the Physics of Notations (Moody, 2009). I used these paradigms to assess SERIES because they provide principles for evaluating visual notations.

A case study serves as an inquiry that is either an *exploratory* preliminary investigation to derive new hypotheses, or a *confirmatory* main investigation to test hypotheses (Easterbrook *et al.*, 2008). I evaluated SPARK using two case studies that involve running simulations of resource variability scenarios with existing datasets. This evaluation had multiple metrics. The metrics “percentage of executed critical task requests” and “average criticality of the executed task requests in comparison to the non-executed ones” show to what extent SPARK can address resource variability by enabling a software system to keep executing critical tasks that require limited or unavailable resources. Hence, a result is positive if SPARK increases the percentage of executed critical tasks and if the tasks that got executed were on average more critical than the ones that did not get executed (if any). The metrics “overhead” and “scalability” show whether SPARK’s adaptation process impacts a software system’s ability to execute tasks in a tolerable waiting time. I evaluated the “overhead” by measuring the running time and evaluated the scalability by increasing the size of SPARK’s input, namely the number of tasks from the datasets.

The two case studies used for this evaluation were derived from real data on the topics of medicine consumption and manufacturing systems, two domains where resource variability is challenging (NHS, 2019; Carvalho *et al.*, 2022). The metrics were used in the evaluation to compare the execution of tasks in the two case studies with variable resources when the adaptive framework is not present, when only the

reactive adaptation part of the framework is used, and when both the proactive and reactive adaptation parts of the framework are used.

1.5 Thesis organisation

The rest of this thesis is organised into the following chapters:

Chapter 2 – Background provides a general understanding of the research area of this thesis and is a prelude to understanding the contents of the other chapters.

Chapter 3 – Literature Review analyses the related work on resource-driven adaptation based on what can be adapted such as components, tasks, and source code. This chapter also presents tables that show the types of resources and systems that each approach targets. Furthermore, this chapter reviews task modelling notations based on whether they comprise characteristics for supporting resource-driven adaptation.

Chapter 4 – Overview of the Work presents an overview of the work that this thesis proposes. It presents the stakeholders, components, and data involved in the solution (SERIES and SPARK) for addressing resource variability by performing resource-driven adaptation.

Chapter 5 – SERIES: A Task Modelling Notation for Resource-Driven Adaptation details the task modelling notation based on CTT (Paterno, Mancini and Meniconi, 1997), which is a notation for representing task models hierarchically using a graphical syntax. SERIES is supported by a software tool that enables software practitioners to create and modify task models.

Chapter 6 – SPARK: A Framework for Resource-Driven Adaptation presents a resource-driven adaptation framework that works proactively and reactively. It supports tasks that use depletable and reusable resource types. Additionally, it supports the generation of unique task priorities using multiple criteria and four types of adaptation.

Chapter 7 – Task Modelling Notation Evaluation (SERIES) assesses SERIES based on existing paradigms for designing notations, namely the Cognitive Dimensions Framework (Green and Petre, 1996) and the Physics of Notations (Moody, 2009). Furthermore, this chapter presents an evaluation study with software practitioners. In this study, the participants explained and created task models using SERIES and then

provided their feedback on the usability of SERIES and the clarity of its semantic constructs.

Chapter 8 – Framework for Resource-Driven Adaptation Evaluation (SPARK) presents a preliminary evaluation conducted to evaluate the feasibility of SPARK by developing a software tool that simulates an automated warehouse system and measures overhead and scalability. Furthermore, this chapter presents an evaluation of SPARK in two case studies that are related to a medicine consumption system and a manufacturing system. In these two case studies, I applied SPARK to existing datasets and measured several metrics including the percentage of executed critical task requests, the average criticality of the executed task requests versus the non-executed ones, overhead, and scalability.

Chapter 9 – Conclusions and Future Work summarises the work and discusses future directions.

2

Background

2.1 Introduction

Section 1.1 discussed how adaptation helps software systems to keep executing critical tasks during periods of resource variability. Additionally, Section 1.3 discussed the research objectives of this thesis, which include creating a task modelling notation and framework for supporting resource-driven adaptation in software systems. This chapter presents background information to explain further the key concepts that are related to the abovementioned research objectives. In this regard, Section 2.2 presents a classification scheme and examples of resource types. Section 2.3 provides examples of tasks and their required resource types. Section 2.4 presents an overview of self-adaptive systems (SASs). Section 2.5 explains the notion of a resource-driven software system and how SASs are a general case of resource-driven adaptive systems (RASs).

2.2 Classification of resource types

A resource is defined as an entity that is needed to carry out a task (Raunak and Osterweil, 2013). There are multiple types of resources. For example, a software system can have several resources of the type “robot” and several resources of the type “box”. The complexity of resource types differs among software systems and domains (Raunak and Osterweil, 2013). For example, a resource type could be as simple as a nail or as complex as manufacturing equipment.

Four resource groups, which represent a way to classify resource types, have been identified in this thesis based on the analysis of existing literature and examples of

multiple types of resources such as CPU (Klein, Maggio, *et al.*, 2014), food ingredients (Bennaceur *et al.*, 2019), and battery (Yan *et al.*, 2019). I chose this classification because it represents differences among resource types, which affect adaptation decisions. For example, a software system should prevent resources that can only be used once from being depleted by non-critical tasks during resource variability. These resource groups are explained below with examples of resource types that fit under each group.

- **Static resource:** does not have a behaviour (actions that the resource performs)
- **Dynamic resource:** has a behaviour that can be adapted
- **Reusable resource:** is available to another task after a task that is using it is done
- **Depletable resource:** is used once

The groups “static” and “dynamic” indicate a resource’s form of behaviour whereas the groups “reusable” and “depletable” indicate its mode of consumption. Table 2.1 shows examples of resource types and their corresponding resource groups. Additionally, a resource type can be related to more than one resource group. For example, batteries are static and reusable since they do not have a behaviour and can be reused when recharged; robots are dynamic and reusable since they have a behaviour and can be reused when a task is completed; raw materials are static and depletable since they do not have a behaviour and can be used once. To clarify further, a robot’s behaviour such as speed can be adapted whereas similar behaviour is not available on batteries and raw materials. Moreover, once a task that is using a battery or a robot is done, then other tasks can reuse these resource types. On the other hand, once a task uses a quantity of raw materials, then this quantity will no longer be available for other tasks.

The abovementioned resource groups affect a software system’s adaptation choices. For example, if non-critical tasks are not restricted before they exhaust scarce depletable resources then critical tasks that need these resources would not be able to execute. Moreover, when reusable resource types are concerned, it is possible to assess whether adaptation is needed by measuring the time spent to gain access to a resource (Grohmann *et al.*, 2021). However, this does not apply to depletable resource types because these are either available for a task to use directly or unavailable and hence adaptation would be needed. Additionally, when depletable resource types are unavailable they have to be ordered from a supplier and cannot be expanded directly

Table 2.1 – Examples of resource types and their groups

Resource Type	Resource Groups			
	Static	Dynamic	Reusable	Depletable
Battery	✓	×	✓	×
RAM	✓	×	✓	×
CPU	✓	×	✓	×
Autonomous vehicle	×	✓	✓	×
IoT device	×	✓	✓	×
Robot	×	✓	✓	×
Food ingredient	✓	×	×	✓
Fuel	✓	×	×	✓
Raw material	✓	×	×	✓

like virtual servers (Van Hoorn *et al.*, 2009; Huber *et al.*, 2017) or discovered for immediate use like software components (Garlan, Cheng, *et al.*, 2004). Furthermore, the behaviour of dynamic resource types can be changed (e.g., through parameters such as a robot's speed). In contrast, static resource types such as RAM and battery cannot change its behaviour.

2.3 Tasks and their required resource types

A task is issued through a software system and needs resources so it can execute. Software systems execute tasks that require multiple types of resources. For example, cloud computing systems rely on hardware resources such as CPU, RAM, and hard drives (Kurp, 2008; Lu *et al.*, 2016). Hospital systems rely on multiple types of resources such as theatres, equipment, and medical materials like bandages and medicine (Hutzschenreuter, Bosman and La Poutré, 2009). IoT systems rely on batteries, sensors, and cameras (Ciccozzi *et al.*, 2017).

As indicated by Raunak and Osterweil (2013), the resources that a software system relies on to operate might not be of the same type. This depends on the diversity of tasks that the system needs to execute (Patel, Patel, and others, 2016). For example, a simple automated lighting system just relies on a motion sensor to execute a task that involves turning on the light when there is movement. On the other hand, an

Table 2.2 – Examples of tasks and their primary required resources

Software System	Example Task	Example of Resource Types
Manufacturing	Manufacture products	Raw Materials and Factory Machinery
Retail Store	Search for a product	RAM and CPU
	View sales report	RAM and CPU
	Print sales report	Printer
Warehouse	Prepare customer order	Robots and Packing Materials
Power Grid	Run generator	Generator and Fuel
Surveillance	Survey area	Drone and Camera

automated warehouse system relies on robots, packing materials, and sensors to execute tasks for customer order preparation and stock replenishment.

Table 2.2 presents a few examples of tasks, from multiple types of software systems alongside examples of resource types that each one requires so it can execute. For example, customers use a retail store’s website to issue the task “search for a product”, which depends on the CPU and RAM of the server for timely execution. Another example is the task “prepare an order for delivery”, which is issued on an automated warehouse management system and is performed by robots. In this example, the availability of the necessary resources can vary, due to reasons such as hardware failure and excess workloads, which can cause financial losses.

2.4 Self-adaptive systems

As explained in Section 1.1, adaptation can help software systems to deal with resource variability and to keep executing tasks that require scarce resources. Hence, this section presents some background information about SASs.

2.4.1 Autonomic manager

A SAS is a system that can adapt itself automatically based on changes in its context (Cheng *et al.*, 2009; De Lemos *et al.*, 2013). SASs are composed of an autonomic manager and a managed system. An autonomic manager corresponds to the control

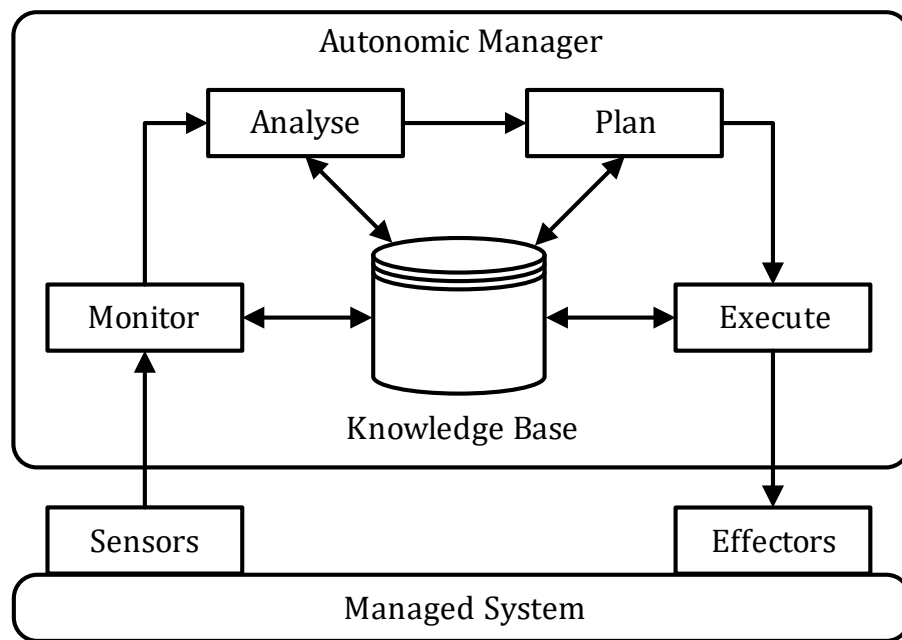


Figure 2.1 – MAPE-K control loop

unit that manages the adaptation process of the managed system. The autonomic manager adapts a managed system.

An autonomic manager is defined using a control loop like the MAPE-K, which proposes components that enable self-adaptation (Kephart and Chess, 2003). MAPE-K is composed of five components: Monitor (M), Analyse (A), Plan (P), Execute (E), and Knowledge (K) base, as illustrated in Figure 2.1.

- **Monitor (M)** collects data related to the managed system and the context using an interface of sensors, where the collected data is stored in the knowledge base.
- **Analyse (A)** performs an analysis of the monitored data and determines if the managed system requires adaptation.
- **Plan (P)** constructs the adaptation actions needed to achieve the managed system's objectives.
- **Execute (E)** carries out the generated actions in the planning phase to adapt the managed system using an interface of effectors.
- **Knowledge (K) base** shares and maintains data resulting from the MAPE components for supporting the autonomic manager.

Table 2.3 – Questions and dimensions of self-adaptive systems

Question	Dimension	Examples
When?	Time	Reactive, proactive
Why?	Reason	Context, resource
Where?	Level	Software system, resource
What?	Technique	Parameter, structure, context
Who?	Executor	Autonomic manager, human
How?	Control	Decision criteria, degree of decentralisation

2.4.2 SAS questions and dimensions

Researchers have proposed questions and dimensions to characterise SASs. Salehie and Tahvildari (2009) presented an overview of self-adaptive software and the 5W1H questions: *When*, *Why*, *Where*, *What*, *Who*, and *How*. Krupitzer et al. (2018) presented an extensive literature review and also proposed the same set of questions except “*Who*” because they considered that the nature of SAS requires automatic adaptation, which is based on using a control loop. Other authors have also used similar questions (McKinley *et al.*, 2004; Buckley *et al.*, 2005). The questions from Salehie and Tahvildari (2009) and Krupitzer et al. (2018) and their corresponding dimensions and examples are shown in Table 2.3 and explained as follows.

“**When to adapt**” relates to **time**, whereby an adaptation can be either reactive or proactive. Reactive is applied when a software system acts after an event happens, whereas proactive is applied before an event happens. Reactive and proactive adaptations consist of similar activities regarding *monitoring*, *planning*, and *executing*, but differ when it comes to *analysing*. Reactive adaptation analyses the monitored data to check if an adaptation is required based on some decision criteria (e.g., rules), whereas proactive adaptation uses the monitored data to forecast the system behaviour or environmental state. Additionally, reactive adaptation could cause a delay if the adaptation is time-consuming, while proactive adaptation prepares the change in advance to reduce the delay. However, the suitability of the prediction algorithms, which are used for proactive adaptation, are dependent on the specific

prediction scenarios, and faulty predictions could cause suboptimal adaptations. Furthermore, the *time* at which an adaptation is needed is linked to the *monitor* and *analyse* components in MAPE-K since the monitored elements are used to determine whether an adaptation is needed.

“**Why** *do software systems adapt*” relates to the **reason** for the adaptation based on changes in the context or the available resources. Hence, the *reason* for performing an adaptation helps in determining the elements that the monitor component monitors. Furthermore, the *reason* for performing an adaptation is related to the *analyse* component in MAPE-K since it determines whether adaptation is needed.

“**Where** *should a change be implemented*” relates to the **level** at which the adaptation occurs in a SAS. As mentioned in Section 2.4.1, a SAS is composed of an autonomic manager and a managed system. Since the autonomic manager often stays the same (i.e., does not get adapted), then the *level* will be related to the managed system’s elements (e.g., tasks and resources). Furthermore, the *level* at which the adaptation occurs is related to the *analyse* component in MAPE-K since it determines where adaptation is needed.

“**What** *type of change is needed*” relates to the applied adaptation **technique** such as parameter, structure, and context adaptation. Parameter adaptation changes the data values of a software system’s parameters (e.g., changing the speed of a robot). Structure adaptation changes the behaviour of a software system (e.g., by replacing a software component with another). Context adaptation relates to explicit adaptation of the environment and resources via the control loop (e.g., changing the physical space that a robot surveys). Furthermore, the *type of change* is related to the *plan* component in MAPE-K since it determines the actions that should be performed to adapt the managed system’s elements.

“**Who** *shall invoke the adaptation*” relates to the **executor** that denotes the level of automation and human involvement in the SAS. The autonomic manager performs adaptation in an automated manner with minimal human intervention. Nonetheless, human involvement can be valuable to improve the manageability of a SAS (e.g., by providing feedback on the adaptation). Furthermore, the executor is related to the *execute* component in MAPE-K since it carries out the actions needed to adapt the managed system. Increasingly there are human-in-the-loop approaches that aim to address interactions and promote collaborations between humans and machines (Cleland-Huang *et al.*, 2022).

“**How is the adaptation performed**” relates to the **control** over the adaptation through decision criteria (e.g., rules and goals) and the degree of decentralisation (e.g., centralised and decentralised). Furthermore, the *control* over the adaptation is related to the *plan* component in MAPE-K since it determines the actions needed to adapt the managed system’s elements.

2.4.3 Self, context, and resources in self-adaptive systems

Self and Context: Salehie and Tahvildari (2009) differentiate between a software system’s self and context. The word “self” denotes the whole multi-layer software system. On the other hand, the word “context” encompasses elements that exist in the operating environment of a software system and affect its properties and behaviour. Hence, they perceive a self-adaptive software system as a closed-loop that receives feedback from itself and its context. A self-adaptive system aims to adjust itself during its operation based on changes from the software system’s self (internal causes) or its context (external events).

Resources versus self and context: Resources that impact the operation of a software system can be diverse as explained in Section 2.2. Based on the definition of Salehie and Tahvildari, it is possible to consider resources to be elements of a software system’s self (e.g., software component) or context (e.g., hardware). However, as explained in Section 2.2, this thesis considers resources as entities needed to carry out tasks; these entities differ from elements of a software system’s self or context in terms of representation and (re)allocation.

Resources can be (re)allocated by a system based on feedback from itself and its context. For example, memory, processing capacity, and the number of allowed connections to a database can be (re)allocated based on things like user privileges, the task at hand, and the time of day. However, this does not apply to context elements such as the type of user (e.g., novice or expert), type of platform (e.g., operating system), and environment-related conditions (e.g., weather). The (re)allocation of resources can involve restriction, shifting, and substitution.

2.5 Resource-driven adaptive systems

A resource-driven adaptive system (RAS) is a type of SAS where the trigger for adaptation is the variability of resources (refer to Section 1.1). In a RAS, the managed

Table 2.4 – Examples of software systems (un)affected by resource variability

Software	Resource Type	Affected by Resource Variability
Calculator	CPU and RAM	×
Text editor	CPU and RAM	×
Warehouse	Robot and Packing Materials	✓
Manufacturing	Raw Material and Factory Machinery	✓

system relies significantly on resources and is therefore a resource-driven system. For example, a “calculator” is a single-user application that relies on a very small amount of CPU and RAM to operate. This means even when the CPU and RAM are facing variability, it is still possible to run this application without adaptation. Hence, the “calculator” is not considered to be resource-driven. On the other hand, an automated warehouse system relies significantly on resources like robots and packing materials to execute tasks that are initiated by its users. The tasks in an automated warehouse system could be competing for scarce resources. Hence, this system is highly affected by resource variability. Table 2.4 shows examples of software systems that are either affected or not affected by resource variability. In a RAS, resources are monitored via the autonomic manager’s monitor component. Additionally, the trigger for adaptation is related to the unavailability or scarcity of resources.

2.6 Chapter summary

This chapter presented background information about concepts that are relevant to this thesis. These concepts include resource types, tasks, and self-adaptive systems. This chapter classified resource types (e.g., robots and materials) under four resource groups static, dynamic, reusable, and depletable. Moreover, it presented examples of tasks and their related resource types from multiple software systems. Furthermore, this chapter presented questions and dimensions of SAS based on the existing literature and explained the components of an autonomic manager as proposed by the MAPE-K control loop. Finally, this chapter explained what types of systems this thesis considers to be resource-driven and how resource-driven adaptive systems (RASs) relate to SASs.

3

Literature Review

This chapter presents a critical analysis that covers the strengths and shortcomings of related work about resource-driven adaptation approaches and task modelling notations. The critical analysis involves discussing the types of resources, adaptation, and software systems that are supported by existing resource-driven adaptation approaches. Furthermore, this chapter reviews existing task modelling notations based on the types of operators and tasks that they support and other characteristics that would specifically benefit resource-driven adaptation. This chapter presents tables that summarise information about the related work.

3.1 Introduction

Resource-driven adaptation approaches are categorised in this chapter according to their key characteristics, which are related to how the adaptation is performed and what is adapted. Some approaches follow the brownout paradigm and temporarily deactivate optional parts of a software system during resource variability (Xu and Buyya, 2019). Other approaches are task-based and do not just deactivate a part of a software system but adapt while considering the tasks that a system executes (Rigole *et al.*, 2007). Scheduling approaches also consider tasks but are particularly concerned with optimising one or more performance measures like the total completion time of tasks (Gawiejnowicz, 2020). The source code of software systems is modified by code-based approaches to reduce the consumption of resources (Christi, Groce and Wellman, 2019). Some approaches use policies (rules) that represent a choice concerning the behaviour of a system (Keeney and Cahill, 2003) while others use

Dynamic Software Product Lines (DSPLs) to produce software systems that are capable of adapting to changes at runtime (Pascual, Pinto and Fuentes, 2015). Reference architectures have also been proposed (Garlan, Poladian, *et al.*, 2004). Moreover, query-based approaches either optimise queries or filter their results to avoid wasting resources due to returning unnecessary data (Gotz *et al.*, 2015).

As mentioned in Section 1.3, task models can serve as input for resource-driven adaptation. Task models are represented using task modelling notations. These notations are beneficial because they support the decomposition of tasks into subtasks that could differ in terms of their priorities and the resources that they require. Task modelling notations can represent several types of tasks and depicts how each task is temporally related to other tasks. Several task modelling notations have been proposed in the literature (Martinie *et al.*, 2019). These notations model tasks and their relationships, mostly by using a graphical representation. This chapter examines existing task modelling notations to see if they support characteristics that are useful for resource-driven adaptation, including properties that represent the priorities of tasks and the types of resources that they use, task variants are useful for performing adaptation by executing similar tasks that consume fewer resources, and properties that indicate which types of adaptation apply to a task. Furthermore, this chapter discusses other characteristics of existing task modelling notations, namely operators and task types. These operators and task types are not specifically meant for supporting resource-driven adaptation but are rather useful for representing tasks and their relationships for any type of software application (including those that need to adapt based on resource variability).

3.2 Resource-driven adaptation approaches

This section presents the state-of-the-art resource-driven adaptation approaches and discusses their strengths and shortcomings with examples. It is important to discuss these approaches because the work presented in this thesis uses resource-driven adaptation to address resource variability.

3.2.1 Brownout

The “brownout” paradigm involves the temporary deactivation of optional parts of a software system, including components and web-page contents; it was inspired by

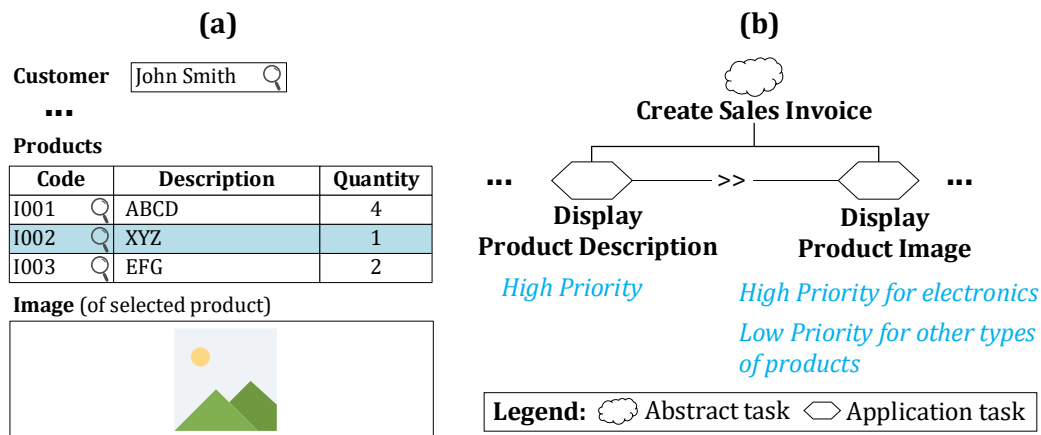


Figure 3.1 – Example of UI and tasks: excerpt from the content of an ERP’s invoice UI (a) and an excerpt of the corresponding task “create invoice” (b)

and named after the intentional voltage drops often used in electrical grids to prevent blackouts through load reduction in case of emergency (Klein, Maggio, *et al.*, 2014). Klein, Maggio, *et al.* (2014) proposed the brownout paradigm to cope with resource variability on the cloud infrastructure due to hardware failures and varying user workload levels without over-provisioning resources. Afterwards, several approaches adopted “brownout” to handle situations such as hardware failures and flash crowds in cloud computing systems (Xu and Buyya, 2019). The brownout approaches are relevant to this thesis because their primary aim is to handle resource variability through adaptation. The brownout paradigm is inspirational because it shows how certain parts of an application can be downplayed in favour of keeping other more important parts running during resource variability. Brownout gracefully stops part of a software system using adaptation, instead of stopping the entire system all at once due to the lack of resources.

Although brownout approaches differ in their objectives, they perform the same type of adaptation that involves the activation and deactivation of optional parts. For example, Moreno *et al.* (2015) proposed a proactive approach to address adaptation latency, which is the lag between performing an adaptation and the effect that is produced from it, Zhao *et al.* (2017) proposed a framework to generate adaptation rules for user goals, and Pandey *et al.* (2016) combined two planning approaches to handle the trade-off between timeliness and optimality of the adaptation plan. The three abovementioned approaches were all evaluated using the brownout example of Klein *et al.* (2014), which involved adapting a web application by deactivating its

optional parts when the CPU (resource) is facing variability. Similarly, Tomas et al. (2014) deactivate optional parts to address the issue of applications that are accepted to run on a cloud server, which does not have sufficient resources. Software systems would have more versatility in coping with resource variability if they support multiple types of adaptation, rather than just deactivating optional parts, because some types of adaptation might not work in some cases. For example, if part of a software system performs mandatory work that cannot be deactivated an alternative could be to process this work in the background at a later time when resources become available. The software system would notify the user when the work is done. This applies the substitution of inoperative robots in an automated warehouse with other types of robots instead of deactivating part of the software system and cancelling the work.

In the brownout paradigm, a controller manages the activation and deactivation of the optional parts based on trade-offs between metrics (Maggio, Klein and Arzén, 2014). The work of Klein, Papadopoulos, et al. (2014) and Dürango et al. (2014) involved performing trade-offs between response time and user experience. Klein, Papadopoulos, et al. (2014) extended the work of Klein, Maggio, et al. (2014) to apply brownout on multiple replicas (rather than just one) of an application using two novel load-balancing algorithms. Dürango et al. (2014) extended the work of Papadopoulos, et al. (2014) by considering other load-balancing algorithms and performing an additional evaluation. The work of Xu, Dastjerdi, and Buyya (2016) and Hasan et al. (2016) involved performing trade-offs between energy consumption and revenue. Xu, Dastjerdi, and Buyya (2016) worked on reducing energy consumption in cloud data centres by performing a trade-off between energy consumption and giving users a discount on the price. In their approach, if the user experience is degraded to reduce energy consumption, then users get a discount on the price. Hasan et al. (2016) aim to achieve green energy awareness in cloud applications. They disable optional web content to increase the usage of green (clean) energies and decrease the usage of brown (polluted) energies while performing a trade-off between user experience and the service provider's revenue. In the brownout approaches, the controller that manages the activation and deactivation of the optional parts has a parameter that represents the probability of running the optional parts. The controller reduces the value of this parameter when the software system is close to saturation. Hence, for example, the optional part may be deactivated or served for every second user. However, performing task prioritisation using multiple criteria provides a more



Figure 3.2 – Example of a component and tasks: two tasks from an ERP system use the same component to export documents to images

granular way of deciding which parts to adapt so the resources are used by the tasks that need them most. For example, some users have more privileged roles and some tasks are more important at specific times of the day. Hence, instead of considering a part of a software system to be either optional or mandatory, its importance would be more accurately represented by prioritising the tasks that use it. This makes it possible to vary adaptation choices among tasks. What follows is an explanation of this point using two examples from an ERP system, one related to the contents of an invoice, shown in Figure 3.1, and another one related to invoice and profit report tasks that use the same components, shown in Figure 3.2.

Consider the example presented in Figure 3.1a, which shows part of an invoice UI from an ERP system, whereby the creation of an invoice involves displaying product information like the description and the image. Consider that displaying a product’s image is optional and can be deactivated when there is a need to avoid saturation (if an application saturates it will be unable to serve users in a timely manner). However, instead of considering the image part to be always optional and deactivating it for random users, its importance would be more accurately represented using a task priority. For example, displaying the image of a product could have a high or a low priority depending on the types of products that the end-user selected, whereby each enterprise has particular values for these types of products (e.g., books and phones).

To clarify further how tasks are related to the product image part of the invoice, consider that the task called “Create Invoice” that is shown in Figure 3.1b corresponds to the invoice example from Figure 3.1a. This task is presented using ConcurTaskTrees (CTT), which is a notation for representing task models hierarchically using a graphical syntax (Paterno, Mancini and Meniconi, 1997; Mori, Paternò and Santoro, 2002). The

task “Create Invoice” has subtasks that include “display product description” and “display product image”. The priority of the task “display product image” would differ according to the value that the end-user chooses for the task’s parameter “product type”.

Consider the example presented in Figure 3.2, which shows a “document to image exporter” component that multiple tasks from an ERP system use. In this case, the component is not optional. Hence, it cannot be deactivated as is done by the brownout approach. Sun et al. (2019) suggest substituting components with alternatives that are less resource-intensive during situations of resource variability. However, instead of substituting the “document to image exporter” component with an alternative one that uses a lower resolution across the entire software system, it is more effective to substitute a task with a similar one depending on the tasks’ priorities. This way the adaptation can vary according to the task rather than being applied to components and disregarding the differences among the tasks that share the component. For example, if the task “Export Invoice to High-Resolution Image” has a higher priority than the task “Export Profit Report to High-Resolution Image”, then the latter can be substituted by a similar task called “Export Profit Report to Low-Resolution Image”. The tasks will still call the same component but with different parameters thereby allowing the high-priority tasks to benefit from a high-resolution output while the low-priority tasks would still run but with a lower resolution to conserve resources. Furthermore, the priority of the task could differ according to the user who is initiating it. Hence, for example, the task “Export Profit Report to High-Resolution Image” has a low priority for all user roles except the manager. Moreover, consider that the “product image” part of the invoice shown in Figure 3.1a is a visual component that is reusable across several UIs. In this case, the component may be optional for the “Create Invoice” task but mandatory for the “Create Product” task. Considering this example, it is possible to say that some resource-intensive parts of a software system are not always optional. Hence, considering the tasks that are executed by software systems provides the necessary granularity to make accurate decisions on when to adapt based on task priorities.

A few brownout approaches have considered prioritising components to decide which ones to deactivate first. Xu et al. (2016) prioritised components based on usage frequency while Sun et al. (2019) also added computational complexity. However, this prioritisation of components does not consider how tasks vary according to multiple criteria such as the time of the day when the task is initiated and the role of the user

who initiated it. Usage frequency is important but is not sufficient for prioritisation if it is not complemented by a managerial view of priorities. For example, management in an enterprise could decide that keeping high-resolution images in an inventory report is of a higher priority than keeping them in a sales report that is more frequently used. Considering both the end-user and managerial perspectives of priorities accommodates the interests of stakeholders during situations of resource variability. Furthermore, Sun et al. (2019) measured component complexity, which they used for prioritisation, by counting the number of files (.class, .html, and .xml) that define a component. This technique does not accurately determine resource intensiveness. For example, if a component used robots as a resource the number of files it has does not indicate how many robots are needed. Alternatively, if tasks were prioritised, high-priority tasks would be given access to the resources that they need while adaptation would be applied to low-priority tasks during situations of resource variability.

Resource types vary from one system to another. For example, some systems primarily depend on reusable hardware resources such as RAM and CPU, while other systems depend on depletable resources such as fuel and raw materials. The existing brownout approaches mostly target CPUs as resources and one work by Nikolov et al. (2014) targets networks. As Xu and Buyya (2019) indicate, brownout approaches would become more comprehensive if they supported multiple types of resources.

3.2.2 Task-based

Some approaches have considered tasks as part of the resource-driven adaptation decision-making process. These approaches target resource-driven adaptation in smart spaces such as smart homes (Wu and Fu, 2011). Rigole et al. (2007) presented an approach for gradual component deployment based on the tasks initiated by end-users to avoid the needless consumption of computing resources and latencies on lightweight mobile devices. A system was developed based on Project Aura (Garlan *et al.*, 2002) to adapt the Quality of Service (QoS) to changes in the environment and the end user's preferences according to the task that he or she is performing (Sousa *et al.*, 2006). An approach was presented for composing services to maximise the QoS for the currently active user task in smart spaces (Davidyuk, Ceberio and Riekkki, 2007). Gajos (2001) presented a system called Rascal for resource mapping and conflict arbitration, whereby resource mapping is similar to web service discovery (i.e., finding a suitable service for a given task) and arbitration decides which task requests are more

important when resources are scarce. The relation between this part of the literature and my thesis is the consideration of tasks, which provides granularity in adaptation decisions because tasks that use the same resources could be different according to their priorities and resource consumption. It is important to analyse this part of the related work to understand how existing approaches use tasks as part of their adaptation process and how they represent these tasks.

Some existing task-based approaches do not address resource variability in scenarios where end-users are initiating tasks that are competing for resources. Rigole et al. (2007) and Sousa et al. (2006) focus on the perspective of a single user who is performing a task on a mobile device. However, there are other cases like enterprises where many end-users (e.g., employees and customers) initiate tasks at the same time, and software systems should decide which tasks to execute as requested and which ones to adapt due to resource limitations. Multi-user scenarios differ from single-user ones because the adaptation must maintain the interest of stakeholders regarding which tasks are more important during resource variability. For example, in enterprises, it is important to consider the perspectives of end-users and corporate management. Hence, even if an end-user wants to execute a resource-intensive report at a specific time of the day, corporate management may consider that other tasks have a higher priority during this time.

Moreover, Perttunen et al. (2007) adopt a leasing mechanism that gives users access to shared resources, such as a screen at an airport, for a limited amount of time. Consider the previously explained invoice example from Figure 3.1 or another example concerning an order preparation task at a warehouse that is automated by robots. Leasing is not ideal for such examples since the tasks require using the resources until completion. Furthermore, as Gajos et al. (2001) state, they decide on the importance of a request using “self-assigned need levels that they describe as a very simple and arbitrary scheme that can be replaced by another system”. Hence, it would be useful to consider task prioritisation that is not arbitrary but rather takes into account multiple criteria.

The adaptation performed by Rigole et al. (2007) focuses on loading software components just-in-time rather than resource variability scenarios. Perttunen et al. (2007) focused on service composition while Garlan et al. (2004) and Sousa et al. (2006) did not discuss specific types of adaptation. As mentioned in Section 3.2.1, software systems would have more versatility in coping with resource variability by

supporting multiple adaptation types. Furthermore, these approaches focused on resources such as the computational resources on mobile phones and hardware equipment that exist in smart environments. However, other types of resources such as depletable resources were not considered.

Concerning task representation, Rigole et al. (2007) represent task models using an existing task modelling notation, namely CTT (used for the example in Figure 3.1b). The other existing works discussed the concept of a task but did not use or present a way for representing the tasks. The representation of different types of tasks using a task modelling notation is useful because some types of adaptation only apply to some types of tasks. For example, tasks such as report generation can be postponed to another time and performed in the background. On the other hand, postponement could negatively impact usability when a task is followed by subsequent user interactions to initiate other tasks (e.g., searching for products and selecting a product from the result). CTT already supports multiple types of tasks and operators. Hence, it presents a viable starting point to represent tasks for resource-driven adaptive systems. Nonetheless, CTT is missing characteristics like task variants (e.g., as demonstrated by the example in Figure 3.2) that are useful for resource-driven adaptation. Furthermore, it would be useful to extend CTT with properties like a task's required resources and priority and task types that are useful for deciding whether an adaptation type applies to a task.

3.2.3 Scheduling

Scheduling approaches aim to solve the problem of allocating resources to tasks in addition to the optimisation of performance measures. These approaches are part of the related work in the sense that they consider that resources, such as computational resources (Zhan *et al.*, 2015) and machinery (Zhou, Zhang and Horn, 2020), are scarce and should be managed in a way that enables the tasks that need them to execute. Hence, they aim to accommodate the execution of tasks within the limitations of the available resources.

Gawiejnowicz (2020) indicates that scheduling approaches mainly use offline algorithms, which require their input to be fully available before executing. However, the input that is required for scheduling tasks is not always available ahead of time. For example, resource variability produces new information at runtime (i.e., while tasks are executing). Additionally, Gawiejnowicz (2020) define scheduling problems by

saying that no resources other than machines are needed to complete the tasks. In this case, machines could represent several types of reusable resources like robots or manufacturing equipment. However, some systems also require depletable resources, which are either available or unavailable. When depletable resources are nearing depletion and replenishment is delayed measures should be taken at runtime to keep these resources available for the critical tasks that need them most. For example, when medicines are facing an unexpected delay in replenishment, the remaining stock should be used for treating the most critical medical conditions. Moreover, changing a task's execution schedule is not always suitable. For example, the end-users of an ERP system expect an immediate result when searching for a product while creating an invoice. Hence, this task cannot be delayed and another type of adaptation is needed.

Mohan et al. (2019) indicate that the majority of existing work is based on static demand and deterministic processing time. Static scheduling problems make several assumptions (Pinedo, 2018). What follows are five of these assumptions and an explanation of how they are limiting in environments that have unforeseen changes. First, all tasks that are going to be processed are available when the software system starts processing the tasks. However, this is not always the case. For example, users can initiate tasks via a software system at varying times during the day thereby creating a dynamic demand. Second, if additional tasks arrive during the day then they would have to be processed the next day. However, there are cases when it is vital to process critical tasks promptly even if these tasks arrive during the day. Third, the machines are always available during the processing period. This is not possible in environments that face resource variability. Fourth, the processing times of the tasks are known and deterministic. However, there are cases where the processing times of tasks are unknown, and these times can change for tasks of the same type (e.g., the time it takes to prepare an order differs from one order to another). Fifth, a task is done using one resource (one machine). However, multiple resources could be needed to execute a task even if it is a sub-task of a parent task. Hence, the relationship between resources and tasks is M-N rather than 1-N. For example, the task of packing products at an automated warehouse requires a robot in addition to boxes and other packing materials.

Based on the abovementioned reasons, environments that face unforeseen changes like resource variability and do not have a static demand for tasks require a dynamic approach to allocate resources to tasks. Priority-generating functions that assign

priorities to tasks are commonly used for scheduling (Strusevich and Rustogi, 2017). However, the generation of those priorities becomes challenging in dynamic environments due to several reasons. First, there is limited availability of resources due to resource variability. For example, an automated warehouse could unexpectedly lose a significant part of its robot capacity due to hardware failures. Second, multiple criteria should be taken into account (e.g., the capabilities of resources like robots). Third, there are multiple competing agents. For example, in an automated warehouse system, multiple robots need boxes to pack orders and multiple tasks need robots to be executed. Fourth, there are multiple types of resources. For example, robots are reusable resources and boxes are depletable resources. Scheduling problems such as flow shop and job shop are NP-Hard. Furthermore, the combination of the abovementioned points makes the scheduling problem intractable (Gawiejnowicz, 2020). Hence, a heuristic is needed to calculate priorities and provide a solution for resource allocation in complex environments.

3.2.4 Code-based

Some adaptation approaches modify the source code of software systems to create new versions that are better fit for some situations. It is useful to explore these approaches as part of the related work to see how they apply adaptation to a software system's source-code and how this compares to performing adaptation while taking tasks into consideration. It is also useful to observe the impact of performing adaptation on the source-code on the ability of these approaches to address the resource variability problems in multiple types of software systems.

Code-based approaches work at design time. However, resource-driven adaptation must be done at runtime because some information, like task priorities and the cost of applying a type of adaptation, are unknown at design time. For example, consider that two enterprises are using the same software system. The task "Export Profit Report to High-Resolution Image", shown in Figure 3.2, could have a high priority for one enterprise and a low priority for another. Furthermore, the cost of applying an adaptation type changes based on input collected from stakeholders such as systems administrators and end-users. For example, concerning the task "Export Profit Report to High-Resolution Image" (Figure 3.2) end-users could favour exporting a report to a low-resolution image to get a result directly when resources are facing variability or they could favour delaying the task's execution until resources become available to get

a high-resolution image. Therefore, adaptation approaches that perform source code reduction at design time are not suitable for addressing resource variability.

A code-based design-time approach was presented to adapt software systems based on available resources (Christi, Groce and Wellman, 2019). This approach performs source-code reduction to make software systems consume less RAM. Annotations are added by programmers on the unit tests to identify which parts of the system are optional and can be removed when RAM consumption is above a threshold. The authors used the NetBeans IDE as a case to assess their work. The main limitation of this approach is that the software system must be recompiled to get an adapted version because the adaptation involves direct changes to the source code. Furthermore, as Christi et al. state, their approach is only valid for Java programs and does not generalise to programs written in other programming languages. Technology independence would improve if prioritisation and adaptation decisions were made at a higher level of abstraction such as tasks instead of being applied to source code. Yan et al. (2019) proposed an approach for adjusting the configuration of software systems to reduce power consumption on Unmanned Aerial Vehicles (UAVs). This approach analyses source code at design time and adjusts the configuration at runtime to avoid having to recompile the software system. However, this approach is specific to power consumption in UAVs and does not apply to other types of software systems and resources.

Huang et al. (2017) and Shao et al. (2014) presented approaches that particularly target mobile apps. The approach of Huang et al. (2017) reduces Android apps by removing the code elements of unwanted features to reduce the consumption of power, CPU, and bandwidth. Even with code modification aside, the idea of removing features from the software system is different from resource-driven adaptation. For example, if there is variability in resources such as CPUs and robots, the software system using these resources should adapt without permanently removing features that are required by the end-users. Huang et al. (2017) assume that in some software systems, some features can be removed because they are not being used at all by the end-users. However, this is not the case in other systems where required features face resource variability. The approach of Shao et al. (2014) detects whether Android apps have been repackaged to avoid the possible insertion of malware. This approach is described as resource-driven because it analyses app resource files such as layout and styles to detect changes. However, although these resources are static and reusable,

they do not face variability since every app has its copy of these files. Hence, the approach of Shao et al. (2014) does not aim to address resource variability through adaptation.

3.2.5 Policy-based

A policy is a rule that represents a choice concerning the behaviour of a system. For example, positive and negative authorisation policies define actions that subjects are authorised or unauthorised to perform respectively (Damianou *et al.*, 2001).

Chisel (2003) uses a policy language to define rules for adapting mobile systems that use services. Chisel dynamically inspects and adapts software systems using Iguana/J (Redmond and Cahill, 2000) with reflection (an API for examining and modifying the behaviour of methods and classes at runtime). The policies demonstrated by Chisel change configuration values to alter the behaviour of the system as a whole, but do not consider how individual tasks should be adapted when these configurations are not sufficient. Similarly, Efstratiou et al. (2002) use policies that include actions on a system-wide level to support the coordination of the adaptive behaviour of multiple mobile applications that share the same reusable resources. For example, one of Chisel's policy examples initiates caching behaviour when memory is low. Consider that instead of memory the number of robots in a warehouse is low due to several unexpected hardware failures. In this case, changing a system setting is not enough to compensate for the missing robots, but performing task prioritisation and adaptation keeps the resources available to the most important parts of the work. Furthermore, using reflection to apply adaptations to the application could have performance implications, but these were not discussed.

VOLARE adapts service requests at runtime based on rules defined using a policy language (Papakos, Capra and Rosenblum, 2010). Like some of the work discussed in Section 3.2.2, VOLARE focuses on single-user mobile application scenarios where the adaptation is based on the resources available in the environment and user preferences, but multi-user scenarios are not considered. For example, the case study done by VOLARE involves binding a mobile app to streaming services whose bitrates depend on the connection speed that is available for the mobile device. If this example was considered for a multi-user scenario, the adaptation will be done based on which tasks are more important rather than just individual user preferences. For example, if many robots at an automated warehouse were malfunctioning the remaining robots

that are working on low-priority tasks could be diverted to work on high-priority tasks until the other robots are repaired.

Policies are also used to maintain service level agreements (SLAs) by setting the QoS (e.g., gold or silver) for software systems that use shared resources (Bandara *et al.*, 2004). David and Ledoux (2003) and Buisson *et al.* (2005) presented frameworks that enable software developers to specify policies for adapting components when resource availability changes at runtime. These policies are useful. However, more granularity in the adaptation decisions could be provided by considering the differences among tasks like priorities and resource consumption rather than adapting the software system or its components as a whole. Hence, for example, instead of considering that an entire software system has a high priority, multiple tasks from several software systems that are sharing resources could benefit from this privilege depending on their importance (e.g., for an enterprise). Furthermore, different types of adaptation could be suitable for each task. For example, usability is hindered by the postponement of tasks that are followed by subsequent user interactions to initiate other tasks (e.g., searching for products while creating an invoice on an ERP system). However, this type of adaptation could be possible for other tasks like the generation of non-critical reports.

3.2.6 Architecture-based

MAPE-K (Kephart and Chess, 2003) and the three-layer architecture (Kramer and Magee, 2007) serve as references for developing self-adaptive systems. Rainbow refines the MAPE-K control loop by adding a resource discovery mechanism to check the resources of the managed system (Garlan, Cheng, *et al.*, 2004). This resource discovery mechanism was also added to MORPH (Braberman *et al.*, 2017), which is a reference architecture that takes inspiration from Rainbow and the three-layered architecture and targets the adaptation of system configuration and behaviour. Rainbow's resource discovery mechanism is limited to discovering and replacing software components, like a video conferencing gateway with existing ones, and it was implemented in a prototype that uses the network-sensitive service discovery mechanism (Huang and Steenkiste, 2003). Although this approach is feasible for software components, it would not work for other types of resources. For example, depletable resources cannot be discovered for direct use since these are delivered by a supplier and are affected by delays in the supply chain. Additionally, consider other resources such as robots. Even if existing resources were discovered these resources

cannot be used without taking into consideration task priorities. Otherwise, low-priority tasks could be given access to resources that are needed by high-priority tasks. Furthermore, as Garlan, Cheng et al. (2004) note, Rainbow could be enhanced with proactive capabilities that will allow it to find improvement opportunities in an anticipatory manner.

Huber et al. (2014, 2017) presented a domain-specific language for describing runtime adaptation at the system architecture level. The adaptation they performed involved dynamically adding and removing virtual CPUs and application servers (resources). Similarly, the SLAstic framework supports dynamic allocation and deallocation of data centre resources through node allocation and deallocation and load balancing (Van Hoorn *et al.*, 2009). However, systems that are unrelated to virtualised environments use other types of resources like robots and raw materials, which cannot be added instantly on demand. Hence, these systems should adapt the way tasks are executed during resource variability using other types of adaptation like changing a task to a similar one that requires fewer resources.

3.2.7 Query-based

A few existing resource-driven adaptation approaches are query-based in the sense that they either directly target query optimisation or filter the results of queries to avoid returning unnecessary data that wastes resources.

Viswanathan et al. (2018) devised an approach for query optimisation that takes resources into account to avoid performance loss in big data systems. Query optimisation is vital in the area of databases and incorporating resources into it is useful, but the approach of Viswanathan et al. is limited to this area and does not apply to other systems where resources differ, and the required adaptation does not merely involve tuning a query to make it faster. Gotz et al. (2015) presented an adaptive knowledge exchange technique that uses runtime models to manage the consumption of energy and memory in cyber-physical systems. The technique presented in this paper is specific to scenarios where robots clean rooms. However, the mentioned resource types are limited, and the meta-model does not illustrate how resources are associated with the software system. Furthermore, the adaptation performed by this technique is specific to changing the amount of data that is being transferred between robots via queries to reduce battery usage based on robot types and states. Hence, it does not generalise to other cases of resource variability.

3.2.8 Dynamic software-product-line

Software Product Lines (SPLs) are used at design time to tailor software systems by creating variations from a set of features, while Dynamic Software Product Lines (DSPLs) are used to produce software systems that are capable of adapting to changes at runtime (Hallsteinsen *et al.*, 2008).

A few approaches have used DSPLs to support resource-driven adaptation. One approach used a genetic algorithm to automatically generate optimal software system configurations from a feature model at runtime according to the available resources (Pascual, Pinto and Fuentes, 2015). This approach helps software systems to cope with resource-constrained environments to improve their performance. Features are mapped to components that are added, removed, or given updated parameter values when a software system needs to adapt. However, working with tasks provides more granularity in adaptation decision-making because multiple tasks that use the same component could differ in terms of priorities and resource consumption. Saller et al. (2013) presented an approach that is concerned with overcoming resource limitations that inhibit the use of DSPLs on mobile devices because of the inability to deploy and explore a complete configuration space due to limited memory and processing capabilities respectively. However, the work of Saller et al. is not concerned with adapting software systems due to resource variability since its concern with resources is limited to the performance of DSPLs. Hence, it does not aim to address resource variability through adaptation.

3.2.9 Other approaches

SARDE is concerned with acting as a self-adaptive ensemble resource demand estimation approach (Grohmann *et al.*, 2021). SARDE dynamically and continuously tunes, selects, and executes an ensemble of resource demand estimation approaches to improve the resulting estimation accuracy. Therefore, its adaptation is related to the tuning and selection of the approaches that are part of this ensemble. On the other hand, this thesis is concerned with making resources available for tasks in the cases where they are most needed and considering viable alternative task execution options when resources are unavailable. Furthermore, SARDE focuses on cloud computing applications and their resources (e.g., estimating resource demands for elastic cloud resource management and auto-scaling where the resources are mainly CPUs).

However, it does not cover a variety of resource types (depletable and reusable). Auto-scaling enables the automatic increase or decrease of cloud services like server capacities when needed. Resource demand estimation for auto-scaling is specific to cloud computing and is not always feasible in other scenarios. For example, it is not possible to auto-scale depletable resources that are facing shortages due to a sudden problem in the supply chain. Additionally, it is costly to compensate for short-term resource variability by over-provisioning reusable and depletable resources such as robots and raw materials respectively. Furthermore, with depletable resources in particular there is no time spent to obtain a service from a resource (resource demand) as assumed by SARDE. The depletable resource is either available for a task to use directly or unavailable, hence the task cannot execute and could be adapted.

A three-way adaptation technique was proposed for optimising the usage of the available resources to satisfy a set of requirements (Bennaceur *et al.*, 2019). This adaptation technique consists of three steps (1) available resources are used for satisfying a set of requirements; (2) unavailable resources are substituted with similar available resources; (3) requirements are adapted based on the available resources. The motivating example for the three-way adaptation is based on a meal planning system, where recipes are considered as requirements and ingredients as resources. Unlike the previously discussed approaches, three-way adaptation targets food, which is a depletable resource. It is interesting to observe in this work an example of depletable resources, namely food, that are affected by resource variability, which is the problem that this thesis aims to address. However, as previously mentioned it would be useful to consider multiple types of resources and tasks as part of the adaptation process.

Samin et al. (2022) presented Pri-AwaRE, which performs decision-making for SASs while considering the priorities of non-functional requirements (NFRs). Hence, their approach performs adaptation with trade-offs between NFRs at runtime. Similar trade-offs are performed by the brownout approaches that were discussed in Section 3.1 (e.g., a trade-off between response time and user experience). However, Pri-AwaRE performs prioritisation, unlike most brownout approaches. Nonetheless, there are cases where the priorities differ by task and not just by NFR. Consider the example shown in Figure 3.1. In this example, response time could be overall more important than user experience for a variant of the task “Display Product Image” while user experience could be more important for other variants.

3.3 Task modelling

As discussed in Section 3.2, considering tasks and their differences, like priorities and resource consumption, is useful because it provides granularity in the decision-making process of a resource-driven adaptation approach. In this regard, a notation is needed to model the tasks of a software system, whereby these models would be used as input when making adaptation decisions. Hence, this section first provides an overview of existing task modelling notations and then discusses their shortcomings in the development of software systems that support resource-driven adaptation. The representation of task variants is one of the important characteristics for supporting resource-driven adaptation. Feature models can represent variations among features. Hence, this section also briefly discusses feature modelling notations to clarify how feature models differ from task models and why task models are more adequate for supporting resource-driven adaptation in software systems.

3.3.1 Task modelling notations

Task modelling notations are useful for representing the tasks and relationships of software systems as task models (Martinie *et al.*, 2019). Task models describe how to perform activities by depicting how a task is divided into subtasks and how these subtasks are temporally related. For example, an automated warehouse system would comprise a task called “prepare customer order”, which is divided into sub-tasks like “locate product in the warehouse” and “pack product in a box”. In this example, the task “locate product in the warehouse” is executed first and is followed by the task “pack product in a box”.

Several task modelling notations were proposed as described by existing surveys (Limbourg and Vanderdonckt, 2004; Guerrero-García, González-Calleros and Vanderdonckt, 2012; Martinie *et al.*, 2019). Furthermore, these notations have been used by model-based software development approaches that target user interfaces (Calvary *et al.*, 2003), games (Vidani and Chittaro, 2009), and collaborative learning systems (Molina *et al.*, 2014).

3.3.1.1 Representation of task models

Task modelling notations represent task models differently. Some notations like UAN (Hartson and Gray, 1992) and GOMS (Kieras, 2004) are textual, while other notations like CTT (Paterno, Mancini and Meniconi, 1997; Mori, Paternò and Santoro, 2002) and HAMSTERS (Martinie, Palanque and Winckler, 2011) are graphical.

Notations that describe the activities of software systems use different forms of representation. For example, “Yet Another Workflow Language” (YAWL) and “Business Process Model and Notation” (BPMN) adopt a graph representation, which enables them to represent workflows that include a sequence of steps with actions, conditions, and loops (van Welie, van der Veer and Eliëns, 1998). Task modelling notations follow a hierarchical structure because this provides the ability to represent abstractions with refinement as tasks and subtasks, which is the objective of these notations. Hence, task modelling notations mostly use graphical representations to visualise a hierarchy of tasks and relationships in a way that is easier to interpret by software practitioners.

3.3.1.2 Operators and task types

Task modelling notations support multiple modelling operators and task types, which are useful for representing software system tasks and their relationships. Although these operators and task types are not specifically related to resource-driven adaptation, they still form the starting point for any task model and are therefore briefly presented in this subsection.

Task modelling notations support multiple operators including choice, concurrency, interruption, iteration, optionality, order independence, and sequence. The choice operator designates the possibility of choosing between multiple tasks so that when one task starts the others are disabled. Concurrency indicates that tasks can be performed simultaneously (i.e., one task can start before the other one finishes). The interruption operator indicates that a task is suspended until another task finishes its work, or a task is completely disabled by another one. Iteration is used to represent repetitive tasks once even though they may occur more than once. The optionality operator specifies whether a task is optional. Order independence and sequence specify that tasks can execute in any order and sequential order respectively. Some notations offer more operators than others. For example, AMBOSS (Giese *et al.*, 2008), HTA (Annett, 2003), GTA (Van Der Veer, Lenting and Bergevoet, 1996), and Diane+

(Tarby and Barthet, 1996) have more operators than TKS (Johnson and Hyde, 2003), GOMS, UAN, ANSI/CEA (2009), and MAD (Scapin and Pierret-Golbreich, 1989). Moreover, CTT and UsiXML (Limbourg *et al.*, 2004) offer the most types of operators.

Existing task modelling notations can represent tasks including abstract, interaction, application, and user. An abstract task represents an action that is divided into sub-tasks. An interaction task involves user interaction with the software system. An application task is done entirely by the software system whereas a user task is done entirely by the user. Some notations like CTT and HAMSTERS support these task types explicitly whereas other notations like Amboss and Diane+ support them implicitly (i.e., tasks that are not labelled as abstract, interaction, application, or user but the meaning can be inferred by a human reader from what the task represents). TOOD (2001) has a “type” property for tasks but does not explicitly specify the values for this property. Moreover, task types in HAMSTERS can be extended with new ones. Explicit support for task types is more useful because software systems would be able to automatically determine the type of a task and make decisions accordingly. For example, in resource-driven adaptation, the choice of the type of adaptation could depend on the task type.

3.3.1.3 Task modelling for resource-driven adaptation

Most task modelling notations have operators and task types for representing the tasks of several types of software systems (e.g., desktop, mobile, and web). Few task modelling notations target particular areas of application. For example, Amboss targets safety-critical systems, and HTA targets some types of industrial systems (e.g., chemical and petroleum refining). However, existing task modelling notations do not target software systems that support resource-driven adaptation and are therefore missing characteristics that would be useful in representing the tasks of these systems as explained below.

Existing task modelling notations do not support the association of resource types and priorities with tasks, which are important to identify potentially adaptable tasks due to resource variability. Although TOOD relates a task to a resource, this is not sufficient because additional information is required for performing resource-driven adaptation. This information includes the type of the resource, whether the task is allowed to use alternative resources, and the quantity of the resource that the task requires. Furthermore, the existing task modelling notations do not support task

variants that differ according to priorities, resource consumption, user roles, and parameter values. Such variants are useful for performing adaptation by executing similar tasks that consume fewer resources. Furthermore, it is important to support task variants because each one could have a different priority than its counterparts. For example, a task variant that consumes few resources and is initiated by users with privileged roles (e.g., manager) could be considered more important than a variant that consumes more resources and is initiated by users with less privileged roles.

Another issue is concerned with the lack of stereotypes (tags) indicating which adaptation types apply to a task. For example, if a task strictly requires a specific type of resource, the system cannot adapt by performing resource substitution and should consider another type of adaptation (e.g., delay the task until the resource becomes available). On the other hand, if the end-users expect an immediate result from a task to perform additional interaction with the system then the task cannot be delayed. In such a case, the execution of a similar task (variant) could be possible.

When adaptation is performed, it is important to give feedback to end-users about the rationale for the adaptation to keep them informed about why the software system made a particular decision. It is also useful to get feedback from the end-users about whether the system should improve its adaptation decisions. However, existing task modelling notations do not have properties for specifying whether and how a software system should present and receive adaptation-related feedback to and from end users.

Considering the abovementioned limitations, it would be useful to have a task modelling notation for representing the tasks of software systems that support resource-driven adaptation. It is not necessary to create this notation from scratch because existing notations offer a good starting point (refer to Section 3.3.1.2 on the supported operators and task types). Hence, the desired characteristics (e.g., resource types and task variants) could be added to one of the existing notations.

3.3.2 Feature modelling notations

A feature model is a hierarchical organisation representing the constraints for valid configurations in a software product line (Hallsteinsen *et al.*, 2008). Feature models are used by Software Product Lines (SPLs) to produce a collection of similar software systems by creating variations from a set of features. Like existing task modelling notations, feature modelling notations do not support resource-driven adaptation.

More specifically, feature modelling notations do not support the representation of resource types; priorities that differ according to parameter values, user roles, time of day, and resource intensiveness; information that affects which adaptation types are applicable; and configuration information regarding whether and how feedback is elicited from the end-users and presented to them.

Feature modelling notations have been extended with additional attributes (e.g., cost of using a feature) by existing work (Benavides, Trinidad and Ruiz-Cortés, 2005). However, resource-driven adaptation could be better supported by extending a task modelling notation because task models represent concrete tasks rather than high-level features, and these tasks could use different types of resources that affect if, when, and how a task is adapted. For example, consider an ERP system that has an “invoice” feature in its feature model. This feature corresponds, on a task model, to tasks such as “search for a product”, “display product description”, “display product image”, and “export invoice to image”. In this example, the software system could cancel the low-priority task “display product image” but not the high-priority task “display product description”. Furthermore, the temporal operators that are supported by task models are useful for anticipating which task will be executed next to inform adaptation decisions. For example, Rigole et al. (2007) used task models to perform gradual component deployment based on the tasks initiated by end-users to avoid the needless consumption of computing resources and latencies on lightweight mobile devices.

One interesting thing about feature models, in comparison to task models, is that they represent variation, which is required for software product lines. As explained in Section 3.3.1.3, the modelling of variation is useful for distinguishing the modes of executing a task when performing resource-driven adaptation. However, variants in feature models are not meant for this purpose but are used for product derivation.

Feature models and task models could be complementary whereby a feature maps to many tasks as explained by existing work on automated product derivation (Pleuss, Botterweck and Dhungana, 2010). Nonetheless, based on what I previously explained, task models are more appropriate for resource-driven adaptation and are therefore used in this thesis.

Table 3.1 provides a summary of the resource-driven adaptation approaches, which were discussed in Section 3.2. This table shows, for each approach, the supported resource types and their corresponding resource groups (refer to Section 2.2). The most common resource groups are static and reusable with resource types like RAM and CPU. Hence, existing adaptation approaches support a limited number of resource types. Furthermore, these approaches do not support the creation of new resource types at runtime. Hence, it is not possible to extend the resource types that are already supported. This limits the applicability of these techniques because software systems could require different types of resources. For example, consider that an ERP system got upgraded to a new version that uses additional resource types. It is more convenient if these resource types are defined through data entry at runtime rather than modifying the source code of the adaptation approach’s implementation.

Table 3.1 – Summary of existing resource-driven adaptation approaches

	Resource Group					Trigger Time			
Existing Work	Static	Dynamic	Reusable	Depletable	Resource Type	Reactive	Proactive	Adaptation Type	Software System Type
Klein <i>et al.</i> (2014)	●	○	●	○	CPU	●	○	activate/deactivate optional contents	Cloud-based
Propose the brownout paradigm to cope with hardware failures and varying user workload levels without over-provisioning resources									
Dürango <i>et al.</i> (2014)	●	○	●	○	CPU	●	○	activate/deactivate optional contents	Cloud-based
Apply brownout on multiple replicas of an application using load-balancing algorithms									
Tomás <i>et al.</i> (2014)	●	○	●	○	CPU	●	○	activate/deactivate optional contents	Cloud-based
Work on the issue of applications that are accepted to run on a cloud server, which does not have sufficient resources									
Maggio <i>et al.</i> (2014)	●	○	●	○	CPU	●	○	activate/deactivate optional components	Cloud-based
Compare multiple control strategies with a trade-off between response time and user experience									
Nikolov <i>et al.</i> (2014)	●	○	●	○	Network	●	○	activate/deactivate optional components	Cloud-based
Manage resource reservations based on resource demands and predefined service level agreements									
Moreno <i>et al.</i> (2015)	●	○	●	○	CPU	○	●	activate/deactivate optional contents	Cloud-based
Propose a proactive approach to address adaptation latency (i.e., the lag between performing an adaptation and its effect)									
Pandey <i>et al.</i> (2016)	●	○	●	○	CPU	●	○	activate/deactivate optional contents	Cloud-based
Combine two planning approaches to handle the trade-off between timeliness and optimality of the adaptation plan									
Xu <i>et al.</i> (2016)	●	○	●	○	CPU	●	○	activate/deactivate optional components	Cloud-based
Prioritise components based on usage frequency and perform a trade-off between energy consumption and revenue									
Hasan <i>et al.</i> (2016)	●	○	●	○	CPU	●	○	activate/deactivate optional contents	Cloud-based
Perform adaptation based on a trade-off between energy consumption and revenue									
Zhao <i>et al.</i> (2017)	●	○	●	○	CPU	●	○	activate/deactivate optional components	Cloud-based
Propose a framework to generate adaptation rules for user goals									
Sun <i>et al.</i> (2019)	●	○	●	○	CPU	●	○	activate/deactivate optional components, and substitute mandatory components	Cloud-based
Suggest substituting components with alternatives that are less resource-intensive during situations of resource variability									

Table 3.2 – Summary of existing task modelling notations

	Representation	Operators							Task Types				Useful Characteristics for Resource-driven Adaptation					Software System Type
		Choice	Concurrency	Interruption	Iteration	Optionality	Order Independence	Sequence	Abstract	Interaction	Application	User	Resource Types	Priorities	Task Execution Type	Task Variants	Feedback Properties	
AMBOSS	Graphical	●	●	○	○	○	●	●	○	○	○	○	○	○	○	○	○	Safety-critical
ANSI/CEA	Textual	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	Electronics devices
CTT	Graphical	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	○	Desktop, mobile, and web
Diane+	Graphical	●	●	○	●	●	●	●	○	○	○	○	○	○	○	○	○	Desktop
GOMS	Textual	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	Desktop
GTA	Graphical	●	●	○	○	○	●	●	○	○	○	○	○	○	○	○	○	Desktop
HAMSTERS	Graphical	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	○	Desktop, mobile, and web
HTA	Graphical	●	○	●	○	○	–	●	○	○	○	○	○	○	○	○	○	Steel production, chemical and petroleum refining, and power generation
MAD	Graphical	●	●	●	○	●	○	●	○	○	○	○	○	○	○	○	○	Desktop
TKS	Graphical	●	○	○	○	○	–	●	○	○	○	○	○	○	○	○	○	Desktop
TOOD	Graphical	●	●	●	○	○	–	●	○	○	○	○	○	○	○	○	○	Desktop
UAN	Textual	●	●	●	●	○	●	●	○	○	○	○	○	○	○	○	○	Desktop
UsiXML	Graphical	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	○	Desktop, mobile, and web
Legend: ● Explicitly Supports ○ Implicitly Supports ○ Does not support – Not specified																		

Table 3.1 also shows the trigger time (reactive and proactive) and the adaptation types that are supported by each approach. The most common adaptation trigger is reactive, and the most common type of adaptation involves the activation and deactivation of optional contents or components. Moreover, as shown in Table 3.1, cloud-based systems and mobile systems are the most common examples of software systems that are targeted by existing resource-driven adaptation approaches. As discussed in Section 3.2, a system becomes more versatile in coping with resource variability if multiple types of adaptation were supported. Additionally, a resource-driven adaptation approach becomes more useful if it supports multiple types of software systems.

Table 3.2 provides a summary of existing task modelling notations that were discussed in Section 3.3.1. This table summarises the operators and task types that task modelling notations support. It is important to explore these operators and task types to see which task modelling notation(s) offers a good starting point for adding characteristics that are useful for resource-driven adaptation but are missing from existing notations as shown in Table 3.2 and discussed in Section 3.3.1.

3.5 Filling the gaps

As explained in Section 1.3, to answer the research questions, this thesis presents a framework and a task modelling notation for supporting resource-driven adaptation in software systems. By supporting resource-driven adaptation, software systems would be able to address the problem of resource variability, which prevents them from executing critical tasks that require scarce resources. The abovementioned contributions aim to fill the gaps that were discussed in Sections 3.2 to 3.4 and are summarised as follows:

1. Consider tasks in the resource-driven adaptation process to provide granularity in the adaptation decision-making based on task differences that include priority, resource consumption, parameters of the task, the role of the user who is initiating the task, when the task is initiated, and the applicability of a type of adaptation to the task
2. Support multiple types of resources that belong to the previously defined four resource groups: reusable, depletable, static, and dynamic, to make the resource-driven adaptation approach more comprehensive
3. Support multiple types of adaptation to provide versatility in addressing resource variability, whereby if one type of adaptation is not applicable to a task the software system would be able to use other types of adaptation

The work presented in this thesis considers tasks to offer more granularity for resource-driven adaptation decision-making in comparison to other approaches such as the ones that work with components or make decisions that apply to the entire software system (e.g., refer to Sections 3.2.1 and 3.2.5). Tasks are represented in task models where they are broken down into subtasks and prioritised to support decision-making on whether they should be adapted in situations of resource variability. By working with tasks and keeping task models available at runtime, it is possible to

support multiple types of software systems and to make dynamic adaptation decisions, unlike the approaches discussed in Section 3.2.4. Furthermore, unlike some of the approaches discussed in Sections 3.2.2 and 3.2.5 that only consider single-user scenarios, this thesis considers scenarios involving multiple users who initiate tasks that compete for shared resources.

Tasks are prioritised based on multiple criteria including the timeframe (from and to time of day) of the task's execution, the role of the user who is attempting to execute the task, the forecasted number of task executions to be done by users, and the parameter values that distinguish task variants (refer to Figure 3.2 for an example). These prioritisation criteria include the perspectives of system administrators who provide task priorities based on their domain knowledge and end-users who indirectly indicate task priorities through actual use. In comparison, for example, Sun et al. (2019) rely on usage frequency only and Gajos et al. (2001) use an arbitrary scheme for indicating a task's importance as discussed in Sections 3.2.1 and 3.2.2 respectively. A heuristic is proposed in this thesis as a solution for performing task prioritisation without affecting a software system's performance when considering environments where changes occur at runtime due to resource variability (refer to the discussion in Section 3.2.3).

Task priorities are used to decide which tasks execute by gaining access to the resources that they need, and which ones get adapted. To provide more versatility than the existing resource-driven adaptation approaches, multiple adaptation types are supported, including the execution of a similar task that requires fewer resources, the substitution of resources with alternative ones, the execution of tasks in a different order, or even the cancellation of low-priority tasks. As shown in Table 3.1 and explained in Section 3.2, existing resource-driven adaptation approaches have less diversity in their supported adaptation types.

Moreover, the work presented in this thesis supports multiple types of resources belonging to the resource groups defined in Section 2.2, namely reusable, depletable, static, and dynamic, and the creation of new types of resources at runtime. As shown in Table 3.1 and explained in Section 3.2, existing resource-driven adaptation approaches support a limited number of resource types, which narrows the applicability of these approaches to specific cases because the way of performing the adaptation differs from one type of resource to another.

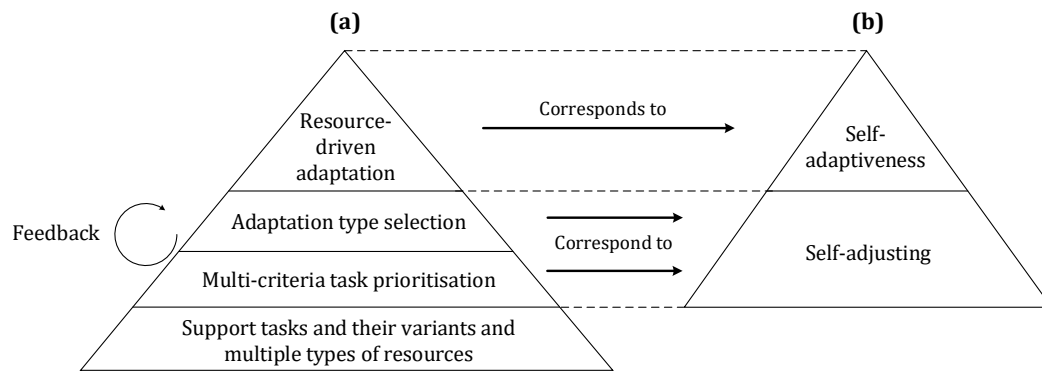


Figure 3.3 – Characteristics of proposed work (a) and their self-* properties (b)

Furthermore, a new task modelling notation is presented and used in this thesis to represent task models that serve as input for resource-driven adaptation. The new notation is based on CTT and fills the gaps that are presented in Table 3.2 and discussed in Section 3.3.1. In comparison, existing task-based approaches discussed in Section 3.2.2 do not specify an explicit way of modelling tasks. Only Rigole et al. (2007) use task models but, as previously mentioned, their work is concerned with loading software components just-in-time rather than addressing resource variability.

In summary, the characteristics of the work that this thesis presents to address resource variability are represented by the hierarchy shown in Figure 3.3a. These characteristics correspond to the self-* properties shown in Figure 3.3b (Salehie and Tahvildari, 2009). Task prioritisation, adaptation type selection, and feedback loops correspond to the property “self-adjusting”. Additionally, resource-driven adaptation, which is used to address resource variability, corresponds to the property “self-adaptiveness”.

4

Overview of the Work

This chapter presents an overview of the work that this thesis presents to support resource-driven adaptation based on MAPE-K. It presents the involved stakeholders, components, and data. This chapter also illustrates how to integrate resource-driven adaptation capabilities into software systems. The concepts that this chapter presents serve as a basis for the resource-driven adaptation framework presented in a subsequent chapter.

4.1 Introduction

Figure 4.1 presents an overview of the work that this thesis proposes for supporting resource-driven adaptation to address the problem of resource variability. As this figure shows, three types of stakeholders are involved in the adaptation process, namely software practitioners, system administrators, and end-users. *Software practitioners* prepare task models that include setup data provided by *system administrators* based on their domain knowledge. *End-users* initiate tasks from the software system and provide their feedback on the adaptations.

This thesis proposes components that work proactively and reactively to support resource-driven adaptation in software systems. These systems can in principle be any type of resource-driven software system. This thesis considers enterprise systems as an example of resource-driven software systems. The data used in the adaptation process includes the task models and setup data in addition to the adaptation plans that are prepared by the corresponding adaptation components. A knowledge base stores this data to make it available for the adaptation decision-making process during resource variability.

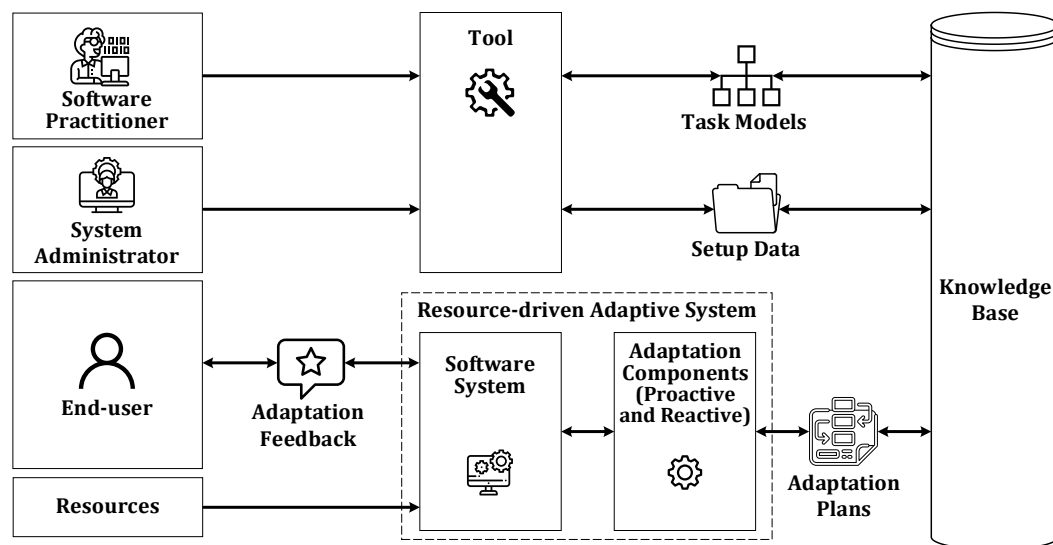


Figure 4.1 – An overview of the proposed work for supporting resource-driven adaptation in software systems

The following sections provide further details on the elements that are represented in Figure 4.1 including stakeholders, adaptation components, and task models, and how the proposed work follows the MAPE-K control loop (Kephart and Chess, 2003). Furthermore, the following sections present a way of integrating the proposed resource-driven adaptation components into software systems. This involves filtering task requests to decide whether to perform an adaptation before executing a task and returning its results to the end user.

4.2 Stakeholders

Figure 4.1 shows three types of stakeholders, namely *software practitioners*, *system administrators*, and *end-users*. Software practitioners work in the software industry in professions such as software design, programming, and deployment. System administrators are information technology (IT) professionals who handle the configuration and upkeep of software systems within an enterprise. End-users use the software system by initiating tasks from it.

Software practitioners are responsible for creating task models because they have the knowledge related to the software system (e.g., tasks and relationships). Software practitioners use a tool to create task models that comprise tasks, which represent the activities that the software system can perform. They could require the domain

expertise of system administrators to specify some information on the task models (e.g., priorities that represent a managerial perspective of the importance of tasks within an enterprise). Software practitioners create the task models via the tool and store them in a knowledge base to be used by the software system when it performs resource-driven adaptation.

System administrators use a tool to specify setup data such as resource types and user roles. The tool stores this setup data in a knowledge base. Software practitioners use this data to assign values to the properties of tasks in a task model (e.g., the type of resource that a task uses and the roles of the users who can initiate a task). System administrators play an important role in many types of software systems. This is especially true for enterprise systems where system administrators assist software practitioners in configuring the software system during the deployment phase. Such configurations could take up to months for large-scale multimillion-dollar enterprise systems (Garg and Venkitakrishnan, 2003). Therefore, it is reasonable to assume that system administrators can play a role in providing setup data for supporting resource-driven adaptation, especially since this data represents part of the domain knowledge that they possess.

End-users interact with the software system by initiating tasks from it. End-users can provide feedback to rate the impact of an adaptation on a task's available functionality and the quality of its outcome. This feedback is used during the preparation of an adaptation plan for choosing suitable types of adaptation. For example, assume that, for the sake of limiting CPU consumption, an adaptation lowered the resolution of images. If the end-users provided a low rating for the resolution quality, then, in the future, the system could choose another type of adaptation (e.g., delay the task's execution instead of lowering the resolution). Furthermore, the opinions of end-users about task priorities are indirectly elicited by monitoring task usage. Hence, the tasks that end-users initiate more frequently would have a higher priority from an end-user perspective. This complements the managerial perspective of priorities provided by the system administrator.

I considered the abovementioned three types of stakeholders, namely software practitioners, system administrators, and end-users, to represent the people who are involved in the adaptation process because their input is sufficient to support resource-driven adaptation in software systems. The software practitioners create the task models, the system administrators provide the domain knowledge, and the end-

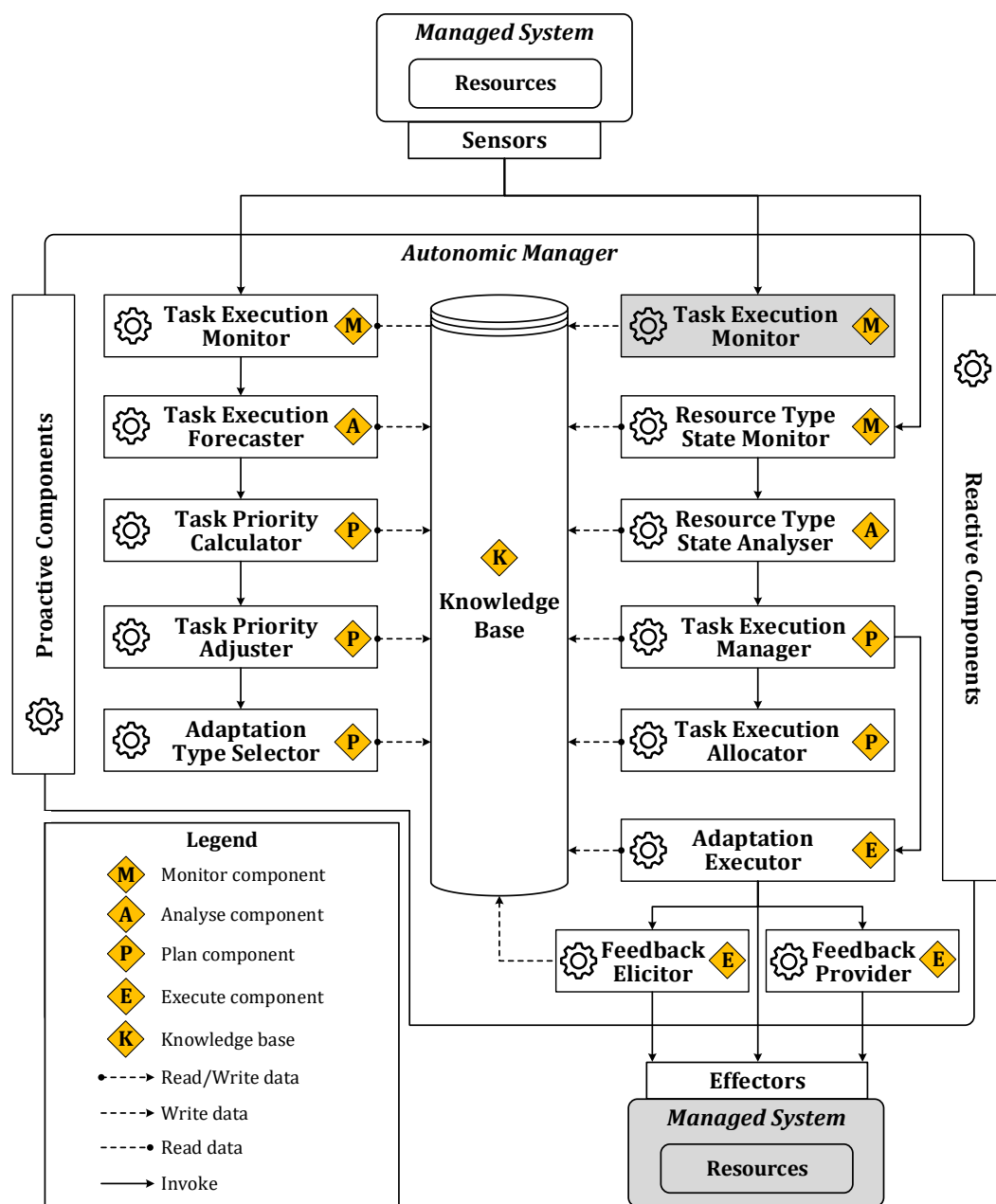


Figure 4.2 – The proposed adaptation components based on MAPE-K (the grey parts are repeated in the figure to avoid overlapping lines)

users provide feedback on the adaptations. People who play these roles are always present. Software practitioners must be present to create software systems for end-users to use.

Moreover, consider enterprise systems as an example, system administrators are present to configure and maintain the software systems of an enterprise. Although people who play other roles, e.g., the managers of an enterprise, also possess domain

knowledge, system administrators can convey the knowledge to software practitioners because they work closely with them during the deployment of the software system.

4.3 Adaptation components

The proposed adaptation components follow the MAPE-K control loop because it serves as a reference for adapting software systems via components that handle actions from the collection of data to the execution of the adaptation (refer to Section 2.4.1 for more information on MAPE-K). Hence, the proposed adaptation components include monitors (M), analysers (A), planners (P), executors (E), and a knowledge (K) base as shown in Figure 4.2. This thesis presents a framework that realises these components, which complement each other to address resource variability through resource-driven adaptation. The relationships between these components involve invoking one another and passing the required data. These relationships are denoted in Figure 4.2 by the arrows that connect the components. As indicated by the legend of Figure 4.2 the arrows that have a solid line represent an invocation between the managed system and the autonomic manager and among the components of the autonomic manager. Moreover, the arrows that have a dashed line represent reading and writing data. These arrows represent data storage and retrieval operations between the adaptation components and the knowledge base.

The adaptation components support proactive and reactive adaptation processes. These adaptation processes rely on the notion of task prioritisation where tasks that use the same types of resources are executed in order of their priorities. The proactive process uses historical data and configurations to calculate the priorities of tasks and prepare an adaptation plan. The reactive process identifies the resource types that are facing variability and decides whether to perform the corresponding type of adaptation.

As shown in Figure 4.1, the adaptation process executes with the support of ten components that work either proactively or reactively. One of the components, namely the *task execution monitor*, works both proactively and reactively. This component is responsible for monitoring the execution of tasks included in the task models and the use of their respective resources reactively. The *task execution monitor* uses a sensor to collect reactively the number of times tasks are initiated and stores it as historical data in the knowledge base. Then, the *task execution monitor* passes this historical data to the *task execution forecaster* so it can use it during the proactive adaptation process.

Sensors act as a bridge between the managed system and the autonomic manager. One sensor enables the *task execution monitor* to reactively identify when tasks are initiated in the managed system. Another sensor enables the *task execution monitor* to proactively decide if it is time to read the historical data from the knowledge base and invoke the *task execution forecaster*. The following subsections explain the remaining adaptation components.

4.3.1 Proactive

The proactive adaptation process includes five components, namely *task execution monitor* (explained above), *task execution forecaster*, *task priority calculator*, *task priority adjuster*, and *adaptation type selector*. The *task execution forecaster* forecasts the number of executions of tasks based on their historical usage. The forecasted task execution is needed because it is part of the data used for calculating the task priorities. The *task priority calculator* and *task priority adjuster* calculate the priorities of tasks and adjust these priorities respectively when tasks with the same priority value require the same type of resources. This way each task is assigned a unique priority value. This helps in differentiating tasks by their importance and deciding whether to execute a task or perform adaptation. Finally, the *adaptation type selector* specifies the adaptation types that will be applied to the tasks. The supported adaptation types include (i) execution of task variants that require fewer resources; (ii) substitution of resource types with alternative ones; (iii) execution of tasks in a different order based on their priorities; and (iv) cancellation of tasks when no other task variant or resource can be used. The use of multiple adaptation types (i)-(iii) provides more versatility in the possible ways of performing adaptation to execute a task that requires resources, which are facing variability. Otherwise, the execution of the task would be cancelled (iv).

4.3.2 Reactive

The reactive adaptation process includes eight components, namely *task execution monitor* (explained above), *resource type state monitor*, *resource type state analyser*, *task execution manager*, *task execution allocator*, *adaptation executor*, *feedback elicitor*, and *feedback provider*. The *resource type state monitor* observes the use of resources by tasks and stores the monitored data in the knowledge base. To obtain this data, the *resource type state monitor* uses a sensor that links it to the managed system and

enables it to observe the resources that are being used by the tasks. The *resource type state analyser* analyses the data that is obtained by the *resource type state monitor* to check whether a resource type is facing variability. The *task execution manager* decides whether to execute or adapt the tasks, which use resources that are facing variability. The *task execution allocator* allocates the number of possible executions for each task based on the forecasted number of task executions, the task priorities, and the available quantities of resources. This helps in keeping resources available for the most important tasks. The *adaptation executor* performs adaptations to address situations of resource variability. The *feedback elicitor* and *feedback provider* gather and provide adaptation-related feedback from and to end-users respectively. The feedback that is elicited from the end-users helps in improving future adaptation-related decisions. Moreover, feedback provided to the end-users helps them to understand the reason behind performing the adaptation.

The *adaptation executor* uses an effector that enables it to make changes to the managed system that are necessary to perform adaptations. Additionally, an effector prompts end-users on the managed system to provide feedback about the adaptations that the *adaptation executor* performed and returns this data to the *feedback elicitor*. Furthermore, the *feedback provider* uses an effector to relay its feedback to end-users as messages on the managed system.

4.3.3 The benefit of combining proactive and reactive adaptation

The proactive adaptation uses forecasting based on historical data to predict future events (Hyndman and Athanasopoulos, 2018). In the adaptation components that this thesis proposes, the proactive adaptation process uses historical data such as task-usage history to compute task priorities and user feedback to select adaptation types. This historical data is not used in reactive adaptation and leveraging it is a benefit of preparing an adaptation plan proactively. Additionally, preparing an adaptation plan reactively over very short periods (e.g., seconds) could be costly and cause interruptions. That is why Krupitzer *et al.* (2018) consider proactive adaptation to be preferable from a user point of view. However, Krupitzer *et al.* also note that the precision of a proactively prepared adaptation plan relies on forecast accuracy. Hence, even if multiple forecasting techniques are used or a suitable technique is selected based on historical data, the results could still have some inaccuracy (Bauer, 2019; Bauer *et al.*, 2020). Therefore, the proactively prepared plan could serve as a starting

point that is complemented by a reactive decision on whether there is a need for adaptation.

4.4 Task models

As shown in Figure 4.1 and explained in Section 4.2, software practitioners use a tool to create task models and store them in a knowledge base. Software practitioners represent the task models using a task modelling notation that supports resource-driven adaptation. This task modelling notation and its supporting tool are part of the contribution of this thesis. The knowledge base stores the task models following the notation's meta-model so the software system can interpret them accordingly.

The task models created by software practitioners serve as input for the preparation of adaptation plans and for making adaptation-related decisions during resource variability. For example, these models comprise knowledge that informs the system about which types of adaptation apply to a task and how to execute a task differently to reduce the strain on resources that are facing variability.

4.5 Integrating resource-driven adaptation in a software system

Figure 4.3 illustrates how this thesis proposes to integrate resource-driven adaptation into software systems. This figure shows the steps from when a user initiates a task request to when they receive a response and provide feedback on the task's outcome passing through the steps that involve adaptation-related decisions. Figure 4.3 indicates the chronological order of these steps by the labels containing numbers 1 to 11.

As shown in Figure 4.3, software systems are considered to have a client-side part and a server-side part. This is common in service-oriented applications such as enterprise systems, which end users access from multiple geographical locations within and outside the enterprise. The client-side part could be a desktop, mobile, or web application that comprises the presentation layer of the software system and runs on the end user's device. On the other hand, the server-side part comprises the other layers of the application including business logic, data access, and web services, and runs on the server.

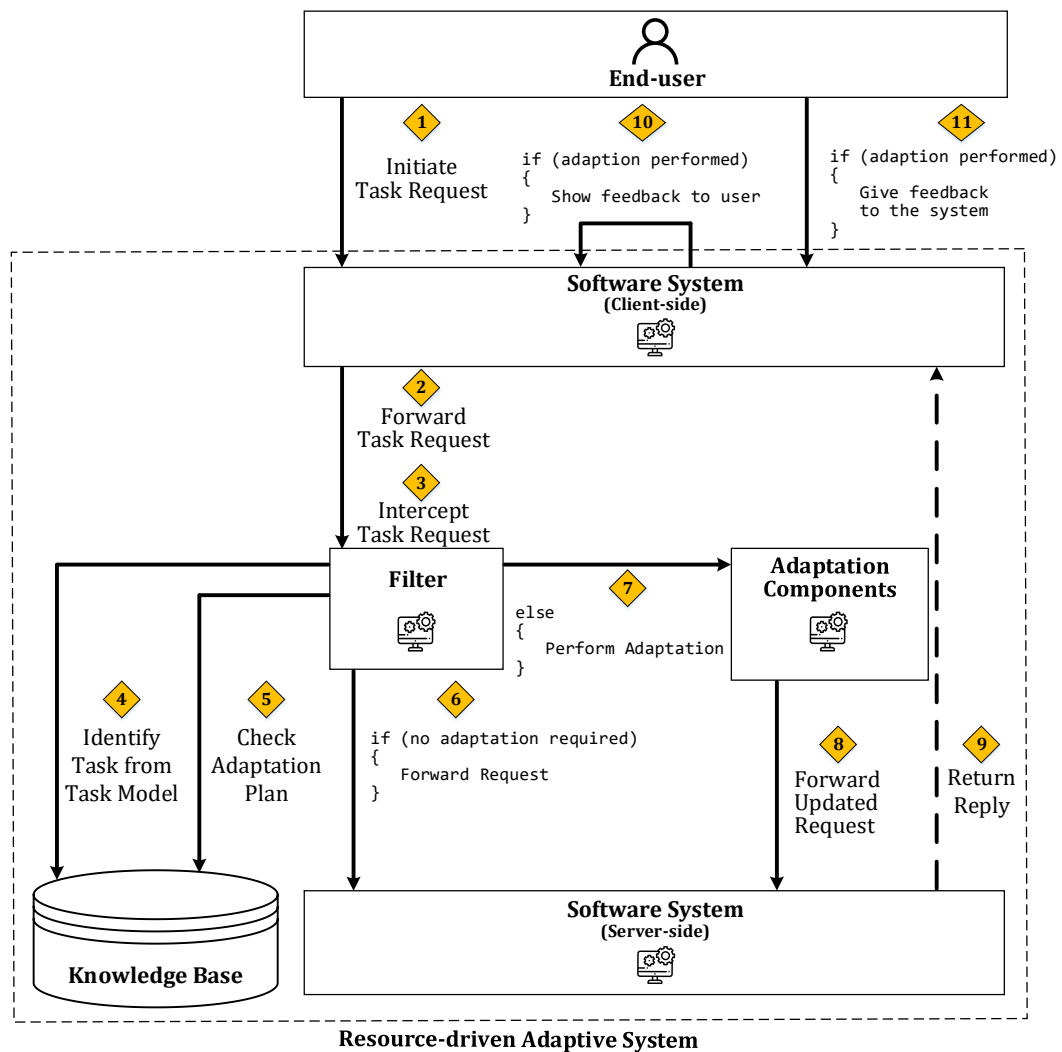


Figure 4.3 – Integrating resource-driven adaptation in a software system

As shown in Figure 4.3, end-users initiate task requests on the client-side part of the software system, which forwards these requests to the server. Filters intercept task requests before reaching the server-side part of the software system. In the work presented in this thesis, the filters are implemented using .NET actions filters, which contain logic that executes before the software system's code executes (Larkin *et al.*, 2021). A filter decides whether to execute adaptation-related logic before forwarding the task request to the software system. This is done after identifying the requested task from the task model that is stored in the knowledge base and checking the adaptation plan to see if adaptation is required due to resource variability.

The adaptation components perform adaptation, when needed, using one of the supported adaptation types (refer to Section 4.3.1). Then, these components forward

an updated request to the server-side part of the software system. For example, if a similar task is to be executed instead of the one that was initiated, then the request would be forwarded with an updated task identifier number. Alternatively, if the adaptation involved resource substitution, then the request would be forwarded with the same task identifier number but a different identifier number for the type of resource. Moreover, if the adaptation delayed the task's execution, then the task request would be added to a queue to be forwarded later. Finally, if the adaptation cancels the task altogether, then the task request would not be forwarded. Regardless of whether adaptation is performed and which type of adaptation is used, a response is returned to the client-side part of the software system. The end-user sees the result of the task, if it executes, and receives and gives adaptation-related feedback if the system performed an adaptation (refer to Section 4.3.2).

4.6 Chapter summary

This chapter presented an overview of the work that this thesis proposes for supporting resource-driven adaptation in software systems. This overview includes the involved stakeholders, namely software practitioners, system administrators, and end-users. This chapter explained the roles of these stakeholders in supporting resource-driven adaptation. In summary, software practitioners create task models, system administrators provide setup data, and end-users offer feedback on the adaptation performed by the system.

This chapter also presented the adaptation components that this thesis proposes in the context of the MAPE-K control loop. It explained these components and their input that is represented as task models. Moreover, this chapter discussed the process of integrating resource-driven adaptation with software systems. The integration uses filters that intercept task requests to decide whether to perform adaptation during resource variability.

This chapter provides an overview of the proposed solution for addressing resource variability through adaptation. Chapter 5 will present the proposed task modelling notation for supporting resource-driven adaptation (SERIES) and give a detailed explanation of the task models. Furthermore, Chapter 6 will present the proposed resource-driven adaptation framework (SPARK) and give a detailed explanation of the adaptation components.

5

SERIES: A Task Modelling Notation for Resource-Driven Adaptation

This chapter presents SERIES, which is a task modelling notation for supporting resource-driven adaptation. First, this chapter explains the meta-model that comprises the constructs of SERIES. Then, it presents an example of a SERIES task model from an automated warehouse system. This example demonstrates the constructs of SERIES and their graphical representation. Afterwards, this chapter presents the supporting tool of SERIES and explains its features.

5.1 Introduction

SERIES is a task modelling notation that supports resource-driven adaptation when a software system's resources are facing variability. Task models that are represented using SERIES contain information that is used by the resource-driven adaptation components introduced in Section 4.3. Moreover, SERIES is based on CTT (Paterno, Mancini and Meniconi, 1997), which is a notation for representing task models hierarchically using a graphical syntax. The graphical representation of SERIES is also inspired by UML class diagrams (Fowler, 2003, pp. 35–52). For example, UML represents a class as a multi-part box with a set of attributes and operations. Similarly, SERIES represents a task as a multi-part box with properties.

As explained in Section 3.3.1, existing task modelling notations have useful features such as tasks and relationships, but they also lack some characteristics that are important for supporting resource-driven adaptation in software systems. Several task modelling notations like CTT, HAMSTERS (Martinie, Palanque and Winckler, 2011),

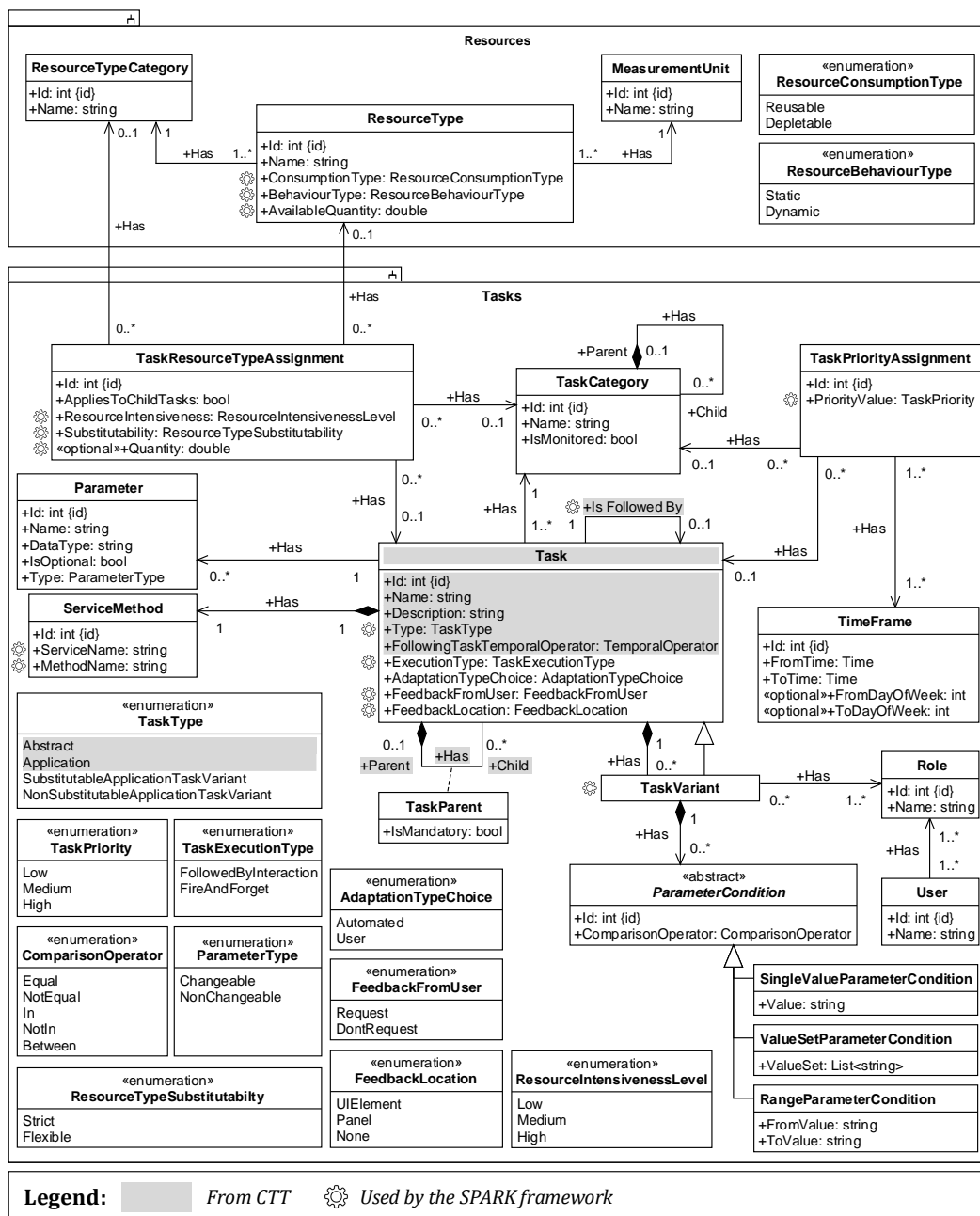


Figure 5.1 – Meta-model of SERIES represented as a class diagram - elements highlighted in grey are from the CTT notation; elements with a gear are used in the SPARK framework)

and UsiXML (Limbourg *et al.*, 2004) could be extended to support resource-driven adaptation.

However, SERIES is based on CTT given its wide use in academia, government, and industry. The tool of CTT has been downloaded over 26,000 times and has over 10,000 registered users (Vigo, Santoro and Paternò, 2017). Additionally, CTT supports useful characteristics like application tasks that SERIES extends with task variants.

5.2 Meta-model of SERIES

Figure 5.1 shows the meta-model of SERIES. This meta-model includes constructs that support resource-driven adaptation in software systems. The part of the meta-model highlighted in grey is incorporated from CTT while the rest is added by SERIES. Moreover, the parts with a gear icon next to them in the meta-model are used in the adaptation components of the SPARK framework, which is explained in more detail in Chapter 6. The following subsections explain the meta-model and use bold and italic to emphasise the names of the constructs.

5.2.1 Constructs that SERIES incorporates from CTT

SERIES incorporates tasks and relationships with temporal operators from CTT. The meta-model represents these constructs by the **Task** class and its self-association “*is followed by*” and property *FollowingTaskTemporalOperator*. Tasks are connected using relationships that are annotated with temporal operators, which express how the tasks relate to each other.

SERIES also incorporates from CTT two task types, namely *abstract task* and *application task*, which are represented by the enumeration *TaskType*. An **abstract task** involves complex actions and is broken down into a sequence of child (sub) tasks, which are represented in the class *Task* by the self-composition parent-child and the association class *TaskParent*. The property “*IsMandatory*” on this association class indicates whether a subtask is mandatory or optional. An **application task** is executed by the software system without user interaction. The child (sub) tasks of an *abstract task* are represented as *application tasks*.

SERIES extends CTT with additional constructs that correspond to the non-grey parts of the meta-model shown in Figure 5.1. The following subsections explain these constructs.

5.2.2 Abstract task

What follows are the characteristics of an abstract task that are extended by SERIES. These characteristics include a description, an execution type, feedback properties, resource types, and parameters.

5.2.2.1 Description

The *name* is a short text that indicates a task's purpose and is available in task modelling notations like CTT. In addition to the *name*, SERIES tasks have a **description**, which is a longer text that provides more explanation about the task. The *task name* is mandatory because it makes the purpose of the task model comprehensible. On the other hand, the *description* is optional and becomes more beneficial when a task model gets larger and more complex in terms of the number of tasks and what the tasks represent in the corresponding domain.

5.2.2.2 Execution type

The **execution type** specifies whether or not end-users require an immediate result from the task to perform additional interaction with the software system. The possible values for the *execution type* are represented by the *TaskExecutionType* enumeration and include “*followed-by-interaction*” and “*fire-and-forget*”.

If the end-users do not require an immediate result from a task to execute another task after it, then the task's *execution type* will be “*fire-and-forget*”, otherwise it will be specified as “*followed-by-interaction*”. A “*fire-and-forget*” task is processed by the software system in the background while the user initiates other tasks. Hence, a user can initiate multiple “*fire-and-forget*” tasks (e.g., as a batch) and check their result later. On the other hand, a user would initiate one task with an execution type “*followed-by-interaction*” and processes its result before deciding what to do next (e.g., stop initiating further tasks or initiate another task with data input that is based on the result of the previous task).

The execution type of tasks affects adaptation decisions during resource variability. In this regard, a task that is “*followed-by-interaction*” cannot be delayed because the user is expecting an immediate result from it. On the other hand, a “*fire-and-forget*” task can be delayed to be executed later when there is less strain on the resource types that it requires and are facing variability.

5.2.2.3 Feedback properties

The **adaptation type choice** indicates whether the type of adaptation is selected manually by the end-user or automatically by the software system. The possible values for *adaptation type choice* are represented by the enumeration *AdaptationTypeChoice*

and include “*automated*” and “*user*”. The “*automated*” selection of the type of adaptation relieves end-users from having to frequently make manual choices. Additionally, if multiple types of adaptation share the lowest cost and the *adaptation type choice* property was set to “*user*”, then the software system prompts end-users to select one of the least costly types of adaptation. Otherwise, the software system automatically selects one of the least costly types of adaptation.

The property **feedback-from-user** specifies whether the software system shall ask end-users to provide their feedback on the outcome of the task when adaptation is performed. This feedback enables the software system to improve its adaptation choices. The possible values are represented by the enumeration *FeedbackFromUser* and include “*request*” and “*don’t request*”. If *feedback-from-user* was set to “*request*”, then the software system would request feedback from end-users on how the adaptation affected their work and use this feedback to adjust the costs of applying multiple types of adaptation. In case the feedback from the user is not required, then the property *feedback-from-user* will be set to “*don’t request*”.

The **feedback location** specifies where the software system shall request and give feedback from and to end-users respectively. The enumeration *FeedbackLocation* represents the possible values for *feedback location*, which include “*UI element*”, “*panel*”, and “*none*”. The “*UI element*” is a suitable option when end-users need to provide immediate feedback as they work. Hence, the software system will give feedback on the part of the UI that corresponds to a task (e.g., as a popup window next to the button that the end-user presses to initiate the task). On the other hand, the “*panel*” option groups multiple feedback messages that end-users can check later. The “*panel*” is dedicated to feedback and is separate from the part of the UI that corresponds to the task. In case no feedback is required, i.e., the property *feedback-from-user* was set to “*don’t request*”, then the *feedback location* will be set to “*none*”.

5.2.2.4 Resource types and their assignment to tasks

Resource types represent entities that a task requires so it can execute. A resource type is represented by the class *ResourceType*, which has several properties. The property *consumption type* specifies whether the resource type is “*reusable*” or “*depletable*” as indicated by the enumeration *ResourceConsumptionType*. A “*reusable*” resource type is available to another task after the task that is using it is done, whereas a “*depletable*” resource type is used once. A resource type also has a *behaviour type* that

is either “static” or “dynamic” as indicated by the enumeration *ResourceBehaviourType*. A “static” resource type does not have a behaviour, whereas a “dynamic” resource type has a behaviour. Refer to Section 2.2 for more information on the types of consumption and behaviour of resource types. A resource type has an *available quantity* specified in terms of a measurement unit that is represented by the class *MeasurementUnit*. The *available quantity* indicates how many resources of a type are available on hand. This is useful for identifying whether resource types are facing shortages.

A **task resource type assignment** represents an association between a *task* and the *resource types* that it requires. The property *AppliesToChildTasks* is set to either “true” or “false” to indicate whether or not the *resource type* that is assigned to a task also applies to its subtasks. By setting this property to “true”, there would be no need to duplicate the effort and re-associate the same resource type with all the sub-tasks. Additionally, the *ResourceIntensiveness* property specifies whether the concerned task has a low, medium, or high consumption for the resource type that is assigned to it. This is useful to identify which tasks place more strain on which types of resources to make adaptation decisions accordingly.

The *task resource type assignment* also specifies the resource type’s substitutability for the task. The substitutability is either “strict” or “flexible” as indicated by the *ResourceTypeSubstitutability* enumeration. A “strict” resource type cannot be substituted with alternatives. Hence, during resource variability situations the software system should seek a type of adaptation that does not involve resource substitution. On the other hand, a “flexible” resource type is substitutable with alternative *resource types*.

5.2.2.5 Categories of resource types and tasks

Resource types are categorisable under **resource type categories**, which are instances of the *ResourceTypeCategory* class. This categorisation helps in speeding up the work when tasks have common *resource types* assigned to them. Hence, if a task requires all the *resource types* in a category, then it would be associated with the category rather than with each *resource type* individually.

Similar tasks are categorisable under **task categories**. Like *resource type categories*, task categories speed up the work by facilitating the association of *resource types* with *tasks*. Hence, if all the *tasks* in a *task category* use a *resource type* then the *task category*

is associated with this *resource type*. Additionally, if all the *tasks* in a *task category* use all the *resource types* in a *resource type category*, then the *task category* is associated with the *resource type category* rather than performing individual associations among the *tasks* and *resource types*.

5.2.2.6 Parameters

Parameters represent a task's expected input data. A parameter has a *name*, *data type*, and *parameter type*. The *data type* specifies what kind of data the parameter holds (e.g., boolean, decimal, and string). On the other hand, the *parameter type* specifies whether a parameter is "*changeable*" or "*non-changeable*" as indicated by the enumeration *ParameterType*. If the parameter type is set to "*changeable*", then the value of the parameter can be changed. Otherwise, the parameter type is set to "*non-changeable*" to indicate that the value of the parameter cannot be changed. The ability to change a parameter's value is important for performing adaptation to execute the task differently. However, in some cases parameter values should remain as they reached the software system (e.g., provided as input by the user or another system). Hence, there is a need for both parameter types "*changeable*" or "*non-changeable*".

5.2.3 Application task

An *application task* has the same characteristics as an *abstract task* (refer to Section 5.2.2) in addition to priorities and services.

5.2.3.1 Priorities

The **priority** that is assigned to a task on the task model represents the task's importance in a domain. Hence, a *priority* is either "*low*", "*medium*", or "*high*" as indicated by the *TaskPriority* enumeration. Priorities are useful for adaptation during resource variability to keep scarce resources available for the most important tasks. The priorities that are specified in the task model are based on domain knowledge.

Priorities could differ among timeframes, which represent intervals of time that are meaningful for a domain. For example, a task could have a "*low*" *priority* in the morning from 8:00 AM to 12:00 PM and a "*high*" *priority* in the afternoon from 12:01 PM to 5:00 PM. The classes *TimeFrame* and *TaskPriorityAssignment* represent timeframes and the assignment of priorities to tasks respectively.

Priority assignments can be applied to *task categories* as well. This helps in speeding up the work when tasks that belong to the same *task category* have the same *priority assignment*. Hence, when a priority is assigned to a category it automatically applies to all the tasks within it.

5.2.3.2 Service method

The tasks represented in a task model correspond to a software system. Hence, the **service method** represents the function in the software system's source code that is called when the task is executed. A *service method* is represented by the class *ServiceMethod*, which has two properties: *method name* and *service name*. A *method name* represents the name of the function that is called to execute a task, whereas a *service name* represents the name of the class where the function is implemented.

Moreover, when the software system receives a request to initiate a task it checks whether adaptation is needed before executing the task. To make adaptation decisions, the software system requires information from the task model (e.g., which type of adaptation applies to the initiated task). Hence, upon receiving a task initiation request the software system identifies the corresponding task from the task model by comparing the names of the class and function that are invoked from the source code to the *service methods* in the task model.

5.2.4 Application task variant

An *application task variant* is a special case of an application task and is needed to (1) avoid treating all executions of an application task in the same way when adapting and (2) identify how to execute an application task with fewer resources. An *application task variant* has the same characteristics as an *application task* (refer to Section 5.2.3), with the addition of parameter conditions, resource intensiveness, roles, and substitutability.

5.2.4.1 Parameter conditions

Parameter conditions specify the parameter values that distinguish *application task variants* from each other. A *parameter condition* is represented by the class *ParameterCondition* and its subclasses "*single value*", "*value set*", and "*range*". *Single value*, *value set*, and *range parameter conditions* compare the value of a parameter to a

single value (e.g., 10), set of values (e.g., 10, 15, and 20), and range of values (e.g., 10 to 20) respectively. Moreover, the comparisons performed in the parameter conditions use one of five comparison operators “*equal*”, “*not equal*”, “*in*”, “*not in*”, and “*between*”, as indicated by the enumeration *ComparisonOperator*. The operators “*equal*” and “*not equal*” are used for *single-value parameter conditions*. Additionally, the operators “*in*” and “*not in*” are used for *value-set parameter conditions*. Furthermore, the operator “*between*” is used for *range parameter conditions*.

5.2.4.2 Resource intensiveness

Resource intensiveness indicates the level of resource consumption of an *application task variant* for a *resource type* (i.e., the strain that an *application task variant* places on a *resource type*). The value of resource intensiveness is either low, medium, or high as indicated by the enumeration *ResourceIntensivenessLevel*. A high *resource intensiveness* means that more resources are required to execute a task, and vice-versa.

The *resource intensiveness* is represented in the class *TaskResourceTypeAssignment* since its value is specified per combination of task variant and resource type. *Resource intensiveness* helps in adaptation decision-making when a software system needs to execute a similar task variant that consumes fewer resources. This reduces the strain on *resource types* that are facing variability.

5.2.4.3 Roles

Although an *application task variant* is performed by the software system it could be initiated after a request from an end-user. **Roles** represent the groups of end-users who are eligible to initiate the *application task variant*. A typical example of a role in an enterprise is a job title like manager or clerk. However, roles are not necessarily related to job titles. For example, a role could be used to indicate the age groups of end-users. Roles and end-users are represented by the classes *Roles* and *Users* respectively. The association between these two classes denotes the assignment of roles to end-users.

Roles are important for adaptation because they can be used by a software system to identify whether privileged users invoke a task variant thereby affecting its priority. For example, a task could be considered more important if it was invoked by a

manager in an enterprise system or by a senior citizen in a public service software system (e.g., transportation).

5.2.4.4 Substitutability

An *application task variant* can be either **substitutable** by another *application task variant* or **non-substitutable**. This depends on the type of *parameter* that is used in the *parameter condition*. If the *parameter* is “*changeable*”, then its value can be changed to invoke an alternative *application task variant* that expects a different value. On the other hand, if the *parameter type* is “*non-changeable*”, then its value cannot be changed. Hence, it is not possible to invoke an alternative *application task variant* that expects a different parameter value. The possible substitutability options, namely “*substitutable application task variant*” and “*non-substitutable application task variant*”, are defined by the enumeration *TaskType* alongside the “*abstract task*” and the “*application task*” explained in Section 5.2.1.

A substitutable task variant could be executed instead of another one that has higher *resource intensiveness* (refer to Section 5.2.4.2). However, non-substitutable task variants are also important for addressing resource variability. Although non-substitutable task variants cannot be interchanged like their substitutable counterparts, they have different priorities and are associated with *parameter conditions* (refer to Section 5.2.4.1) that specify in which cases these task variants are executed. This helps software systems in making adaptation decisions that reduce the strain on limited resources to keep them available to the most important task variants.

5.3 Example task model from an automated warehouse system

Consider an example of a warehouse for a retail store that receives customer orders throughout the day. In this example, the warehouse is automated by robots that perform order preparation tasks and pack items into boxes. To prepare a customer’s order, robots locate the respective items in the warehouse, pack the items in boxes, and decorate the boxes to make them ready for delivery (e.g., seal boxes, and attach labels with addresses).

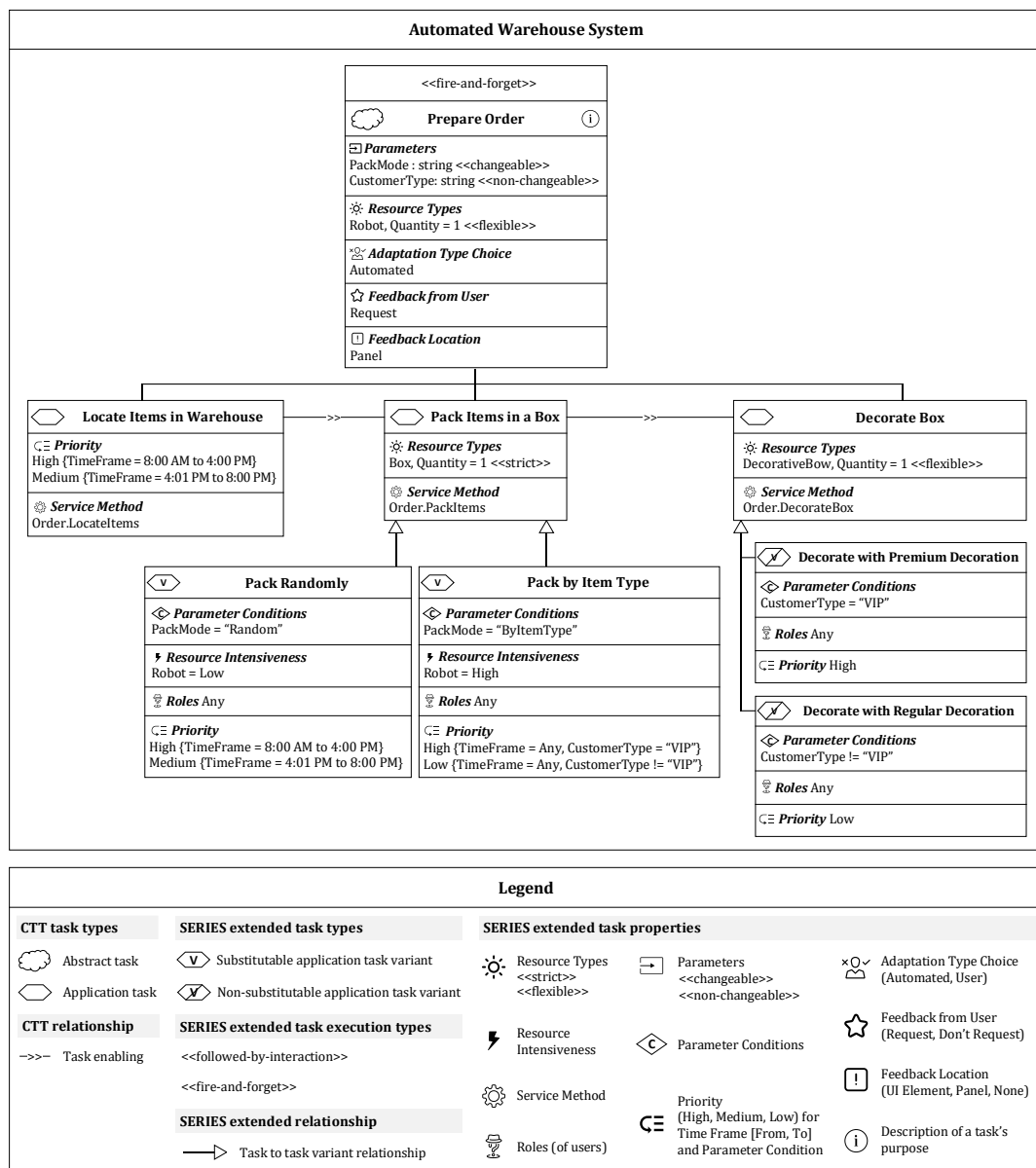


Figure 5.2 – A task model example from an automated warehouse system (excerpt)

Consider that robots should place items of the same type together in a pile inside the boxes. Robots and boxes are essential resources for the retail store's operations. Robots can temporarily go out of service due to unexpected errors or due to the need for recharging, thereby, delaying order fulfilment and causing financial losses. Similarly, due to the high demand for orders, the warehouse may run out of boxes to pack and deliver the orders. Figure 5.2 illustrates an excerpt of a task model for the automated warehouse system example. This task model is represented using the SERIES notation. Additionally, Figure 5.2 also shows a legend, which illustrates the constructs that are used in the task model. The legend consists of task types and a

relationship that SERIES uses from CTT, as well as task types, task execution types, relationships, and task properties that SERIES adds as extensions to CTT.

5.3.1 Abstract task: Prepare Order

The task model consists of an *abstract task* called “Prepare Order”, which is divided into three application tasks and represents the activity of preparing customer orders for delivery.

5.3.1.1 Description

The task “Prepare Order” has a *description*, which is accessed via the information icon displayed in the top right corner of the box that represents the task. This description explains further the purpose of the task “Prepare Order” and comprises the following text “This task represents order preparation at the warehouse before delivering the ordered products to the customers”.

5.3.1.2 Execution type

The task “Prepare Order” has a “*fire-and-forget*” *execution type* since end-users do not require a result from the task to interact with the system and execute another task. For example, the software system could execute a batch of order preparation tasks. The software system would process the batch and assign order preparation tasks from it to the robots at the warehouse. When the batch is done, a warehouse control employee checks the outcome of the batch of tasks altogether (e.g., if the software system returned any messages regarding changes to some customer orders).

5.3.1.3 Parameters

The task “Prepare Order” has two *parameters*. The first one is a *changeable parameter* called “PackMode”, which specifies the mode for packing items in a box. The second one is a *non-changeable parameter* called “CustomerType”, which specifies the type of customer for whom the order is being prepared.

The *parameter* “PackMode” is set to be *changeable* because it is possible to change its value to indicate that the packing should be done differently (e.g., items are sorted by their type or randomly). Hence, it is possible to consider “PackMode” as a type of

configuration parameter. On the other hand, the “CustomerType” parameter is set to be *non-changeable* because it is based on domain-related data such as the total amount of purchases or the number of years as a customer, which indicate loyalty to the store. Hence, in this case, it is not possible to downgrade a VIP customer to a regular customer for performing adaptation.

5.3.1.4 Resource types

The task “Prepare Order” has a single *flexible resource type* called “Robot”, which is used for preparing a customer order. The robot is set as a *flexible resource type* to indicate that it is substitutable by another robot during resource variability (e.g., due to unexpected robot malfunctions).

5.3.1.5 Feedback properties

The *adaptation type choice* is set to “*automated*” for the task “Prepare Order”. This means the software system will automatically select the least costly adaptation type to apply to this task. Additionally, the property *feedback-from-user* is set to “*request*”. This means when an adaptation type is applied to this task, feedback will be requested from the end-user to provide input on the outcome of the adaptation.

Moreover, the *feedback location* for “Prepare Order” is set to “*panel*”. This means end-users can provide feedback on a separate panel from the user interface that is used to run the task. For example, consider that a batch of “Prepare Order” tasks have finished executing and that adaptations were performed due to resource variability. The software system would list the performed adaptations as messages underneath each other in a side panel to inform the end-users about its decisions. The users could scroll through these messages and provide their feedback on the outcomes of the tasks. To provide a comparable example, the feedback panel discussed here resembles the notifications panel in Windows.

5.3.2 Application tasks

The *abstract task* “Prepare Order” is divided into three *application tasks* “Locate Items in Warehouse”, “Pack Items in a Box”, and “Decorate Box”. These *application tasks* represent the sequence of actions that are required for preparing an order.

5.3.2.1 Application task 1: Locate items in the warehouse

The first *application task* that is executed as part of the “Prepare Order” *abstract task* is called “Locate Items in Warehouse”. This *application task* has a “high” *priority* between 8:00 AM and 4:00 PM and a “medium” *priority* between 4:01 PM and 8:00 PM. This example shows how a task’s *priority* can differ per timeframe. Additionally, the *service method* is set to “Order.LocateItems”, where “Order” and “LocateItems” represent the *service name* and *method name* respectively.

5.3.2.2 Application task 2: Pack items in a box

The second *application task* is called “Pack Items in a Box”. This task has a single *resource type* called “Box”, which represents a container that is used to pack the items of the customer order. The *resource type* “Box” is set to *strict* to indicate that it is not substitutable. For example, in this case, it is not possible to substitute a box with another container such as a bag because the box provides better protection. However, there could be other cases where a *resource type* “Box” could be set to *flexible* to indicate that it is substitutable; the choice depends on the requirements for a particular domain. Additionally, the *service method* is set to “Order.PackItems”, where “Order” and “PackItems” represent the *service name* and *method name* respectively.

5.3.2.3 Application task 3: Decorate box

The third *application task* is called “Decorate Box”. This task has a single *flexible resource type* called “DecorativeBow”, which is used to decorate the box once the items are packed in it. The “DecorativeBow” is set to *flexible* since it can be substituted with other types of decorations during resource variability. Additionally, the *service method* is set to “Order.DecorateBox”, where “Order” and “DecorateBox” represent the *service name* and *method name* respectively.

5.3.2.4 Order of the application tasks

These three *application tasks* “Locate Items in Warehouse”, “Pack Items in a Box”, and “Decorate Box” execute in sequential order and are therefore linked by a “task enabling” relationship to indicate that one task enables the other. This means that “Locate Items in Warehouse” will execute first. Once the items are located the task

“Pack Items in a Box” will execute. Finally, after the items are packed in a box the task “Decorate Box” will execute to add decorations to the box.

5.3.3 Application task variants for “pack items in a box”

The *application task* “Pack Items in a Box” has two *application task variants*, which are “Pack Randomly” and “Pack by Item Type”. These *application task variants* are substitutable because it is possible to change the value of the “PackMode” parameter to execute one variant instead of another.

Furthermore, the variants of the task “Pack Items in a Box” are linked to it via a “task to task variant” relationship. This relationship indicates that these variants are special cases of “Pack Items in a Box”, whereby each variant performs the packing differently. The “task to task variant” relationship is similar to a generalisation relationship in UML. Hence, the *application task variants* use (inherit) the information found in their parent *application task* (e.g., resource type and service method). In this example, both “Pack Randomly” and “Pack by Item Type” use the *resource type* “Box”. Furthermore, both of these variants call the same service method (Order.PackItems) but with a different value for the parameter “PackMode”.

5.3.3.1 Application task variant 1: Pack randomly

The *application task variant* “Pack Randomly” is executed when the “PackMode” parameter is equal to “Random”. This variant has a “low” *resource intensiveness* on the robot *resource type* since the robot will perform the packing randomly without taking additional time to sort the items before placing them in the box. Additionally, end-users with any role can execute the “Pack Randomly” *application task variant*. Hence, the initiation of this task variant is not restricted to particular types of employees within the warehouse and its importance is not affected by who initiated it.

Furthermore, “Pack Randomly” has a “high” *priority* value between 8:00 AM and 4:00 PM and a “medium” *priority* value between 4:01 PM and 8:00 PM. This is an example of how *priorities* can differ among timeframes for an *application task variant*. The choice of timeframes with priorities depends on what is required for a domain. For example, the abovementioned priority and timeframe were chosen for the task variant “Pack Randomly” because customer orders are mostly fulfilled between 8:00 AM and

4:00 PM and packing items randomly would reduce the strain on the resources during this busy time.

5.3.3.2 Application task variant 2: Pack by item type

The *application task variant* “Pack by Item Type” is executed when the “PackMode” parameter is equal to “ByItemType”. This variant has a “high” *resource intensiveness* on the robot *resource type* since it would take the robot more time to identify the items and pack each type in a separate pile in comparison to packing items randomly. Like “Pack Randomly”, the task variant “Pack by Item Type” can be executed by users of any role. For example, this includes employees from the warehouse with any job title.

Furthermore, “Pack by Item Type” has a “high” *priority* value for VIP customers and a “low” *priority* value for non-VIP customers. The priorities, in this case, apply to any timeframe. The task variant “Pack by Item Type” places more strain on the robots (resources) in comparison to its counterpart “Pack Randomly”. Hence, it was given a low priority for non-VIP customers. This example shows how the priority of an *application task variant* can be the same for any timeframe but differ according to parameter values.

5.3.4 Application task variants for “decorate box”

The *application task* “Decorate Box” has two *non-substitutable application task variants*, which are “Decorate with Premium Decoration” and “Decorate with Regular Decoration”. Like the variants of the task “Pack Items in a Box”, the variants of “Decorate Box” are linked via a “task to task variant” relationship to indicate that the task variants are special cases of the application task.

These variants are non-substitutable because the value of the “CustomerType” parameter cannot be changed. This means that it is not possible to change an order that is designated for a VIP customer to an order for a regular customer. Hence, it is not possible to execute the task variant “Decorate with Premium Decoration” instead of “Decorate with Regular Decoration” and vice-versa.

5.3.4.1 Application task variant 1: Decorate with Premium decoration

The *application task variant* “Decorate with Premium Decoration” is executed when the parameter “CustomerType” is equal to “VIP”. This means that VIP customers will

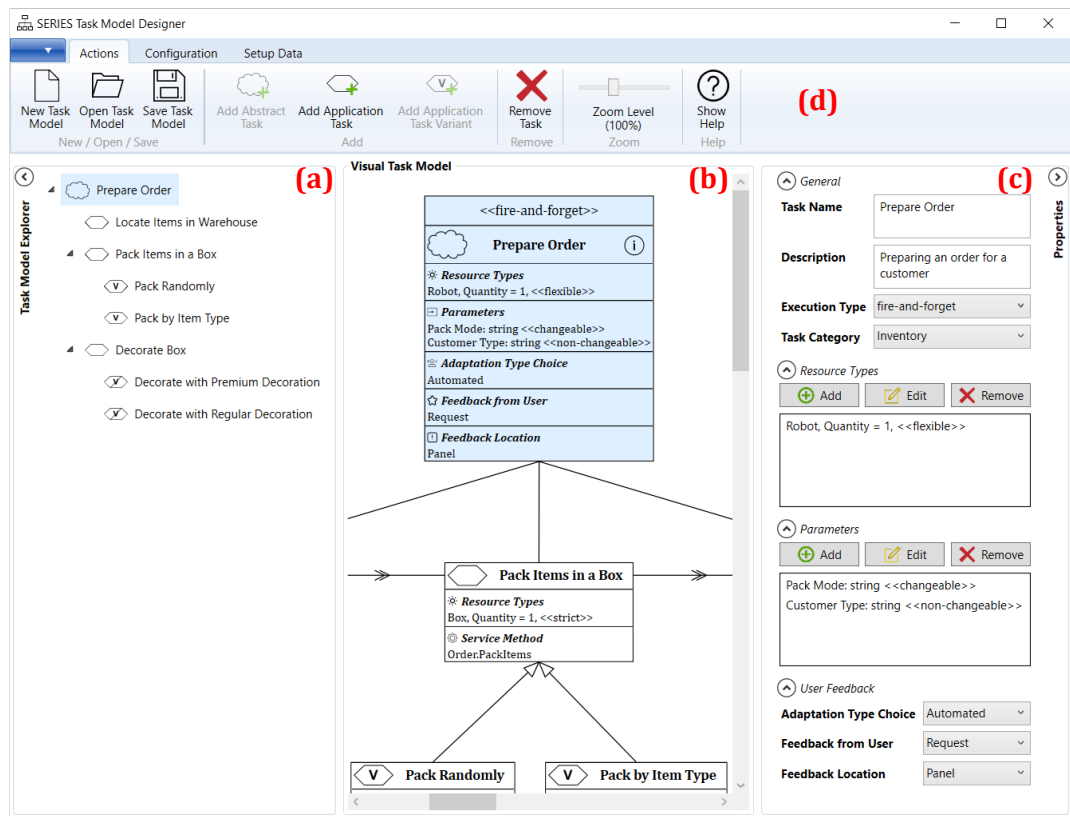


Figure 5.3 – Tool for creating and modifying task models using SERIES

receive a premium decoration for their order. Additionally, end-users with any role can execute “Decorate with Premium Decoration”. Moreover, “Decorate with Premium Decoration” has a “high” *priority* value at any timeframe, which means it is important for VIP customers to receive a premium decoration on their order regardless of when the order is prepared. This is an example of how a single *priority* value can be given to an *application task variant*.

5.3.4.2 Application task variant 2: Decorate with Regular decoration

The *application task variant* “Decorate with Regular Decoration” is executed when the parameter “CustomerType” is not equal to “VIP”. This means non-VIP customers will receive a regular decoration for their orders. Additionally, end-users with any role can execute “Decorate with Regular Decoration”. Furthermore, “Decorate with Regular Decoration” has a “low” *priority* value at any timeframe, which means it is not important for non-VIP customers to receive a decoration for their order. For example, in case there is a shortage in the decorative bow *resource type*, then non-VIP customers

The figure displays six data entry windows arranged in a 3x2 grid. Each window has a title bar with a close button (X) and a specific icon representing its function. The windows are as follows:

- Resource Type Data:** Contains fields for Name (Robot), Quantity (1), and Substitutability (radio buttons for Strict and Flexible, with Flexible selected). Buttons for OK and Cancel are at the bottom.
- Parameter Data:** Contains fields for Name (Pack Mode), Data Type (string), and Parameter Type (radio buttons for Changeable and Non-changeable, with Changeable selected). Buttons for OK and Cancel are at the bottom.
- Priority Data:** Contains fields for Priority (High), From Time (8:00 AM), and To Time (4:00 PM). Buttons for OK and Cancel are at the bottom.
- Parameter Condition Data:** Contains fields for Parameter (Pack Mode), Operator (=), and Value (Random). Buttons for OK and Cancel are at the bottom.
- Resource Intensiveness Data:** Contains fields for Resource Type (Robot) and Resource Intensiveness (Low). Buttons for OK and Cancel are at the bottom.
- Role Data:** Contains a field for Role (Manager). Buttons for OK and Cancel are at the bottom.

Figure 5.4 – Data entry windows for properties that represent sets of values (the data displayed in these windows are examples)

might not receive a decorated box. This could be achieved by performing an adaptation that cancels the execution of a “Decorate with Regular Decoration” task variant.

5.4 Supporting tool of SERIES

SERIES has a supporting tool that offers functionality for creating and modifying task models. A screenshot of this tool is shown in Figure 5.3. This tool was developed using C# and the Windows Presentation Foundation (WPF).

5.4.1 Panels: Task model explorer, visual task model, and properties

The supporting tool of SERIES is divided into three panels: (a) Task Model Explorer, (b) Visual Task Model, and (c) Properties, which are shown in Figure 5.3a, Figure 5.3b, and Figure 5.3c respectively.

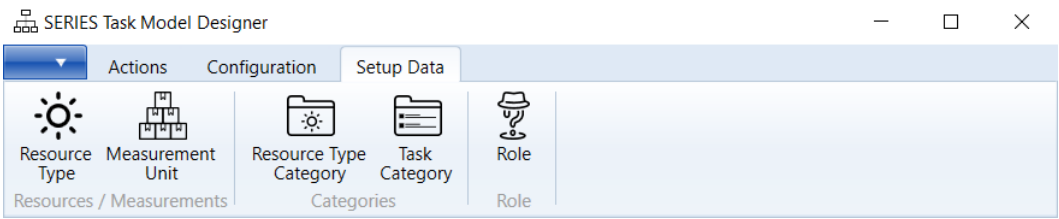


Figure 5.5 – Supporting tool of SERIES – setup data tab

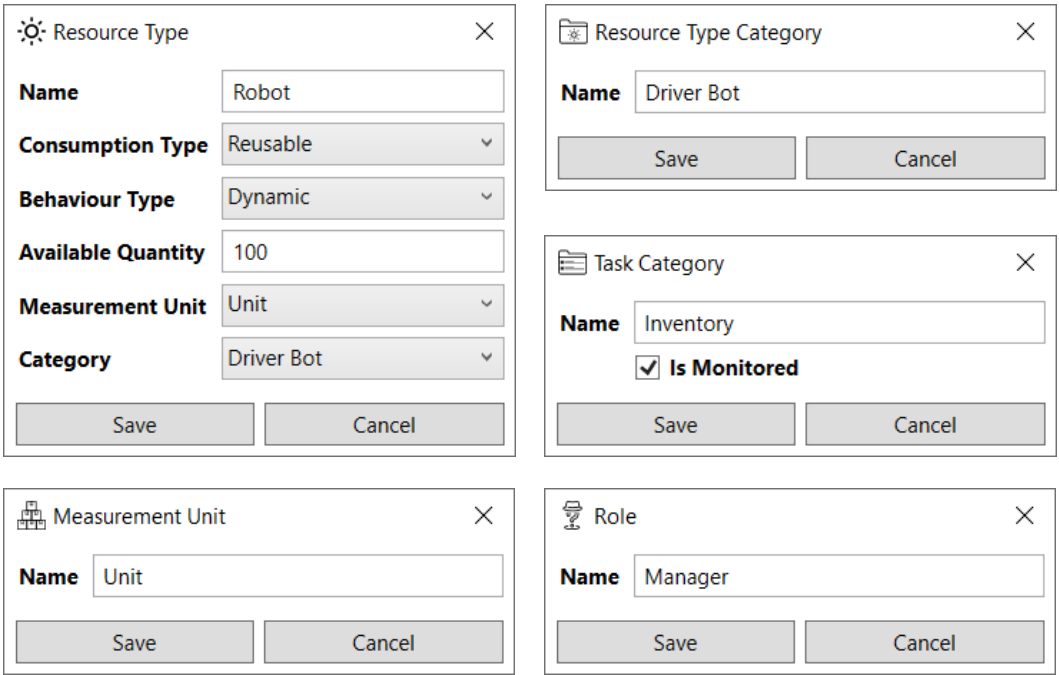
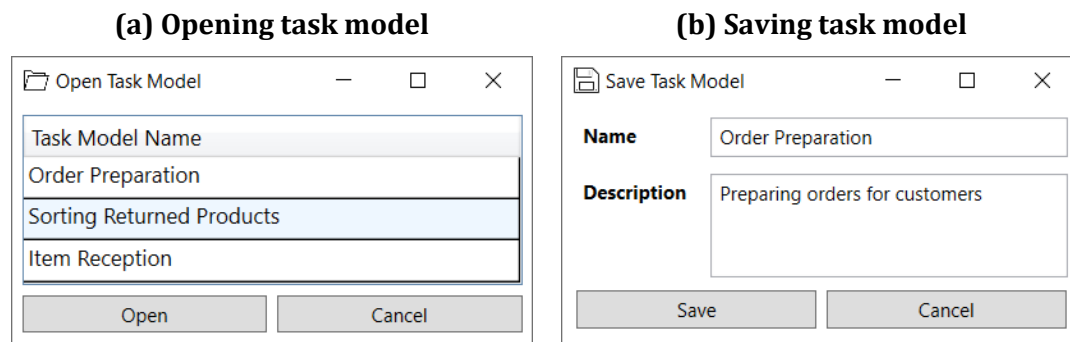


Figure 5.6 – Data entry windows for setup data properties (the data displayed in these windows are examples)

The “Task Model Explorer” panel offers a hierarchical view of a task model. This panel shows how the tasks are ordered under each other using a compact tree structure that enables users to navigate to the different parts of the model. The “Visual Task Model” panel displays the task model graphically using the SERIES notation. Moreover, it is possible to zoom in and out of the task model in the “Visual Task Model” panel using the “Zoom” slider.

The “Properties” panel displays the properties of the selected task. Users use this panel to edit the values of the properties (e.g., change the name of a task). A property can either have a single value like *task name* and *description* or a set of values like *resource types* and *parameters*. Properties with lists of values are shown in the “Properties” panel with “Add”, “Edit”, and “Remove” buttons. The buttons “Add” and “Edit” open windows for inputting and editing the data of the respective property.



**Figure 5.7 – Opening and saving task models from and to a database
(the data displayed in these windows are examples)**

Figure 5.4 shows screenshots of those data entry windows with example data. On the other hand, the button “Remove” deletes the selected value.

The possible set of values for the properties resource type, resource type category, measurement unit, role, and task category that are used in the data entry windows (Figure 5.4) are defined beforehand. The data entry windows for these properties are accessible through the “Setup Data” tab, shown in Figure 5.5, where the buttons open the corresponding data entry windows that are shown in Figure 5.6.

The automated-warehouse-system example shown in Figure 5.2 is displayed in the supporting tool in Figure 5.3. The “Task Model Explorer” panel shows the task model as a hierarchical view, which starts from the abstract task “Prepare Order”, followed by the application tasks and their corresponding application task variants. The “Visual Task Model” panel displays the task model, which is partially shown in Figure 5.3 to keep the text on the figure readable in the limited space on the page. The “Properties” panel shows the properties and their values for the *abstract task* “Prepare Order” that is selected in the task model. The name and description are editable as text, execution type and user feedback are selected from combo boxes, while resource types and parameters can be added, edited, or removed. Properties and values are displayed depending on which part of the task model is selected. For example, if a task variant was selected the “Properties” panel would also display parameter conditions, resource intensiveness, and roles.

5.4.2 Actions: Creating, loading, and saving task models

The actions tab shown in Figure 5.3d provides buttons for invoking the actions that are needed to create and modify task models. To clear the currently loaded task model

Listing 5.1 – JSON representation of the “Prepare Order” abstract task

```
1.  {
2.      "Id":1,
3.      "Name":"Prepare Order",
4.      "Description":"Preparing an order for a customer",
5.      "ExecutionType":"<<fire-and-forget>>",
6.      "TaskCategory":"Inventory",
7.      "AdaptationTypeChoice":"Automated",
8.      "FeedbackFromUser":"Request",
9.      "FeedbackToUser":"Panel",
10.     "ResourceTypes":
11.     [
12.         {
13.             "Name":"Robot",
14.             "Quantity":1,
15.             "Substitutability":"<<flexible>>"
16.         }
17.     ],
18.     "Parameters":
19.     [
20.         {
21.             "Name":"Pack Mode",
22.             "DataType":"string",
23.             "ParameterType":"<<changeable>>"
24.         },
25.         {
26.             "Name":"Customer Type",
27.             "DataType":"string",
28.             "ParameterType":"<<non-changeable>>"
29.         }
30.     ]
31. }
```

and create a new one, the user presses the “New Task Model” button. A user loads an existing task model by pressing the “Open Task Model” button and selecting the corresponding name from a list of task models that were previously saved to a database (Figure 5.7a). Similarly, a user saves a task model to a database by pressing the “Save Task Model” button and specifying a task model name (Figure 5.7b). The database that stores the task models is a realisation of the knowledge base introduced in Section 4.1.

The tool stores SERIES task models in JSON format. JSON stands for “JavaScript Object Notation” and is a well-known format that uses human-readable text to store data objects. The JSON format is a serialised (textual) representation of objects from a software system. The serialised objects in the supporting tool of SERIES represent task models. JSON is useful for storing task models in the database and also for transmitting

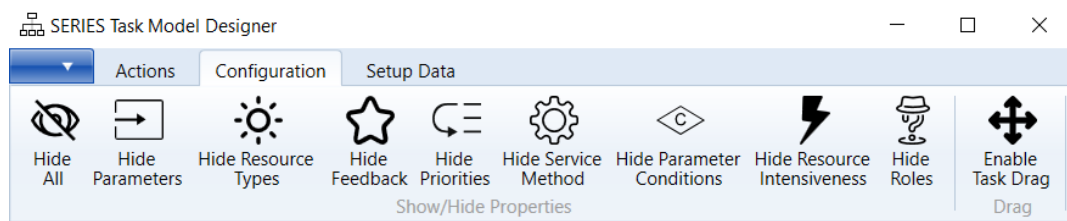


Figure 5.8 – Supporting tool of SERIES – configuration tab

them over a network to be processed by a software application. The JSON documents that store the data of SERIES task models consist of all the tasks and task variants in a task model’s hierarchy and their property values. For example, as shown by the “Properties” panel in Figure 5.3c, the abstract task “Prepare Order” has data corresponding to the *task name*, *description*, *execution type*, *task category*, *resource types*, *parameters*, *adaptation type choice*, and *feedback*. This data is represented using JSON in Listing 5.1.

5.4.3 Adding, modifying, and removing tasks

The hierarchy of a task model starts with an *abstract task*. Hence, the first step in the creation of a task model is the addition of an *abstract task* by pressing the button “Add Abstract Task”. Moreover, when an *abstract task* is added and selected the button “Add Application Task” is enabled. Upon pressing this button an application task is added as a subtask of the abstract task. Task variants for an application task are added by pressing the button “Add Application Task Variant”.

Upon selecting a task or a task variant on the “Task Model Explorer” (Figure 5.3a) or the “Visual Task Model” (Figure 5.3b), the corresponding properties are automatically shown in the “Properties” panel (Figure 5.3c). When a user edits the values of these properties, the new values are directly reflected in the “Task Model Explorer” and the “Visual Task Model”. For example, when a user changes the name of a task in the “Properties” panel, the new name directly appears on the graphical representation of the task model.

Additionally, users can remove a task from the task model by selecting the task and clicking on the button “Remove Task”. This includes any child tasks that are linked to the removed task. For example, if an application task is removed and it has a set of application task variants, then those task variants will be removed as well.

5.5 Graphical representation of SERIES task models

As mentioned in Section 5.1 and shown by the example presented in Section 5.3, the graphical representation of the constructs in SERIES is based on CTT and also inspired by UML class diagrams.

5.5.1 Representation of tasks and task variants

In CTT, a task is represented by an icon with the task name written underneath. However, SERIES represents tasks and task variants as boxes that resemble classes in a UML class diagram. The box representation is needed because SERIES extends CTT with additional characteristics that should be represented graphically. Hence, these boxes have multiple parts that represent the properties of the tasks and task variants.

SERIES uses the icons of CTT for the abstract and application tasks (Paterno, Mancini and Meniconi, 1997). Furthermore, SERIES adds the letter “v” to the icon of the application task and uses the new icon for the application task variants. These icons are displayed in the top left corner of the boxes that represent tasks and task variants as shown by the example in Figure 5.2. Moreover, SERIES uses both a textual description and an icon for the properties of tasks and task variants (Figure 5.2). For example, the property that represents parameters has the word “Parameters” as a textual description alongside an icon of an arrow pointing inside a rectangle to denote an input that is given to the system.

5.5.2 Representation of relationships

A relationship that connects an abstract task to its subtasks (application tasks) is represented as a line without an arrow as is done in CTT. Additionally, a relationship that connects subtasks is represented as a line with a temporal operator as is done in CTT as well.

On the other hand, a relationship that connects an application task to an application task variant resembles the generalisation relationship in UML. This relationship is represented as a line with a white triangle as an arrow tip. The generalisation relationship in UML connects a concept to its special cases (e.g., a class to its subclasses). The same principle applies to tasks and task variants, whereby the task

variants are special cases of a task. Hence, the graphical representation of the task-to-task-variant relationship was inspired by UML's generalisation relationship.

5.5.3 Different levels of detail

The “Configurations” tab, shown in Figure 5.8, has a set of options for showing or hiding parts of the task model. This enables users to show more or fewer details as needed. Hence, it is possible to hide the details that are not needed at the moment to show a bigger part of the task model on the screen. For example, it is possible to hide all the properties from the “Visual Task Model” and only keep the task names while retaining the ability to browse and edit the properties in the “Properties Box”. It is also possible to hide properties that are less frequently used for a particular project.

The ability to control the level of detail on the task model is inspired by UML class diagrams where it is possible to hide the parts of the boxes that represent the classes as needed. In UML class diagrams the box parts represent the class name, attributes, and operations. On the other hand, in a SERIES task model, the box parts represent the name of a task or a task variant and the properties (e.g., priority, parameters, resource intensiveness, etc.).

5.5.4 The layout of task models

SERIES adopts a hierarchical layout for task models. This type of layout is typically used by task modelling notations like CTT. Hence, the abstract task is displayed at the top of the task model. Then, the subtasks (application tasks) are displayed under the abstract task and the task variants are displayed under the subtasks.

The supporting tool of SERIES automatically adjusts the layout of the task model when new tasks are added or removed. This facilitates the management of task models without having to worry about overlapping lines and the necessity of performing manual adjustments to keep the task model readable. Nonetheless, if a user wishes to perform a manual rearrangement of the task model's layout, the tool also supports the dragging and dropping of tasks. The ability to drag and drop tasks is enabled by pressing the button “Enable Task Drag” from the “Configurations” tab, which is shown in Figure 5.8.

Table 5.1 – Mapping concepts from SERIES meta-model to SPARK components

SERIES		SPARK
<i>Class name</i>	<i>Class property</i>	<i>Used by adaptation component</i>
Resource type	Consumption type Available quantity	Resource type state monitor Resource type state analyser Task execution allocator
	Behaviour type	Adaptation executor
Task priority assignment	Priority value	Task priority calculator
Task resource type assignment	Resource intensiveness	Adaptation executor (task variant substitution)
	Substitutability	Adaptation executor (resource substitution)
	Quantity	
Service method	Service name	Task execution monitor
	Method name	
Task and Task variant	Feedback from user	Feedback elicitor
	Feedback location	Feedback provider
	Execution type	Adaptation executor (delay task execution)
	Type	Adaptation executor (task variant substitution)

5.6 Mapping of concepts from SERIES to SPARK

Table 5.1 shows the mapping of concepts from the meta-model of SERIES to the adaptation components of SPARK. In the class *resource type*, the *consumption type* and *available quantity* properties are used in three components, namely resource type state monitor, resource type state analyser, and task execution allocator. These properties are used to monitor the available quantities of reusable and depletable resources to check if they are facing variability and allocate accordingly the number of possible task executions. Moreover, the property *behaviour type* is used by the adaptation executor component to check whether the adaptation is applied to the software system or the resource accordingly to whether the type of resource type is static or dynamic respectively.

In the class *task priority assignment*, the property *priority value* is used as input to the task priority calculator component to calculate the initial priority value for a task. In the class *task resource type assignment*, the property *resource intensiveness* is used by the adaptation executor component to check which task variant gets substituted by another one. The properties *substitutability* and *quantity* are used by the adaptation executor component to check if the resource type is substitutable and check the available quantities for the substitutable resource types respectively. In the class *service method*, the properties *service name* and *method name* are used by the task execution monitor component to identify which task (variant) is initiated by an end-user so the appropriate adaptation decisions can be made accordingly.

In the classes *task* and *task variant*, the property *feedback-from-user* is used by the feedback elicitor component to check if it needs to request feedback from the end-user when adaptation is performed. Moreover, the property *feedback location* is used by the feedback provider component to specify the location for providing feedback when an adaptation is performed (directly on the UI or in a separate panel). The property *execution type* is used by the adaptation executor component to identify whether a task can be delayed. The property *type* is used by the adaptation executor to identify whether a task variant is substitutable by another one. If one type of adaptation does not apply, the framework would apply another type.

5.7 Chapter summary

This chapter presented a task modelling notation called SERIES, which this thesis proposes for supporting resource-driven adaptation in software systems. SERIES is based on CTT and its graphical representation is also inspired by UML.

The meta-model of SERIES was presented as a class diagram and its concepts were explained. The explanation differentiated between the concepts that were incorporated from CTT and those that were added by SERIES. Moreover, I have mapped the concepts from the SERIES meta-model to the adaptation components of the SPARK framework. Additionally, SERIES was demonstrated through an example task model that corresponds to an automated warehouse system. Furthermore, the supporting tool of SERIES was presented as screenshots and its features were explained. Moreover, the graphical representation of SERIES was explained and related to the example task model from the automated warehouse system.

6

SPARK: A Framework for Resource-Driven Adaptation

This chapter presents SPARK, which is a framework for supporting resource-driven adaptation. First, this chapter explains SPARK's proactive and reactive adaptation components. Then, it presents an example from an automated warehouse system to demonstrate the calculations that are performed by SPARK's adaptation components. Afterwards, this chapter gives an overview of a prototype implementation of these components.

6.1 Introduction

The SPARK framework executes resource-driven adaptation due to variations in resources used by software systems. SPARK realises the adaptation components introduced in Section 4.3, which is based on the MAPE-K control loop approach (Kephart and Chess, 2003). These components are defined, according to the MAPE-K, as monitors (M), analysers (A), planners (P), executors (E), and a knowledge (K) base. An example from an automated warehouse system is used in this chapter to explain the adaptation components.

As explained in Section 3.2, existing resource-driven adaptation approaches target a limited number of resource types. SPARK supports tasks that use *depletable* and *reusable* resource types. Additionally, SPARK supports the generation of unique task priorities using multiple criteria including the task's parameter values, the time of day when the task was initiated, the task's resource intensiveness, the role of the user who initiated the task, and how critical the task is in its respective domain. Furthermore,

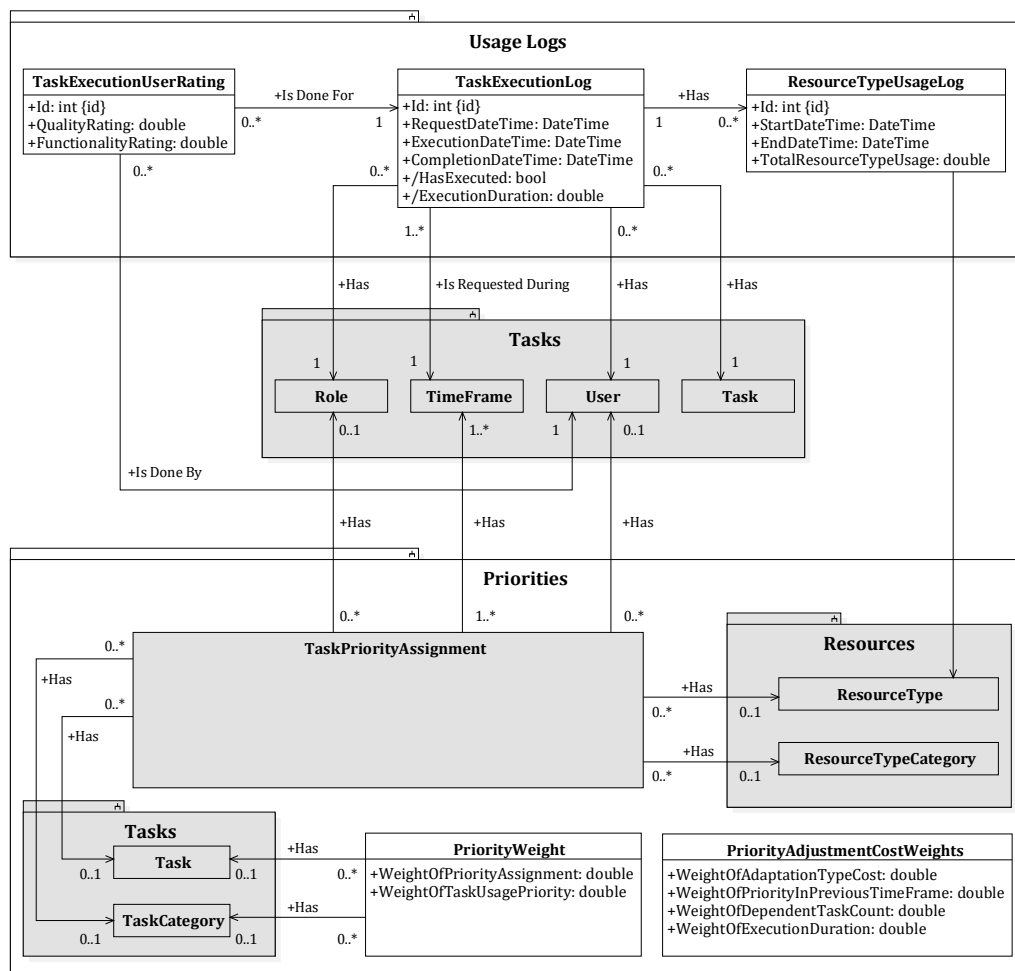


Figure 6.1 - Logging and prioritisation of tasks
(the grey parts represent the same classes used in the SERIES meta-model that was presented in Section 5.2)

SPARK supports four types of adaptation, namely: (i) execution of other task variants that require fewer resources; (ii) substitution of resource types with alternative ones; (iii) execution of tasks in a different order based on their priorities; and (iv) cancellation of tasks when no other task variant or resource can be used.

Figure 6.1 shows the classes that are used in SPARK for the prioritisation and logging of tasks in a software system. The grey parts in the class diagram represent the same classes that are used in the SERIES meta-model in Section 5.2. The explanation of the adaptation components in the next sections will reference Figure 6.1 in terms of the classes that are related to an adaptation component. Moreover, Figure 6.2 show these adaptation components, where each adaptation component is mapped to a MAPE-K component to illustrate the flow of the adaptation components' execution.

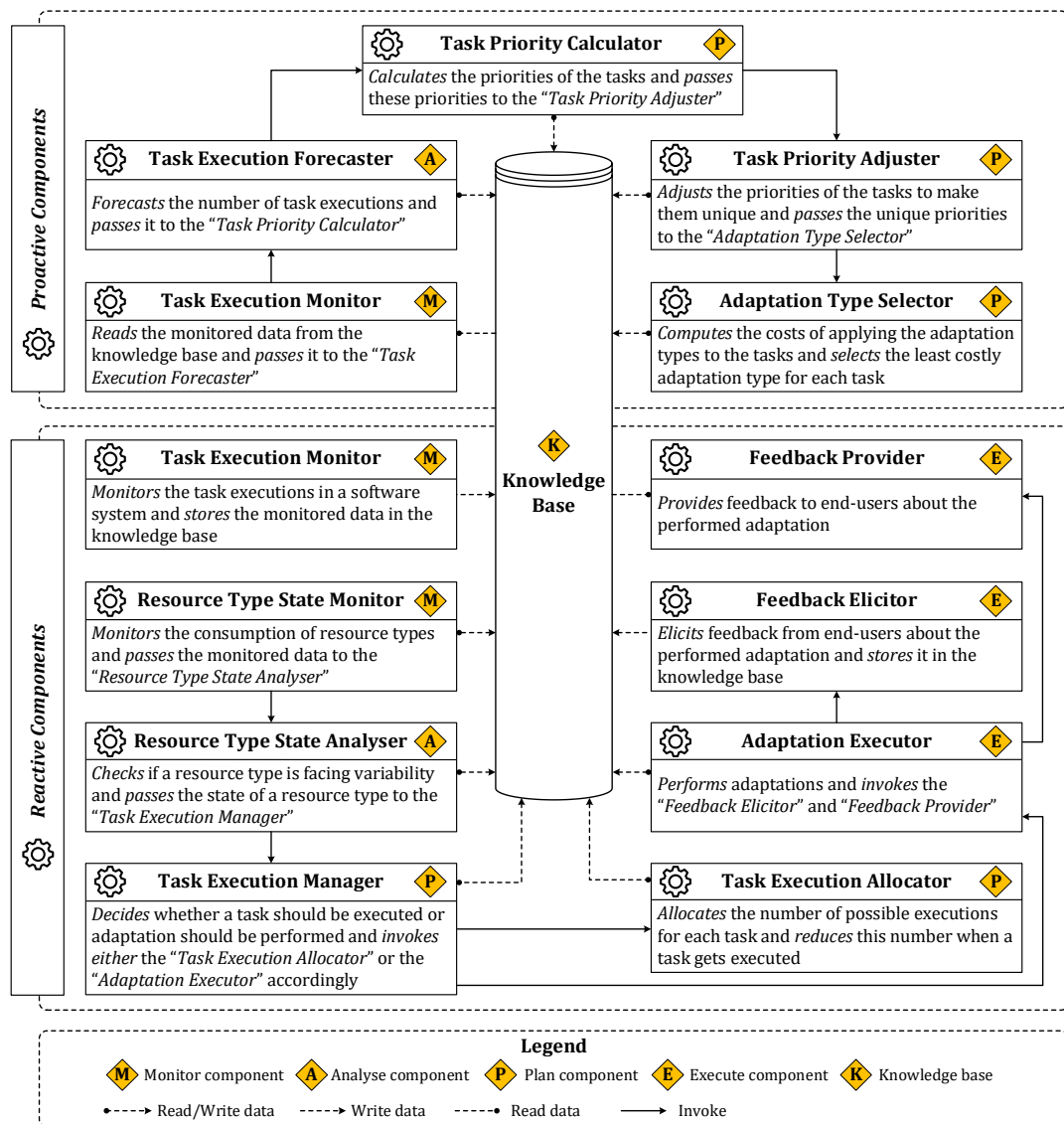


Figure 6.2 – SPARK proactive and reactive adaptation components (based on MAPE-K)

Furthermore, the calculations performed by the adaptation components will be illustrated via an example of a warehouse for a retail store, which consists of tasks, namely "Pack Item in Container" (T_1), "Receive Item to Warehouse" (T_2), "Restock Item in Warehouse" (T_3), "Sort Returned Items" (T_4), and "Dispatch Item from Warehouse" (T_5). In this example, robots pack the respective items in a container to prepare a customer order, receive items that are delivered to the warehouse, restock the items in the warehouse, sort returned items from customers, and dispatch items from the warehouse. This example comprises the abovementioned five tasks, a reusable resource type (robot), and a depletable resource type (container).

6.2 Proactive adaptation components

SPARK implements five proactive adaptation components that include a *task execution monitor*, *task execution forecaster*, *task priority calculator*, *task priority adjuster*, and *adaptation type selector*, which are illustrated in Figure 6.2 and explained in the following subsections.

6.2.1 Task execution monitor

SPARK monitors the execution of tasks in a software system via the *task execution monitor* component. The monitored data is logged in the knowledge base and is used as historical data for forecasting future executions of tasks. The classes shown under the “Usage Logs” package in Figure 6.1 are related to logging task executions. The class *TaskExecutionLog* represents who (Users) had used or attempted to initiate what task (or task variant) and during which *TimeFrame*. A user could attempt to initiate a task without being successful due to the lack of resources. This is indicated by the “HasExecuted” property. Nonetheless, the attempts are also logged to reflect what tasks the users find important during a timeframe. Both the number of task executions and the number of task execution attempts reflect the task’s priority from an end-user perspective.

In order to prevent end-users from abusing their ability to impact the priority of a task through improper usage, SPARK avoids storing consecutive task execution attempts within milliseconds of each other. This is done by using throttling (Azure, 2022), which reduces the trigger rate. Throttling is used in event-driven programming to ensure that a function is called at most once in a specified period (e.g., once every 2 seconds).

6.2.2 Task execution forecaster

The data represented by the class *TaskExecutionLog*, shown in Figure 6.1, is used as historical data to forecast the number of task executions, which represents the number of times Users are expected to execute a *Task* (T) during a *TimeFrame* (TF). This is done via the *task execution forecaster* component. The forecasted task execution (FTE) represents a number of expected task executions based on a forecast from historical data. FTE is obtained using a regression algorithm from the ML.NET framework

(2021), where FTE is the dependent variable and T and TF are the independent variables. The independent variables are used to forecast the value of the dependent variable. Regression analysis is used for analysing the relationship between variables for forecasting purposes. The type of regression analysis that is used in our case is multiple linear regression because there is one dependent variable (FTE) and multiple independent variables (T and TF). Moreover, FTE is used during the process of task prioritisation, which is discussed next.

6.2.3 Task prioritisation

Priorities play an important role to determine whether tasks are executed by directly gaining access to the resources or are executed after adaptation is performed. The priorities of tasks are calculated by the *task priority calculator* and the *task priority adjuster* components. The calculated priority value (P_v) is a real number between 1 and n as shown in Equation 6.1. Furthermore, each task's P_v is then classified (i.e., $C(P_v)$) as *high*, *medium*, or *low* based on a threshold that provides an equivalent distribution for the priorities. The classification is based on the following ranges specified in Equation 6.2: high-priority $\in [1-2)$; medium-priority $\in [2-3)$; low-priority $\in [3-n]$.

6.2.3.1 Task priority calculator

The *task priority calculator* computes initial priority values for tasks. The initial priority value for a task is based on two inputs that provide complementary perspectives: (i) domain priority and (ii) priority from the forecasted task execution.

The domain priority (DP) reflects the importance of a task for a domain. Its value is obtained from someone who knows the domain like the system administrator. Tasks are assigned DPs as denoted by the class *TaskPriorityAssignment*, which is shown in Figure 6.1. The DP takes into consideration the following criteria: the timeframe of the task execution, the role of the user who is attempting to execute the task, and the task variants. Timeframes represent time intervals that are meaningful for a domain.

For example, in a warehouse system, order preparation has a higher priority than sorting returned products during the daytime, when most of the orders are shipped. Roles characterise users and differ among software systems. For instance, roles can represent job titles such as warehouse clerk and manager in a warehouse system. Task priorities can differ according to roles because roles indicate that certain users are

$$P_v = \{ p \mid p \in \mathbb{R} \wedge 1 \leq p \leq n \}$$

Equation 6.1 – Priority value range

$$C(P_v) = \begin{cases} High & 1 \leq P_v < 2 \\ Medium & 2 \leq P_v < 3 \\ Low & 3 \leq P_v \leq n \end{cases}$$

Equation 6.2 – Priority value classification

$$TH = \frac{\min(FTEC) + \max(FTEC)}{\max(P_v)}$$

Equation 6.3 – Threshold calculation

$$TFTEP = \begin{cases} 1 & FTEC < TH \\ 2 & TH \leq FTEC \leq TH \times 2 \\ 3 & FTEC > TH \times 2 \end{cases}$$

Equation 6.4 – Threshold forecasted task execution priority calculation

$$P_I = (DP \times W_{DP}) + (TFTEP \times W_{TFTEP})$$

Equation 6.5 – Initial task priority calculation

more privileged, and a task can have a higher priority when it is initiated by someone with a more privileged role. For example, in a warehouse system, a shelf stock count task has a higher priority when it is initiated by a manager.

The *forecasted task execution* (FTE) is calculated from the logged historical data (see Section 6.2.1). The *forecasted task execution counter* (FTEC) is then computed by sorting the FTE values in descending order, where the highest FTE value has an FTEC value that is equal to one. Moreover, SPARK uses thresholds to calculate priority values within the range specified in Equation 6.1. Here, a threshold (TH) value is calculated as described in Equation 6.3, whereby tasks with a lower FTEC have a higher priority. Furthermore, a threshold is applied to FTEC to ensure its value is within the priority range of 1 and 3 (mapping the priority levels of *high*, *medium*, and *low*). The result is a *threshold forecasted task execution priority* (TFTEP) as shown in Equation 6.4. The initial priority (P_I) is then computed by multiplying each input (i.e., DP and TFTEP) by its corresponding weight as shown in Equation 6.5, where the sum of the weights is equal to one.

Table 6.1 – Initial priority calculation example

Tasks	DP	FTE	FTEC	TH	TFTEP	P _I
T ₁	2	30	1	2	1	1.5
T ₂	1	20	2	2	2	1.5
T ₃	1	10	3	2	2	1.5
T ₄	2	7	4	2	2	2.0
T ₅	2	5	5	2	2	2.0

*Assuming, for this example, the following priority weights: $W_{DP} = 0.5$ and $W_{TFTEP} = 0.5$

Moreover, the *TaskUsageLog* data is combined with the *PriorityAssignment* to calculate the priorities during a *TimeFrame*. Each input has a weight that is indicated as part of the *PriorityWeight* class under “Priorities” in Figure 6.1. These weights can be the same for all tasks and can differ from one Task or *TaskCategory* to another. For example, the priority assignment could have a higher weight for some tasks that the management of an enterprise deems important (managerial view of priorities). In other cases, there could be a higher weight for task usage (user view of priorities).

6.2.3.2 Task priority calculator: example

Assume that all five tasks (T₁ to T₅) are monitored by SPARK via the *task execution monitor* component, and the *task execution forecaster* component forecasted the number of task executions (FTE) for each task as shown in Table 6.1. The forecasted task execution counter (FTEC) is then computed by sorting FTE in descending order and providing an incremental counter value for each task. The threshold (TH) value is computed by adding the minimum and maximum values of FTEC and then dividing by the maximum priority value. In this case, TH will be equal to 2, which is then used for computing TFTEP. Once TFTEP is computed, and the domain priority (DP) is specified, both inputs will be used to calculate the initial priority (P_I) for each task. This is done by multiplying each input by its corresponding weight and then adding the results. The weights used in this example for the inputs are specified in Table 6.1.

6.2.3.3 Task priority adjuster

It is possible to have tasks that share the same resources with the same initial priority values (P_I). The *task priority adjuster* component adjusts the P_I of tasks to

$$C_{DT} = C_{AD} \times W_{CAD} + \sum P_A \times W_{\sum P_A} + |DT| \times W_{|DT|} + ED \times W_{ED}$$

Equation 6.6 – Cost function for deprioritising a task

$$\varepsilon_0 = \frac{NTG_{P_I} - CTG_{P_I}}{|CTG| + 1}, \text{ where } C(NTG_{P_I}) = C(CTG_{P_I})$$

Equation 6.7 – Initial epsilon value calculation

$$P_A = P_I + \varepsilon_k, \text{ where } C(P_A) = C(P_I) \text{ and } \varepsilon_k = \begin{cases} \varepsilon_0 & k = 0 \\ \varepsilon_{k-1} + \varepsilon_0 & k > 0 \end{cases}$$

Equation 6.8 – Priority adjustment calculation

ensure that each task gets a unique priority value. Tasks with the same P_I are grouped, and the cost function shown in Equation 6.6 is applied to deprioritise a task (C_{DT}) and adjust the priorities of the tasks in each group. The cost function takes into consideration the following inputs: the cost of performing adaptation for the task, the sum of adjusted priorities from previous timeframes, the total number of dependent tasks, and the estimated duration of the task's execution. Moreover, each input has a corresponding weight to specify its importance. These weights are illustrated under the package "Priorities" in Figure 6.1, where the class *PriorityAdjustmentCostWeight* represents the four weights used for the calculation of C_{DT} . The abovementioned inputs of the cost function are explained next.

The *first input* is the result of a cost function that calculates the cost of adaptation. When a task cannot execute, due to the lack of resources, adaptation is performed using one of the supported adaptation types. The use of an adaptation type has a cost (to be discussed in Section 6.2.4). If an adaptation is needed, the type of adaptation that has the lowest cost is performed. When tasks have the same priority and the resources are not enough for all of them, it is possible to decide which one gets the resources based on how costly it would be to perform the adaptation for each of them.

The *second input* involves summing up the task's priorities in previous timeframes. This value will be larger for the task that had lower priorities and would increase the overall value of C_{DT} . Hence, the advantage is given to the tasks that got fewer chances of being executed so far.

The *third input* is the task's number of dependent tasks. As explained in Section 5.2.1, a task can enable its following task. For example, in the automated warehouse example, "locate items in the warehouse" enables "pack items in a box", "pack items in a box" enables "decorate box", and "decorate box" does not enable any following task (i.e., does not have dependents). The number of dependent tasks for "locate items in the warehouse", "pack items in a box", and "decorate box" is two, one, and zero. Therefore, tasks with a higher number of dependents are given a higher priority because without them other tasks would not get executed.

The *fourth input* is the task's estimated execution duration, which is obtained from the historical data by computing a task's average execution duration from the *TaskExecutionLog* shown in Figure 6.1. The priority adjuster gives a higher priority to the tasks that require on average less time to complete. This way, more tasks can execute within a timeframe.

Furthermore, that have the same P_I are placed in a group and are sorted by the values of the cost function C_{DT} . Each P_I value for a task is incremented by an epsilon value ϵ_k , where k represents the task order (sequence number) in a group. An initial value of ϵ , represented as ϵ_0 , is computed based on Equation 6.7 to ensure that the values of the adjusted priorities are between the priority values of the tasks in the current group (CTG_{P_I}) and the priority values of tasks in the next group (NTG_{P_I}). It is important to note that the priority classification (high, medium, or low) of the tasks under NTG_{P_I} should be equal to the priority classification of the tasks under CTG_{P_I} . Hence, the adjusted priority value (P_A) and P_I should have the same priority classification. For example, if CTG_{P_I} is equal to 1.5 and NTG_{P_I} is equal to 2.2, then both groups have different classifications. Therefore, NTG_{P_I} becomes 2.0 instead of 2.2, which allows the adjusted priority values for the tasks in CTG to remain in the same priority classification. Moreover, the first task in CTG (i.e., when $k = 0$) has an adjusted priority value P_A equal $P_I + \epsilon_0$ as shown in Equation 6.8. As for the rest of the tasks in the group (i.e., when $k > 0$), the value of ϵ_k is computed by adding the previous epsilon value (ϵ_{k-1}) with the initial epsilon value (ϵ_0). The value of ϵ_k is then added with P_I to compute a task's P_A , which is also shown in Equation 6.8.

6.2.3.4 Task priority adjuster: example

Based on the values in Table 6.1, three tasks have the same P_I of 1.5 and two tasks with the same P_I of 2.0. In this case, the cost function for deprioritising tasks (C_{DT}) is

Table 6.2 – Adjusted priority calculation example

Tasks	C_{AD}	$\sum P_A$	$ DT $	ED	C_{DT}	ϵ	P_A
T_1	2	1.5	1	2	1.625	0.125	1.625
T_2	2	1.7	1	2	1.675	0.25	1.75
T_3	3	1.8	2	3	2.45	0.375	1.875
T_4	3	2.2	2	3	2.55	0.33	2.33
T_5	4	2.5	2	3	2.875	0.66	2.66

*Assuming, for this example, the following values for the input weights: $W_{C_{AD}} = 0.25$, $W_{\sum P_A} = 0.25$, $W_{|DT|} = 0.25$, and $W_{ED} = 0.25$

computed based on its four inputs (refer to Section 6.2.3.3). Assume that the four inputs have the values specified in Table 6.2. This table also shows the computed values for C_{DT} . Moreover, two groups G_1 and G_2 are formed, where G_1 has three tasks T_1 , T_2 , and T_3 with P_i equal to 1.5, and G_2 has two tasks T_4 and T_5 with P_i equal to 2.0. For each task in G_1 and G_2 , the values of P_A should be distinct and range between [1.5-2.0) and [2.0-3) respectively. Moreover, for the groups G_1 and G_2 , ϵ will be equal to 0.125 and 0.33 respectively. By sorting the tasks based on C_{DT} , ϵ is then incremented by its value for each task in the group. This gives us the values found in Table 6.2. Therefore, the adjusted priority (P_A) for each task is then computed by adding ϵ to the task's P_i . The P_A of each task is shown in Table 6.2.

6.2.4 Adaptation type selector

After computing task priorities, the *adaptation type selector* component selects for each task the adaptation types that are viable when resources are facing variability. SPARK supports four adaptation types, which are explained next.

6.2.4.1 Supported adaptation types

A task is changed into a similar one by executing a task variant that takes alternative parameter values and consumes fewer resources. One example from an automated warehouse system involves using an alternative packing method for customer orders such as “Pack Randomly” and “Pack by Item Type” task variants for the “Pack Items in a Box” task. Another example is related to variants of a “Verify Order” task, which checks whether the products packed in the box are the ones ordered by the customer.

One task variant performs the verification by scanning the products while the second one weighs the box and compares the result to the expected weight. The trade-off here is between accuracy and speed whereby scanning is generally more accurate whereas weighing the box is faster and has a satisfactory accuracy when the products are not very light. The faster variant is beneficial when many robots malfunction unexpectedly.

Resources are substituted with alternative ones, so a task can be executed when the resources it requires are not available. For example, in an automated warehouse, a robot that needs repairs is substituted by another type of robot to avoid interrupting high-priority tasks. In a semi-automated warehouse, where humans collaborate with robots, a malfunctioning robot that was supposed to perform a high-priority task can be substituted by a human employee who is working on a low-priority task.

Alternative task executions are considered when resources are not available by postponing the execution of low-priority tasks to another time until the required resources become available. For instance, the low-priority task of “sorting returned products” is postponed until more robots are available. This enables high-priority order preparation tasks to complete on time. Another example involves queuing robot repair tasks to be processed by order of their priority when a single robot repair bay is operational because the others are out-of-service due to machinery malfunctioning or technicians being sick.

The execution of tasks can be *cancelled* when no other adaptation type is applicable. An example is cancelling the transmission of optional log data from a robot to a server to conserve battery power. Another example is cancelling a task that involves using optional decorative items like stickers to decorate a box of products that were ordered by non-VIP customers. This task is cancelled in situations where there is a low stock of decorative items due to an unexpected delay in the supply chain. Therefore, the addition of these optional items would only be done for the orders of VIP customers.

Moreover, the abovementioned adaptation types consider the computed priorities for the tasks. For example, the resources of a task are not substituted with alternatives that are needed by higher-priority tasks. Additionally, when alternative tasks are executed the delayed task is executed at a time when resources are available, and it is important enough to execute.

$$Input = \{ v \mid v \in \mathbb{R} \wedge 1 \leq v \leq 5 \}$$

Equation 6.9 – Input value range

$$C_{AD} = CWTE + \frac{1}{n} \sum_{i=1}^n SF_i + \frac{1}{n} \sum_{i=1}^n SQ_i + \frac{1}{n} \sum_{i=1}^n FET_i$$

Equation 6.10 – Cost function for adaptation

6.2.4.2 Adaptation type selection

First, the applicability of the adaptation types to a task is checked. Hence, if a task is “*followed-by-interaction*” (refer to Section 5.2.2.2) it is not possible to adapt by delaying the task because the users need the result immediately to perform additional interaction with the software system. On the other hand, this is possible for “*fire-and-forget*” tasks. Additionally, if a task does not have variants (refer to Section 5.2.4) then it is not possible to adapt by executing a task variant that consumes fewer resources. Furthermore, if a task’s required resource is set to “*strict*” (refer to Section 5.2.2.4) then it is not possible to adapt by substituting this resource with another one.

Then, one of the applicable adaptation types is selected for a task based on a cost, which is calculated using a cost function that takes four inputs. Each input is represented via a rating value that is on a scale of 1 to 5 as shown in Equation 6.9. The cost function for selecting an adaptation type is shown in Equation 6.10. In this cost function, the inputs for the cost of adaptation (C_{AD}) represent sub-costs related to changing when a task is executed (CWTE), sacrificing functionality (SF), sacrificing quality (SQ), and financing a task’s execution (FET). These inputs are explained in the following subsections.

6.2.4.3 Changing when a task is executed (CWTE)

CWTE represents the effect of changing when a task is executed. For example, the cost of delaying the generation of a financial report. The cost of CWTE is determined from both initial configurations and data that is collected from end users. The initial configurations are done by assigning a cost value on a five-point scale based on Equation 6.9 (a system administrator does this assignment). Moreover, end-users can

provide their feedback to express to what extent the task's delayed execution has affected them.

6.2.4.4 Sacrificing functionality (SF)

SF means that some functionality is not available during the execution of a task. A piece of functionality represents the ability to execute a task in a specific way (e.g., by disabling options or passing values to the task's parameters). Functionality can be sacrificed by cancelling the execution of a task or by substituting a task with a similar one. The total cost of the sacrificed functionalities (SF) is defined as the sum of the costs of all the sacrificed functionalities (if any). Examples of software functionality that could be sacrificed include product recommendations in an online retail system and automated product identification via computer vision by robots that are preparing parcels for delivery (used to double-check the packed products).

The cost of sacrificing functionality is determined through end-users feedback, whereby end-users are asked to rate to what extent they were negatively affected when part of the functionality was sacrificed due to limited resources. End-users provide ratings on a five-point scale (Equation 6.9).

6.2.4.5 Sacrificing quality (SQ)

SQ means quality is reduced during the execution of a task. Multiple types of qualities could be sacrificed by performing an adaptation. The total cost of the sacrificed qualities (SQ) is defined as the sum of the costs of all the sacrificed qualities (if any). Examples of software-related qualities that could be sacrificed include the amount of data a user is allowed to view and the resolution of an image or video. Furthermore, some qualities that are not software related could also be affected in particular domains. For example, in food manufacturing, performing an adaptation like resource substitution could reduce the nutritional value (amount of nutrients) of a food product or the quality of its packaging.

In the cases where sacrificed quality was due to sacrificed functionality (refer to Section 6.2.4.4), the cost shall be added to both the cost of SF and SQ. For example, packing items randomly in a box rather than by type means that functionality is sacrificed and the quality of the item's layout in the box is also negatively affected.

Table 6.3 – Cost of adaptation calculation example

Type of Adaptation	Cost Function Input Values				Total
	CWTE	SF	SQ	FET	
$C_{\text{Execute similar task (task variant)}}$	0	1	2	1	5
$C_{\text{Perform resource substitution}}$	0	2	2	1	6
$C_{\text{Execute alternative task (delay task)}}$	2	0	0	5	7
$C_{\text{Cancel task execution}}$	3	2	2	3	10
Minimum Cost					5
Selected Adaptation Type: Execute similar task (task variant)					

The costs of sacrificing qualities are computed as a weighted average from both setup data (initial configurations) and data that is collected through end-user feedback. The setup data involves assigning a cost value on a five-point scale (Equation 6.9) to each type of quality. Consider the packaging of a product, where the safest packaging has no cost (scale value = 1) because there is no reduction in quality. The cost of sacrificing quality increases when the safety of the packaging is reduced. Hence, the two values are inversely proportional. Moreover, end-users can express to what extent the sacrifice in quality affected them. For example, if they deemed the packaging of a product to be acceptable they could give it a rating of 1 or 2, otherwise, they would give it a higher rating value to indicate that the cost was high.

6.2.4.6 Financing a task's execution (FET)

FET is a monetary value converted to a rating scale and represents the financial cost of performing an adaptation type. For example, the substitution of a malfunctioning cheap robot for an expensive one adds extra financial costs for an automated warehouse. The reason is that all robots will have to be replaced at some point and it costs more to replace the expensive robot. The cost of financing a task's execution is determined from setup data. The configuration is set for each task as a monetary value, which is then converted to a rating on a scale ranging from 1 to 5 (Equation 6.9). The conversion is applied based on min-max normalisation (Grus, 2015), where the set of values $[a, b]$ represents the scale range between 1 and 5, and $[\min, \max]$ represents the minimum and maximum financial values of the tasks respectively.

6.2.5 Adaptation type selector: example

After the prioritisation of tasks, the *adaptation type selector* component selects the least costly adaptation type that can be applied to a task in case an adaptation is required to execute a task. As mentioned in Section 6.2.4, SPARK supports four types of adaptation. The cost of applying each type of adaptation is computed using the cost function shown in Equation 6.10. Moreover, the cost function relies on four inputs, which are elicited either from a system administrator, end-users, or both.

Consider the input values in Table 6.3, where the cost function for each type of adaptation is computed for a task based on the values of the four inputs. Among the four types of adaptation, the first type (i.e., executing a similar task) is the least costly one for the task in this example. This means in case the task needs to be adapted, then a similar task (task variant) will be executed. The cost function values are stored in the knowledge base as part of the adaptation plan.

6.3 Reactive adaptation components

In addition to the proactive adaptation components explained in Section 6.2, SPARK also consists of reactive adaptation components. These components, which are illustrated in Figure 6.2, include a *resource type state monitor*, *resource type state analyser*, *task execution manager*, *task execution allocator*, *adaptation executor*, *feedback elicitor*, and *feedback provider*. The following subsections explain each reactive adaptation component used in SPARK.

6.3.1 Resource type state monitor

Resources are monitored by the *resource type state monitor* component, which observes the use of resources by tasks. Information related to resource consumption is used by SPARK to identify if there are sufficient resources for the tasks to execute or to perform adaptation when resources are unavailable or have reached a low quantity. The historical usage of resource types is illustrated under “Usage Logs” in Figure 6.1 as a class related to logging resource type usage. The *ResourceTypeUsageLog* shows which task used a resource type, as well as the total usage of the resource type. Moreover, it uses different measurements to monitor depletable and reusable resource types due to the differences between them.

SPARK monitors depletable resources based on their quantity of hand (QOH), critical stock level (CSL), replenishment delay (RD) and critical replenishment delay (CRD). When a resource reaches its critical stock level, an order is placed to acquire new resources from a supplier. A critical stock level is calculated taking into consideration a maximum consumption of a resource and its delivery lead time, as well as replenishment and critical replenishment delays. The critical replenishment delay is used to determine when the adaptation process should start so that there are still enough resources to be used for executing high-priority tasks.

Additionally, SPARK monitors reusable resource types based on the average execution duration (AED) of tasks that consume these resources. In order to reduce the overhead of monitoring reusable resources, SPARK monitors a sample of the most frequently executed high-priority tasks that use all resource types, rather than monitoring every task in the system. The average execution duration of a task is measured by the average between the sum of the execution duration of the task and the number of times the task is executed.

Moreover, other ways have been considered for monitoring reusable resource types based on (i) resource usage conditions (e.g., robot usage > 90%), (ii) percentage of malfunctioning resources (e.g., 30% of robots are malfunctioning), or (iii) task response time. For (i), resources should not remain idle while tasks are waiting to be executed. For (ii), the remaining functioning resources could be enough to execute the tasks. For (iii), acceptable response time differs among tasks; for example, the acceptable response time of an automated order preparation task is different than that of a task that involves searching for a product in a database. Hence, AED was used as explained above.

6.3.2 Resource type state analyser

The *resource type state analyser* component identifies whether a resource type is facing variability based on notifications from the *resource type state monitor* (refer to Section 6.3.1). A depletable resource type faces variability when its quantity on hand (QOH) is below its critical stock level (CSL) and its replenishment delay (RD) is higher than its critical replenishment delay (CRD). A reusable resource type is facing variability when its use by a task causes an increase in the average execution duration (AED) of this task. Moreover, the *resource type state analyser* informs the *task executor manager* component about resource types that are facing variability.

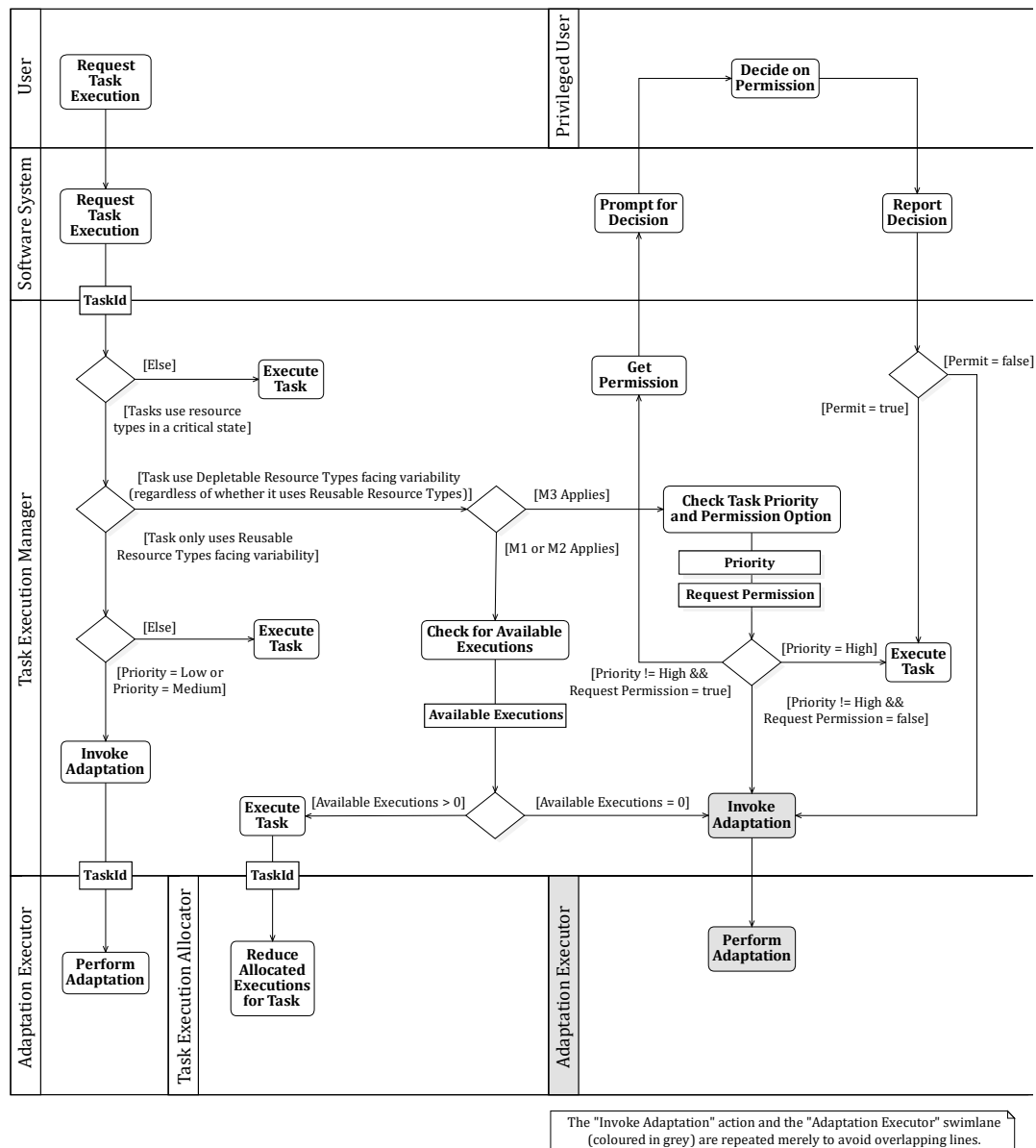


Figure 6.3 – Activity diagram of the decision-logic for task execution

6.3.3 Task execution manager

Tasks are executed with the support of the *task execution manager* and *task execution allocator* components. The *resource type state analyser* informs the *task executor manager* about resource types that are facing variability. The *task execution manager* decides whether to execute a task or to invoke an adaptation. The *task execution manager* also informs the *task execution allocator* about changes to the states of the resource types to allocate a number of execution permissions to a task.

Additionally, Figure 6.3 shows the activity diagram for the decision-making process of the *task execution manager*. As shown in the diagram, in case a task needs reusable resource types which are facing variability, the decision of whether to execute the task or perform an adaptation type depends on the task's priority. If the task has a low or a medium priority and the resource type is facing variability, then adaptation is performed. Otherwise, the task is executed directly.

Moreover, in case a task uses depletable resources which are facing variability, the decision of whether to execute the task or to perform an adaptation depends on the task's consumption of the depletable resources (regardless of whether the task also uses reusable resource types). There are two types of resource consumption fixed and variable. For example, the task "Decorate Box" uses one decorative bow and is an example of a task with fixed resource consumption. On the other hand, the number of boxes (resources) consumed by the task "Pack Items in a Box" depends on the dimensions of the items being packed and is an example of a task with variable resource consumption.

In case a task has a fixed consumption of depletable resources, the task execution manager executes the task as requested if the number of executions that were allocated to it by the task execution allocator is greater than zero. Otherwise, the task execution manager invokes an adaptation by calling the Adaptation Executor. In case a task has a variable resource consumption, the task execution manager executes the task if it has a high priority. If the task has a low or medium priority, the task execution manager checks the setup data to see if the task's execution requires the permission of a user. If it does not require permission, then adaptation is performed. Otherwise, permission is requested, and the task is executed if permission is given otherwise adaptation is performed.

Moreover, adaptations are performed by the *adaptation executor* component, using the appropriate types of adaptation as determined by the *adaptation type selector* and specified in the adaptation plans during the proactive process.

6.3.4 Task execution allocator

The *task execution allocator* specifies the number of times in which tasks that use depletable resource types can be executed, based on a task's forecasted number of task executions and the available resources. The allocation of task executions is based on

Method M1:

1. Let $\{T\}$ be the set of all tasks
2. Order $\{T\}$ by task priority (highest first)
3. Loop around the tasks in $\{T\}$ from highest to lowest priority
4. Determine the depletable resources used by each task
5. Allocate to each task as many of its forecasted number of executions (NATE) as the available resources it requires can support
6. Reduce the available resource quantities
7. If there are no resources left to allocate additional task executions, then break the loop else continue

Figure 6.4 – Method M1 steps

three methods: M1, M2, and M3. Methods M1 and M2 are applied to tasks that use, per execution, fixed quantities of depletable resources. Method M3 is applied to tasks that use, per execution, variable quantities of depletable resources, where these quantities cannot be determined beforehand.

M1 allocates as many as possible of the number of forecasted task executions by order of task priority until the resources cannot support further task executions. This is done by determining for each task (from highest to lowest priority) the depletable resources needed. M1 stops executing when no depletable resources are left to allocate task executions. The steps of M1 are presented in Figure 6.4.

M2 allocates a percentage of the number of forecasted task executions by order of task priority until the resource cannot support further allocations. This is done by assigning to each task (from highest to lowest priority) a counter C (between 1 and n) and a percentage of forecasted executions to be allocated (PFEA). Additionally, a number of allocated task executions (NATE) is computed for the high-priority tasks first and, if possible, the remaining non-high-priority tasks. The steps of M2 and the equations for calculating PFEA and NATE are presented in Figure 6.5.

M3 only executes high-priority tasks on a first-come-first-serve basis. Hence, with M1 and M2, in case of available resources, tasks with medium or low priorities may be allocated resources and executed; while with M3, only high-priority tasks are executed given that resources required by a task cannot be determined beforehand. In M3, tasks that do not have a high priority could be executed if permission is given by a user.

Method M2:

1. Let $\{T\}$ be the set of all tasks
2. Order $\{T\}$ by task priority (highest first)
3. Loop around the tasks in $\{T\}$ from highest to lowest priority
4. Assign to each task a number C that represents a counter from 1 to n , where $n_1 = 1, n_2 = 2, \dots, n_k = k$.
5. Loop around the tasks in $\{T\}$ from highest to lowest priority
6. Assign to each task T a number that represents the percentage of forecasted executions to be allocated (PFEA), which is calculated as follows where $|T|$ is the total number of tasks:

$$PFEA_T = 1 - \frac{C_T}{|T|}, \text{ where } \frac{C_T}{|T|} \in (0,1]$$

7. Let $\{ST\} \subset \{T\}$, where $\text{Priority}(T) = \text{High}$
8. Loop around the tasks in $\{ST\}$ highest to lowest priority
9. Allocate to each task T a number of executions (NATE), which is calculated as follows (decimal values are rounded), where FTE is the forecasted task executions:

$$NATE_T = \lceil PFEA_T \times FTE_T \rceil$$

10. Repeat the loop at Line 8 from the beginning while there are sufficient resources left to allocate task executions
11. If there are resources left to allocate tasks executions, then Let $\{ST\} \subset \{T\}$, where $\text{Priority}(T) \neq \text{High}$
12. Repeat Lines 8 to 10 (allocate executions to the medium and low priority tasks based on the remaining resources)

Figure 6.5 – Method M2 steps

The decision to apply method M1 or method M2 for tasks that use fixed depletable resources is done by system administrators for the entire system, based on their domain knowledge of the system. M2 is more adequate for situations in which it is preferable to spread the execution of tasks with high priorities. For example, in the case in which high-priority tasks are financially critical and it is important to ensure that a percentage of all these tasks are executed such as spreading the purchase of shares in top companies. On the other hand, M1 is more useful for situations in which tasks should be allocated based purely on their priority, even if fewer tasks are executed. For example, when tasks involve medical operations that are more life-threatening than others.

Table 6.4 – Example data showing how M1 and M2 calculate NATE

			M1	M2				
Task	Priority	FTE	NATE	Counter	PFEA	Iterations		NATE
						#1	#2	
T1	1.625	30	30	1	80%	24	6	30
T2	1.75	20	20	2	60%	12	8	20
T3	1.875	10	10	3	40%	4	4	8
T4	2.33	7	0	4	20%	1	1	2
T5	2.66	5	0	5	0%	0	0	0

**Assuming, for this example, that all tasks use, per execution, 1 unit of depletable resource type of which there are 60 units available*

6.3.5 Task execution allocator: example

This section shows an example of how the *task execution allocator* works. This example illustrates how the computations corresponding to methods M1 and M2 are performed to obtain the number of executions that shall be allocated to a task. Based on the activity diagram shown in Figure 6.3, when a depletable resource type is facing variability, the *task execution manager* component checks which method (i.e., M1, M2, or M3) is used by the *task execution allocator* component. If M1 or M2 is selected, then a set of steps, shown in Figure 6.4 and Figure 6.5 for M1 and M2 respectively, are applied to calculate the number of allocated task executions (NATE).

To show the difference between methods M1 and M2, consider the data shown in Table 6.4. In this example, tasks T1 to T5 use a depletable resource type with an available quantity of 60 units. For simplicity, assume that each task uses one unit of the depletable resource type per execution. Each task has a forecasted number of executions (FTE). Tasks are sorted in ascending order based on their adjusted priorities (P_A). M1 specifies NATE based on FTE and the available units of a depletable resource type. M2 specifies a counter and computes PFEA for each task. Multiple iterations are applied for allocating the number of task executions based on FTE and the available depletable resource type units. NATE is then computed by summing up for each task the values in each iteration. As shown in Table 6.4, for M1, tasks T4 and T5 were not executed, while task T4 was allocated some executions for M2.

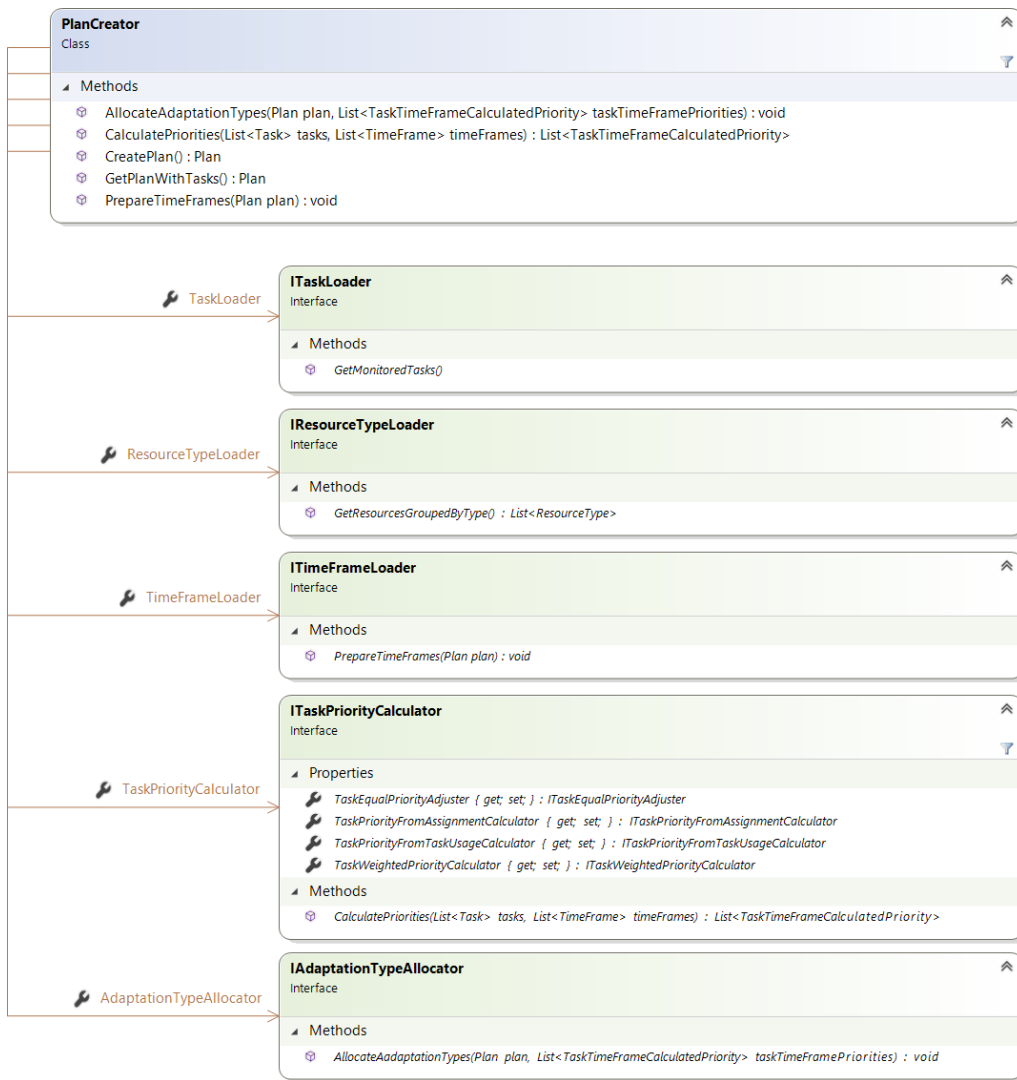


Figure 6.6 – Plan creation

6.3.6 Feedback elicitor and provider

SPARK's feedback elicitor and feedback provider components elicit and provide feedback, respectively, from and to users about the performed adaptations. The feedback elicited from end-users indicates their opinions about the actions taken by the framework and it is used to improve the adaptation process in terms of choice of the types of adaptation for similar future situations. This improvement is done by the adaptation type selector component, which uses the feedback to recompute the cost of applying a type of adaptation.

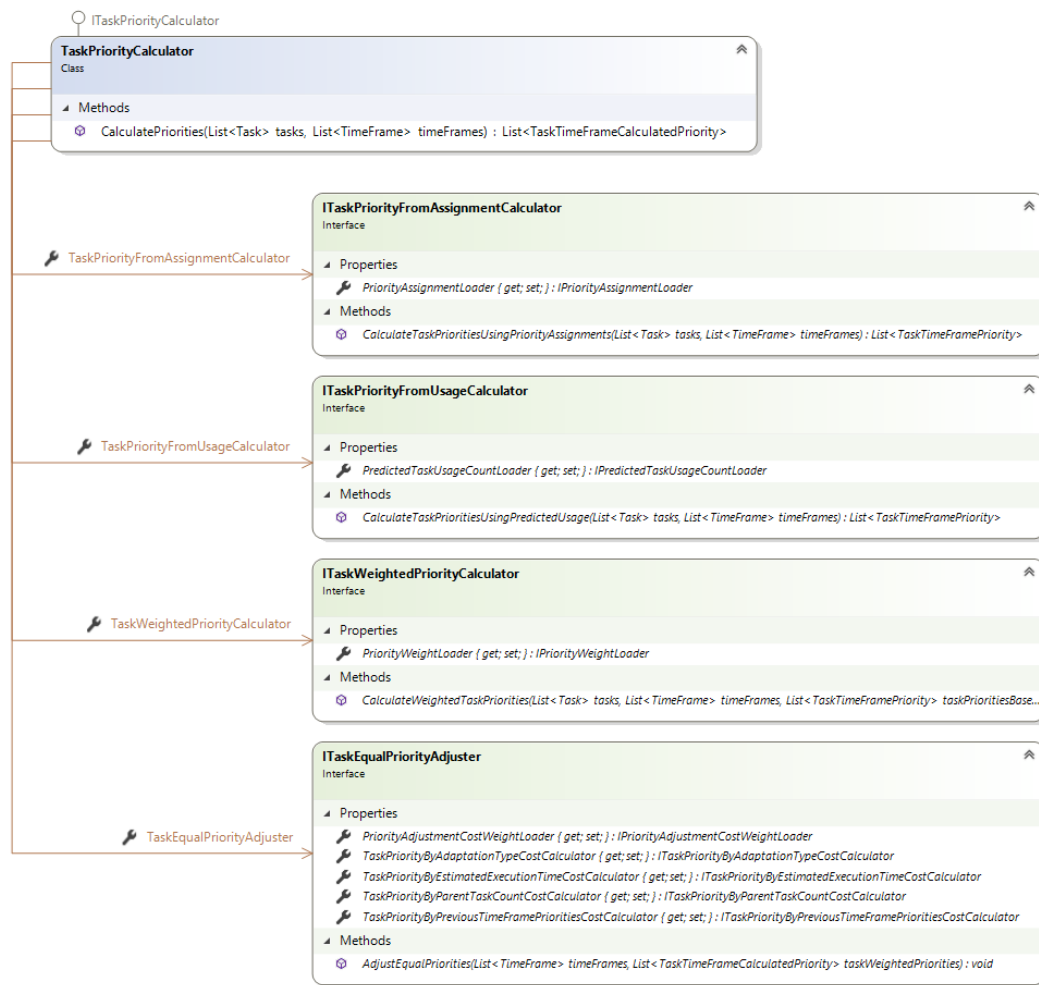


Figure 6.7 – Task priority calculation overview

Moreover, feedback provided to end-users informs them about the reasons for adaptation. This keeps the users in the loop of what is happening to avoid situations where the software system performs changes that the end users do not understand (e.g., the reduction in video quality should not be perceived by the end users as a problem with the system when it is done to address resource variability).

6.4 Implementation of adaptation components

The adaptation components of SPARK were implemented as a prototype using C#. This section provides an overview of SPARK's implementation and presents example class diagrams and source code.

Listing 6.1 – Priority calculation source code (excerpt)

```

1.      CalculateTaskPrioritiesUsingPriorityAssignments(tasks, timeFrame)
2.      {
3.  POL O(n)    var priorityAssignments ← PriorityAssignmentLoader.
4.      .      GetTaskPriorityAssignments(timeFrame);
5.  POL O(n)    return priorityAssignments.Select(p =>
6.      .      new TaskTimeFramePriority {
7.      .      Task = p.Task,
8.      .      TimeFrame = timeFrame,
9.      .      Priority = p.Priority }).ToList();
10.     }
11.
12.     CalculateTaskPrioritiesUsingPredictedUsage(tasks, timeFrame)
13.     {
14.  POL O(n)    predictedTaskUsagesCount ← PredictedTaskUsageCountLoader.
15.     .      LoadPredictedTaskUsageCount(tasks, timeFrame);
16.  CON O(1)    var counter ← 0, lastUsage ← -1;
17.  POL O(n)    foreach (var usageCount in predictedTaskUsagesCount)
18.     .      {
19.     .      if (usageCount.Usage != lastUsage)
20.     .      counter++;
21.     .      usageCount.SequenceNumber ← counter;
22.     .      }
23.  CON O(1)    var minSequenceNumber ← 1, numberOfPriorities ← 3;
24.  CON O(1)    var maxSequenceNumber ← counter;
25.  CON O(1)    var threshold ← (minSequenceNumber +
26.     .      maxSequenceNumber) / numberOfPriorities;
27.  CON O(1)    List<TaskTimeFramePriority> calculatedTaskPriorities ← new();
28.  POL O(n)    foreach (var _ in predictedTaskUsagesCount)
29.     .      {
30.  CON O(1)    TaskTimeFramePriority taskPriority = new() {
31.     .      Task = _.Task,
32.     .      TimeFrame = timeFrame,
33.     .      PredictedUsage = _.Usage };
34.
35.  CON O(1)    if (_.SequenceNumber < threshold)
36.     .      taskPriority.Priority ← Priority.High;
37.  CON O(1)    else if (_.SequenceNumber > threshold * 2)
38.     .      taskPriority.Priority ← Priority.Low;
39.  CON O(1)    else
40.     .      taskPriority.Priority ← Priority.Medium;
41.
42.  CON O(1)    calculatedTaskPriorities.Add(taskPriority);
43.     }
44.  CON O(1)    return calculatedTaskPriorities;
45.     }

```

The class *PlanCreator* shown in Figure 6.6 is responsible for preparing adaptation plans that are followed to perform adaptation during resource variability. When

SPARK creates an adaptation plan it stores it in the knowledge base (refer to Section 6.1), which is implemented as a SQL Server database.

Figure 6.6 shows five interfaces that represent the dependencies of *PlanCreator*. The interfaces *ITaskLoader*, *IResourceTypeLoader*, and *ITimeFrameLoader* are implemented by classes that retrieve tasks, resource types, and time frames respectively to be used as input data for the preparation of an adaptation plan. The tasks are assigned priorities for each time frame during the planning and the resource types are associated with the tasks to plan the choices of adaptation types (e.g., resource substitution).

As shown in Figure 6.7, the interface *ITaskPriorityCalculator* is implemented by the class *TaskPriorityCalculator*, which is responsible for computing the priorities of tasks (refer to Section 6.2.3.2). Figure 6.7 also shows four interfaces that are the dependencies of the *TaskPriorityCalculator*. The interfaces *ITaskPriorityFromAssignmentCalculator* and *ITaskPriorityFromUsageCalculator* are implemented by classes that are responsible for computing priorities based on domain priorities and task usage respectively. The *ITaskWeightedPriorityCalculator* interface is implemented by a class that computes a weighted priority by combining the abovementioned domain and task usage priorities (refer to Section 6.2.3.2). The *ITaskEqualPriorityAdjuster* is implemented by a class that adjusts equal priorities to provide each task with a unique priority (refer to Section 6.2.3.4).

An excerpt of the priority calculation algorithm is shown in Listing 6.1. The first method calculates the priorities of the tasks using their assigned domain priorities (refer to Section 6.2.3.2). The second method calculates the priorities of the tasks based on the forecasted task executions. Thresholding is applied to the priorities in the second method to set the priority values within the required range.

Listing 6.1 also shows the running times using the “Big O” notation (Cormen *et al.*, 2009), where POL and CON denote polynomial and constant running times respectively. Based on the “Big O” notation, the running times are $O(2N)$ and $O(3N)$ for the first and second methods respectively. This means that the overall priority calculation has a polynomial running time equal to $O(N)$.

The rest of SPARK’s algorithms also have polynomial running times. This is not just shown only shown by the “Big O” notation but also by a performance and scalability evaluation that is reported later in this thesis.

6.5 Chapter summary

This chapter presented a framework called SPARK, which this thesis proposes for supporting resource-driven adaptation to address resource variability. SPARK is based on MAPE-K and consists of proactive and reactive adaptation components. SPARK uses task models, represented using SERIES, as input for making adaptation decisions.

This chapter started by introducing the proactive adaptation components that include the task execution monitor, task execution forecaster, task priority calculator, task priority adjuster, and adaptation type selector. Afterwards, it introduced the reactive adaptation components that include the resource type state monitor, resource type state analyser, task execution manager, task execution allocator, adaptation executor, feedback elicitor, and feedback provider.

Then, this chapter demonstrates SPARK's adaptation components through an example from an automated warehouse system. The example showed how these components perform their calculations. Finally, this chapter explained the prototype implementation of SPARK's adaptation components by showing example class diagrams and an excerpt of source code related to the calculation of task priorities.

7

Task Modelling Notation Evaluation (SERIES)

This chapter presents the evaluation of SERIES, which is the task modelling notation proposed in this thesis for supporting resource-driven adaptation. Figure 7.1 presents an overview of this evaluation. SERIES was assessed based on existing recommendations for designing graphical notations. Additionally, I evaluated SERIES through a study with software practitioners.

7.1 Introduction

As indicated by (Moody, 2009), graphical notations are composed of syntax (form) and semantics (content). Hence, a meta-model represents semantic constructs that are visualised by graphical symbols. Section 5.2 presents the meta-model and graphical symbols of SERIES. Based on Moody's explanation, an example of a semantic construct in SERIES is a "Task Variant". The graphical symbol that represents this semantic construct is a box with its corresponding icon.

SERIES represents task models graphically like most existing task modelling notations such as CTT, HAMSTERS, UsiXML, and Amboss (refer to Table 3.2). Graphical representations favour legibility and understanding of models (Chattratchart and Kuljis, 2002). Task modelling notations propose hierarchical task decomposition, which is graphically visualised in a way that is easier to interpret by software practitioners. The concept of hierarchical representation is rooted in psychology (Sebillotte, 1988) and presents task models in the way that people structure their activities and enables the creation of several levels of abstraction and refinement.

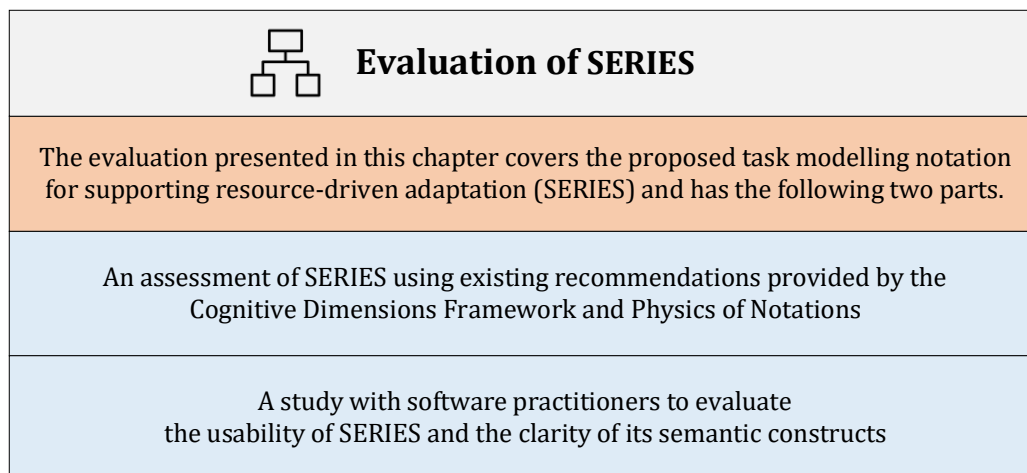


Figure 7.1 – Overview of the evaluation of SERIES

Hence, in SERIES, software practitioners do not have to read dense text to understand how abstract tasks are decomposed into subtasks and how these subtasks are refined into task variants.

I assessed SERIES based on existing recommendations for designing the syntax of graphical notations, which are provided by the Cognitive-Dimensions Framework (Green and Petre, 1996) and the Physics of Notations (Moody, 2009). Additionally, I evaluated SERIES through a study with software practitioners. The participants of this study explained and created SERIES task models and then provided feedback on the usability of SERIES and the clarity of its semantic constructs.

7.2 Assessment of SERIES using existing recommendations

As mentioned in Section 5.5, the visual representation of SERIES is inspired by CTT and UML class diagrams. However, since SERIES introduces new constructs and visual syntax it was assessed based on the recommendations of two paradigms, namely the Cognitive Dimensions Framework and the Physics of Notations. These two paradigms are widely used for assessing visual notations (Genon, Heymans and Amyot, 2011), and are used to assess SERIES because they provide principles that serve as a starting point for evaluating visual notations.

7.2.1 Cognitive Dimensions Framework

The Cognitive Dimensions Framework proposes dimensions that serve as discussion tools for assessing notations (Green and Petre, 1996). What follows is an assessment of SERIES based on these dimensions (their names are marked in bold).

SERIES represents tasks **consistently** using boxes with multiple parts, which are based on the syntax of UML classes. These box parts represent properties that support resource-driven adaptation. The representation of relationships is also consistent. For example, the relationship between a task and its variants is represented by a line with a white triangle on top. This is inspired by the generalisation relationship in UML. Furthermore, the used terms have a consistent meaning throughout a task model. For example, “priorities” and “resource types” have the same meaning when used with different tasks and task variants.

SERIES is **abstraction-tolerant** since it supports the representation of task models using predefined visual elements of tasks, properties, and relationships. It is not possible to add new visual elements. However, new abstractions are defined in the form of parent tasks that comprise default property values, which do not need to be specified for the child tasks.

A **premature commitment** is not needed since a SERIES task model may comprise part of the tasks and property values at design time. Task variants and properties, like roles and priorities, are set at runtime when their corresponding data is available. Furthermore, a SERIES task model is arranged hierarchically to avoid “*visual spaghetti*” that could occur with some box-and-line notations. Even when new tasks are added at a later stage, the model is automatically rearranged without needing to look ahead to avoid a messy layout.

Concerning **diffuseness**, each meaning of a task or property in SERIES is denoted by one box part that has an icon and a description that makes it easy to remember. The ability to add default property values at the parent task level reduces the number of properties in the boxes. This improves the overall visibility of the task model and makes the visual notation terse (compact) enough to represent multiple tasks on the screen. Furthermore, it is possible to suppress a group of task properties by hiding its box parts as is done with UML class diagrams. Hence, the notation supports multiple levels of terseness that are changeable according to how much detail a person wants to see.

SERIES does not have complex conditionals that create **hard mental operations**. Parameter conditions are defined as simple textual statements such as “PackMode = Random”. Hence, these conditions do not use complex line connections that cause software practitioners to resort to tracking what is happening with their fingers.

There are no **hidden dependencies** between the elements of a SERIES task model. The dependencies between tasks are shown as relationships. For example, task variants are linked to their base task using relationships that resemble UML generalisation to indicate that the variants are special cases of a general case. Furthermore, properties like parameter conditions and priorities are shown on the model without hidden formulas (e.g., like the ones in spreadsheets).

Concerning **role-expressiveness**, a task’s name indicates its purpose. It is also possible to add a description that further explains a task’s purpose. This description is a secondary notation and is viewed by clicking on an information icon, which appears on tasks that have a description.

SERIES has a **low viscosity** since little effort is needed to change tasks and properties using the supporting tool’s panels that include task model explorer, visual canvas, and properties box. The users can select a task by simply clicking on its representation in the task model explorer or on the visual canvas. Then, the corresponding properties can be modified by typing or selecting new values in the properties box.

7.2.2 Physics of Notations

The Physics of Notations is a theory that offers principles for visual notation design (Moody, 2009). What follows is an assessment of SERIES based on these principles (their names are marked in bold).

SERIES provides **semiotic clarity** since it has a one-to-one correspondence between each symbol and the concept that it represents. Hence, there is no redundancy because no concept is represented by more than one symbol and there is no overload because no symbol represents more than one concept. In SERIES, a different icon is used for each property (e.g., priorities, roles, parameters, and parameter conditions). Furthermore, icons distinguish the boxes that represent abstract tasks, application tasks, and application task variants. The use of icons to differentiate task types is common practice in other task modelling notations like CTT and HAMSTERS.

SERIES provides **semantic transparency** because its task model nodes are represented hierarchically and connected by lines that look different, thereby allowing users to infer the relationships between tasks, subtasks, and task variants.

The principle of **complexity management** is maintained in SERIES by representing tasks hierarchically with the *abstraction* (abstract tasks) shown at the top of the hierarchy and *decomposition* (subtasks and task variants) shown at the lower levels. A task model is browsable hierarchically in the supporting tool either on the visual canvas or on the task model explorer where nodes are collapsible. The hierarchical display also helps with **perceptual discriminability** since each level of the hierarchy displays task model elements that belong to the same category. The categories include abstract tasks, subtasks, and task variants.

SERIES applies the principle of **dual coding** using text to complement graphics. All the properties in SERIES have an icon and a textual description. For example, the property that represents parameters has the word “Parameters” as a textual description alongside an icon that represents a parameter (an arrow pointing inside a box to denote an input). Additionally, a description that acts as a secondary notation can be used to provide a complementary textual description to tasks and task variants that are represented by a graphical shape with an icon.

SERIES abides by the principle of **graphic economy** since its number of graphical symbols is cognitively manageable. SERIES provides six main symbols that represent abstract tasks, application tasks, substitutable and non-substitutable application task variants, task-to-sub-task relationships, and task-to-variant relationships. Although other symbols represent the properties of tasks and task variants, these symbols just appear on the diagram once the user modifies values in the properties box. Hence, the user is not required to choose and add these symbols from a toolbox as is done with the tasks, task variants, and relationships.

7.2.3 Further evaluation

Sections 7.2.1 and 7.2.2 explained how SERIES considers the recommendations of the Cognitive Dimensions Framework and the Physics of Notations. Nonetheless, further evaluation is required to determine how well software practitioners can use SERIES for task modelling. Such evaluation also determines how software practitioners

perceive SERIES concerning its usability and the clarity of its semantic constructs. Therefore, a study was conducted for this purpose as explained in the next section.

7.3 A study to evaluate SERIES with software practitioners

As mentioned in Section 4.2, software practitioners are responsible for creating task models using SERIES to support resource-driven adaptation in software systems. Hence, I conducted a study with software practitioners to evaluate SERIES. First, this section provides an overview of the participants' background information. Then, it explains the design of the study. Afterwards, this section presents and discusses the results and the threats to validity.

7.3.1 Participants

This study had 20 participants. The number of participants is comparable to other studies that evaluate visual notations (Batra, Hoffler and Bostrom, 1990; Shoval and Shiran, 1997). Furthermore, the participants represent the target population, namely software practitioners. The participants provided some background information about their experience in the software industry and with visual modelling notations. What follows is an overview of this information.

All the participants have experience with software engineering. The majority of them, 16 out of 20, are currently working as software practitioners at software companies in the following countries: Lebanon, the United States, Canada, Denmark, Egypt, Germany, and the Netherlands (Figure 7.3). The remaining participants, 4 out of 20, are currently working as researchers at The Open University in the United Kingdom, but three of them had previous experience in the software industry.

As shown in Figure 7.2, thirteen of the participants have between 1 and 5 years of experience in the software industry. Three participants have 6 to 10 years of experience. Another three participants have less than one year of experience. Only one participant never had any experience in the software industry. However, this participant has some personal experience in developing non-commercial software applications. The participants' collective experience includes the development of software systems for multiple domains including business, education, electronics, games, government, and multimedia. These software systems cover several software paradigms including web, mobile, desktop, and virtual reality.

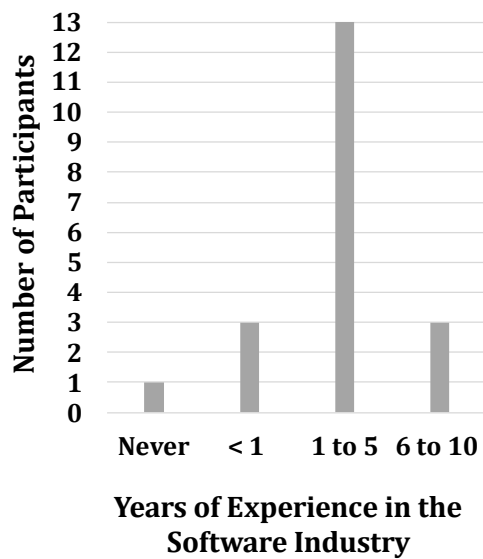


Figure 7.2 – Experience of the participants in the software industry

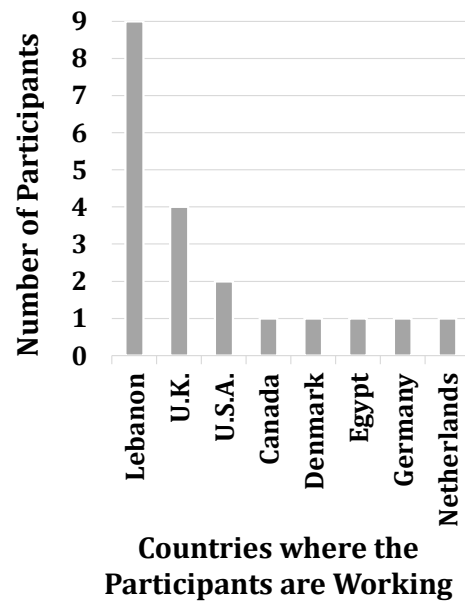


Figure 7.3 – Countries where the participants are working

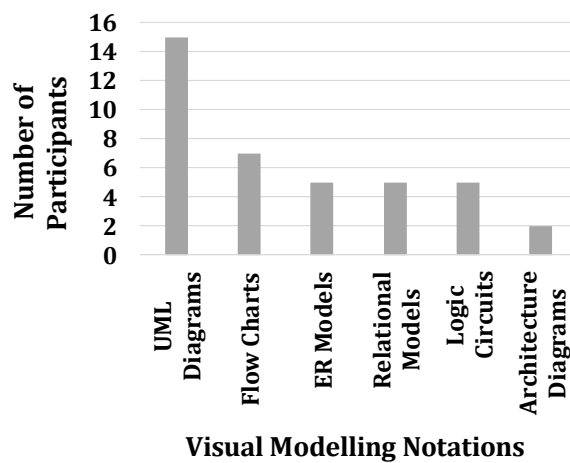


Figure 7.4 – Experience of the participants with visual modelling notations (each participant could list multiple notations)

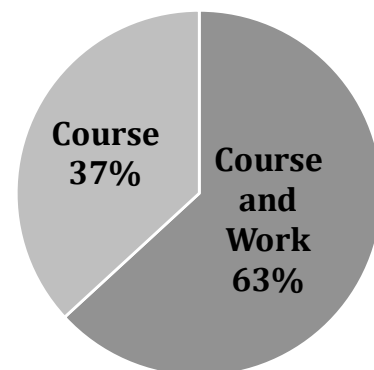


Figure 7.5 – Where the participants used visual modelling notations

All the participants have previous experience in using visual modelling notations. As shown in Figure 7.4, their collective experience includes using UML diagrams, flow charts, ER models, relational models, logic circuits, and architecture diagrams. The participants have also used a variety of modelling tools including StarUML, Draw.io, Cadence, Dia, Jira, Lucid Chart, PgAdmin, and Umlet. As shown in Figure 7.5, thirteen of the participants (63%) have indicated that they have used visual modelling notations both at the work and in a course. The remaining seven participants (37%) have only

Introduction		
Watch a brief tutorial video about SERIES and its supporting tool (7 minutes)		
Use SERIES via its Tool		
Explain task models represented using SERIES	Create task models using SERIES	
Answer Questionnaire		
Provide some background information	Provide feedback on clarity of semantic constructs	Provide feedback on usability

Figure 7.6 – Overview of the study’s activities

used visual modelling notations in a course. The chart shown in Figure 7.4 might not be comprehensive because I asked the participants to recall and list the visual modelling notations that they have used. However, the examples they listed are indicative of their experience.

The participants who are working as software practitioners (16 out of 20) were recruited from the Computer Science graduates (alumni) of Notre Dame University – Louaize (NDU), Lebanon, and the rest of the participants (4 out of 20) were recruited from the Computing researchers at The Open University (OU), United Kingdom.

7.3.2 Design of the study

Each participant took on average 55 minutes to complete this study, which involved the activities that are shown in Figure 7.6. Before beginning the study, I asked the participants to watch a brief seven-minute tutorial video on SERIES and its supporting tool. A tutorial video is used so that participants would receive the same explanation of SERIES, understand the semantic constructs of SERIES, and understand how the semantic constructs can be used via the supporting tool of SERIES. Then, I asked them to explain and create task models using SERIES via its supporting tool because these activities provide the means to check the usability of SERIES from the perspective of the participants. Afterwards, I asked the participants to complete a questionnaire to provide some background information (refer to Section 7.3.1) and offer their feedback on the usability of SERIES and the clarity of its semantic constructs.

Considering that the participants are from and working in different countries, it is important to note that the study was conducted in English. Hence, the participants' comments did not require translation into English from other languages. All the participants are proficient in English. The participants who were recruited from the Computer Science graduates (alumni) of NDU have all received their education in English (like many universities in Lebanon NDU offers all its courses in English). The participants who were recruited from the OU are living in the United Kingdom and are therefore either native speakers or have a working proficiency in English.

7.3.2.1 Hypothesis

The hypothesis for this study is described as follows:

H₁: The use of SERIES will result in good user (software practitioner) performance in the interpretation and creation of task models for resource-driven adaptation.

This hypothesis was assessed based on the correctness of the answers given by the participants when explaining and creating SERIES task models and the feedback that they gave to indicate their perception of the clarity of the semantic constructs. It was also assessed based on the feedback that the participants gave on the usability of SERIES to see whether they found usability issues that hinder their ability to use this notation.

7.3.2.2 Environment and data collection

I conduct the study online due to the Covid-19 pandemic. However, this did not obstruct any of the planned activities. I gave the participants remote control of my computer via Zoom (videoconferencing software program). This way they were able to use the supporting tool of SERIES (refer to Section 5.4). I chose Zoom because it provided the best performance for remote access and videoconferencing with minimal to no lagging over an internet connection. I tested Zoom and compared it to two alternatives, namely Microsoft Teams and TeamViewer.

The participants' verbal input (feedback) and their work on the supporting tool of SERIES were captured using audio and screen recordings respectively. The task models that the participants created were also saved as files from the tool. The participants provided written input using a questionnaire that I presented to them as a Word document.

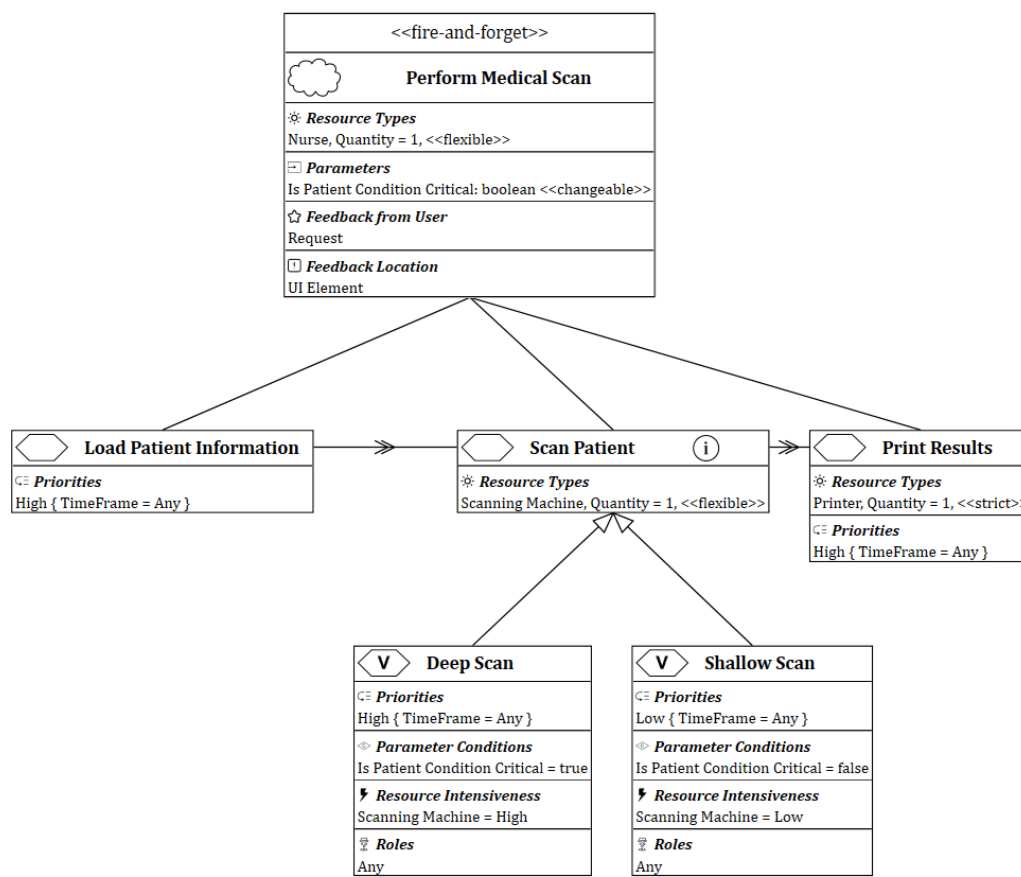


Figure 7.7 – Example: task model that the participants explained (the participants who chose the hospital domain explained this task model)

7.3.2.3 Explaining and creating task models

The explanation and creation of models are recommended activities for studies that evaluate visual notations (Bork and Roelens, 2021). What follows is a description and examples of the SERIES task models that I asked the participants of this study to explain and create.

Multiple domain choices: This study aims to assess SERIES rather than the participants' knowledge of a domain. Hence, I asked the participants to select the domain of the task models that they were required to explain and create. The study included three domain options, namely hospital, manufacturing, and surveillance. Although there were three domain options, there was no discrepancy among them in the difficulty of the task models that the participants were asked to explain and create.

(1) Add an abstract task named "Book Medical Operation". This task has two parameters: an "Operation Type" that is non-changeable and a "Respiratory Mode" that is changeable. It also requires a "General Practitioner (GP)" resource type that is "flexible" with a quantity of 1. For this task, feedback from the user shall be requested and the feedback location shall be a panel.

(2) Add three application tasks as subtasks of "Book Medical Operation". These three subtasks are named "Reserve Operation Material", "Reserve Respiratory Device", and "Reserve Operation Room" respectively.

"Reserve Operation Material" requires a "Material" resource type that is "flexible" with a quantity of 10.

"Reserve Respiratory Device" has the following description: "Reserve a ventilator or oxygen tank for the patient". This task requires a "Respiratory Device" resource type that is "flexible" with a quantity of 1.

"Reserve Operation Room" requires a "Medical Room" resource type that is "strict" with a quantity of 1. This task has a high priority at any time frame.

(3) Add two application task variants for "Reserve Operation Material".

- The first variant is named "Reserve Material for Urgent Operation" and has a high priority and a parameter condition that specifies "Operation Type=Urgent".
- The second variant is named "Reserve Material for Elective Operation" and has a low priority and a parameter condition that specifies "Operation Type=Elective".

Add two application task variants for "Reserve Respiratory Device".

- The first task variant is named "Reserve Ventilator" and has a high priority, a parameter condition that specifies "Respiratory Mode=Ventilator", and a low resource intensiveness for the "Respiratory Device" resource type.
- The second task variant is named "Reserve Oxygen Tank" and has a low priority, a parameter condition that specifies "Respiratory Mode=Oxygen Tank", and a high resource intensiveness for the "Respiratory Device" resource type.
- Both task variants have a "Role" that is equal to "Any".

**Figure 7.8 – Example: requirements for creating a task model
(the participants who chose the hospital domain used these requirements)**

Hence, the task models from the three domains had the same number and types of tasks and task variants. Additionally, each task and task variant had the same number and types of properties as its counterparts from the other domains. This way, each participant could choose the domain that they prefer while maintaining the same level of difficulty for all the participants.

Multiple levels of difficulty: The task models that the participants were expected to explain and create were presented to them with three levels of increasing difficulty, where level 1 is basic and level 3 is advanced. The complexity of the task model

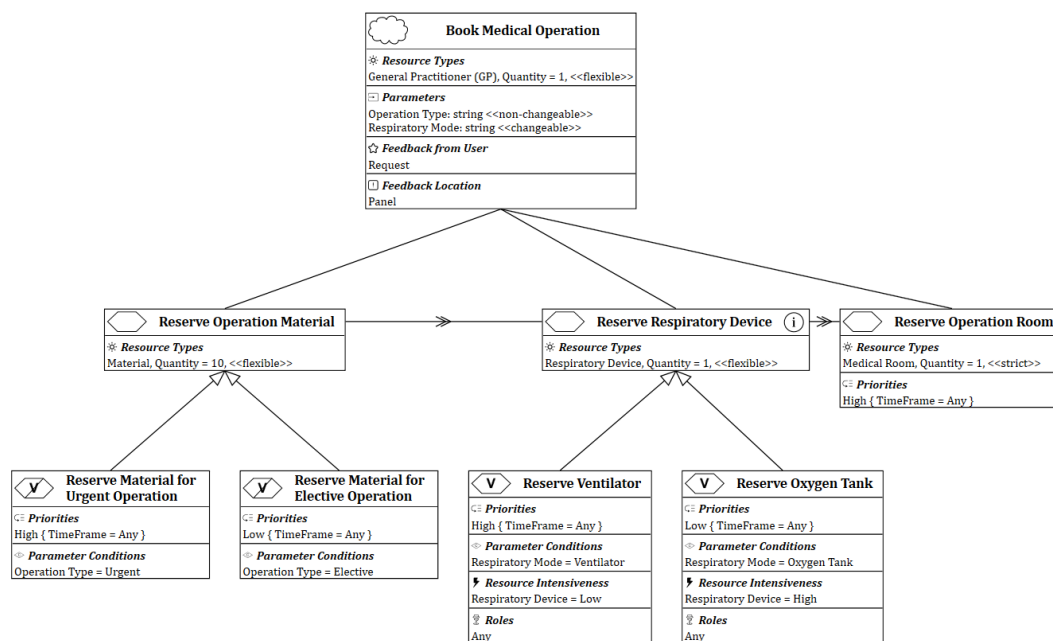


Figure 7.9 – Example: task model that the participants created (the participants who chose the hospital domain were expected to create this task model from the corresponding requirements)

hierarchy and the number of elements increased at each level. At level 1, the task model contains an abstract task with four properties including resource type, parameter, feedback-from-user, and feedback-location. At level 2, the abstract task is divided into three application subtasks that have priorities and resource types as properties. At level 3, task variants are added with properties that include priority, parameter conditions, resource intensiveness, and roles. These levels show whether the participants would face difficulties in performing task modelling with SERIES when the complexity of the task model hierarchy increases.

Examples of task models and requirements from the study: Figure 7.7 shows an example task model that I asked the participants to explain. I gave this task model to the participants who chose the hospital domain. The text shown in Figure 7.8 is an example of the requirements given to the participants to create a task model. I gave these requirements to the participants who chose the hospital domain. Figure 7.9 shows the task model that the participants were expected to produce from the requirements shown in Figure 7.8. The task models and requirements corresponding to the other two domains (manufacturing and surveillance) are shown in Appendix A.

Process of explanation and creation of task models: The task models that I asked the participants to explain were presented to them using the SERIES notation within its

supporting tool. I asked them to go through the task model elements and explain their meanings. For creating task models, the participants were presented with written requirements that were displayed in a panel within the supporting tool of SERIES. I asked them to read these requirements and create the corresponding task models using the tool. I asked the participants to think aloud when they are creating the task models. The think-aloud protocol is commonly practised during studies where participants are required to report what they are thinking while performing an activity (Oh and Wildemuth, 2009). In this study, the activity is task modelling using SERIES and its supporting tool. For example, this could show whether the participants are confused about how they should associate a requirement with its corresponding semantic construct(s) or whether they find it difficult to perform an action using the tool.

7.3.2.4 Feedback questionnaire

After the participants explained and created SERIES task models, they were asked to provide their feedback by completing the questionnaire shown in Appendix A.

The participants answered questions to indicate their perception of how well the semantic constructs of SERIES convey a clear meaning that enables a user (software practitioner) to explain and create task models using this notation. I also asked them to recommend changes if necessary. These questions complement the explanation and creation of task models (refer to Section 7.3.2.3) because even if the participants can explain and create SERIES task models, they could still have some feedback on possibilities for further improvement.

The participants also answered questions that convey their perception of the usability of SERIES. I asked them to answer a set of five questions about the overall ease of use of SERIES and give any additional comments that they may have. The five questions on ease of use have been proposed in the literature and used in other studies that evaluate modelling notations (Davis, 1985; Batra, Hoffler and Bostrom, 1990). The answers given by the participants to these questions are used to compute a rating between one and five, where one is worst and five is best.

Then, the participants selected three Product Reaction Cards (PRCs) that they thought were most suitable for describing SERIES. The PRCs were developed by Microsoft to understand the aspect of desirability corresponding to a user's experience with a product (Benedek and Miner, 2002). I asked the participants to choose from a set of 16 PRCs (8 positive PRCs and 8 negative ones). The selection was not restricted to

either positive or negative. Hence, the participants were able to select any three PRCs, all positive, all negative, or a mixture of both. Additionally, the PRCs were not labelled as “positive” and “negative” to allow the participants to provide their interpretations. Hence, after choosing the three PRCs, I asked the participants to explain their choice and suggest improvements if necessary. Finally, I asked the participants if they had any remaining comments before concluding the study.

7.3.3 Processing and presenting the data

The data from the study should be processed and presented before being analysed. The quantitative data from ratings given on a scale are directly presented using box plots alongside descriptive statistics (mean, median, and standard deviation). Similarly, the selected PRCs are counted and presented using a bar chart. However, the outcomes of the participants’ explanation and creation of task models are scored first. The scores represent how well the participants were able to explain and create task models using SERIES. Furthermore, the qualitative data from the participants’ comments are categorised so they can be analysed. Hence, this section explains how the outcomes of the participants’ explanation and creation of task models are scored. It also explains how qualitative comments are categorised and presented anonymously.

7.3.3.1 Scoring participants’ explanations and created task models

I scored the participants’ explanations of the task models to see if they correctly interpreted the meanings of the semantic constructs. Similarly, I scored the task models that the participants created to see if they used the semantic constructs to correctly represent the given requirements.

An answer key with a three-level scoring scheme (incorrect, partially correct, and correct) is used. A score is computed for each of the explained and created task models at the three levels of difficulty (refer to Section 7.3.2.3). An answer is incorrect if it does not have any correct part or if it is completely missing because a participant did not know how to answer. If part of the answer is correct, then it is partially correct. On the other hand, correct answers do not have any incorrect or missing parts. The scores that are awarded to incorrect, partially correct, and correct answers are 0, 0.5, and 1 respectively. The grading scheme covers the semantic constructs of SERIES used in the task models. This way, each task model is divided into parts that are scored separately to

identify if the participants faced difficulty in a specific part. The task model parts are scored on the abovementioned three-level scoring scheme in the form of a rubric, where the rows are the task model parts and the columns are scoring options (incorrect, partially correct, and correct). A total score for a task model is computed over 100 from the scores of all the parts, which are given equal weights.

An example from the grading scheme for explaining a task model at level 3 includes being able to explain a *relationship between a task and a task variant*, two *priority properties*, two *parameter-condition properties*, two *resource-intensiveness properties*, and two *role properties*. In this case, when there are two properties, an answer is considered partially correct if one of these properties is explained correctly. Furthermore, for example, in the case of the priority properties, a correct answer covers both the priority value and the corresponding time frame. On the other hand, for the explanation of the *relationship between a task and a task variant* the answer is either correct or incorrect (i.e., there is no partial credit).

An example from the grading scheme for creating a task model at level 1 includes being able to add elements corresponding to the requirements that include an *abstract task*, a *resource type* (with quantity and substitutability), *two parameters* (with data type and parameter type), and *two feedback properties* (feedback-from-user and feedback-location). In this case, the abstract task is either correct or incorrect (i.e., there is no partial credit). On the other hand, the resource type property is considered correct if both its quantity and substitutability are added correctly. If only one of them is correct then, the resource type property is considered partially correct. The two parameters are considered correct if both of them are added with their correct data types and parameter types. Otherwise, if only one of the two parameters was added correctly then the answer would be partially correct. The same applies to the feedback properties. For correcting the task models created by the participants, the tasks, task variants, relationships, and properties are compared one by one to a predefined task model that represents the correct answer.

7.3.3.2 Quoting the participants and classifying their comments

All the participants of this study have consented to be quoted anonymously. Hence, upon presenting the results the participants are quoted by using a reference number (e.g., P1) as a pseudonym next to the corresponding comment. The comments are italicised and placed between quotes.

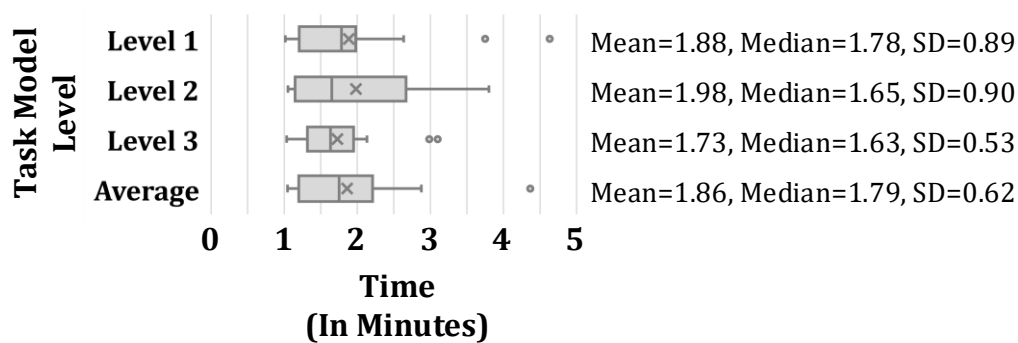


Figure 7.10 – Time taken by the participants to explain the task models
(an “x” on the boxplot represents a mean and a circle represents an outlier)

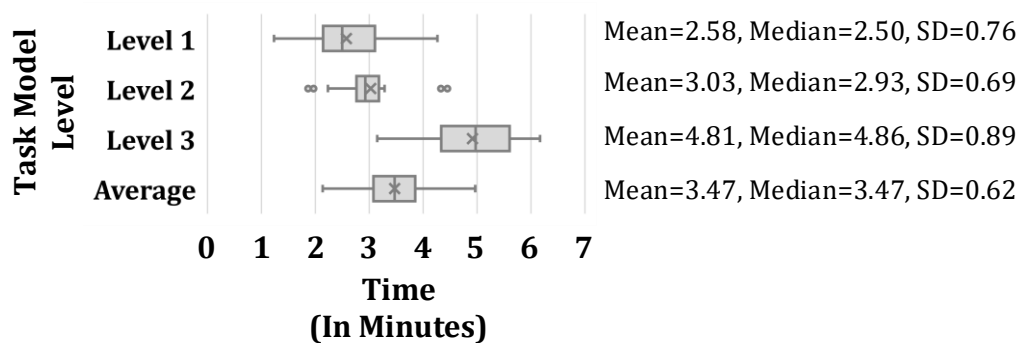


Figure 7.11 – Time taken by the participants to create the task models
(an “x” on the boxplot represents a mean and a circle represents an outlier)

Furthermore, the comments given by the participants represent qualitative data that complement the quantitative data that is obtained from the ratings given on a scale and the scores of the explanation and creation of task models. Categorisation is common in studies that involve qualitative data (Khalid *et al.*, 2015; Neale, 2016). Hence, I categorised the comments given by the participants of this study to identify what they perceive as strengths of SERIES and what they suggest for potential improvement. I also classified the comments under broad themes that provide a general overview of what the participants said about their perception of SERIES. These themes were deduced from the abovementioned categories. I computed the percentage of comments in each theme.

7.3.4 Results of participants’ explanation and creation of task models

As explained in Section 7.3.2.3, the participants were able to choose a domain that they prefer from three choices including hospital, surveillance, and manufacturing. All three domains were selected by some participants. The hospital, surveillance, and

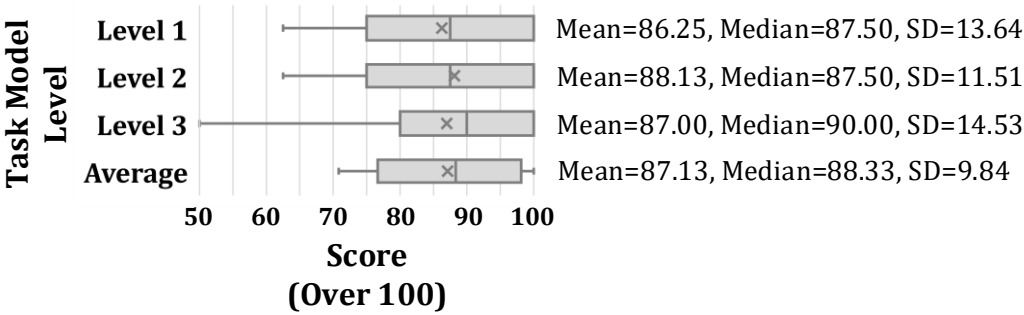


Figure 7.12 – Scores on the participants’ explanation of task models (an “x” on the boxplot represents a mean and a circle represents an outlier)

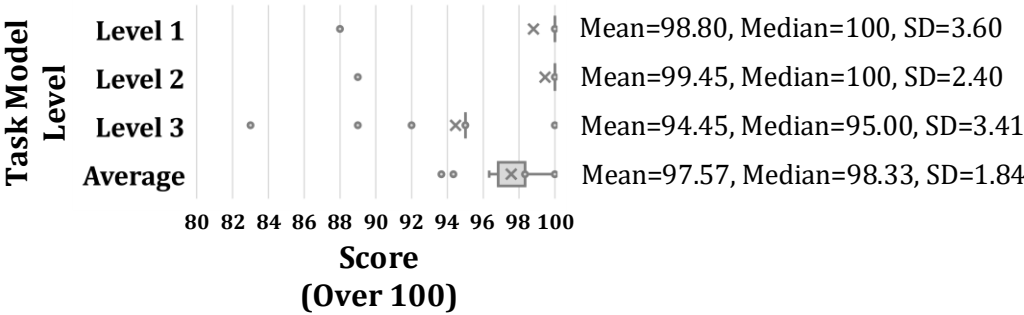


Figure 7.13 – Scores on the participants’ creation of task models (an “x” on the boxplot represents a mean and a circle represents an outlier)

manufacturing domains were selected by 50%, 30%, and 20% of the participants respectively. Each participant explained and created task models from the domain that they selected. The explanations and created task models were scored as explained in Section 7.3.3.1. The results are as follows.

7.3.4.1 Results of the explanation of task models

As Figure 7.10 shows, the participants took on average 1.86 minutes to explain the task models for each one of the three levels. The participants averaged close scores on the explanation of all three task model levels. As Figure 7.12 shows, their scores (over 100) on the explanation were on average 86.25, 88.13, and 87, for levels 1, 2, and 3 respectively, and their average score across all three levels was 87.13. These results show that the participants exhibited very good performance when explaining task models that are represented using SERIES. The close scores on the three task model levels indicate that the participant did not face difficulty when the complexity of the task model hierarchy increased. The average score on level 1 was slightly lower than

the scores on the other two levels, which are more complex. A reason for this could be that the purpose of the abstract task from level 1 became clearer once sub-tasks were shown at level 2. Several participants mentioned this point when they moved from the first level to the second one.

Considering the overall explanation scores and that the mistakes were not focused on a particular semantic construct, it is possible to say that there is no major ambiguity in the meaning of a particular semantic construct in SERIES. Examples of the participants' explanation mistakes include explaining a task without discussing its execution type, explaining a parameter property without discussing its parameter kind, and explaining a priority property without discussing its corresponding timeframe. Although there is room for improving the explanations to reach perfect scores, it is important to mind that the participants were only given a brief tutorial about SERIES and that they never had any previous experience with this notation.

7.3.4.2 Results of the creating of task models

As Figure 7.11 shows, the participants took on average 3.47 minutes to create the task models that correspond to the given requirements for each one of the three levels. The participants averaged close scores on the creation of task models for all three levels. As Figure 7.13 shows, their scores (over 100) were on average 98.80, 99.45, and 94.45, for levels 1, 2, and 3 respectively, and the average score across all three levels was 97.57. These results show that the participants exhibited excellent performance when creating task models using SERIES via its supporting tool. The mean score on level 3 was slightly lower than the scores of the other two levels. Nonetheless, it is still very high (94.45). There were some outlier scores as shown in Figure 7.13. However, these scores were mostly ≥ 88 . The mistakes were overall minor. For example, some participants did not specify the "non-substitutable" property on a pair of the task variants or specified the "role" property on one of the task variants but forgot to specify it on the other variant. Another example of a mistake is giving a parameter the wrong name (typo). These mistakes are mostly due to lapses that do not indicate the existence of any major difficulties in using SERIES to create task models.

7.3.4.3 Result comparison

As explained above, both the explanation and the creation of task models yielded high scores. However, the mean scores for the creation of task models (Figure 7.13) were higher than the mean scores for the explanation of task models (Figure 7.12). This could be due to the participants performing the explanation of the task models first. Hence, this gave them some additional exposure to examples of SERIES task models before they created task models using this notation.

As mentioned in Section 7.3.1, the level of experience of this study's participants is diverse. I compared the scores of the participants to see whether the level of experience affected their ability to explain and create task models.

Concerning the explanation of task models, the four participants with little or no experience (< 1 year - never) averaged a score of 85 across the three task model levels. The thirteen participants with medium experience (1 to 5 years) averaged a close score of 86.02. The three participants with high experience (6 to 10 years) averaged a score of 94.72 across the three task model levels. Although the average score of the participants with the most experience is higher, the other participants still averaged high scores (≥ 85) considering it's the first time they work with SERIES.

Concerning the creation of task models, there was little to no difference among the scores for participants of all levels of experience. This is expected considering that the overall scores are very high. The four participants with little or no experience (< 1 year - never) averaged a score of 97.33 across the three task model levels. The thirteen participants with medium experience (1 to 5 years) averaged the same score (97.33). On the other hand, the participants with high experience (6 to 10 years) averaged a score of 98.88 across the three task model levels. Since the scores are overall very close, the level of experience did not affect the ability of the participants to create task models using SERIES.

7.3.5 Results of participants' feedback on clarity of semantic constructs

The participants rated the clarity of the semantic constructs of SERIES on a scale that ranged between 1 and 5, where 1 is the worst and 5 is the best. Figure 7.14 shows these ratings. We can see that the mean values for the semantic constructs ranged between 4.5 and 5.0. These results indicate that the participants considered the semantic constructs of SERIES to convey a clear meaning that enables software practitioners to explain and

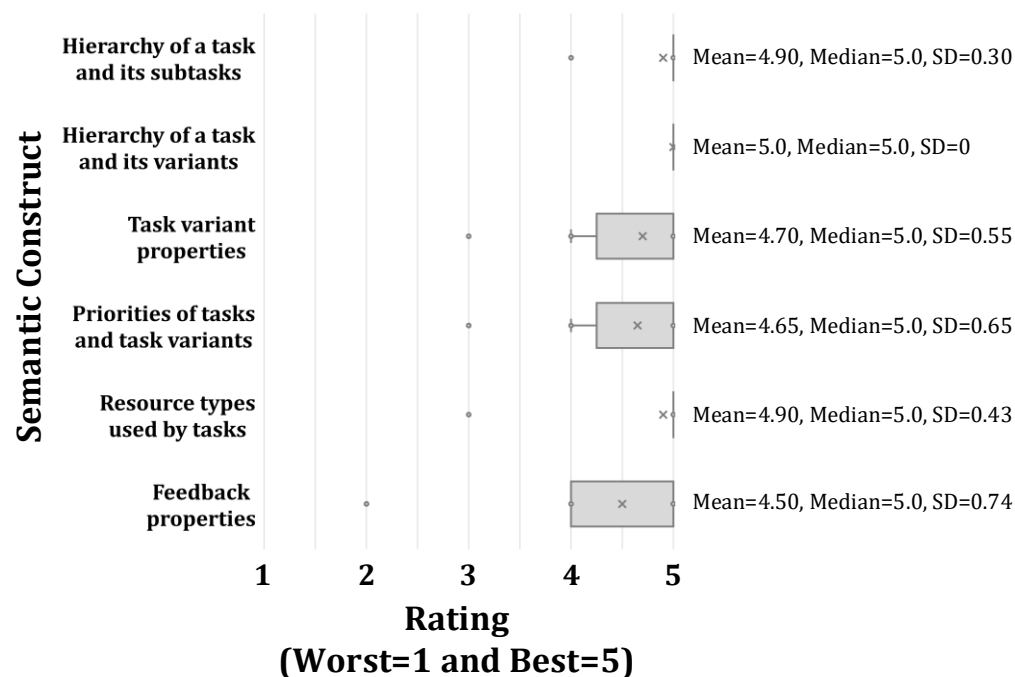
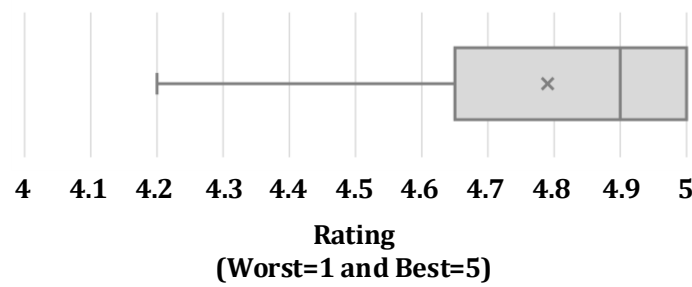


Figure 7.14 – Participants’ feedback on the semantic constructs of SERIES – this feedback represents the participants’ perception of whether the semantic constructs convey a clear meaning for explaining and creating SERIES task models (an “x” on the boxplot represents a mean and a circle represents an outlier)

create task models. As Figure 7.14 shows, four of the semantic constructs had one outlier with a rating of three or two over five. In this case, the participants felt that it would be useful to have some more clarification about the respective constructs. However, these values do not affect the participants’ overall positive perception of the clarity of the semantic constructs as shown by the mean ratings that are ≥ 4.5 . Some participants gave additional comments regarding the clarity of the semantic constructs of SERIES. These comments were classified under four categories as explained next. The names of the categories are marked in bold.

The first category included comments that said **the given tutorial is sufficient to clarify the meanings of the semantic constructs**. The tutorial referred to in these comments is the brief video given to the participants at the beginning of the study (refer to Figure 7.6). In this regard, P10 said, “*The constructs are clear once we see the tutorial.*” and P20 said, “*The brief tutorial was enough for me to understand the semantic constructs. I do not think that any changes are needed to these constructs.*” These comments indicate that software practitioners could learn SERIES without significant time and effort. P8 noted the same thing and elaborated further by saying, “*Notation is easy to use after a quick tutorial. Terms mean the same throughout (e.g., resources for tasks and subtasks).*”



Mean = 4.79, Median = 4.90, SD = 0.28

Figure 7.15 – Participants' feedback on the ease of use of SERIES
(the "x" on the boxplot represents the mean)

Output diagram is easily readable, you can directly notice if the design matches the requirements." The idea presented by P8 on the terms meaning the same thing throughout the model is due to SERIES adhering to the dimension of *consistency* from the Cognitive Dimensions Framework as discussed in Section 7.2.1. It is also due to SERIES adhering to the principle of *semiotic clarity* from the Physics of Notations as discussed in Section 7.2.2.

The second category of comments involved a **minor clarification**. In this regard, P12 said, *"Everything was mostly clear, maybe a bit more information on the priority times."* P14 also mentioned priorities and timeframes by saying *"At the beginning, the timeframe and its relation with the priority was a bit confusing."* P12 and P14 mentioned this point because the brief tutorial that was given to the participants at the beginning of the study only demonstrated the association of priorities with a specific time frame (e.g., 8:00 AM to 12:00 PM). However, during the study, the participants were asked to set the priorities for any time frame. Nonetheless, even with the brief coverage of this concept, the participants' explanation, use, and feedback demonstrated that they know how to work with priorities and timeframes. Hence, this point did not cause a major issue overall. For example, both P12 and P14 rated the corresponding semantic constructs (priorities of tasks and task variants) with a 4 over 5 (i.e., "clear") because the concepts got clarified after some reflection. Nonetheless, in a longer tutorial, more examples could be given to explain the relationship between priorities and timeframes.

One comment involved a **minor change suggestion**, whereby P17 suggested having *"colour coding for priorities"*. The implementation of this suggestion is simple and could be useful to attract attention to the high-priority tasks and task variants. For example,

the priority property could be presented in red colour when its value is high. This should not visually saturate SERIES since colours are not used in other properties.

One comment emphasised that the **task hierarchy is clear**. In this comment, P12 said, *“This is actually one of the things that I liked about SERIES. It is the ability to create the subtasks very clearly and to define how they follow each one after the other.”* This indicates that the visual presentation of the hierarchy helped P12 to understand the meaning of the relationship between a task and its subtasks and among subtasks and relates to the principles of *semantic transparency* and *perceptual discriminability* from the Physics of Notations as explained in Section 7.2.2.

7.3.6 Results of participants’ feedback on usability

This section reports on the results of the participant feedback on the usability of SERIES. These results include the participants’ ratings of ease of use, selected PRCs, and additional comments that justify their answers and provide suggestions.

7.3.6.1 Ratings for ease of use

The participants were given five questions on ease of use that covered difficulty, clarity, understandability, frustration, and mental effort. The result reported in Figure 7.15 is computed from the means of the answers that the participants gave to all five questions. The mean rating given by the participants on the ease of use questions is 4.79 over 5 (1 is the worst and 5 is the best). This result indicates that the participants considered SERIES to be overall very easy to use. The participants gave some additional comments on the ease of use of SERIES and its supporting tool. These comments are classified under three categories as explained next. The names of the categories are marked in bold.

P14 mentioned how the three-level SERIES **task hierarchy is clear**. In this regard, P14 said *“The three-level hierarchy is very clear and helpful in analysing and formalising a complex task. The visual representation of the task, sub-tasks, and variants of the sub-tasks is very well organized and lets the user have every information at hand.”* This comment by P14 complements the comment that P12 gave on the clarity of the task hierarchy (refer to Section 7.3.5) and indicates that both the meaning and visual representation of the task hierarchy are clear.

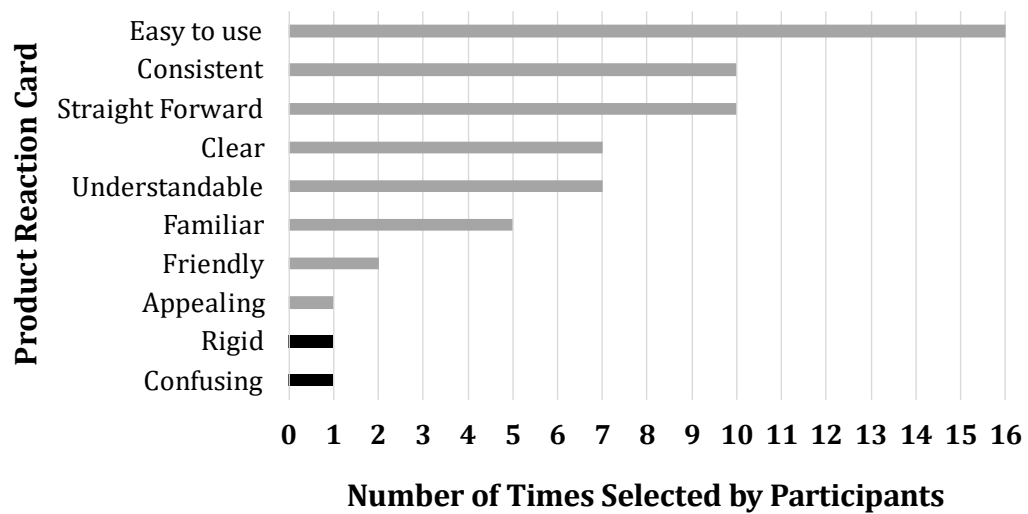


Figure 7.16 – PRCs selected by the participants to describe SERIES – each participant was asked to select three PRCs (positive PRCs are shown in grey while the negative ones are shown in black)

Some comments explained why the **UI of the tool is usable**. P7 said “*The UI is very user-friendly. All functionality is easily visible.*” Additionally, P14 said, “*The user interface lets the user reach every window in an easy and organized manner – i.e., the user does not need to scroll menus to find and fill in the necessary information.*” These two comments reflect a positive perception of the supporting tool’s usability, which complements the usability of the notation and facilitates task modelling using SERIES.

There was one **minor change suggestion**. P8 suggested adding a “*small UI*” under the selection boxes of resources, priorities, and parameters so these properties can be added quickly without having to open a new (popup) window. P11 also preferred having a quicker way of setting these properties without having to open a new window but did not suggest a specific solution. It is worth noting that existing software development tools such as Visual Studio open new windows, in many cases, for setting properties from a properties box. Nonetheless, there could be an alternative. However, the addition of “*small UIs*” as P8 suggested could over-clutter the properties box if these UIs are always visible. Hence, one possibility could be to use dropdown UIs that the user opens via the selection boxes or buttons without navigating to a new window.

7.3.6.2 Selected product reaction cards

The participants were each asked to select three PRCs that they thought best describe SERIES. As Figure 7.16 shows, the PRCs that the participants selected were mostly positive (58 out of 60). The selected PRCs complement the results reported in Section 7.3.6.1, which indicates that the participants have a very positive perception of the usability of SERIES. The participants justified their choice of PRCs by giving additional comments. These comments are categorised under the corresponding PRCs. In some cases, the participants gave a single comment to justify their choice of multiple PRCs. Such comments are only listed under one of the corresponding PRCs. The PRCs are listed from the most to the least selected. The PRCs and the names of the categories of comments are marked in bold.

1) Easy to Use: The PRC “easy to use” was the most selected by the participants. The comments that the participants gave for choosing this PRC are classified under three categories as follows.

The first category includes comments that said **the given tutorial is sufficient or even unnecessary to use SERIES easily**. In this regard, P11 gave a combined justification for selecting “easy to use” and “understandable” by saying *“The overall system was simple, and most of the controls were instinctive to use. I felt that I could’ve figured out the system without even having the tutorial, as the names and types were generally self-explanatory.”* P14 thought SERIES was easy to use because a *“short introduction was enough to be able to easily work with it.”* These two comments by P11 and P14 complement what participants P8, P10, and P20 said on the tutorial being sufficient to clarify the meanings of the semantic constructs (refer to Section 7.3.5).

The second category includes comments that complement the abovementioned (first) category by saying that **SERIES notation is understood quickly**. In this regard, P18 said, *“I got accustomed to it quite fast.”* and P19 said, *“SERIES was very easy to use. I did not find any difficulty understanding the notation.”* P2 provided a combined comment for choosing “straightforward” and “easy to use” by saying SERIES is *“straightforward and easy to use because there are not many concepts to deal with meaning that there are 3 main tasks (abstract task, application task, and application task variant).”* These comments alongside those from the abovementioned (first) category indicate a positive perception of the learnability of SERIES. Furthermore, the comment given by P2 on SERIES not having too many concepts is related to how

SERIES adheres to the principle of *graphic economy* from the Physics of Notations as explained in Section 7.2.2.

The third category includes comments that said **the SERIES notation and the UI of its supporting tool have a well-designed layout**, which improves usability. P16 commended the layout of SERIES and the friendly UI and particularly liked how tasks are automatically connected. In this regard, P16 said, *“I found SERIES really easy to use since everything is laid out in front of me. Simple operations and a friendly interface. The particular thing that I found that’s really helpful is how the tasks are automatically connected without the need to draw lines and to choose specific relationships.”* When tasks are added to the task model they are automatically connected and the task model is rearranged to avoid “visual spaghetti” in SERIES (refer to Section 7.2.1 – dimension about *premature commitment* from the Cognitive Dimensions Framework). P17 said, *“everything is very to the point and isn’t complicated.”* P3, P5, and P13 noted that the UI is usable. In this regard, P3 commented on the layout and labelling of the UI by saying, *“The interface is very simple and well divided, everything is well labelled to properly represent what is needed.”* Additionally, P5 mentioned the simplicity of the UI by saying *“The UI is clean and simple to use.”* Furthermore, P13 noted how the UI made it easy to use SERIES via its supporting tool by saying, *“It is easy to use and clear because of the UI. It doesn’t take time to understand how to use the software.”*

2) Consistent / Straightforward: The PRCs “consistent” and “straightforward” are equally the second most selected by participants after “easy to use”. The comments related to these two PRCs are classified under two categories as explained next.

The first category includes comments that said **SERIES task model elements and information are consistently represented**. P14 commented on the consistency of the visual representation at the three levels of the task model hierarchy by saying, *“Each of the three-level-hierarchy parts uses similar panels to fill in the required pieces of information. Furthermore, the representation of the information in the main window – where the task model is represented – is consistent for each part, making it easily interpretable.”* Similarly, P18 considered that the visual elements constituting the different task types are consistent and said, *“All elements of the different task types are at the same place, making it easy to expect where the properties, resource types, etc... fall.”* P5 considered that there is consistency in the wording and the design of the notation by saying, *“SERIES uses the same wording and design elements across the diagram. It is easy to understand what elements mean across tasks/variants.”* The

consistency in SERIES, which is discussed in the abovementioned comments is a result of taking into consideration the dimension of *consistency* from the Cognitive Dimensions Framework (refer to Section 7.2.1).

The second category includes comments saying said the **UI provides a consistent and effortless way to work with tasks and properties**. Concerning consistency, P9 said, *“The UI was consistent throughout when creating either abstract tasks, application tasks, or application task variants.”* Additionally, P17 commented on the consistency of the design by saying *“The whole application follows the same design for entering new tasks and data.”* Furthermore, P15 mentioned the similarity in the way all tasks are created by saying *“creation of all types of tasks is similar to one another (name, description, properties...)”*. P12 justified choosing “straightforward” by giving a comment that complements that of P15, whereby P12 said, *“It was very easy to use with the UI, adding, removing and editing tasks and their resources, parameters, etc... was straightforward and didn’t require any effort.”*

3) Clear / Understandable: The PRCs “clear” and “understandable” are equally the third most selected by the participants. The comments corresponding to these two PRCs are categorised under the following three categories.

The comments in the first category noted that the **task hierarchy is clear**. P4 considered that the task organisation provides clarity and makes the use of SERIES straightforward. In this regard, P4 said, *“The different types of tasks were ordered hierarchically in a way that is straightforward and clear.”* Additionally, P20 considered that the task hierarchy is organised clearly by saying *“The hierarchy of tasks is organised in a clear way that makes it easy to navigate through the model and understand relationships among the tasks.”* These two comments by P4 and P20 complement what P12 said about the clarity of the task hierarchy of SERIES as mentioned in Section 7.3.5 about the clarify of the semantic constructs. Additionally, these comments reflect how the hierarchical representation of task models in SERIES abides by the principle of *semantic transparency* from the Physics of Notations explained in Section 7.2.2.

The comments in the second category pointed out that there is **clear labelling and understandable information**. P4 and P19 commented on the clarity of the labelling whereby P4 said, *“The panels are clear, the labels are enough to understand the fields needed to be filled.”* and P19 said, *“Everything was clear and labelled out.”* The labelling that is part of the notation is a design choice that adheres to the principle of dual

coding in the Physics of Notations that recommends using text to complement graphics (refer to Section 7.2.2). Additionally, P14 and P15 noted that the information presented on the screen is understandable, whereby P14 said, *“All the information is well summarized and represented in the main window in an understandable way.”* and P15 said, *“There is a help me button, but everything is already self-explanatory.”* The terms “understandable” and “self-explanatory” indicate that P14 and P15 perceive SERIES to be intuitive, which means that it does not need a lot of effort and explanation to understand. P2 mentioned this point explicitly by saying *“I find it very intuitive.”*

Two comments mentioned that the **UI is clean and not cluttered**. P18 justified choosing “understandable” by saying, *“I got accustomed to it quite fast (same reason for choosing easy to use), in addition to the fact that the UI is not cluttered, it’s friendly”*. Additionally, P5 said, *“The clean UI helps in understanding where all the buttons are. There is enough space for the diagram.”* In this comment, P5 is saying that it is easy to locate functionality on the UI, which is important for working without hassle. P5 is also saying that the space (section of the UI), that is allocated for displaying and managing the task model is enough for the user to work comfortably.

4) Familiar: This PRC was the fourth most selected by the participants (considering that two pairs of PRCs shared an equal number of selections). The comments on this PRC are classified under two categories as explained below.

As mentioned in Section 5.5, the visual representation of SERIES is inspired by UML class diagrams (e.g., tasks are represented as boxes like classes in UML). The benefit of this design choice is reflected in comments that said there is **familiarity due to the visual resemblance between SERIES and UML**. P2 said that SERIES is *“familiar because it is similar to UML.”* Additionally, P9 said, *“SERIES was familiar to use, as it builds on what the user has already been exposed to in the past, particularly UML.”* Furthermore, P5 said, *“The diagram uses simple design elements similar to UML which makes it familiar to use.”* P20 also elaborated further on this point by saying, *“Certain elements of the task model’s visual representation resemble UML class diagrams. Tasks are represented as boxes like classes and task variants are related to tasks using an arrow that resembles generalization relationships.”* Although the purpose of SERIES is different than that of UML class diagrams, the visual resemblance in parts of the notation has created familiarity that helps software practitioners in learning SERIES as noted in the abovementioned comments. For example, in UML class diagrams the main

semantic construct, namely a class, is represented as a box. The same is done in SERIES but the main semantic construct is a task rather than a class.

Other comments stated that there is **familiarity due to the resemblance between the UI of the SERIES tool and the UIs of other tools**. P2 said, *“The tool resembles other tools making it very easy to learn.”* Additionally, P13 said, *“The UI of the tool is familiar because it resembles existing IDEs.”* This was also a design choice because the panels of the tool were designed to resemble panels that are common in existing integrated development environments (IDEs). For example, the “Task Model Explorer” and the “Properties” panels in the supporting tool of SERIES resemble the solution explorer and properties box respectively in the Visual Studio IDE.

5) Appealing / Friendly: P12 and P7 justified selecting “appealing” and “friendly” respectively, by saying that the tool is helpful, and the functionality is easily visible on the UI. P12 considered the tool to be appealing because it fits its purpose and said, *“I found that this tool would be very helpful to use to model such tasks which made it very appealing to me.”* Additionally, P7 positively commented on the usability of the UI by saying, *“The UI is very user friendly. All functionality is easily visible.”*

6) Confusing / Rigid: Very few negative PRCs were selected, namely “rigid” and “confusing” were each selected once as shown in Figure 7.16. It is worth noting that participants P6 and P11, who chose these two PRCs, also chose two positive PRCs including “easy to use” and “consistent” for P6 and “easy to use” and “understandable” for P11. Furthermore, negative PRCs only constituted 3.33% (2 out of 60) of the total selected PRCs, whereas positive PRCs constituted 96.66% (58 out of 60). The participants who selected these two PRCs explained their choices by giving additional comments. Overall the reasons for their choices are minor and could be addressed as explained below.

P6 chose “confusing” because some clarification is required for two terms, namely “changeable” and “strict”, which are used to annotate parameters and resource types respectively. P6 said, *“The reason why I chose confusing is due to the usage of some not-so straightforward terms (changeable and strict).”* Although P6 found these two terms to be confusing the overall results of the explanation and creation of task models and the participants’ feedback are positive. Hence, it is possible to consider this to be a minor issue that can be resolved with some further explanation of the meanings of these terms and possibly additional examples.

P11 chose “rigid” and said, *“The system did feel relatively rigid, as there were a very limited number of things I could do at any given time.”* P11 explained further by saying that adding an application task requires selecting an abstract task and then clicking on “add application task”. Hence, the same actions have to be repeated for adding more application tasks. P11 preferred being able to add multiple application tasks without having to re-click on the abstract task. The supporting tool of SERIES works this way because it is shifting the focus to the newly added application task rather than keeping it on the abstract task. The focus is shifted so users would directly edit the properties of a newly added application task without having to select it. A possible solution for P11’s request could be to enable the addition of multiple application tasks with one click. This just requires a minor adjustment to the supporting tool of SERIES.

7.3.7 Results: participants’ final comments

A few participants gave comments at the end of the study. Some were just general positive observations. For example, P3 said, *“An interesting tool with a lot of potential applications.”* Other comments were more specific. P7 noted that **SERIES task models are clear even when the task hierarchy has multiple levels**. In this regard, P7 said, *“The diagram remains very clear even when the relationships between the tasks become deeper.”* This indicates that for P7 the increase in the level of difficulty, as explained in Section 7.3.2.3, did not make it harder to understand and work with SERIES task models. P7 also compared SERIES task models to UML diagrams by saying that **SERIES task models are easier to manage than UML diagrams**. In this regard, P7 said, *“In UML diagrams when the hierarchy starts getting deep everything gets convoluted making it harder to see the relationships. Here it is nice how the diagram organizes itself.”* Hence, P7 found the auto-arrange feature that is offered by the supporting tool of SERIES to be useful. This feature automatically adjusts the layout of the task hierarchy when new tasks or task variants are added to avoid the “visual spaghetti” that affects some box-and-line notations (refer to Section 7.2.1 on the Cognitive Dimensions Framework).

P5 suggested a **minor change** to the tool concerning default values and said, *“It would be nice to have good defaults where it makes sense. For example, if I don’t specify a ‘Role’ property it could be assumed to be ‘Any’.”* The tool is currently a prototype which is why it does not have a default values settings feature. Nonetheless, this feature could be added as a dynamic settings window that enables the specification of default values for properties such as role, resource quantity, and so on.

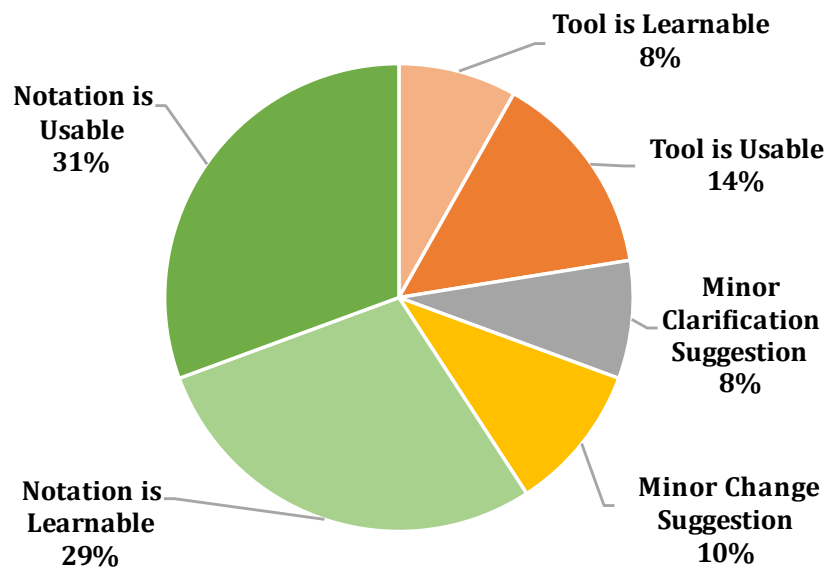


Figure 7.17 – Percentage of comments in each theme

7.3.8 Discussion of the results

Concerning the results from quantitative data, the participants exhibited very good performance in the explanation and creation of task models using SERIES as indicated by the scores that were reported in Section 7.3.4. The mean scores over 100 were 87.13 for the explanation and 97.57 for the creation of task models. Additionally, the feedback of the participants also showed that they perceived the meanings of the semantic constructs to be clear, where the mean ratings of clarity ranged between 4.5 and 5 over 5, where 1 is the worst and 5 is the best (refer to Section 7.3.5). Furthermore, the feedback from the participants showed that they perceived SERIES to be usable. The mean of the ratings that the participants gave for ease of use was 4.79 over 5, where 1 is the worst and 5 is the best. Additionally, 96.67% of the PRCs that the participants selected to describe SERIES were positive (refer to Section 7.3.6). These results provide a positive indication of the ability of software practitioners to use SERIES.

Concerning the results from qualitative data, as explained in the previous sections, the participants gave comments that expressed their feedback on the clarity of the semantic constructs and the usability of SERIES and its supporting tool. These comments were presented and categorised in Sections 7.3.5, 7.3.6, and 7.3.7. Six themes are defined here to provide a broad overview of the content of these comments. These themes were deduced from the abovementioned comment categories and cover the usability and

learnability of the SERIES notation and its supporting tool and the suggested minor clarifications and changes.

The six themes are “Notation is Usable”, “Tool is Usable”, “Notation is Learnable”, “Tool is Learnable”, “Minor Clarification Suggestions” and “Minor Change Suggestions”. The themes “Notation is Usable” and “Tool is Usable” are related to usability and indicate that the notation and tool can be used by specified users, namely, software practitioners, to achieve specified goals with effectiveness, efficiency, and satisfaction (ISO 9241, 2008). On the other hand, the themes “Notation is Learnable” and “Tool is Learnable” are related to learnability and indicate that users, namely software practitioners, do not take a lot of time to understand how to use the notation and tool (ISO 9241, 2008). The themes “Minor Clarification Suggestions” and “Minor Change Suggestions” are related to minor suggestions given by the participants about clarifying a concept further and adding a basic feature, respectively.

As Figure 7.17 shows, the participants discussed why they perceive the SERIES notation to be usable and learnable in 31% and 29% of their comments respectively. This indicates that the participants have a positive perception of SERIES concerning its syntax (form) and semantics (content), which constitute graphical notations as noted by (Moody, 2009). Additionally, the participants discussed how they perceive the tool to be usable and learnable in 14% and 8% of their comments respectively. Considering that the tool complements the notation, it is interesting to see that the participants also perceive it to be usable and learnable. Furthermore, the participants made suggestions for minor changes and minor clarifications in 10% and 8% and their comments respectively. As previously explained, these suggested changes and clarifications are minor and do not hinder the ability of software practitioners to use SERIES for task modelling. Nonetheless, they will be taken into consideration in the future.

Based on the results of the study, as discussed above, the hypothesis that is defined in Section 7.3.2.1 is accepted because the use of SERIES resulted in very good user (software practitioner) performance in the interpretation and creation of task models for resource-driven adaptation. This was indicated by both the results of the activities that the participants performed using SERIES and the feedback that they provided.

7.3.9 Threats to validity

This study involved 20 participants. This could limit the generalisability of the results. However, these participants accurately represent the group of people who are expected to use SERIES, namely software practitioners. Hence, the participants were able to give insights based on their knowledge and experience. Furthermore, the sample of participants was diverse in terms of the level of experience in the software industry and experience with software modelling notations as explained in Section 7.3.1. This diversity yields feedback from people with different perspectives and capabilities.

A comparison was conducted in Section 7.3.4 among the scores of the participants from different levels of experience. However, it is important to clarify that this study is not intended or designed to be a between-groups comparison study based on level of experience. The latter is not this study's objective and requires a larger number of participants in each group. For example, there were only three participants with six to ten years of experience. Hence, this comparison is only meant to provide a basic idea about possible differences in the ability to explain and create SERIES task models.

7.4 Chapter summary

This chapter presented the evaluation of the proposed task modelling notation for supporting resource-driven adaptation (SERIES). It presented an assessment of SERIES using existing guidelines for designing notations given by the Cognitive Dimensions Framework and Physics of Notations. Moreover, this chapter presented a study with software practitioners to evaluate SERIES. The study was divided into four parts: (i) watching a brief tutorial video about SERIES and its supporting tool; (ii) explaining tasks models that are represented using SERIES; (iii) creating a task model using SERIES via its supporting tool; (iv) completing a questionnaire to provide background information and feedback about the usability of SERIES and the clarity of its semantic constructs. The results of the study showed a very good user (software practitioner) performance in explaining and creating SERIES task models. Moreover, the participants gave positive feedback regarding the usability of SERIES and the clarity of its semantic constructs.

8

Framework for Resource-Driven Adaptation Evaluation (SPARK)

This chapter presents the evaluation of the proposed framework for resource-driven adaptation (SPARK). Figure 8.1 presents an overview of this evaluation. The evaluation of SPARK has two parts, a preliminary evaluation with generated data and two case studies with existing datasets. The metrics that I used in the evaluation include the percentage of executed critical task requests, the average criticality of the executed task requests versus the non-executed ones, overhead, scalability, and the intrusiveness of integrating SPARK into a software system.

8.1 Introduction

I conducted a preliminary evaluation of SPARK by developing a simulation tool for an automated warehouse where robots are responsible for preparing customer orders to be shipped. This tool served as a proof-of-concept prototype for evaluating SPARK's feasibility. Additionally, I evaluated SPARK's overhead and scalability using a varying number of business tasks that are commonly found in enterprise systems (e.g., "view sales report"). The preliminary evaluation used generated data.

Afterwards, I evaluated SPARK in two case studies with existing datasets corresponding to (i) a medicine consumption system and (ii) a manufacturing system. These datasets include multiple tasks and types of resources. The two case studies involved measuring two metrics, namely the percentage of executed critical task requests and the average criticality of the executed task requests versus the non-expected ones during resource variability. The outcomes of these two metrics were


 Evaluation of SPARK
<p>The evaluation presented in this chapter covers the proposed framework for resource-driven adaptation (SPARK) and has the following two parts.</p>
<p>Initial evaluation that involves developing software prototypes for simulating an automated warehouse and task requests</p>
<p>Two case studies with existing datasets corresponding to a medicine consumption system and a manufacturing system</p>

Figure 8.1 – Overview of the evaluation of SPARK

compared when using SPARK’s proactive and reactive adaptation, reactive adaptation only, and no framework. Additionally, the overhead and scalability of SAPRK were measured using a varying number of tasks from the datasets. Furthermore, intrusiveness was evaluated by measuring the lines of code to be added or modified for integrating SPARK into a software system (based on what I proposed in Section 4.5).

8.2 Preliminary evaluation of SPARK

A preliminary evaluation of SPARK’s feasibility, overhead, and scalability was conducted. This section explains the design of this preliminary evaluation and presents its results and threats to validity.

8.2.1 Evaluating feasibility with an automated warehouse simulation

Some types of automated warehouse management systems like Ocado use robots that move on top of a grid to store and pick up items and prepare customer orders for delivery (*Ocado Solutions*, 2018; Mason, 2019). As shown in Figure 8.2, robots can malfunction and cause resource variability until repairs are performed. This delays the fulfilment of customer orders. I developed the software tool shown in Figure 8.4 to simulate a grid-based automated warehouse where robots are responsible for executing requests of the “Prepare Order” task that is shown in Figure 8.3. The settings shown in Figure 8.4a are used to specify the simulation parameters including the

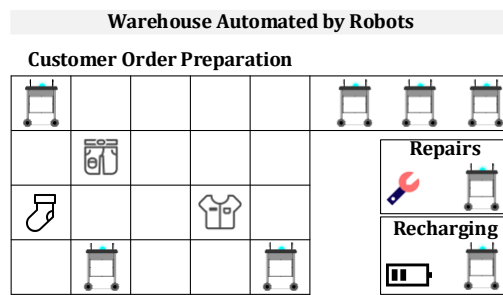


Figure 8.2 – Automated warehouse example (used in preliminary evaluation of SPARK)

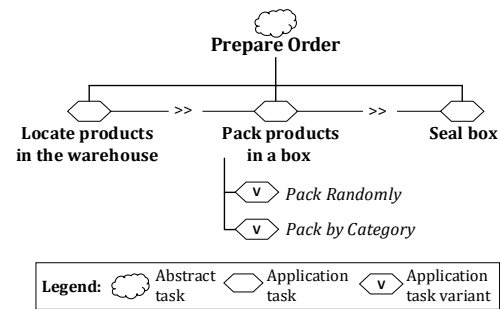


Figure 8.3 – Prepare order task (used in preliminary evaluation of SPARK - shown in summarised form of SERIES)

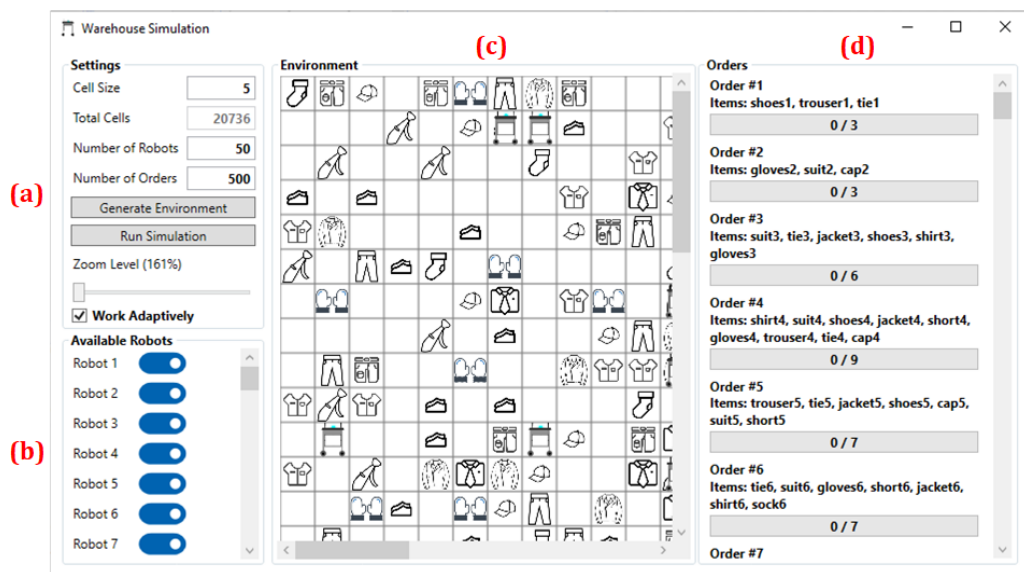


Figure 8.4 – Automated warehouse simulation software (developed for the preliminary evaluation of SPARK)

number of robots and the number of customer orders. Figure 8.4b shows the availability of the simulated robots. The simulated warehouse environment is visualised in Figure 8.4c. The completion progress of the customer orders is shown in Figure 8.4d.

8.2.1.1 Design of the simulation

The robots execute a task called “prepare order”, which has three subtasks “locate products in the warehouse”, “pack products in a box”, and “seal box” (Figure 8.3). The task “pack products in a box” has two variants “pack randomly” and “pack by category”. These two variants have a trade-off between presentation and speed, whereby “pack randomly” is faster but “pack by category” is more elegant.

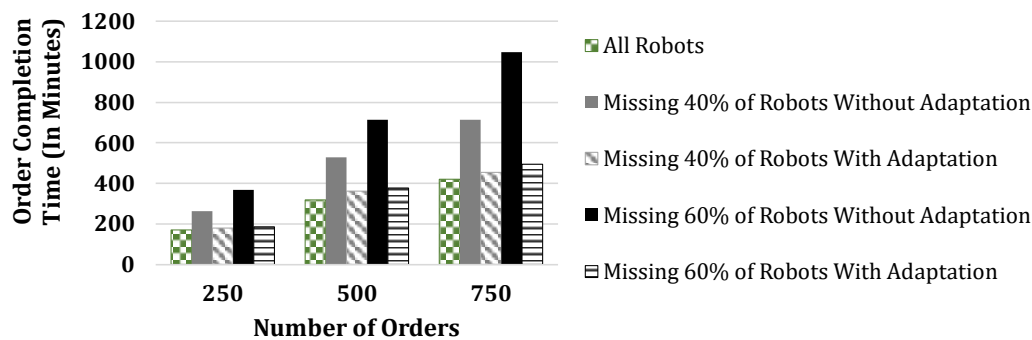


Figure 8.5 – Customer order preparation during resource variability with and without adaptation

Customer orders were generated with random products and the products in stock were dispersed across the grid of the warehouse. Ten types of products were used and included clothing items like shirts, jackets, shorts, trousers, ties, and caps. Upon running the simulation each robot is assigned a customer order that it should prepare. Then, the robots move around the grid and locate the products of the customer orders. Once a robot collects all the products for a customer order, it is assigned another customer order to prepare.

The simulation included cases with full robot capacity and others where 40% and 60% of the robots are missing. These cases included an increasing number of customer orders, namely 250, 500, and 750. In the cases where robots were missing, the simulation was executed with and without adaptation to observe the difference. The adaptation involved executing the faster variant of the “pack products in a box” task, namely “pack randomly” to speed up order preparation during resource variability (i.e., when robots are missing). What follows are the results of this simulation during resource variability with and without adaptation.

8.2.1.2 Results

The chart presented in Figure 8.5 shows how adaptation improved the order completion time when robots were unavailable. As this figure shows, in the cases where 40% or 60% of the robots were unavailable, adaptation reduced the order completion time by almost half in comparison to the cases with missing robots but without adaptation. Adaptation reduced order completion time by an average of 66% and 50% when 40% and 60% of the robots were missing, respectively.

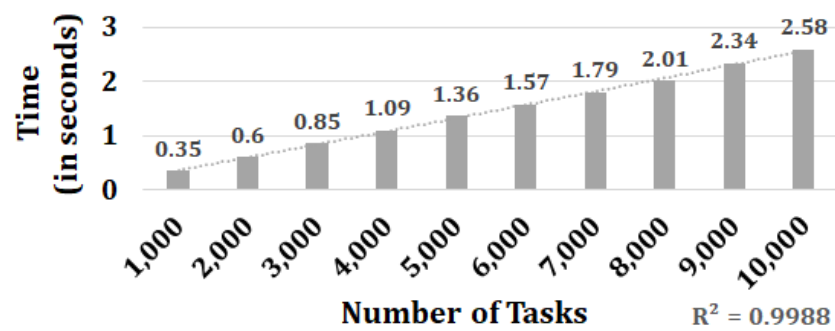


Figure 8.6 – Running time scalability of proactive adaptation planning

These results show that, with adaptation, a smaller number of robots can prepare customer orders without major delays, until the malfunctioning robots are repaired. This simulation demonstrates the feasibility of SPARK as a resource-driven adaptation framework, which enables software systems to keep executing tasks during resource variability.

8.2.2 Evaluation of overhead and scalability

I evaluated the overhead and scalability for two parts of SPARK. The first part is proactive adaptation planning. The second part is the reactive identification of the invoked tasks and variants so SPARK can make adaptation choices. Refer to Sections 6.2 and 6.3 for a detailed explanation of these parts. The overhead and scalability were evaluated using a varying number of business tasks that are commonly found in enterprise systems (e.g., “issue sales invoice”, “view sales report”, and “make item reception”). The evaluation was done on a Windows 10 computer with a Core i7 1.8 GHz CPU and 16 GB of RAM.

8.2.2.1 Proactive adaptation planning

As shown in Figure 8.6, the running time ranges between 0.35 and 2.58 seconds when the number of tasks ranges between 1000 and 10,000. It is possible to say that SPARK has a minor overhead, especially when considering that proactive adaptation planning is not executed with every task request. Furthermore, as shown in Figure 8.6, the fitting curve of the running time is polynomial with R^2 equal to 0.9988. Hence, the algorithm for proactive adaptation planning is scalable.

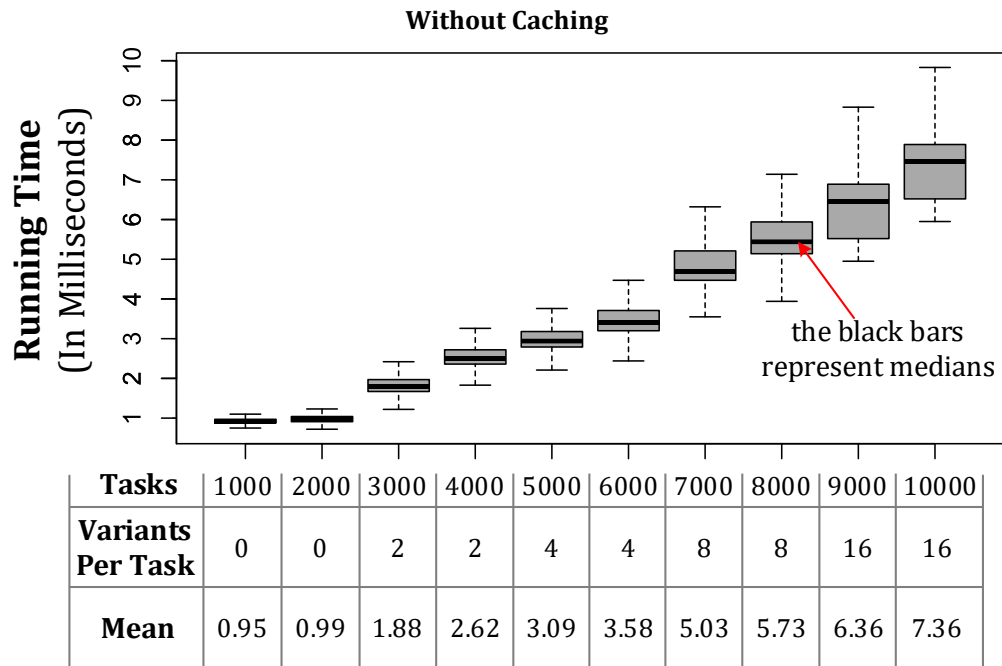


Figure 8.7 – Overhead and scalability evaluation without caching for the identification of tasks and task variants from SERIES task models

8.2.2.2 Reactive identification of tasks and task variants

NBomber (2021) was used to simulate user requests to C# web service methods that represent tasks in a software system. Then, the evaluation involved measuring the time it took to identify the tasks and task variants, in a SERIES task model, which correspond to the service methods being called.

Multiple test runs were performed with an increasing number of tasks and variants ranging from 1,000 to 10,000 tasks and 0 to 16 variants per task. The evaluation was done with two implementations, one that caches the task models in memory and another one that does not perform caching. NBomber simulated user requests for 10 minutes per test run. The results are shown in Figure 8.7 (without caching) and Figure 8.8 (with caching). The mean running time was measured in milliseconds and ranged between 0.95 and 7.36 without caching and 0.0047 and 0.0075 with caching. Hence, the overhead is minor. Furthermore, both fitting curves of the mean running times are polynomial with R^2 equal to 0.9924 (without caching) and 0.9797 (with caching). This indicates that the algorithm for the reactive identification of tasks and task variants is scalable. The use of caching is favourable since it reduces overhead without burdening

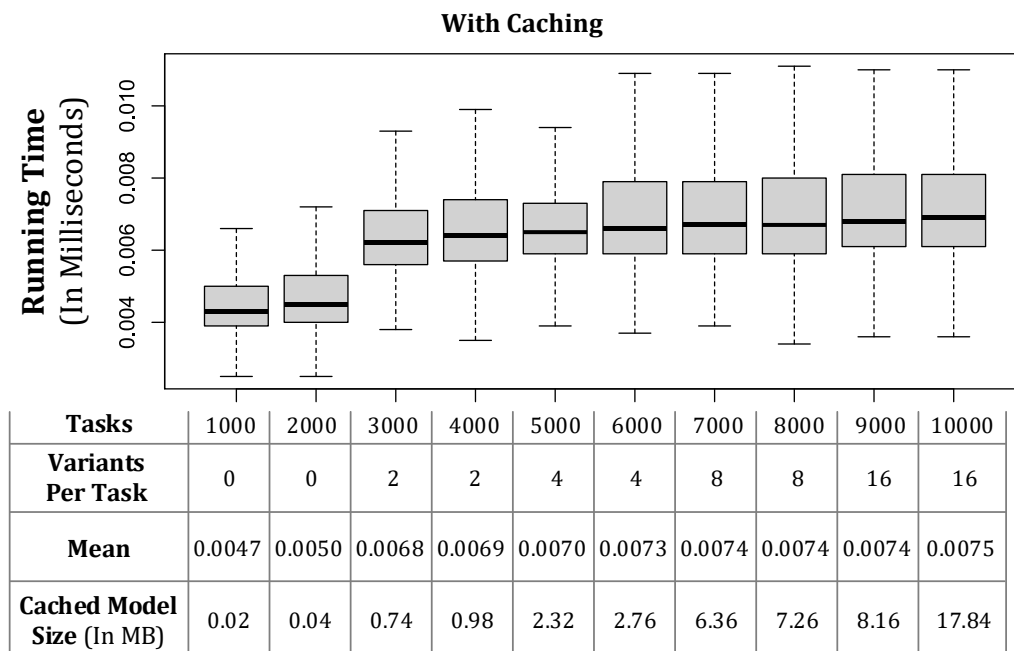


Figure 8.8 – Overhead and scalability evaluation with caching for the identification of tasks and variants from SERIES task models

the memory. As shown in Figure 8.8, the size of the cached model ranged from 0.02 MB to 17.84 MB, which is minor for modern RAM capacity.

8.2.3 Summary of the results

The automated warehouse management system example, which was used in this preliminary evaluation, demonstrates SPARK's feasibility and its ability to support software systems to keep executing tasks during resource variability. Furthermore, the preliminary overhead and scalability evaluation showed that SPARK is scalable and has a minor overhead that does not hinder a software system's ability to execute task requests in a tolerable waiting time (e.g., 2 to 4 seconds as indicated by Nah, (2004)).

8.2.4 Threats to validity

The automated warehouse management system example only considers one type of system, task, resource, and adaptation. Furthermore, the preliminary evaluation uses data that was generated by the researcher. Therefore, two case studies representing different systems were conducted to evaluate SPARK with scenarios that involve multiple types of tasks, resources, and adaptation. These two case studies are presented in the next section.

8.3 Evaluating SPARK with two case studies

I evaluated SPARK with two case studies involving two datasets from a medicine consumption system and a manufacturing system. In this evaluation, I measured four metrics including (i) the percentage of executed critical task requests, (ii) the average criticality of the executed task requests in comparison to the non-executed ones, (iii) the overhead of the approach, and (iv) the scalability of the approach. What follows are the details of these two case studies, including a description of the datasets, an explanation of the design of the case studies, and a presentation and discussion of the results and threats to validity.

8.3.1 Datasets

The two datasets used for evaluating SPARK were selected after searching publicly available datasets on several platforms including AWS Data Exchange (Amazon, 2019), Data.Mendeley (2013), Data.World (2016), IEEE DataPort (*IEEE*, no date), Kaggle (2010), Office for National Statistics (UK Statistics Authority, 1996), United States Census Bureau (US Department of Commerce, 1902), and Zenodo (CERN and OpenAIRE, 2013). The target datasets were expected to contain data for simulating software systems that execute resource-dependent tasks. Hence, candidate datasets were examined using four criteria to check if they contain (i) tasks and task variants; (ii) resources; (iii) task (variant) requests with a chronological order that could be replicated in a simulation; and (iv) association between each task (variant) request and the resources it requires. The characteristics of the datasets are summarised according to the abovementioned criteria (i)-(iv) in Table 8.1 and are elaborated below.

The first selected dataset corresponds to a medicine consumption system (Ghodki, 2021). This dataset contains 783 variants of a medicine allocation task. The task variants differ according to a parameter that represents the medical condition of the patient. This dataset has 153,385 task requests, whereby each one allocates a quantity of a medicine to treat a patient's medical condition. This dataset has depletable resources, namely medicines. It also contains feedback provided by patients to rate, on a scale, the effectiveness of the medicines that they were given to treat their medical conditions. SPARK uses this type of user feedback to adjust its adaptation type choices (refer to Section 6.3.6).

Table 8.1 – Characteristics of the datasets used in the evaluation

	Dataset 1: Medicine Consumption	Dataset 2: Manufacturing
Number of tasks and task variants	783	7
Number of resource types	3,260	17
Number of task (variant) requests	153,385	275
Association between each task (variant) request and the resources it requires	✓	✓

The second selected dataset corresponds to a manufacturing system (Mota *et al.*, 2020). This dataset contains 4 tasks and one of these tasks has 4 variants. It also contains 275 task requests. This dataset has both depletable and reusable resources, namely raw materials and machines respectively. These resources are used to execute tasks related to the manufacturing of hang tags, which are “*tags attached to an article of merchandise giving information about its material and proper care* (Webster, 2022)”.

The two selected datasets have four differences, which create diversity in the case studies conducted to evaluate SPARK. The first difference is in the types of resources. The types of resources in the medicine consumption dataset are depletable whereas the ones in the manufacturing dataset are depletable and reusable. The second difference is in the number of tasks and task variants. The medicine dataset has one task with a lot of variants whereas the manufacturing dataset has a few more tasks with few variants. The third difference is in the number of task requests. The medicine dataset has a larger number of task requests than the manufacturing dataset. The fourth difference is in the presence of user feedback. The medicine consumption dataset has user feedback in the form of ratings as previously mentioned, whereas the manufacturing dataset does not have such feedback data.

8.3.2 Design of the case studies

This section explains the evaluation metrics and how they are applied to different cases of task criticality, resource variability, and modes of adaptation. It also presents the hypothesis that is evaluated through the case studies.

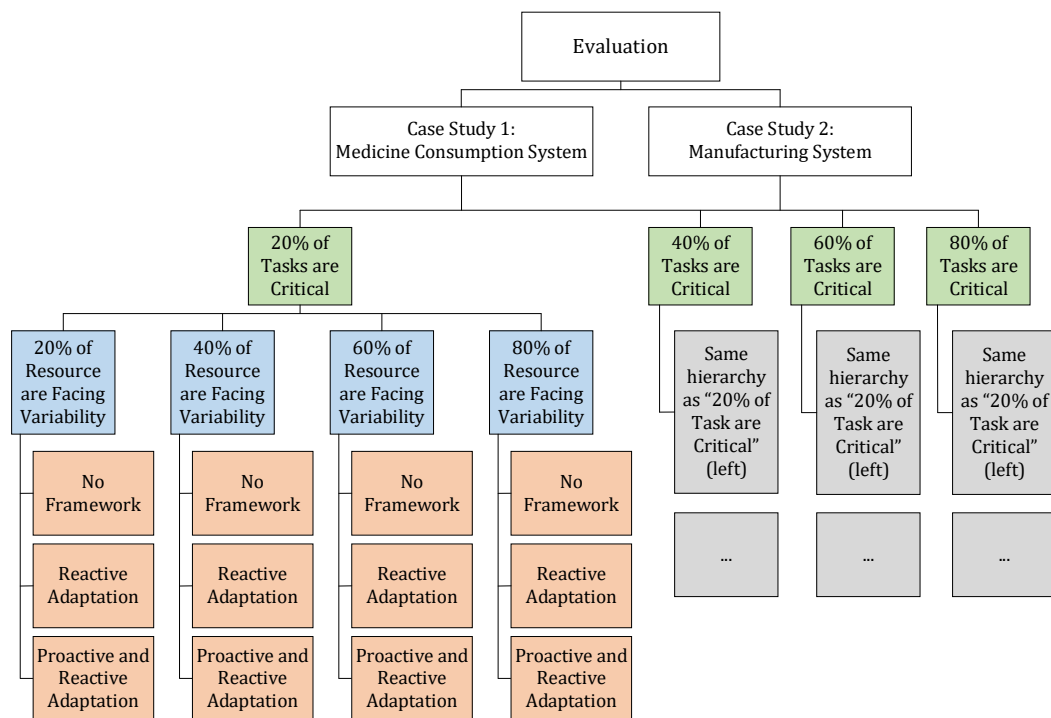


Figure 8.9 – Evaluation case studies with subcases of task criticality, resource variability, and mode of adaptation

8.3.2.1 Metrics and cases

Four metrics were applied in both case studies. These metrics included measuring the percentage of executed critical task requests, the average criticality of the executed task requests in comparison to the non-executed ones, the overhead of the approach, and the scalability of the approach.

The metrics “percentage of executed critical task requests” and “average criticality of the executed task requests in comparison to the non-executed ones” are related to task criticality. A critical task has high importance for its domain and should be more privileged in accessing the resources it needs in comparison to non-critical tasks. Three levels of criticality for the tasks were considered, where the first level is “critical” and the other two levels are “non-critical” namely tasks of moderate or little importance for the domain.

The metric “percentage of executed critical task requests” is measured by counting the critical task requests that got executed and dividing this count by the total number of critical task requests. The metric “average criticality of the executed task requests in

comparison to the non-executed ones” is measured as follows. First, the executed and non-executed task requests are each grouped by the abovementioned three levels of criticality. Then, the difference between the percentages of executed task requests and non-executed ones is computed for each level of criticality. Afterwards, a weighted average of the percentages is computed (critical task requests have the highest weight).

For the two case studies, the abovementioned two metrics are applied to multiple cases of task criticality, resource variability, and mode of adaptation as shown in Figure 8.9. There are **four cases of task criticality**, where 20%, 40%, 60%, and 80% of the overall tasks and variants are critical. Additionally, within each of the task criticality cases, there are **four cases of resource variability**, where 20%, 40%, 60%, and 80% of the resource types are facing variability. Furthermore, within each resource variability case, there are **three cases of the mode of adaptation**, namely (i) no use of the framework, (ii) use of the framework for the reactive process only, and (iii) use of the framework for the proactive and reactive processes.

Concerning the three cases of adaptation, “*no use of the framework*” means that the task requests are executed as they arrive without any adaptation. The way of using the framework (SPARK) with both the “*proactive and reactive*” was explained in Sections 6.2 and 6.3. On the other hand, using the “*reactive*” process only means that there is no proactive prioritisation. Hence, the framework reactively uses a task’s criticality as its priority without considering other factors such as historical task usage, which are used in the proactive computation of priorities. The reactive adaptation executes a critical task if the required resource type is not facing variability, otherwise, adaptation is applied. Additionally, since the priority values are not unique, resources are allocated to tasks on a first-come-first-serve basis when multiple tasks have the same priority.

I evaluated the overhead and scalability for three parts of SPARK. The first part is proactive adaptation planning, which is responsible for preparing an adaptation plan proactively. The second part is the reactive identification of tasks and variants, which is responsible for identifying which task or task variant is being invoked so SPARK can make the necessary adaptation choices. The identification is done by associating the name of the service method being called with its corresponding task in the SERIES task model. The third part is the reactive task execution allocation, which is responsible for allocating executions to tasks when resources are facing variability. Refer to Sections 6.2 and 6.3 for a detailed explanation of these parts. I evaluated the overhead by measuring the running time in each of these three parts to see if using SPARK impacts

a software system's ability to execute tasks in a tolerable waiting time. Furthermore, an increasing number of tasks (variants) is used from the dataset when measuring the running time to determine whether SPARK is scalable based on the type of trendline (e.g., logarithmic, polynomial, or exponential).

8.3.2.2 Simulation tool and adaptation types

I developed a simulation tool to simulate the task requests found in the datasets. I implemented the modes of adaptation mentioned in Section 8.3.2.1. The simulation tool invoked the task requests while SPARK handled the resource-driven adaptation using multiple types of adaptation as explained next.

The first case study involved, two types of adaptation, namely resource substitution and task cancellation. The substitutability of the depletable resource types (medicines) with one another was deduced from the dataset. Furthermore, this dataset contains user feedback that rates the effectiveness of medicines in treating medical conditions. These ratings are used for resource substitution to decide which potential substitute is the best choice. The adaptation types involving the execution of a task variant instead of another and delaying a task do not apply to this case study. Since the medical conditions are different from each other and are not interchangeable, the variants of the medicine allocation task are non-substitutable. For example, it is not possible to consider that patients have diabetes if they have allergies. Furthermore, delaying a task is not applicable because the resource types are depletable meaning that they are either available or unavailable. Hence, task requests do not need to wait their turn as is the case with reusable resources or when there are scheduled replenishments for the depletable resources which is not the case here.

The second case study included all four of SPARK's types of adaptation because the dataset of this case study contains both reusable and depletable resource types. This means when a reusable resource type is unavailable, it can be substituted with another one or the task request can be delayed and executed later. Additionally, this case study had substitutable task variants that produce a similar outcome but differ according to the time needed to execute. Hence, it is possible to execute one variant instead of another when needed to reduce the strain on the reusable resource types (machines). Moreover, the depletable resource types are also substitutable. Furthermore, task requests are cancelled when no other type of adaptation is applicable.

8.3.2.3 Hypothesis

Software systems comprise tasks that require resources. Hence, a software system can execute a task if the required resources are available. During resource variability, it is important to keep the resources available for critical tasks that need them most. Accordingly, the hypothesis **H₁** is described as follows:

H₁: *When resources are facing variability, SPARK's proactive and reactive adaptation improves a software system's ability to execute critical tasks.*

The hypothesis is primarily evaluated using the metrics related to measuring the percentage of executed critical task requests and the average criticality of the executed task requests in comparison to the non-executed ones. Furthermore, the overhead and scalability of SPARK are measured to see whether using this framework to improve a software system's ability to execute critical tasks is done while maintaining a tolerable waiting time.

8.3.3 Case study 1: Medicine consumption system

The first case study is related to a medicine consumption system. The results of the metrics for this case study are reported and discussed in the following subsections.

8.3.3.1 Metric 1: Percentage of executed critical task requests

Figure 8.10 (a to d) shows the percentage of executed critical task requests for the four cases of task criticality, four cases of resource variability, and three cases of the mode of adaptation that were explained in Section 8.3.2.1. For all cases of task criticality and resource variability, using SPARK's *proactive and reactive adaptation* resulted in a higher percentage of executed critical task requests in comparison to using *reactive adaptation*. Additionally, using *reactive adaptation* resulted in a higher percentage of executed critical task requests in comparison to using *no framework*. *Proactive and reactive adaptation* increased the executed critical task requests by 1% to 6% and 10% to 31% in comparison to using *reactive adaptation* and *no framework* respectively. In this case study, each 1% of the abovementioned increase represents 243, 901, 1137, and 1336 task requests when 20%, 40%, 60%, and 80% of the tasks are critical respectively. What follows is an analysis of these results based on the number of critical task requests and the number of resource types that are facing variability.

Medicine Consumption System

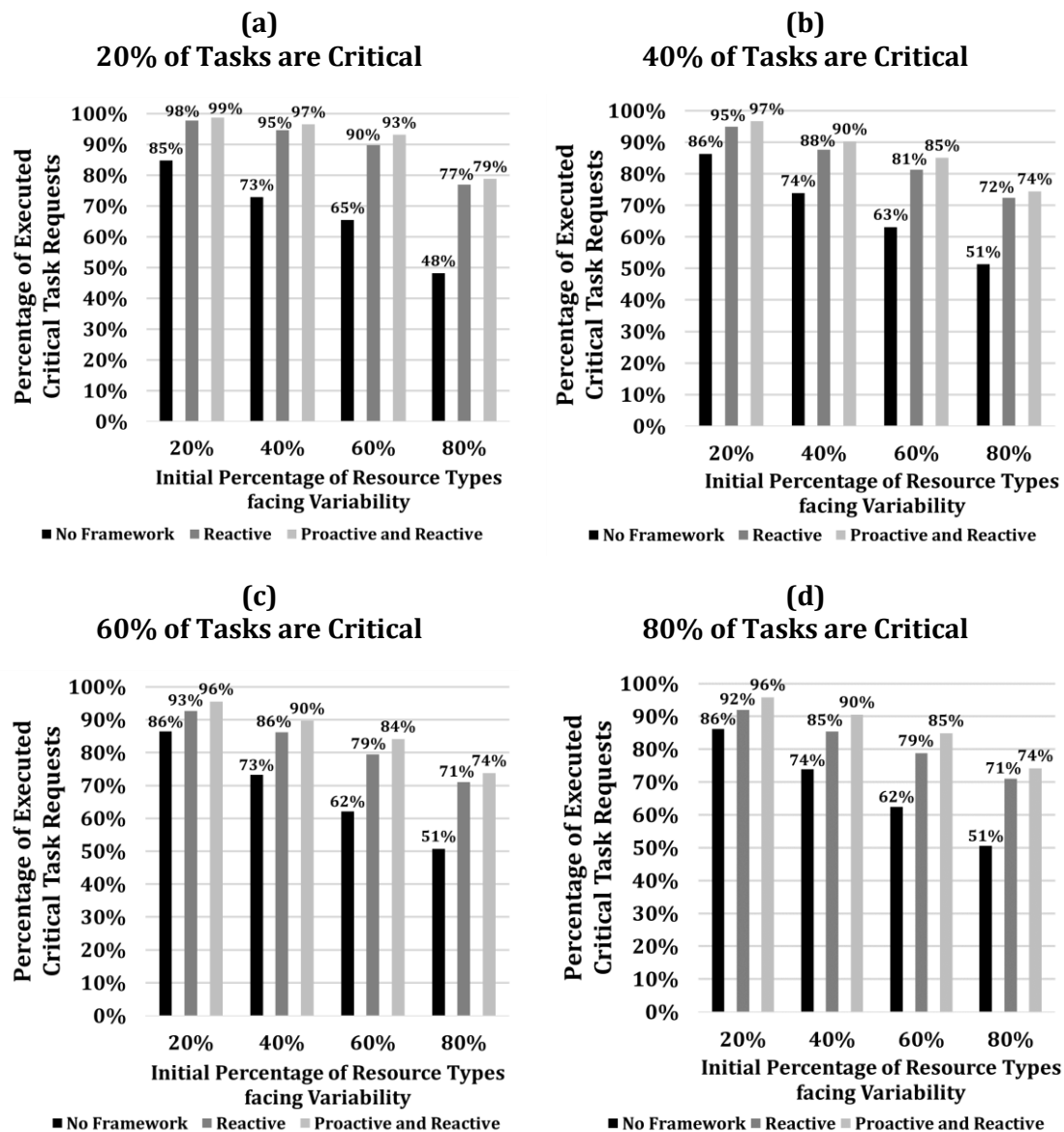


Figure 8.10 – Case Study 1: percentages of executed critical task requests

Figure 8.10a represents a case where 20% of the tasks are critical (in this case study, these are variants of a task as explained in Section 8.3.1). Table 8.2 shows the number of critical task requests and their corresponding number of resource types that are either facing or not facing variability for the case where 20% of tasks (variants) are critical. There were 24,264 critical task requests in total. When 20% of the resource types were facing variability, 19,266 critical task requests used 886 resource types that are not facing variability and 4,998 critical task requests used 210 resource types that are facing variability. Some of the 4,998 critical task requests were

Table 8.2 – Case Study 1: the case where 20% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 157 (20%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20% = 652	886 not facing variability	153	19,266
	210 facing variability	100	4,998
40% = 1,304	661 not facing variability	144	14,319
	435 facing variability	128	9,945
60% = 1,956	466 not facing variability	123	11,139
	630 facing variability	141	13,125
80% = 2,608	219 not facing variability	86	4,008
	877 facing variability	153	20,256

able to use the resource types they originally needed, and most of the remaining task requests were executed after adaptation was performed. Hence, most critical task requests were able to access the resource types that they need. This explains the high percentage of executed critical task requests in this case, where 98% and 99% of the critical task requests were executed when using reactive adaptation and proactive and reactive adaptation respectively.

Moreover, there is a linear drop in the percentage of executed critical task requests between the cases where 40% and 60% of resource types are facing variability (Figure 8.10a). However, there is a higher drop between the cases where 60% and 80% of the resource types are facing variability. The reason for this higher drop is that, in the case where 80% of the resource types are facing variability, 4,008 critical task requests required 219 resource types not facing variability, and 20,256 critical task requests required 877 resource types facing variability. This means that most of the critical task requests required resource types that are facing variability. In comparison, in the case where 60% of the resource types are facing variability, the number of critical task requests (11,139) that require resource types not facing variability is close to the number of task requests (13,125) that require resource types facing variability.

When 40%, 60%, and 80% of tasks (variants) are critical (Figure 8.10b-d), there is a linear drop in the percentage of executed critical tasks between all the cases of resource variability. When comparing the cases where 20% (Figure 8.10a) and 40% (Figure 8.10b) of the tasks are critical, the percentage of executed critical tasks was 6% and 8% lower when 40% and 60% of the resources were facing variability respectively. This is due to a significant increase in the number of critical task requests. This number was 24,264 in the case where 20% of the tasks were critical and it became 90,105 in the case where 40% of the tasks are critical. Hence, the task requests increased by 271%. This increase is smaller when comparing the cases where 40% (Figure 8.10b) and 60% (Figure 8.10d) of the tasks are critical. The number of task requests was 90,105 in the case where 40% of the tasks are critical and it became 113,688 in the case where 60% of the tasks are critical. Hence, the task requests only increased by 26%. Appendix B shows additional information on the number of critical task requests and their corresponding number of resource types for the cases where 40%, 60%, and 80% of tasks (variants) are critical.

8.3.3.2 Metric 2: Average criticality of executed tasks

Figure 8.11(a-d) presents percentages that show how much more critical the executed task requests were in comparison to the non-executed ones. These percentages are presented for the four cases of task criticality, four cases of resource variability, and three cases of the mode of adaptation explained in Section 8.3.2.1. For all cases of task criticality and resource variability, the task requests that are executed when using *proactive and reactive adaptation* are on average more critical than the ones that are executed when using *reactive adaptation*. Additionally, using *reactive adaptation* gave a better result than using *no framework*.

With *proactive and reactive adaptation*, the executed task requests were on average 49% to 95% more critical than the non-executed ones. *Proactive and reactive adaptation* had better results than *reactive adaptation* with a difference of 1% to 3%, 3% to 7%, 5% to 10%, and 7% to 13% in the cases where 20%, 40%, 60%, and 80% of tasks (variants) are critical respectively. This shows that SPARK was able to execute task requests of a higher criticality as the number of critical task requests increased.

Additionally, as shown in Figure 8.11, the percentage is small (1% to 3%) when *no framework* is used in the case where 80% of the resource types are facing variability. The reason behind this result is that around half of the critical task requests were

Medicine Consumption System

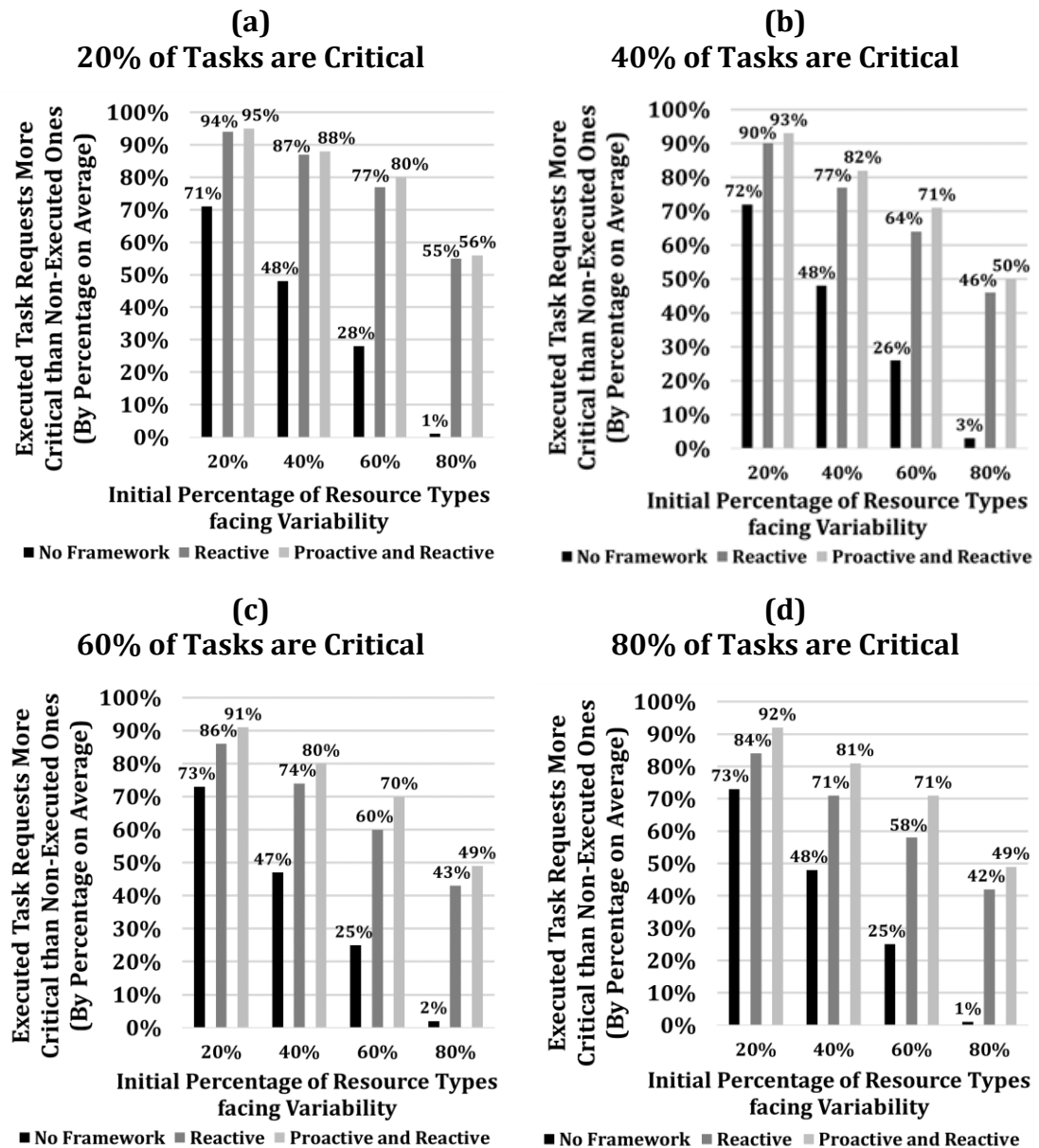


Figure 8.11 – Case Study 1: executed task requests that are more critical than non-executed ones (by percentage on average)

executed, whereas the rest of the executed task requests were non-critical because without the framework the task requests just execute as they arrive. Hence, non-critical task requests that arrived early depleted the resources that are needed for executing critical task requests that arrived late.

Moreover, there is a linear drop between the cases where 20% and 40% and 40% and 60% of resource types are facing variability. However, a higher drop is shown

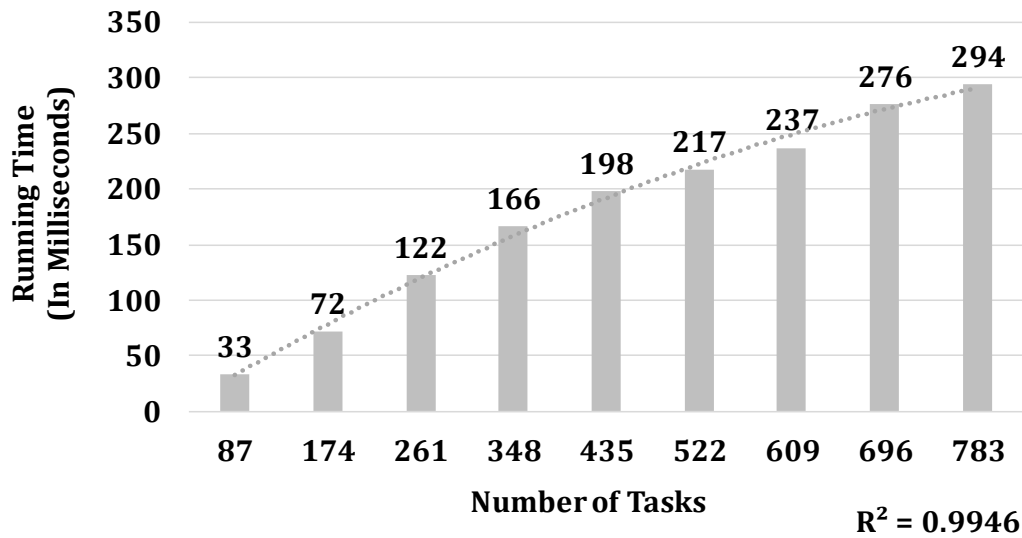


Figure 8.12 – Case Study 1: proactive adaptation planning running time

between the cases where 60% and 80% of resource types are facing variability. The reason for this higher drop is that most of the critical task requests required resource types that are facing variability.

8.3.3.3 Metrics 3 and 4: Overhead and scalability

I evaluated overhead and scalability in three parts of SPARK, namely proactive adaptation planning, reactive identification of tasks and variants, and reactive task execution allocation. What follows are the results of these two metrics.

Proactive adaptation planning. The running time, shown in Figure 8.12, ranges between 33 and 294 milliseconds when the number of task variants is between 87 and 783. The abovementioned values do not add a significant overhead to a software system, especially when considering that proactive adaptation planning is not executed with every task request. Additionally, the fitting curve in Figure 8.12 is polynomial with R^2 equal to 0.9946, which indicates that the algorithm for proactive adaptation planning is scalable.

Reactive identification of tasks and variants. The mean running time, shown in Figure 8.13, ranges between 0.0035 and 0.0040 milliseconds when the number of tasks is between 87 and 783. This overhead is minor, even though the identification is performed with every task request. Additionally, the fitting curve is polynomial with R^2

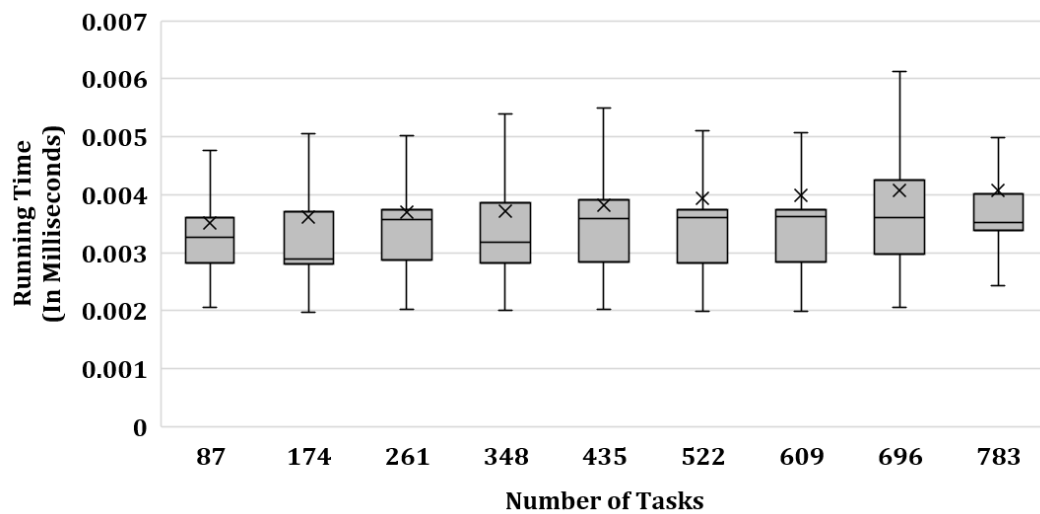


Figure 8.13 – Case Study 1: identification of tasks/variants running time (an “x” on the box plot represents the mean)

equal to 0.9807, which indicates that the algorithm for identifying tasks and variants is scalable.

Reactive task execution allocation. The running time for reactive task execution allocation is shown in Figure 8.14 and ranges between 1.81 and 10.92 milliseconds when the number of tasks is between 87 and 783. A few milliseconds are minor and do not constitute significant overhead. Additionally, the fitting curve in Figure 8.14 is polynomial with R^2 equal to 0.9988, which makes the algorithm for task execution allocation scalable.

The maximum overheads for the abovementioned three parts of SPARK in this case study were 294, 0.0040, and 10.92 milliseconds respectively. This overhead is minor and does not hinder a software system’s ability to execute tasks with a tolerable waiting time. For example, web users find it tolerable to wait for 2 to 4 seconds (Nah, 2004).

I compared the overhead values from this case study to those from the preliminary evaluation reported in Section 7.4. Consider the cases of 783 task variants in the case study and 1000 tasks in the preliminary evaluation since these are the closest to each other in the number of tasks (variants). The overhead reported in this case study was similar to that reported in the preliminary evaluation. The overhead for the proactive adaptation in this case study is 294 milliseconds, which is comparable to the 350 milliseconds from the preliminary evaluation. Additionally, the overhead for the

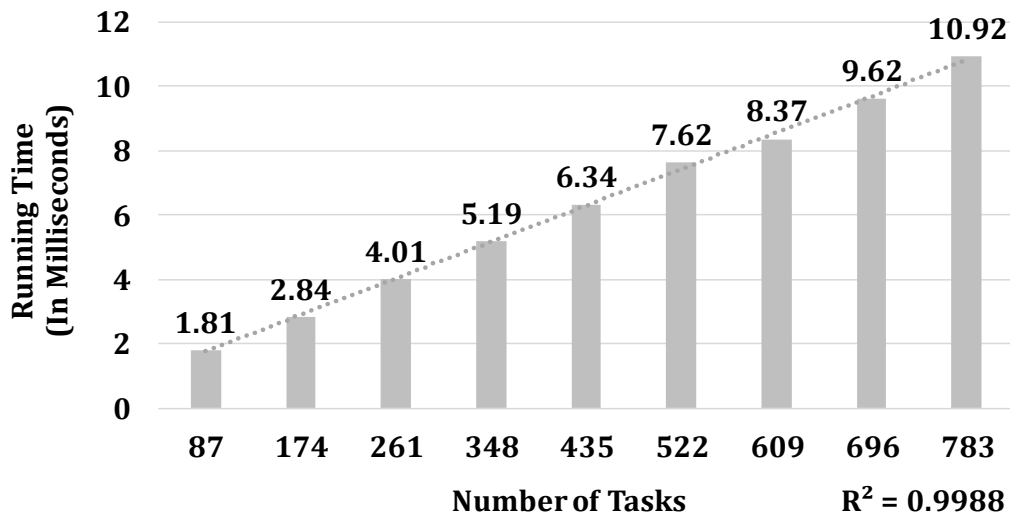


Figure 8.14 – Case Study 1: task execution allocation running time

reactive identification of tasks and variants in this case study is 0.0040 milliseconds, which is comparable to the 0.0047 milliseconds from the preliminary evaluation. Furthermore, the scalability is shown to be polynomial in both the preliminary evaluation and this case study.

8.3.3.4 Summary of case study 1

We can conclude from the results of this case study that using SPARK's proactive and reactive adaptation produces better results for the first two metrics compared to using reactive adaptation only and to not using the framework. SPARK improved the percentage of executed critical task requests by 1% to 6% and 10% to 31% in comparison to using reactive adaptation only and no framework, respectively. It also improved the average criticality of executed task requests by 1% to 13% and 18% to 55% in comparison to using reactive adaptation only and no framework, respectively. Furthermore, SPARK does not add significant overhead and is scalable. These results from this case study satisfy the hypothesis established in Section 8.3.2.3 because they show that SPARK helps software systems in increasing the number of executed critical task requests during resource variability without exceeding a tolerable waiting time concerning overhead and scalability.

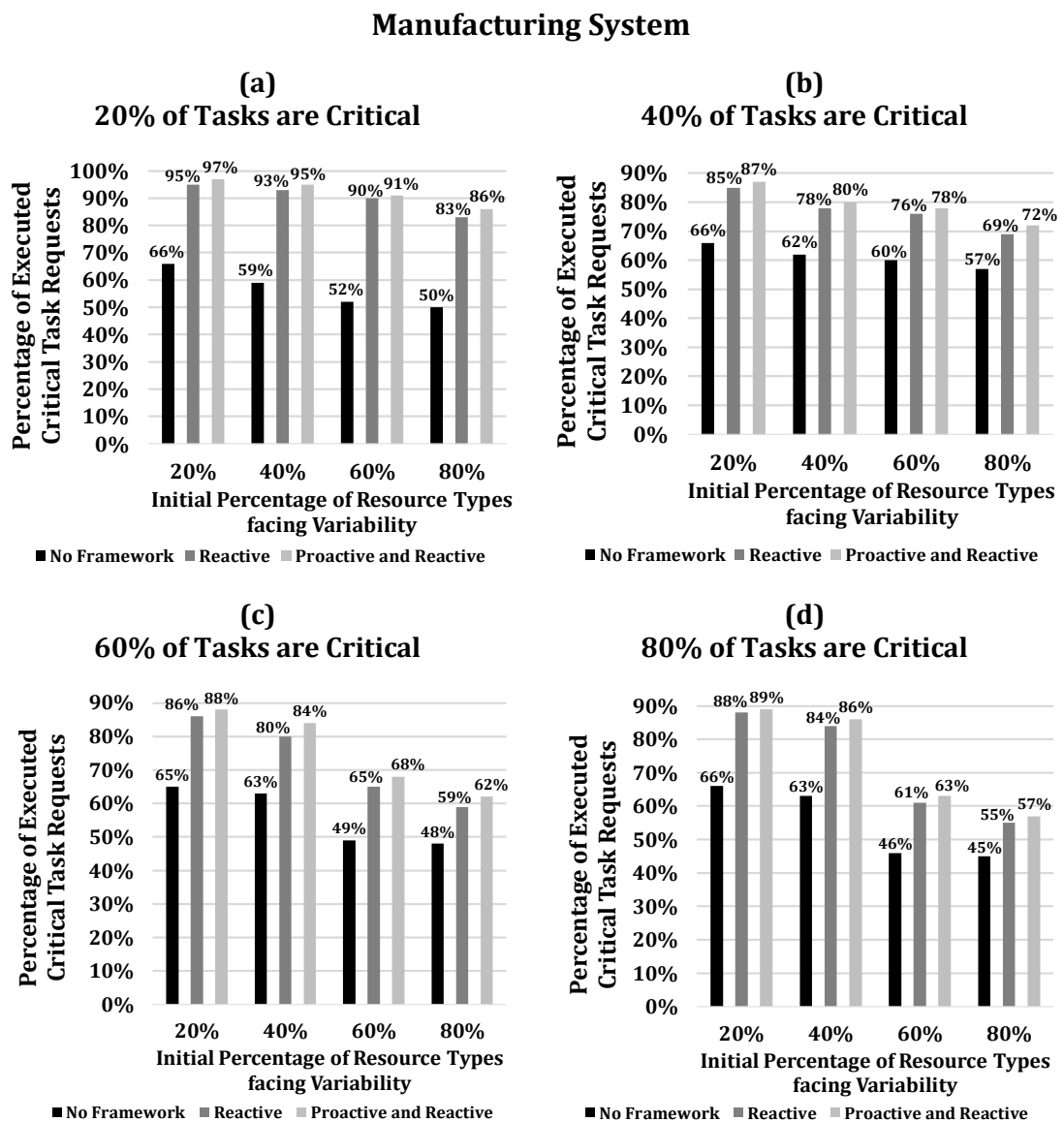


Figure 8.15 – Case Study 2: percentages of executed critical task requests

8.3.4 Case study 2: Manufacturing system

The second case study is related to a manufacturing system. The results of the metrics for this case study are reported and discussed in the following subsections.

8.3.4.1 Metric 1: Percentage of executed critical task requests

Figure 8.15 (a to d) shows the percentage of executed critical task requests for the four cases of task criticality, four cases of resource variability, and three cases of the mode of adaptation that were explained in Section 8.3.2.1. For all cases of task

Table 8.3 – Case Study 2: the case where 20% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 2 (20%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20%	<i>Not facing variability:</i> 6 depletable and 2 reusable	2	65
	<i>Facing variability:</i> 3 depletable and 1 reusable	2	46
40%	<i>Not facing variability:</i> 4 depletable and 2 reusable	2	42
	<i>Facing variability:</i> 5 depletable and 1 reusable	2	69
60%	<i>Not facing variability:</i> 4 depletable and 1 reusable	2	42
	<i>Facing variability:</i> 5 depletable and 2 reusable	2	69
80%	<i>Not facing variability:</i> 2 depletable and 1 reusable	2	25
	<i>Facing variability:</i> 7 depletable and 2 reusable	2	86

criticality and resource variability, using SPARK's *proactive and reactive adaptation* resulted in a higher percentage of executed critical task requests in comparison to using *reactive adaptation* only. Additionally, using *reactive adaptation* resulted in a higher percentage of executed critical task requests in comparison to using *no framework*. *Proactive and reactive adaptation* increased the executed critical task requests by 1% to 4% and 12% to 39% in comparison to using *reactive adaptation* and *no framework* respectively. What follows is an analysis of these results based on the number of critical task requests and the number of resource types that are facing variability.

Figure 8.15a represents a case where 20% of the tasks are critical (in this case study, these are tasks and task variants as explained in Section 8.3.1). Table 8.3 shows the number of critical task requests and their corresponding number of resource types when facing and not facing variability for the case where 20% of tasks (variants) are critical. There were 111 critical task requests in total. When 20% of the resource types were facing variability, 65 critical task requests used 6 depletable and 2 reusable resource types that are not facing variability and 46 critical task requests used 3

depletable and 1 reusable resource types that are facing variability. Some of the 46 critical task requests were able to use the resource types they originally needed, and most of the remaining task requests were executed after adaptation was performed. Hence, most critical task requests were able to access the resource types that they need. This explains the high percentage of executed critical task requests in this case, where 95% and 97% of the critical task requests were executed when using reactive adaptation and proactive and reactive adaptation respectively.

Moreover, there is a linear drop in the percentage of executed critical task requests in the four cases of resource variability when 20% and 40% of tasks are critical (Figure 8.15a-b). However, there is a higher drop in the cases where 60% and 80% of the resource types are facing variability when 60% and 80% of tasks are critical (Figure 8.15c-d). The reason for this higher drop is that there is an increase in the number of critical task requests, which required a reusable resource type that is facing variability with no possible alternatives to execute the task requests. Appendix B shows additional information on the number of critical task requests and their corresponding number of resource types for the cases where 40%, 60%, and 80% of tasks (variants) are critical.

8.3.4.2 Metric 2: Average criticality of executed tasks

Figure 8.16 (a-d) presents percentages that show how much more critical the executed task requests were in comparison to the non-executed ones. These percentages are presented for the four cases of task criticality, four cases of resource variability, and three cases of the mode of adaptation explained in Section 8.3.2.1. For all cases of task criticality and resource variability, the task requests that are executed when using *proactive and reactive adaptation* are on average more critical than the ones that are executed when using *reactive adaptation*. Additionally, using *reactive adaptation* gave a better result than using *no framework*.

With *proactive and reactive adaptation*, the executed task requests were on average 6% to 75% more critical than the non-executed ones. *Proactive and reactive adaptation* had better results than reactive adaptation with an improvement of 1% to 4%, 1% to 6%, 1% to 7%, and 1% to 4% in the cases where 20%, 40%, 60%, and 80% of tasks (variants) are critical respectively. This shows that SPARK was able to execute task requests of a higher criticality as the number of critical task requests increased.

Manufacturing System

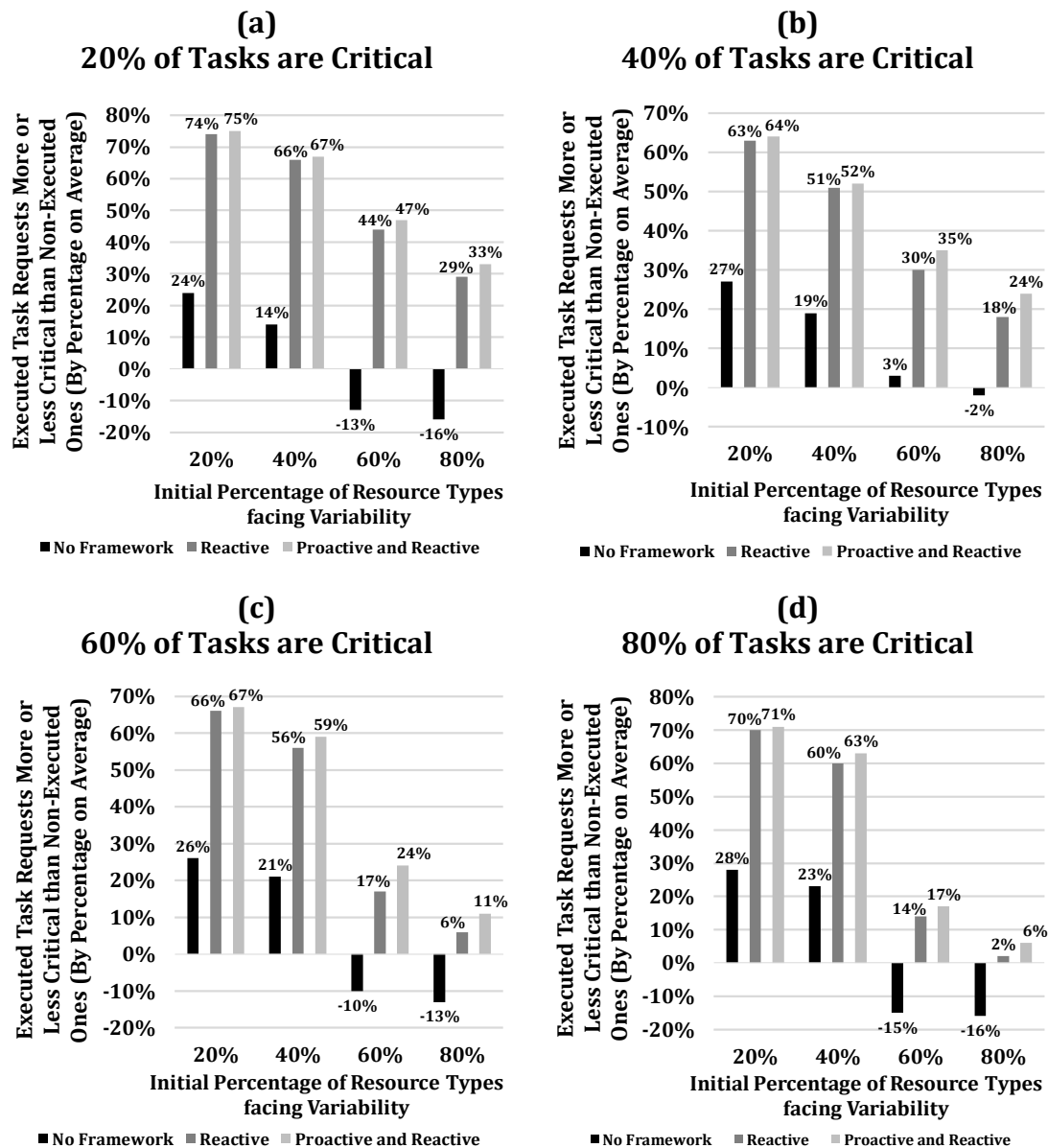


Figure 8.16 – Case Study 2: executed task requests are more or less critical than non-executed ones - by percentage on average
 (positive percentages denote that the executed task requests are more critical while negative percentages denote that the executed task requests are less critical)

As shown in Figure 8.16, the percentage becomes negative when *no framework* is used in the cases where 60% and 80% of the resource types are facing variability. A negative percentage means that the executed task requests are on average less critical than the non-executed ones. The reason behind this result is that a portion of the task requests required a reusable resource type that is facing variability and there is no

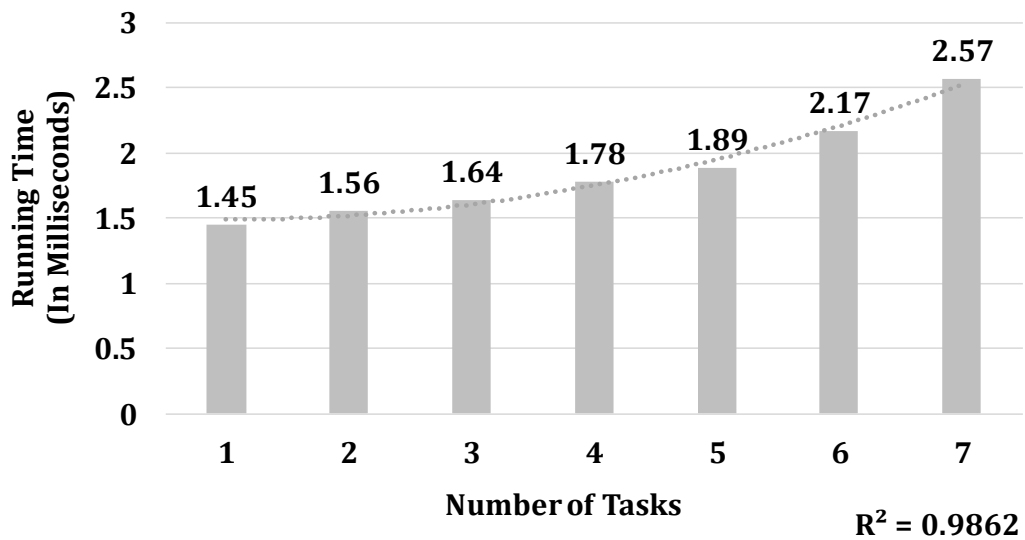


Figure 8.17 – Case Study 2: proactive adaptation planning running time

framework to perform adaptation to address this issue. Hence, the resource was not kept available for the critical tasks that need it most.

8.3.4.3 Metrics 3 and 4: Overhead and scalability

Similar to the first case study, I evaluated the overhead and scalability in three parts of SPARK: (i) proactive adaptation planning, (ii) reactive identification of tasks and variants, and (iii) reactive task execution allocation. What follows are the results of these two metrics.

Proactive adaptation planning. The proactive adaptation planning running time, shown in Figure 8.17, ranges between 1.45 and 2.57 milliseconds when the number of tasks and variants is between 1 and 7. Additionally, the fitting curve in Figure 8.17 is polynomial with R^2 equal to 0.9862, which indicates that the algorithm for proactive adaptation planning is scalable.

Reactive identification of tasks and variants. The reactive identification of tasks and variants running time, shown in Figure 8.18, ranges between 0.0020 and 0.0023 milliseconds when the number of tasks and variants is between 1 and 7. Additionally, the fitting curve is polynomial with R^2 equal to 0.9899, which indicates that the algorithm for the reactive identification of tasks and variants is scalable.

Reactive task execution allocation. The reactive task execution allocation running time, shown in Figure 8.19, ranges between 0.0394 and 0.103 milliseconds when the

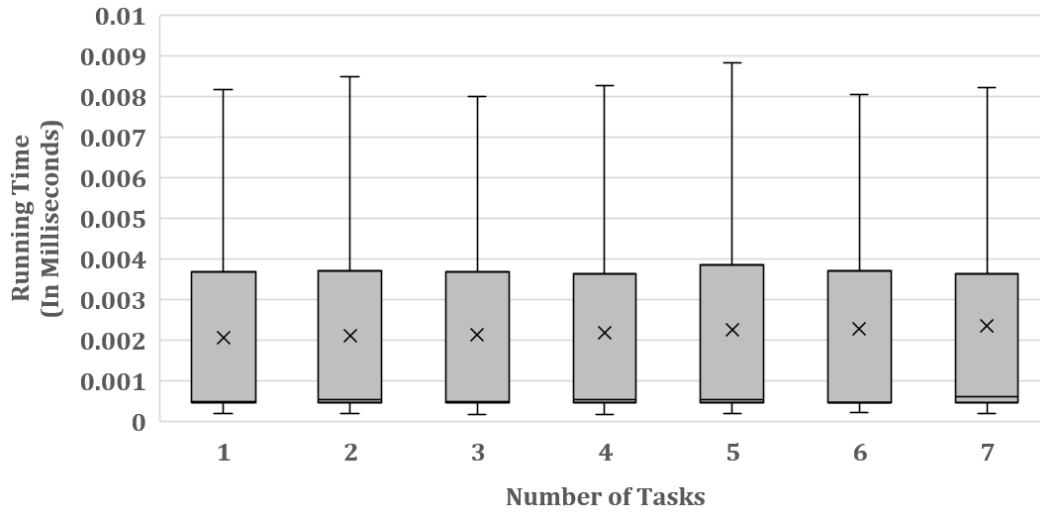


Figure 8.18 – Case Study 2: identification of tasks/variants running time (an “x” on the box plot represents the mean)

number of tasks and variants is between 1 and 7. Additionally, the fitting curve in Figure 8.19 is polynomial with R^2 equal to 0.9781, which indicates that the algorithm for reactive task execution allocation is scalable.

Unlike the first case study, the results of overhead and scalability were not compared to the preliminary evaluation because the dataset of this case study has a smaller number of tasks and variants than the preliminary evaluation. However, like the first case study, SPARK’s overhead is minor and does not hinder a software system’s ability to execute tasks with a tolerable waiting time (e.g., 2 to 4 seconds as indicated by (Nah, 2004)).

8.3.4.4 Summary of case study 2

A conclusion that is drawn from the second case study is that SPARK’s *proactive and reactive adaptation* offers better results in the first two metrics compared to using *reactive adaptation* only and *no framework*. Proactive and reactive adaptation increased the percentage of executed critical task requests by 1% to 4% in comparison to reactive adaptation and by 12% to 39% in comparison to having no framework. Additionally, proactive and reactive adaptation increased the criticality of executed task requests by 1% to 7% in comparison to reactive adaptation and by 22% to 60% in comparison to having no framework. Furthermore, the results show that SPARK has low overhead and is scalable. The results of the four metrics satisfy the hypothesis

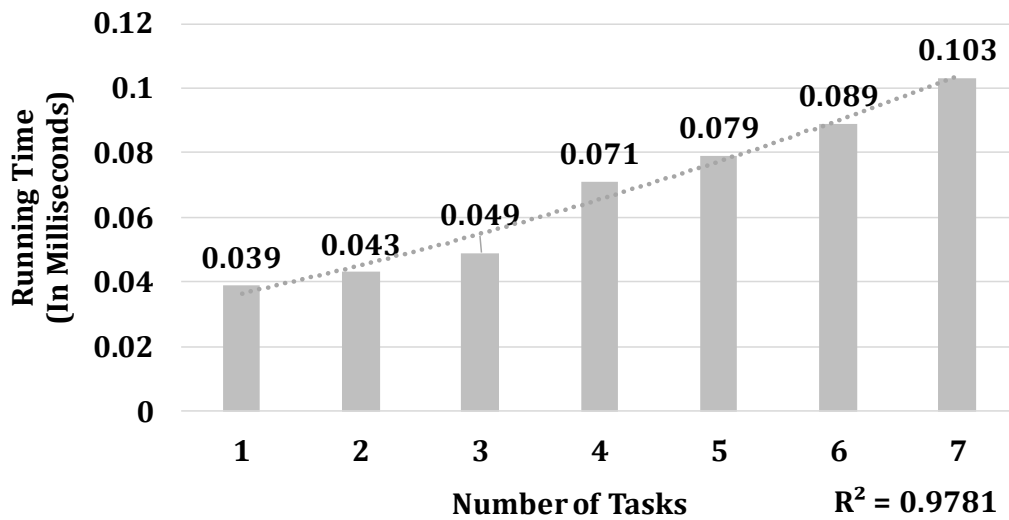


Figure 8.19 – Case Study 2: task execution allocation running time

since SPARK helps software systems in increasing the number of executed critical task requests during resource variability without exceeding a tolerable waiting time concerning overhead and scalability.

8.3.5 Comparison between the two case studies

The average percentage of executed critical task requests is close in both case studies. For the four cases of resource variability and the four cases of task criticality, on average overall, 88% and 80% of the critical task requests were executed when using proactive and reactive adaptation in the first and second case study respectively. Additionally, the improvement that proactive and reactive adaptation provides over proactive adaptation is also close ranging from 1% to 6% in the first case study and 1% to 4% in the second case study.

In both case studies, the use of proactive and reactive adaptation provided an improvement over the use of reactive adaptation. This improvement is also close in both case studies and ranges from 1% to 6% in the first case study and 1% to 4% in the second case study. Additionally, in both case studies, having no framework yielded the lowest percentages of executed critical tasks compared to performing adaptation.

In terms of overhead and scalability, both case studies show that SPARK has low overhead and is scalable, whereby the different parts of this framework have running times in milliseconds and the fittings curves of these running times are polynomial.

8.3.6 Threats to validity

The evaluation results presented in this thesis have both internal and external threats to validity. An internal threat is related to the percentage of critical tasks and resource types facing variability, which were selected by the researcher. However, the evaluation considered multiple cases with an increasing percentage of critical tasks (20%, 40%, 60%, and 80%). Furthermore, each of the abovementioned cases had an increasing percentage of resource types that are facing variability (20%, 40%, 60%, and 80%). These cases show what would be the outcome when there is an increase in the percentage of critical tasks and the percentage of the resource types facing variability. Furthermore, since the percentages are increasing and reach up to 80%, the majority of tasks and resource types are included in the selection.

As for the external threat to validity, the evaluation covered three types of systems, namely a medicine consumption system and a manufacturing system from the two case studies, in addition to the automated warehouse system from the preliminary evaluation. Therefore, the results may not be generalisable. However, the evaluation shows that different types of systems that depend on resources can benefit from SPARK.

8.4 Evaluating intrusiveness

The use of a framework with a software system requires some changes to the software system's source code. These changes should be minimally intrusive for the source code because they would require less effort to perform (Klein, Maggio, *et al.*, 2014). Furthermore, there would be less possibility of introducing errors into the source code of a software system. Intrusiveness is measured in terms of the number of lines of code (LOC) and the number of source-code files that are added or modified to perform the integration. If minor changes are needed, then the integration is non-intrusive. For example, consider that an enterprise system would benefit from using SPARK for performing resource-driven adaptation and that this system uses resources for many tasks (e.g., invoicing, manufacturing, reception, etc.). These tasks typically have many corresponding source code files at each layer of a multi-layer software architecture that includes business logic, controllers, infrastructure, and presentation (Martin, 2009). A minimally intrusive integration would only perform small changes to the smallest possible number of these source code files.

A prototype software application was developed to evaluate the intrusiveness of integrating SPARK into it. The integration was performed as I proposed in Section 4.5. The tasks being initiated in a software system should be intercepted and identified for SPARK to make adaptation decisions. Tasks are initiated by making a call to a service method. For example, to initiate an order preparation task a call is made to a corresponding service method called “PrepareOrder” in the software system. A service method is associated with its corresponding task on a SERIES task model as explained in Section 5.2.3.2. Hence, the prototype software application had service methods that correspond to tasks defined in a SERIES task model.

Actions filters (Larkin *et al.*, 2021) were used to intercept the service method calls and execute SPARK’s algorithm for identifying tasks and task variants. This way when a task or one of its variants is initiated SPARK can identify it and decide whether adaptation is needed. The implementation included one action filter with 136 LOC, which includes the functionality for intercepting tasks and caching task models to reduce execution time. The 136 LOC were added to two source-code files globally for the entire software system. This means that the number of LOC required for the integration of SPARK does not increase as the number of tasks in a software system increases. Based on this, it is possible to say that the integration is non-intrusive because it only requires the addition of a small number of LOC to a small number of source code files to make SPARK work for any number of tasks and task variants. Task-specific source code files like business logic were not modified. This makes it easier to use SPARK with software systems that comprise thousands of tasks (e.g., enterprise systems). On the other hand, if the integration required the addition and modification of many LOC in several source-code files per task, then the intrusiveness would increase as the number of tasks increases, due to the widespread changes.

8.5 Chapter summary

This chapter presented the evaluation of the proposed framework for resource-driven adaptation (SPARK). I conducted a preliminary evaluation of SPARK’s feasibility by developing a simulator of an automated warehouse system. Additionally, this preliminary evaluation involved measuring SPARK’s overhead and scalability. Furthermore, I evaluated SPARK through two case studies that include datasets from a medicine consumption system and a manufacturing system. I measured several metrics including the percentage of executed critical task requests, the average

criticality of the executed task requests versus the non-executed ones, overhead, and scalability. The results showed that SPARK's proactive and reactive adaptation increased the number of executed critical task requests and the average criticality of the executed task requests during resource variability in comparison to using reactive adaptation only and no framework. SPARK did not add significant overhead and was shown to be scalable. Furthermore, a few lines of code are needed to make SPARK work with a software system; this means that it is non-intrusive.

9

Conclusions and Future Work

This chapter summarises the work described in the thesis by presenting an overview of the contributions and evaluation results. Moreover, this chapter presents ideas for future work and a few concluding final remarks.

9.1 Conclusions

This thesis contributed an approach for supporting resource-driven adaptation when software systems are facing resource variability. Chapter 3 presented a literature review of existing task modelling notations and resource-driven adaptation approaches. This literature review compared existing task modelling notations based on their task types and operators and their support for resource-driven adaptation (e.g., resource types and task variants). This helped in identifying the gaps in existing task modelling notations and in formulating the first research question (RQ1) as specified in Section 1.4.1. Additionally, the literature review categorised and critically analysed existing resource-driven adaptation approaches (e.g., based on the types of resources and adaptation that they support). This helped in formulating the second research question (RQ2) specified in Section 1.3.1. To answer these questions, this thesis proposed a task modelling notation called SERIES and a framework called SPARK to support resource-driven adaptation in software systems.

The proposed notation (SERIES) and framework (SPARK) fill the gaps (1-3) presented in Section 3.5. The *first gap* is related to considering tasks in the resource-driven adaptation process to provide granularity in the adaptation decision-making based on task differences. In this regard, SERIES supports the representation of tasks and their variants with properties like resource consumption and the role of the user

who is initiating the task. Additionally, SPARK uses this data in the resource-driven adaptation process to compute unique priorities for tasks and their variants and to decide whether an adaptation type is applicable to a task. The *second gap* is related to supporting multiple types of resources, to make the resource-driven adaptation approach more comprehensive. In this regard, SERIES supports the association of tasks with types of resources that are defined at runtime according to what the software system requires. Moreover, SPARK considers the differences among resources when making adaptation decisions. For example, it determines whether a resource is facing variability by observing the quantity on hand for depletable resources and the effect of reusable resources on the execution duration of tasks. The *third gap* is related to supporting multiple types of adaptation to provide versatility in addressing resource variability. In this regard, SPARK supports four types of adaptation, namely (i) execution of task variants that require fewer resources; (ii) substitution of resource types with alternative ones; (iii) execution of tasks in a different order based on their priorities; and (iv) cancellation of tasks when no other task variant or resource can be used. Hence, if one type of adaptation is not applicable to a task another type of adaptation would be chosen.

Chapter 4 presented an overview of the proposed work. This overview showed the involved *stakeholders*, *adaptation components*, and *data*. The *stakeholders* include system administrators and software practitioners, whose role is to provide setup data and define task models related to the software system respectively. The stakeholders also include end-users who initiate tasks and provide feedback to the software system when it performs adaptation. Moreover, the *adaptation components* have proactive and reactive capabilities. The proactive components are mainly responsible for uniquely prioritising tasks based on multiple criteria and selecting the types of adaptation to perform based on their cost. The reactive components are mainly responsible for identifying if a type of resource is facing variability, managing the execution of the tasks, and eliciting and providing feedback from and to end-users. The *data* required by the proposed framework (SPARK) to perform the adaptation includes setup data (e.g., type of resources) and task models that are represented using the proposed task modelling notation (SERIES). Furthermore, the overview that Chapter 4 presented paved the way for the contributions of Chapters 5 and 6.

Chapter 5 presented the proposed task modelling notation (SERIES). The meta-model of SERIES consists of constructs for representing task models. Examples of

those constructs include resource types, task priorities, task variants, and feedback properties. SERIES represents task models graphically to make them more legible and understandable. Hence, software practitioners can understand how abstractions are refined (e.g., abstract tasks to subtasks) without having to read dense text. Chapter 5 presented an example from an automated warehouse system to illustrate these constructs. Furthermore, Chapter 5 presented a tool for supporting the creation of task models using SERIES and for managing setup data (e.g., resource types and user roles).

Chapter 6 presented the proposed resource-driven adaptation framework (SPARK). This chapter explained the proactive and reactive adaptation components of SPARK. Moreover, Chapter 6 presented an example from an automated warehouse system to demonstrate the calculations that SPARK's adaptation components perform. Examples of those calculations included the calculation of task priorities, the selection of adaptation types, and the allocation of task executions. Furthermore, Chapter 6 discussed the implementation of the adaptation components as a prototype.

Chapter 7 presented an assessment of SERIES based on two paradigms, namely the Cognitive Dimensions Framework and the Physics of Notations. I used these paradigms because they provide useful principles for evaluating visual notations. Additionally, Chapter 7 presented an evaluation of SERIES via a user study with software practitioners. The purpose of the study was to measure the ability of software practitioners to explain and create task models, as well as the usability of SERIES and the clarity of its semantic constructs. The results showed that software practitioners performed very well when explaining and creating task models using SERIES. These results were reflected in the task modelling activities that the participants performed as well as in their positive feedback regarding the usability of SERIES and the clarity of its semantic constructs. The user study also provided some additional insights that were not part of the original research question. In this regard, the feedback of the participants on the SERIES notation included comments that were related to the dimensions and principles of the Cognitive Dimensions Framework and Physics of Notations paradigms respectively. This was an additional insight because the user study did not include questions that are directly related to these paradigms. The feedback provided by the participants complemented the initial assessment that I did with these two paradigms and showed that SERIES adhered to their recommendations. Additionally, participants with different levels of experience in the software industry achieved high scores on both the explanation and creation of SERIES task models.

Although it is not the study's objective to compare these scores by level of experience, I was able to perform this comparison and get some insights because the sample of participants was diverse and included software practitioners whose levels of experience ranged from less than one year up to ten years.

Chapter 8 presented a preliminary evaluation with generated data for SPARK via a tool that simulates an automated warehouse scenario. The tool served as a proof-of-concept prototype to evaluate SPARK's feasibility. Furthermore, Chapter 8 presented two case studies that evaluated SPARK with existing datasets that are related to a medicine consumption system and a manufacturing system. The two case studies involved measuring the percentage of executed critical task requests, and the average criticality of the executed task requests versus the non-executed ones, overhead, and scalability. The results showed an increase in the number of executed critical task requests during resource variability when using SPARK. Moreover, the time it took to prepare and apply adaptation plans did not add significant overhead that affects the software system's ability to execute tasks in a tolerable waiting time. Furthermore, the results showed that SPARK is scalable relative to the increase in the number of tasks. Additionally, the preliminary evaluation done on the SPARK framework provided some additional insights that were not part of the original research question. In this regard, the identification of tasks and variants was initially applied without caching the tasks and variants. However, the results showed that the performance can be improved further. Hence, I created an alternative implementation that performs caching and compared both implementations in the evaluation. This shows that caching improved the performances and was therefore subsequently used in the case studies. Moreover, SPARK originally added a fixed epsilon value to the initial task priorities to make them unique if they were equal. However, I saw that the epsilon value should be changed when the number of tasks increases. Therefore, I adjusted the original design to have a changeable epsilon value, which makes the calculation of the adjusted priorities applicable to any number of tasks.

9.2 Limitations

The work presented in this thesis has some limitations that are discussed in the following subsections. These limitations do not undermine the contribution but are rather either out of the scope of this work or could complement it through future work.

9.2.1 Using SPARK in different types of software systems

The evaluation of the SPARK framework was done with three types of software systems. However, the results may not be generalisable to some types of software systems that have different requirements. For example, embedded systems, such as domestic appliances, have a limited amount of computational resources and a small number of tasks running at the same time. In this type of system, SPARK would be excessive because it is intended for more complex cases of resource variability.

9.2.2 Using SERIES with other frameworks

Frameworks other than SPARK could potentially use SERIES for modelling tasks. Moreover, frameworks could rely on metrics as part of their adaptation process as is done, for example, in some brownout approaches (Xu and Buyya, 2019). SERIES task models reflect metrics like speed and aesthetics indirectly through task variants. However, SERIES was not designed to provide concepts for explicitly defining metrics. Therefore, SERIES would have to be extended if such metrics are explicitly needed by other frameworks.

9.2.3 Integrating SPARK into applications

I integrated SPARK into prototype software applications to evaluate it. However, an application programming interface (API) is needed to enable software practitioners to integrate SPARK into their software systems. This API should work with existing software development technologies such as Integrated Development Environments. This thesis did not present an integration API for SPARK because it is outside the scope of this work.

9.2.4 Providing additional analysis in SPARK

One of SPARK's adaptation components forecasts the number of task execution requests in software systems. SPARK uses this forecasted number to estimate future workloads so it can prioritise tasks and allocate task executions accordingly. However, SPARK does not currently forecast the use of resources by future tasks before making resource substitution decisions. The forecasted resource usage would enable SPARK to

analyse which critical tasks may need certain types of resources in the future, to avoid allocating these resources to non-critical tasks that need to execute in the present.

9.3 Future work

SPARK and SERIES help software systems that rely on multiple resource types in addressing resource variability through resource-driven adaptation. Nonetheless, as explained in this section, there is room for extending SPARK and SERIES in the future.

9.3.1 Extending SERIES and its supporting tool

It is possible to investigate further how to elicit and present feedback to end-users based on the adaptation-related feedback properties of SERIES. In this regard, a study with end-users would provide ideas about how they prefer to receive and provide feedback when a software system performs adaptation to address resource variability. Based on the outcome of this study, it is possible to extend SERIES with additional properties. Furthermore, besides the extension of the notation, it is possible to create and refine a UI for end-user feedback. Software practitioners would integrate this UI with software systems that face resource variability and require resource-driven adaptation.

It is also possible to extend the tool of SERIES to support additional features that would be useful for software practitioners. In this regard, the extension could target the generation of code that corresponds to the service methods of tasks in a SERIES task model. This enables software practitioners to automatically link the task models that they create using SERIES to the source code of their software systems. Additionally, having a search mechanism would enable software practitioners to search through multiple task models and find elements that match advanced search criteria (e.g., composite conditions with wild characters). Then, the software practitioners would be able to perform an update simultaneously on all the matching tasks and relationships from the search result.

9.3.2 Integrating SPARK with real-world software systems

In this thesis, I integrated SPARK with a prototype software system to evaluate its intrusiveness (refer to Section 8.4). It is possible to develop further the integration

mechanism by creating libraries that software practitioners can use through an application programming interface (API). This enables software practitioners to leverage resource-driven adaptation capabilities in the software systems that they are developing using existing IDEs. For example, software practitioners would be able to use classes from these libraries that represent concepts like tasks and resources and invoke services to perform adaptation when needed. These capabilities are already present in SPARK but having an API makes them more accessible during software development.

By developing the abovementioned API and extending the supporting tool (refer to Section 9.3.1), it is possible to improve the adoption of resource-driven adaptation in the software industry. This would be due to the improvement of the ceiling of the tool (i.e., what it can achieve) and the ability to use the proposed framework as an out-of-the-box solution like any software development framework.

9.3.3 Exploring other techniques

We also plan to investigate how to expand the proposed work by using other techniques. We discuss these possible expansions with the following techniques: (i) knapsack, (ii) operations research, (iii) AI planning, and (iv) benchmarks.

Knapsack problem. The multidimensional knapsack problem (Lust and Teghem, 2012) is an optimisation problem that involves making an optimal selection from items that have multidimensional weights. Methods M1 and M2 (refer to Section 6.3.4) allocate executions to tasks based on priorities. The items and weights in this case are the tasks and their priorities, respectively. The priorities are computed based on multiple criteria before applying methods M1 or M2. Hence, tasks that have a higher priority get allocated more executions. Since the multidimensional knapsack problem is known to be NP-hard (Chu and Beasley, 1998), we plan to investigate the use of the prioritisation technique and methods M1 and M2 proposed in this thesis for solving existing knapsack problems, like financial portfolio selection with a small amount of computational effort.

Operations research. Operations research (OR) encompasses problem-solving techniques, such as simulation and mathematical optimisation, which support decision-making. Operations research is used for business decisions and helps companies in setting up their decision support systems (Gupta *et al.*, 2022). Future

work could consider using OR for placing constraints on metrics that are meaningful for a domain (e.g., resource consumption rate, cost of resources, loss of human life, etc.). The work presented in this thesis indirectly reflects these metrics through priorities and task variants. For example, in the given automated warehouse example the task “pack items in a box” has two variants “pack randomly” and “pack by item type”, which improve speed and aesthetics respectively. It would be useful to explore whether the understandability of these metrics would improve if they were stated explicitly instead of keeping them implicitly stated via the task variants and priorities.

AI planning. Future work could also explore how addressing resource variability could benefit from AI planning, which involves choosing actions to perform and structuring them in a plan (Wilkins, 2014, pp. 3–5). In this regard, it is possible to consider how AI planning could benefit both SERIES and SPARK.

More specifically, SERIES could be extended with concepts that are offered by the Planning Domain Definition Language (PDDL+), which is a family of languages for defining a planning problem. PDDL+ has the concepts of events and effects (Coles and Coles, 2014). An event represents an occurrence within a system and an effect specifies a change that shall be done when the event occurs. The events and effects in PDDL+ could complement the work presented in this thesis in the case of dynamic resources. These resources have settings variables that a software system can change when there is resource variability. Examples include changing the brightness of a display to reduce battery consumption. In this thesis, these settings are changed via parameters of task variants to take into consideration differences among tasks. However, if some settings apply in the same way to all tasks, it could be possible to change them as an effect of resource variability events. Furthermore, if a language such as PDDL+ was used to complement SERIES it is important to evaluate its usability in comparison to the graphical notation of SERIES. Hence, some additional evaluation with software practitioners will provide insights into their ability to understand and use concepts like events and effects in a textual language like PDDL+.

The PDDL+ events and effects shall be defined on SERIES task models and given as input to SPARK. Hence, SPARK shall identify when these events occur and apply their corresponding effects. To achieve this, SPARK shall define two additional adaptation components, namely an event monitor and an effector. The event monitor shall monitor the occurrence of events and notify the effector accordingly. In turn, the effector shall apply an event's corresponding effect, which will change the settings of a

dynamic resource to address a situation of resource variability. Some additional evaluation would also be needed on SPARK to assess its ability to handle events and apply their corresponding effects without adding significant overhead to software systems.

Benchmarks. Benchmarks are important for comparing frameworks that have the same objective. Presently, there is a lack of benchmarks for evaluating resource-driven adaptation frameworks (Xu and Buyya, 2019). It is challenging to create such benchmarks because existing approaches have different objectives and capabilities. For example, SPARK supports depletable and reusable resources whereas other approaches only support one type of resource. Hence, an important direction for future work would be to create a set of benchmark metrics and datasets that can be used to evaluate and compare resource-driven adaptation frameworks. These benchmarks should consider different objectives of existing work on resource-driven adaptation, as well as different types of resources, and different ways of adapting the systems. The metrics and datasets that I used for evaluating SPARK could serve as a starting point to create benchmarks. Alternatively, benchmarks from AI planning could also be explored such as the ones that are used in international planning competitions (Coles *et al.*, 2012). Examples of these benchmarks include controlling ground traffic at an airport (Botea *et al.*, 2005) and resupplying lines in a faulty electricity grid (Hoffmann *et al.*, 2006).

9.4 Final thoughts

The contributions of this thesis are not meant to replace existing software development approaches, but rather to further empower software practitioners to develop software systems that are capable of addressing resource variability through resource-driven adaptation. This in turn benefits end-users because software systems will be capable of executing important tasks when resources are scarce or unavailable.

Although the focus of this thesis was on enterprise systems as motivating examples, SPARK and SERIES are not restricted to these systems. I chose enterprise systems because they encompass a variety of tasks and resources and their ability to function affects multiple domains (e.g., manufacturing and medical). Nonetheless, it is possible to use SPARK and SERIES to support resource-driven adaptation in other types of software systems that rely on resources, which are affected by variability.

The work presented in this thesis contributed to the field of self-adaptive systems by placing further emphasis on the importance of considering resource variability as a trigger for adaptation. Furthermore, this work does not only consider computing resources (e.g., CPU) that a software system needs, but it also considers different types of resources that are needed to execute the tasks that are meant to be fulfilled by these systems. Research on self-adaptive systems has been ongoing for years and it empowered software systems to manage themselves and dynamically adapt to change. Nonetheless, it is still possible to do further research work. Hence, in these final thoughts, it is worth mentioning that increasing the adoption of adaptation frameworks is important to benefit from the contributions of the research on self-adaptive systems. APIs and tools can integrate adaptation frameworks and software development frameworks (e.g., web development frameworks that practitioners use in the software industry). Moreover, it would be interesting to see future research that explores what software practitioners think about adaptation frameworks and how they use them. The outcome could be a set of guidelines that informs research on self-adaptive systems. Hence, when designing an adaptation framework, researchers would not only take into account how the framework works and whether it produces the desired outcome, but would also consider how it shall be used in practice by assessing it against a set of guidelines.

Bibliography

- Adelstein, F. et al. (2005) Fundamentals of mobile and pervasive computing. McGraw-Hill New York.
- Akiki, P., Zisman, A. and Bennaceur, A. (2022) ‘SERIES: A Task Modelling Notation for Resource-driven Adaptation’, in. 24th International Conference on Enterprise Information Systems, Online Streaming: SCITEPRESS - Science and Technology Publications, pp. 29–39. Available at: <https://doi.org/10.5220/0011001800003179>.
- Akiki, P., Zisman, A. and Bennaceur, A. (2023) ‘Modelling Software Tasks for Supporting Resource-driven Adaptation’, in. Springer (Lecture Notes in Business Information Processing).
- Akiki, P.A. (2021) ‘Towards an approach for resource-driven adaptation’, in. ESEC/FSE ’21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens Greece: ACM, pp. 1625–1629. Available at: <https://doi.org/10.1145/3468264.3473098>.
- Akiki, P.A., Zisman, A. and Bennaceur, A. (2021) ‘Work With What You’ve Got: An Approach for Resource-Driven Adaptation’, in. 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), DC, USA: IEEE, pp. 105–110. Available at: <https://doi.org/10.1109/ACSOS-C52956.2021.00030>.
- Allemang, D. and Hendler, J. (2011) Semantic web for the working ontologist: effective modeling in RDFS and OWL. Elsevier.
- Amazon (2019) AWS Data Exchange. Available at: <https://aws.amazon.com/data-exchange/>.
- Annett, J. (2003) ‘Hierarchical task analysis’, Handbook of cognitive task design, 2, pp. 17–35.
- Azure (2022) Throttling pattern. Available at: <https://learn.microsoft.com/en-us/azure/architecture/patterns/throttling>.
- Bandara, A.K. et al. (2004) ‘A goal-based approach to policy refinement’, in Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004. IEEE, pp. 229–239.
- Batra, D., Hoffler, J.A. and Bostrom, R.P. (1990) ‘Comparing representations with relational and EER models’, Communications of the ACM, 33(2), pp. 126–139. Available at: <https://doi.org/10.1145/75577.75579>.

- Bauer, A. (2019) 'Challenges and Approaches: Forecasting for Autonomic Computing', in *Organic Computing: Doctoral Dissertation Colloquium 2018*. kassel university press GmbH, p. 1.
- Bauer, A. et al. (2020) 'Time Series Forecasting for Self-Aware Systems', *Proceedings of the IEEE* [Preprint].
- Benavides, D., Trinidad, P. and Ruiz-Cortés, A. (2005) 'Automated reasoning on feature models', in *International Conference on Advanced Information Systems Engineering*. Springer, pp. 491–503.
- Benedek, J. and Miner, T. (2002) 'Measuring Desirability: New methods for evaluating desirability in a usability lab setting', *Proceedings of Usability Professionals Association*, 2003(8–12), p. 57.
- Bennaceur, A. et al. (2019) 'Won't Take No for an Answer: Resource-Driven Requirements Adaptation', in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Montreal, QC, Canada: IEEE, pp. 77–88. Available at: <https://doi.org/10.1109/SEAMS.2019.00019>.
- Blue Link (2022) 'Import & Export Software'. Available at: <https://www.bluelinkerp.com/import-export-inventory-accounting-software/>.
- Bork, D. and Roelens, B. (2021) 'A technique for evaluating and improving the semantic transparency of modeling language notations', *Software and Systems Modeling*, 20(4), pp. 939–963. Available at: <https://doi.org/10.1007/s10270-021-00895-w>.
- Botea, A. et al. (2005) 'Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators', *Journal of Artificial Intelligence Research*, 24, pp. 581–621. Available at: <https://doi.org/10.1613/jair.1696>.
- Braberman, V. et al. (2017) 'An extended description of morph: A reference architecture for configuration and behaviour self-adaptation', in *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, pp. 377–408.
- Buckley, J. et al. (2005) 'Towards a taxonomy of software change', *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), pp. 309–332.
- Buisson, J., André, F. and Pazat, J.-L. (2005) 'A framework for dynamic adaptation of parallel components', in *International Conference ParCo*, p. 65.
- Calvary, G. et al. (2003) 'A unifying reference framework for multi-target user interfaces', *Interacting with computers*, 15(3), pp. 289–308.
- Carvalho, H. et al. (2022) 'The resilience of on-time delivery to capacity and material shortages: An empirical investigation in the automotive supply

- chain', *Computers & Industrial Engineering*, 171, p. 108375. Available at: <https://doi.org/10.1016/j.cie.2022.108375>.
- CEA, A. (2009). Available at: <https://www.w3.org/2005/Incubator/model-based-ui/wiki/ANSI/CEA-2018>.
- CERN and OpenAIRE (2013) 'Zenodo'. CERN. Available at: <https://doi.org/10.25495/7GXX-RD71>.
- Chattratchart, J. and Kuljis, J. (2002) 'Exploring the Effect of Control-Flow and Traversal Direction on VPL Usability for Novices', *Journal of Visual Languages & Computing*, 13(5), pp. 471–500. Available at: <https://doi.org/10.1006/jvlc.2002.0240>.
- Cheng, B.H.C. et al. (2009) 'Software Engineering for Self-Adaptive Systems: A Research Roadmap', in B.H.C. Cheng et al. (eds) *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 1–26. Available at: https://doi.org/10.1007/978-3-642-02161-9_1.
- Christi, A. and Groce, A. (2018) 'Target selection for test-based resource adaptation', in 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). IEEE, pp. 458–469. Available at: <https://doi.org/10.1109/QRS.2018.00059>.
- Christi, A., Groce, A. and Gopinath, R. (2017) 'Resource adaptation via test-based software minimization', in 2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). IEEE, pp. 61–70. Available at: <https://doi.org/10.1109/SASO.2017.15>.
- Christi, A., Groce, A. and Wellman, A. (2019) 'Building Resource Adaptations via Test-Based Software Minimization: Application, Challenges, and Opportunities', in 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, pp. 73–78. Available at: <https://doi.org/10.1109/ISSREW.2019.00046>.
- Chu, P.C. and Beasley, J.E. (1998) 'A Genetic Algorithm for the Multidimensional Knapsack Problem', *Journal of Heuristics*, 4(1), pp. 63–86. Available at: <https://doi.org/10.1023/A:1009642405419>.
- Ciccozzi, F. et al. (2017) 'Model-driven engineering for mission-critical iot systems', *IEEE software*, 34(1), pp. 46–53.
- Cleland-Huang, J. et al. (2022) 'Extending MAPE-K to support human-machine teaming', in *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '22: 17th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Pittsburgh Pennsylvania: ACM, pp. 120–131. Available at: <https://doi.org/10.1145/3524844.3528054>.

- Coles, Amanda et al. (2012) 'A Survey of the Seventh International Planning Competition', *AI Magazine*, 33(1), pp. 83–88. Available at: <https://doi.org/10.1609/aimag.v33i1.2392>.
- Coles, Amanda and Coles, Andrew (2014) 'PDDL+ Planning with Events and Linear Processes', *Proceedings of the International Conference on Automated Planning and Scheduling*, 24, pp. 74–82. Available at: <https://doi.org/10.1609/icaps.v24i1.13647>.
- Cormen, T.H. et al. (2009) *Introduction to algorithms*. MIT press.
- Damianou, N. et al. (2001) 'The ponder policy specification language', in *International Workshop on Policies for Distributed Systems and Networks*. Springer, pp. 18–38.
- Data.World (2016). Data.World, Inc. Available at: <https://data.world/>.
- David, P.-C. and Ledoux, T. (2003) 'Towards a framework for self-adaptive component-based applications', in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, pp. 1–14.
- Davidyuk, O., Ceberio, J. and Riekkki, J. (2007) 'An algorithm for task-based application composition', in *Proc. of the 11th IASTED International Conference on Software Engineering and Applications (SEA07)*, Cambridge, Massachusetts, USA. Citeseer.
- Davis, F.D. (1985) *A technology acceptance model for empirically testing new end-user information systems: Theory and results*. PhD Thesis. Massachusetts Institute of Technology.
- De Lemos, R. et al. (2013) 'Software engineering for self-adaptive systems: A second research roadmap', in *Software Engineering for Self-Adaptive Systems II*. Springer, pp. 1–32.
- Dürango, J. et al. (2014) 'Control-theoretical load-balancing for cloud applications with brownout', in *53rd IEEE Conference on Decision and Control*. IEEE, pp. 5320–5327.
- Easterbrook, S. et al. (2008) 'Selecting empirical methods for software engineering research', in *Guide to advanced empirical software engineering*. Springer, pp. 285–311.
- Efstratiou, C. et al. (2002) 'A platform supporting coordinated adaptation in mobile systems', in *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, pp. 128–137.
- ERP for Healthcare (2022) 'Best ERP Software for Healthcare'. ERP Research. Available at: <https://www.erpresearch.com/en-us/best-erp-for-healthcare>.
- Fowler, M. (2003) *UML distilled: a brief guide to the standard object modeling language*. 3rd ed. Boston: Addison-Wesley.

- Gajos, K. (2001) 'Rascal—a resource manager for multi agent systems in smart spaces', in International Workshop of Central and Eastern Europe on Multi-Agent Systems. Springer, pp. 111–120.
- Garg, V.K. and Venkitakrishnan, N. (2003) Enterprise Resource Planning: concepts and practice. PHI Learning Pvt. Ltd.
- Garlan, D. et al. (2002) 'Project aura: Toward distraction-free pervasive computing', IEEE Pervasive computing, 1(2), pp. 22–31.
- Garlan, D., Cheng, S.-W., et al. (2004) 'Rainbow: Architecture-based self-adaptation with reusable infrastructure', Computer, 37(10), pp. 46–54.
- Garlan, D., Poladian, V., et al. (2004) 'Task-based self-adaptation', in Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, pp. 54–57.
- Gawiejnowicz, S. (2020) 'A review of four decades of time-dependent scheduling: Main results, new topics, and open problems', Journal of Scheduling, 23(1), pp. 3–47.
- Genon, N., Heymans, P. and Amyot, D. (2011) 'Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation', in B. Malloy, S. Staab, and M. van den Brand (eds) Software Language Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg (Lecture Notes in Computer Science), pp. 377–396. Available at: https://doi.org/10.1007/978-3-642-19440-5_25.
- Ghodki, D. (2021) Medicine. Available at: <https://www.kaggle.com/datasets/deepalighodki/medicine-review>.
- Giese, M. et al. (2008) 'AMBOSS: a task modeling approach for safety-critical systems', in Engineering Interactive Systems. Springer, pp. 98–109.
- Gotz, S. et al. (2015) 'Adaptive Exchange of Distributed Partial Models@run.time for Highly Dynamic Systems', in 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Florence, Italy: IEEE, pp. 64–70. Available at: <https://doi.org/10.1109/SEAMS.2015.25>.
- Green, T.R.G. and Petre, M. (1996) 'Usability analysis of visual programming environments: a “cognitive dimensions” framework', Journal of Visual Languages & Computing, 7(2), pp. 131–174.
- Grohmann, J. et al. (2021) 'SARDE: A Framework for Continuous and Self-Adaptive Resource Demand Estimation', ACM Transactions on Autonomous and Adaptive Systems (TAAS), 15(2), pp. 1–31.
- Grus, J. (2015) Data science from scratch: first principles with Python. First edition. Sebastopol, CA: O'Reilly.

- Guerrero-García, J., González-Calleros, J. and Vanderdonckt, J. (2012) 'A Comparative Analysis of Task Modeling Notations.', *Acta Universitaria*, 22, pp. 90–97.
- Gupta, S. et al. (2022) 'Artificial intelligence for decision support systems in the field of operations research: review and future scope of research', *Annals of Operations Research*, 308(1–2), pp. 215–274. Available at: <https://doi.org/10.1007/s10479-020-03856-6>.
- Hallsteinsen, S. et al. (2008) 'Dynamic software product lines', *Computer*, 41(4), pp. 93–95.
- Hartson, H.R. and Gray, P.D. (1992) 'Temporal aspects of tasks in the user action notation', *Human-Computer Interaction*, 7(1), pp. 1–45.
- Hasan, M.S. et al. (2016) 'Enabling green energy awareness in interactive cloud application', in 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, pp. 414–422.
- Hoffmann, J. et al. (2006) 'Engineering Benchmarks for Planning: the Domains Used in the Deterministic Part of IPC-4', *Journal of Artificial Intelligence Research*, 26, pp. 453–541. Available at: <https://doi.org/10.1613/jair.1982>.
- Huang, A.-C. and Steenkiste, P. (2003) 'Network-sensitive service discovery', *Journal of grid computing*, 1(3), pp. 309–326.
- Huang, J. et al. (2017) 'UI driven Android application reduction', in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 286–296.
- Huber, N. et al. (2014) 'Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments', *Service Oriented Computing and Applications*, 8(1), pp. 73–89.
- Huber, N. et al. (2017) 'Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language', *IEEE Transactions on Software Engineering*, 43(5), pp. 432–452. Available at: <https://doi.org/10.1109/TSE.2016.2613863>.
- Hutzschenreuter, A.K., Bosman, P.A. and La Poutré, H. (2009) 'Evolutionary multiobjective optimization for dynamic hospital resource management', in *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, pp. 320–334.
- Hyndman, R.J. and Athanasopoulos, G. (2018) *Forecasting: principles and practice*. OTexts.
- IEEE (no date). IEEE. Available at: <https://ieee-dataport.org/>.
- ISO 9241 (2008) ISO 9241-12:1998 - Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs) -- Part 12: Presentation of information. Available at: <https://www.iso.org/standard/16884.html>.

- Johnson, H. and Hyde, J. (2003) 'Towards modeling individual and collaborative construction of jigsaws using task knowledge structures (TKS)', *ACM Transactions on Computer-Human Interaction (TOCHI)*, 10(4), pp. 339–387.
- Kaggle (2010). Kaggle, Inc. Available at: <https://www.kaggle.com/>.
- Keeney, J. and Cahill, V. (2003) 'Chisel: A policy-driven, context-aware, dynamic adaptation framework', in *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. IEEE, pp. 3–14.
- Kephart, J.O. and Chess, D.M. (2003) 'The vision of autonomic computing', *Computer*, 36(1), pp. 41–50. Available at: <https://doi.org/10.1109/MC.2003.1160055>.
- Khalid, H. et al. (2015) 'What Do Mobile App Users Complain About?', *IEEE Software*, 32(3), pp. 70–77. Available at: <https://doi.org/10.1109/MS.2014.50>.
- Kieras, D. (2004) 'GOMS Models for Task Analysis. The Handbook of Task Analysis for Human-Computer Interaction, Ed. Dan Diaper, Neville A. Stanton'. Lawrence Erlbaum Associates.
- Klein, C., Maggio, M., et al. (2014) 'Brownout: building more robust cloud applications', in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. the 36th International Conference, Hyderabad, India: ACM Press*, pp. 700–711. Available at: <https://doi.org/10.1145/2568225.2568227>.
- Klein, C., Papadopoulos, A.V., et al. (2014) 'Improving cloud service resilience using brownout-aware load-balancing', in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE, pp. 31–40.
- Kramer, J. and Magee, J. (2007) 'Self-managed systems: an architectural challenge', in *2007 Future of Software Engineering*. IEEE Computer Society, pp. 259–268.
- Krupitzer, C. et al. (2018) 'A survey on engineering approaches for self-adaptive systems (extended version)'.
- Kurp, P. (2008) 'Green computing', *Communications of the ACM*, 51(10), pp. 11–13.
- Larkin, K. et al. (2021) *Filters in ASP.NET Core*. Available at: <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-5.0>.
- Limbourg, Q. et al. (2004) 'USIXML: A language supporting multi-path development of user interfaces', in *IFIP International Conference on Engineering for Human-Computer Interaction*. Springer, pp. 200–220.

- Limbourg, Q. and Vanderdonckt, J. (2004) 'Comparing task models for user interface design', *The handbook of task analysis for human-computer interaction*, 6, pp. 135–154.
- Lu, Y. et al. (2016) 'RVLBPNN: A workload forecasting model for smart cloud computing', *Scientific Programming*, 2016.
- Lucas, W.T., Xu, J. and Babaian, T. (2013) 'Visualizing ERP Usage Logs in Real Time.', in *ICEIS* (3), pp. 83–90.
- Lust, T. and Teghem, J. (2012) 'The multiobjective multidimensional knapsack problem: a survey and a new approach', *International Transactions in Operational Research*, 19(4), pp. 495–520. Available at: <https://doi.org/10.1111/j.1475-3995.2011.00840.x>.
- Maggio, M., Klein, C. and Arzén, K.-E. (2014) 'Control strategies for predictable brownouts in cloud computing', *IFAC proceedings volumes*, 47(3), pp. 689–694.
- Mahfoudhi, A., Abed, M. and Tabary, D. (2001) 'From the formal specifications of users tasks to the automatic generation of the HCI specifications', in *People and Computers XV—Interaction without Frontiers*. Springer, pp. 331–347.
- Martin, R.C. (2009) *Clean code: a handbook of agile software craftsmanship*. Pearson Education.
- Martinie, C. et al. (2019) 'Analysing and demonstrating tool-supported customizable task notations', *Proceedings of the ACM on human-computer interaction*, 3(EICS), pp. 1–26.
- Martinie, C., Palanque, P. and Winckler, M. (2011) 'Structuring and composition mechanisms to address scalability issues in task models', in *IFIP Conference on Human-Computer Interaction*. Springer, pp. 589–609.
- Mason, R. (2019) 'Developing a Profitable Online Grocery Logistics Business: Exploring Innovations in Ordering, Fulfilment, and Distribution at Ocado', in *Contemporary Operations and Logistics*. Springer, pp. 365–383.
- McKinley, P.K. et al. (2004) 'Composing adaptive software', *Computer*, 37(7), pp. 56–64.
- Mendeley Data (2013). Elsevier. Available at: <https://data.mendeley.com/>.
- ML.NET Framework (2021). Available at: <https://dotnet.microsoft.com/en-us/apps/machinelearning-ai/ml-dotnet>.
- Mohan, J., Lanka, K. and Rao, A.N. (2019) 'A review of dynamic job shop scheduling techniques', *Procedia Manufacturing*, 30, pp. 34–39.

- Molina, A.I. et al. (2014) 'Evaluating a graphical notation for modeling collaborative learning activities: A family of experiments', *Science of Computer Programming*, 88, pp. 54–81.
- Moody, D. (2009) 'The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering', *IEEE Transactions on software engineering*, 35(6), pp. 756–779.
- Moreno, G.A. et al. (2015) 'Proactive self-adaptation under uncertainty: a probabilistic model checking approach', in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pp. 1–12.
- Mori, G., Paternò, F. and Santoro, C. (2002) 'CTTE: support for developing and analyzing task models for interactive system design', *IEEE Transactions on software engineering*, 28(8), pp. 797–813.
- Mota, B. et al. (2020) 'Production line dataset for task scheduling and energy optimization - Schedule Optimization'. Zenodo. Available at: <https://doi.org/10.5281/ZENODO.4106746>.
- Nah, F.F.-H. (2004) 'A study on tolerable waiting time: how long are web users willing to wait?', *Behaviour & Information Technology*, 23(3), pp. 153–163.
- NBomber (2021). Available at: <https://nbomber.com/>.
- Neale, J. (2016) 'Iterative categorization (IC): a systematic technique for analysing qualitative data: Systematic technique for analysing qualitative data', *Addiction*, 111(6), pp. 1096–1106. Available at: <https://doi.org/10.1111/add.13314>.
- NHS (2019) A Guide to Managing Medicines Supply and Shortages. Available at: <https://www.england.nhs.uk/publication/a-guide-to-managing-medicines-supply-and-shortages/>.
- Nikolov, V. et al. (2014) 'Cloudfarm: An elastic cloud platform with flexible and adaptive resource management', in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE, pp. 547–553.
- Ocado Solutions (2018). Available at: <https://www.youtube.com/watch?v=hdyHnak61OI>.
- Oh, S. and Wildemuth, B. (2009) 'Think-aloud protocols', *Applications of social research methods to questions in information and library science*, pp. 178–188.
- Oz, E. (2009) *Management information systems*. 6th ed. Boston, Mass: Thomson/Course Technology.
- Pandey, A. et al. (2016) 'Hybrid planning for decision making in self-adaptive systems', in *2016 IEEE 10th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, pp. 130–139.

- Papakos, P., Capra, L. and Rosenblum, D.S. (2010) 'Volare: context-aware adaptive cloud service discovery for mobile systems', in *Proceedings of the 9th International Workshop on Adaptive and Reflective Middleware*, pp. 32–38.
- Pascual, G.G., Pinto, M. and Fuentes, L. (2015) 'Self-adaptation of mobile systems driven by the common variability language', *Future Generation Computer Systems*, 47, pp. 127–144. Available at: <https://doi.org/10.1016/j.future.2014.08.015>.
- Patel, K.K., Patel, S.M., and others (2016) 'Internet of things-IOT: definition, characteristics, architecture, enabling technologies, application & future challenges', *International journal of engineering science and computing*, 6(5).
- Paterno, F., Mancini, C. and Meniconi, S. (1997) 'ConcurTaskTrees: A diagrammatic notation for specifying task models', in *Human-computer interaction INTERACT'97*. Springer, pp. 362–369.
- Perttunen, M., Jurmu, M. and Riekk, J. (2007) 'A QoS model for task-based service composition', in *Proc. 4th International Workshop on Managing Ubiquitous Communications and Services*, p. 11.
- Pinedo, M. (2018) *Scheduling: theory, algorithms, and systems*. Springer.
- Pleuss, A., Botterweck, G. and Dhungana, D. (2010) 'Integrating automated product derivation and individual user interface design'.
- Preece, J., Rogers, Y. and Sharp, H. (2015) *Interaction design: beyond human-computer interaction*. 4. ed. Chichester: Wiley.
- Pressman, R.S. (2010) *Software engineering: a practitioners approach*. 7. ed., alternate ed. Boston, Mass. [u: McGraw-Hill Higher Education.
- Raunak, M.S. and Osterweil, L.J. (2013) 'Resource Management for Complex, Dynamic Environments', *IEEE Transactions on Software Engineering*, 39(3), pp. 384–402. Available at: <https://doi.org/10.1109/TSE.2012.31>.
- Redmond, B. and Cahill, V. (2000) 'Iguana/J: Towards a dynamic and efficient reflective architecture for Java', in *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*.
- Rigole, P. et al. (2007) 'Task-driven automated component deployment for ambient intelligence environments', *Pervasive and Mobile Computing*, 3(3), pp. 276–299.
- Salehie, M. and Tahvildari, L. (2009) 'Self-adaptive software: Landscape and research challenges', *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), pp. 1–42. Available at: <https://doi.org/10.1145/1516533.1516538>.
- Saller, K., Lochau, M. and Reimund, I. (2013) 'Context-aware DSPLs: model-based runtime adaptation for resource-constrained systems', in *Proceedings*

- of the 17th International Software Product Line Conference co-located workshops, pp. 106–113.
- Samin, H., Bencomo, N. and Sawyer, P. (2022) ‘Decision-making under uncertainty: be aware of your priorities’, *Software and Systems Modeling*, pp. 1–30.
- Scapin, D. and Pierret-Golbreich, C. (1989) ‘Towards a method for task description: MAD’, *Work with display units*, 89, pp. 371–380.
- Sebillotte, S. (1988) ‘Hierarchical planning as method for task analysis: the example of office task analysis’, *Behaviour & Information Technology*, 7(3), pp. 275–293. Available at: <https://doi.org/10.1080/01449298808901878>.
- Shao, Y. et al. (2014) ‘Towards a scalable resource-driven approach for detecting repackaged android applications’, in *Proceedings of the 30th Annual Computer Security Applications Conference*, pp. 56–65.
- Shaw, M. (2002) ‘What makes good research in software engineering?’, *International Journal on Software Tools for Technology Transfer*, 4(1), pp. 1–7.
- Shoval, P. and Shiran, S. (1997) ‘Entity-relationship and object-oriented data modeling — An experimental comparison of design quality’, *Data & Knowledge Engineering*, 21(3), pp. 297–315. Available at: [https://doi.org/10.1016/S0169-023X\(97\)88935-5](https://doi.org/10.1016/S0169-023X(97)88935-5).
- Sousa, J.P. et al. (2006) ‘Task-based adaptation for ubiquitous computing’, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(3), pp. 328–340. Available at: <https://doi.org/10.1109/TSMCC.2006.871588>.
- Strusevich, V.A. and Rustogi, K. (2017) *Scheduling with Time-Changing Effects and Rate-Modifying Activities*. 1st ed. 2017. Cham: Springer International Publishing: Imprint: Springer (International Series in Operations Research & Management Science, 243). Available at: <https://doi.org/10.1007/978-3-319-39574-6>.
- Sun, Y., Cai, X. and Loparo, K.A. (2019) ‘Learning-based Adaptation Framework for Elastic Software Systems.’, in *SEKE*, pp. 281–372. Available at: <https://doi.org/10.18293/SEKE2019-00>.
- Tarby, J.-C. and Barthet, M.-F. (1996) ‘The DIANE+ Method.’, in *CADUI*, pp. 95–119.
- Tomás, L. et al. (2014) ‘The straw that broke the camel’s back: safe cloud overbooking with application brownout’, in *2014 International Conference on Cloud and Autonomic Computing*. IEEE, pp. 151–160.
- UK Statistics Authority (1996) UK Statistics Authority. Available at: <https://www.ons.gov.uk/>.

- US Department of Commerce (1902) US Department of Commerce. Available at: <https://www.census.gov/>.
- Van Der Veer, G.C., Lenting, B.F. and Bergevoet, B.A. (1996) 'GTA: Groupware task analysis—Modeling complexity', *Acta psychologica*, 91(3), pp. 297–322.
- Van Hoorn, A. et al. (2009) 'An adaptation framework enabling resource-efficient operation of software systems', in *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, pp. 41–44.
- Vidani, A.C. and Chittaro, L. (2009) 'Using a task modeling formalism in the design of serious games for emergency medical procedures', in *2009 Conference in Games and Virtual Worlds for Serious Applications*. IEEE, pp. 95–102.
- Vigo, M., Santoro, C. and Paternò, F. (2017) 'The usability of task modeling tools', in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, pp. 95–99.
- Viswanathan, L., Jindal, A. and Karanasos, K. (2018) 'Query and resource optimization: Bridging the gap', in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, pp. 1384–1387. Available at: <https://doi.org/10.1109/ICDE.2018.00156>.
- Webster, M. (2022) Definition of hangtag. Available at: <https://www.merriam-webster.com/dictionary/hangtag>.
- van Welie, M., van der Veer, G.C. and Eliëns, A. (1998) 'An Ontology for Task World Models', in P. Markopoulos and P. Johnson (eds) *Design, Specification and Verification of Interactive Systems '98*. Vienna: Springer Vienna (Eurographics), pp. 57–70. Available at: https://doi.org/10.1007/978-3-7091-3693-5_5.
- Wilkins, D.E. (2014) *Practical planning: extending the classical AI planning paradigm*. Elsevier.
- Wu, C.-L. and Fu, L.-C. (2011) 'Design and realization of a framework for human–system interaction in smart homes', *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 42(1), pp. 15–31.
- Xu, M. and Buyya, R. (2019) 'Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions', *ACM Computing Surveys (CSUR)*, 52(1), pp. 1–27.
- Xu, M., Dastjerdi, A.V. and Buyya, R. (2016) 'Energy efficient scheduling of cloud application components with brownout', *IEEE Transactions on Sustainable Computing*, 1(2), pp. 40–53.
- Yan, S. et al. (2019) 'Conf-Adaption: Adaptive Adjustment of Software Configuration On UAV by Resource Dependency Analysis', in *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence*

- Conference (ITAIC). IEEE, pp. 155–161. Available at: <https://doi.org/10.1109/ITAIC.2019.8785785>.
- Zhan, Z.-H. et al. (2015) ‘Cloud computing resource scheduling and a survey of its evolutionary approaches’, *ACM Computing Surveys (CSUR)*, 47(4), pp. 1–33.
- Zhao, T. et al. (2017) ‘A reinforcement learning-based framework for the generation and evolution of adaptation rules’, in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, pp. 103–112.
- Zhou, L., Zhang, L. and Horn, B.K.P. (2020) ‘Deep reinforcement learning-based dynamic scheduling in smart manufacturing’, *Procedia CIRP*, 93, pp. 383–388. Available at: <https://doi.org/10.1016/j.procir.2020.05.163>.

Appendices

A

Artefacts from the evaluation of SERIES

This appendix presents the requirements and task models used in the user study from the manufacturing and surveillance domains, as well as the questionnaire.

A.1 Manufacturing Domain: Requirements and Task Models

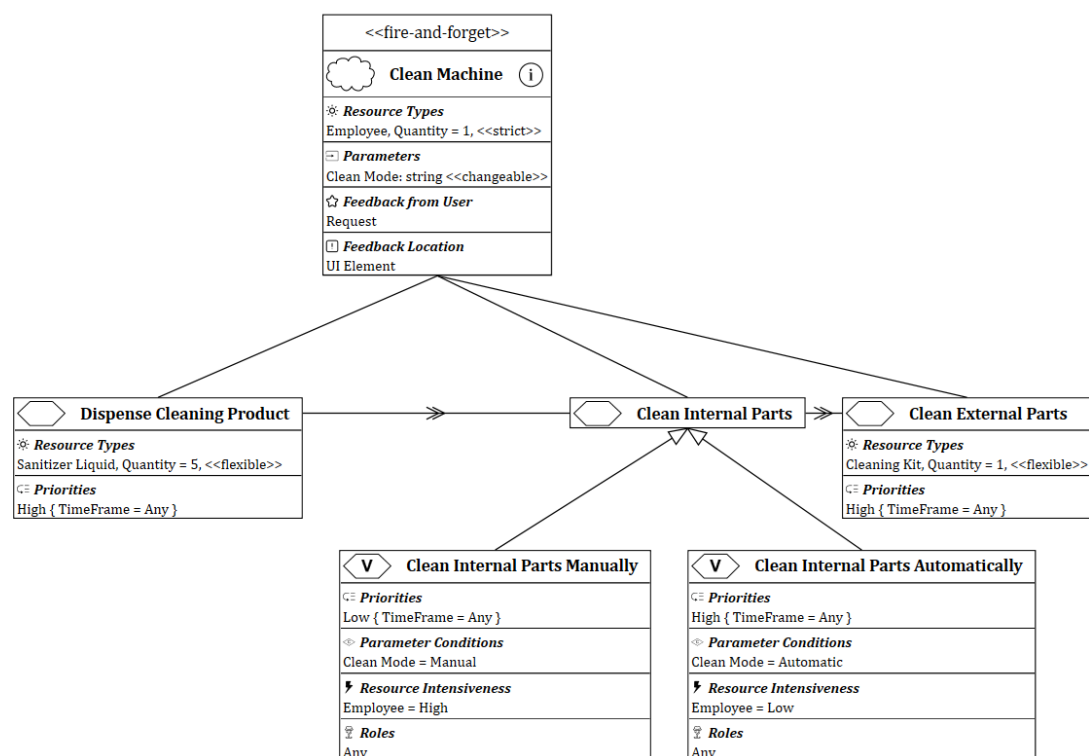


Figure A.1 – Task model for explanation (manufacturing)

(1) Add an abstract task named "Manufacture Food Product". This task has two parameters: a "Product Quantity" that is non-changeable and a "Marking Type" that is changeable. It also requires "Ingredients" resource type that is "flexible" with a quantity of 10. For this task, feedback from the user shall be requested and the feedback location shall be a panel.

(2) Add three application tasks as subtasks of "Manufacture Food Product". These three subtasks are named "Mix Ingredients", "Add Flavour", and "Mark Expiration Date" respectively.

"Mix Ingredients" requires a "Mixer" resource type that is "strict" with a quantity of 1.

"Add Flavour" has the following description: "Set a flavour for the food product". This task requires a "Product Flavour" resource type that is "flexible" with a quantity of 1.

"Mark Expiration Date" requires a "Marking Machine" resource type that is "strict" with a quantity of 1. This task has a high priority at any time frame.

(3) Add two application task variants for "Mix Ingredients".

- The first variant is named "Fast Mix" and has a high priority and a parameter condition that specifies "Product Quantity > 100", and a high resource intensiveness for the "Mixer" resource type.
- The second variant is named "Slow Mix" and has a low priority and a parameter condition that specifies "Product Quantity <= 100", and a low resource intensiveness for the "Mixer" resource type.

Add two application task variants for "Mark Expiration Date".

- The first task variant is named "Mark as Stamp" and has a low priority, a parameter condition that specifies "Marking Type=Stamp", and a high resource intensiveness for the "Marking Machine" resource type.
- The second task variant is named "Mark as Sticker" and has a high priority, a parameter condition that specifies "Marking Type=Sticker", and a low resource intensiveness for the "Marking Machine" resource type.
- Both task variants have a "Role" that is equal to "Any".

Figure A.2 – Requirements to create a task model (manufacturing)

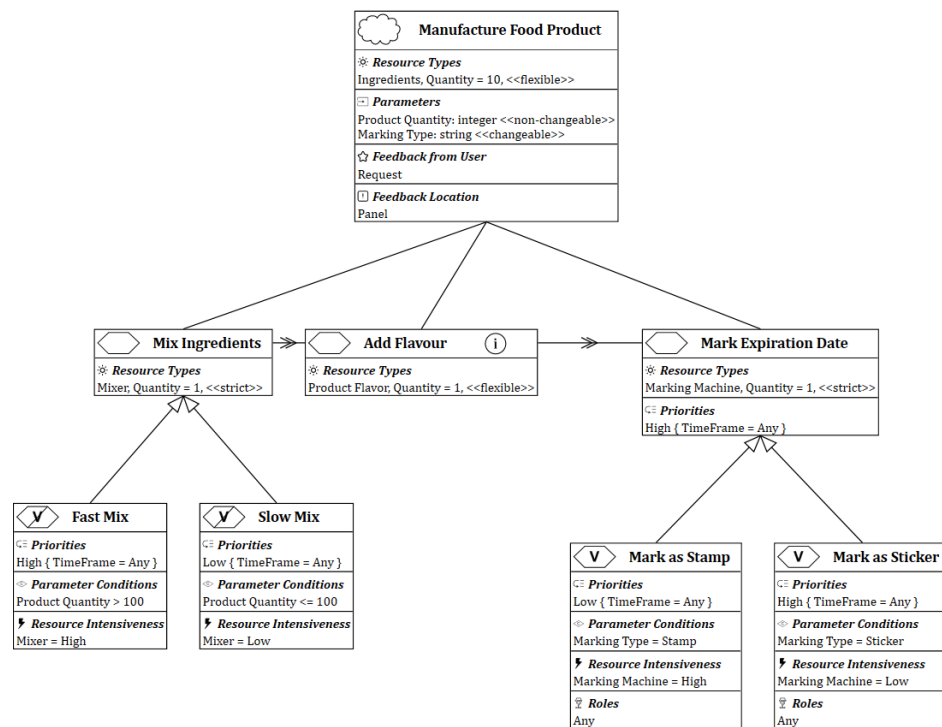


Figure A.3 – Expected task model based on requirements (manufacturing)

A.2 Surveillance Domain: Requirements and Task Models

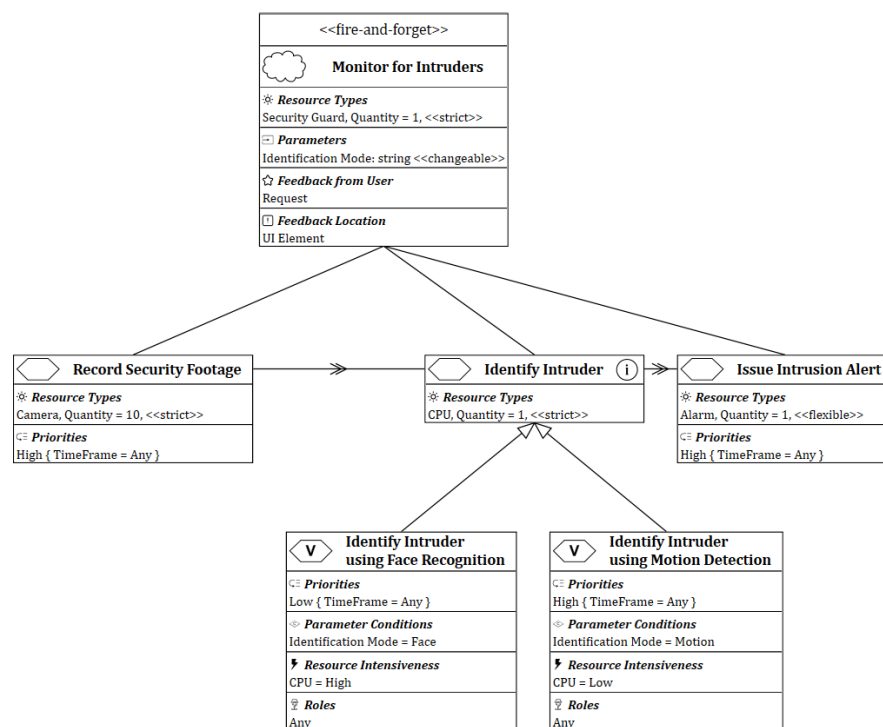


Figure A.4 – Task model for explanation (surveillance)

(1) Add an abstract task named "Monitor Area". This task has two parameters: a "Robot Type" that is non-changeable and a "Video Quality" that is changeable. It also requires "Camera" resource type that is "strict" with a quantity of 10. For this task, feedback from the user shall be requested and the feedback location shall be a panel.

(2) Add three application tasks as subtasks of "Monitor Area". These three subtasks are named "Set Surveillance Type", "Set Footage Quality", and "Record Footage" respectively.

"Set Surveillance Type" requires a "Robot" resource type that is "strict" with a quantity of 10.

"Set Footage Quality" has the following description: "HD or SD footage quality". This task requires a "Battery" resource type that is "strict" with a quantity of 1.

"Record Footage" requires a "Bandwidth" resource type that is "strict". This task has a high priority at any time frame.

(3) Add two application task variants for "Set Surveillance Type".

- The first variant is named "Set to Aerial Surveillance" and has a high priority and a parameter condition that specifies "Robot Type=Drone".
- The second variant is named "Set to Ground Operation" and has a low priority and a parameter condition that specifies "Robot Type=Driver Bot".

Add two application task variants for "Set Footage Quality".

- The first task variant is named "Set to HD Quality" and has a high priority, a parameter condition that specifies "Video Quality=HD", and a high resource intensiveness for the "Battery" resource type.
- The second task variant is named "Set to SD Quality" and has a low priority, a parameter condition that specifies "Video Quality=SD", and a low resource intensiveness for the "Battery" resource type.
- Both task variants have a "Role" that is equal to "Any".

Figure A.5 – Requirements to create a task model (surveillance)

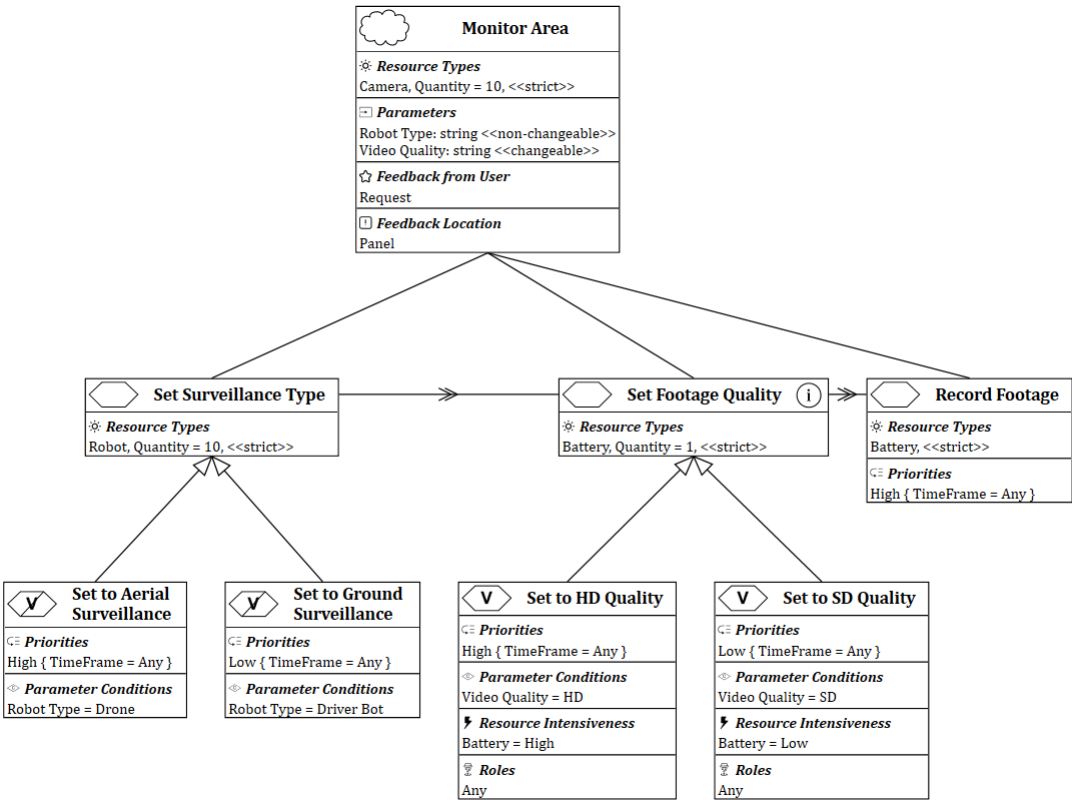


Figure A.6 – Expected task model based on requirements (surveillance)

Do you have any comments regarding ease of use?

2.2. Please choose **three** out of the following words to describe SERIES. The words that you choose may be positive, negative, or a combination of both.

<input type="checkbox"/> Appealing	<input type="checkbox"/> Confusing
<input type="checkbox"/> Easy to use	<input type="checkbox"/> Difficult
<input type="checkbox"/> Consistent	<input type="checkbox"/> Hard to use
<input type="checkbox"/> Clear	<input type="checkbox"/> Inconsistent
<input type="checkbox"/> Familiar	<input type="checkbox"/> Intimidating
<input type="checkbox"/> Friendly	<input type="checkbox"/> Overwhelming
<input type="checkbox"/> Straight Forward	<input type="checkbox"/> Rigid
<input type="checkbox"/> Understandable	<input type="checkbox"/> Incomprehensible

Please clarify your choice of terms and mention any suggested improvements that you may have.

3. Would you like to make any final comments?

B

Artefacts from the evaluation of SPARK

This appendix presents the remaining cases (40%, 60%, and 80%) of task criticality and their four cases of resource variability, for the two case studies from Chapter 8.

B.1 Case Study 1: Medicine Consumption System

Table B.1 – Case Study 1: the case where 40% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 314 (40%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20% = 652	1,673 not facing variability	303	71,369
	406 facing variability	188	18,736
40% = 1,304	1,252 not facing variability	288	52,140
	827 facing variability	246	37,965
60% = 1,956	842 not facing variability	253	35,100
	1,237 facing variability	277	55,005
80% = 2,608	420 not facing variability	180	16,222
	1,659 facing variability	301	73,883

Table B.2 – Case Study 1: the case where 60% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 471 (60%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20% = 652	2,114 not facing variability	448	90,828
	524 facing variability	279	22,860
40% = 1,304	1,590 not facing variability	421	66,073
	1,048 facing variability	365	47,615
60% = 1,956	1,068 not facing variability	371	44,334
	1,570 facing variability	415	69,354
80% = 2,608	539 not facing variability	270	21,534
	2,099 facing variability	452	92,154

Table B.3 – Case Study 1: the case where 80% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 628 (80%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20% = 652	2,407 not facing variability	599	106,799
	611 facing variability	377	26,825
40% = 1,304	1,804 not facing variability	562	79,150
	1,214 facing variability	499	54,474
60% = 1,956	1,208 not facing variability	497	53,179
	1,810 facing variability	559	80,445
80% = 2,608	610 not facing variability	365	25,296
	2,408 facing variability	605	108,328

B.2 Case Study 2: Manufacturing System

Table B.4 – Case Study 2: the case where 40% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 3 (40%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20%	<i>Not facing variability:</i> 7 depletable and 2 reusable	3	75
	<i>Facing variability:</i> 3 depletable and 1 reusable	3	62
40%	<i>Not facing variability:</i> 5 depletable and 2 reusable	3	52
	<i>Facing variability:</i> 5 depletable and 1 reusable	3	85
60%	<i>Not facing variability:</i> 4 depletable and 1 reusable	3	47
	<i>Facing variability:</i> 6 depletable and 2 reusable	3	90
80%	<i>Not facing variability:</i> 2 depletable and 1 reusable	3	30
	<i>Facing variability:</i> 8 depletable and 2 reusable	3	107

Table B.5 – Case Study 2: the case where 60% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 4 (60%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20%	<i>Not facing variability:</i> 6 depletable and 2 reusable	4	115
	<i>Facing variability:</i> 3 depletable and 1 reusable	4	88
40%	<i>Not facing variability:</i> 6 depletable and 2 reusable	4	82
	<i>Facing variability:</i> 6 depletable and 1 reusable	4	121
60%	<i>Not facing variability:</i> 4 depletable and 1 reusable	4	57
	<i>Facing variability:</i> 8 depletable and 2 reusable	4	146
80%	<i>Not facing variability:</i> 2 depletable and 1 reusable	4	40
	<i>Facing variability:</i> 10 depletable and 2 reusable	4	163

Table B.6 – Case Study 2: the case where 80% of tasks (variants) are critical (number of critical task requests and their corresponding number of resource types)

Critical Tasks (variants) = 6 (80%)			
Percentage of Resource Types facing variability	Number of Resource Types facing and not facing variability	Critical Tasks (variants)	Critical Task Requests
20%	<i>Not facing variability:</i> 10 depletable and 2 reusable	6	126
	<i>Facing variability:</i> 3 depletable and 1 reusable	4	88
40%	<i>Not facing variability:</i> 7 depletable and 2 reusable	6	93
	<i>Facing variability:</i> 6 depletable and 1 reusable	4	121
60%	<i>Not facing variability:</i> 5 depletable and 1 reusable	6	68
	<i>Facing variability:</i> 8 depletable and 2 reusable	4	146
80%	<i>Not facing variability:</i> 2 depletable and 1 reusable	5	45
	<i>Facing variability:</i> 11 depletable and 2 reusable	6	169