OPTIMAL MODELING LANGUAGE AND FRAMEWORK FOR SCHEDULABLE SYSTEMS

Güner Orhan

October 31, 2019

OPTIMAL MODELING LANGUAGE AND FRAMEWORK FOR SCHEDULABLE SYSTEMS

DISSERTATION

to obtain
the degree of doctor at the Universiteit Twente,
on the authority of the rector magnificus,
Prof.dr. T.T.M. Palstra,
on account of the decision of the graduation committee
to be publicly defended
on Thursday 31 October 2019 at 16.45 uur

by
Güner Orhan

born on 25^{th} of July, 1989
in Ankara, Turkey

This dissertation has been approved by:

Supervisor:
Prof. dr. M. Aksit

This is a cooperation framework between Aselsan and the university of Twente. The framework actually consists of a set of individual projects, which are carried out concurrently and cooperatively by a PhD student. The project PLOS proposes a productline architecture for designing optimal schedulers for the digital receivers that takes care of application semantics in scheduling, can cope with dynamically changing context, can deal with variations in scheduling objectives, optimizes the scheduling criteria and causes an acceptable overhead. The productline approach enables to effectively reuse the basic building elements of the scheduler asset base in different application settings.

aselsan

UNIVERSITY OF TWENTE.

Typeset with LaTeX
Cover design: The image is retrieved from [101]
Printed by: GildePrint
ISBN: 978-90-365-4873-1
DOI: 10.3990/1.9789036548731
Available online at https://doi.org/10.3990/1.9789036548731

Graduation Committee:

Chairman / Secretary:	prof.dr. J.N. Kok
Supervisor:	prof.dr.ir. M. Aksit
Committee Members:	prof.dr.ir. M. Aksit
	prof.dr.ir. G.J.M. Smit
	dr. L. Ferreira Pires
	prof. dr. ir. B. Tekinerdogan
	prof. dr. C. De Roover
	prof. dr. A. Dogru
	dr. M. Dursun

Acknowledgements

The journey from an idea to the thesis begins with difficulties such as leaving hometown and family for the first time, traveling to not only another city, but another country. That's one small step for mankind, but it was one giant leap for me. In this journey, I have accumulated lots of unforgettable memories. Thanks to some of my Turkish friends, colleagues, and especially our group secretary Ida, for showing me their generous hospitality, I could manage to finalize this activity.

First of all, I would like to show my gratitude to my supervisor Mehmet Akşit. From the beginning of my journey to the Netherlands, he has always behaved me like my elder family member. One of the main reasons, which avoids me giving up this study, is his warm and kind attitude towards me. Like most Turkish students, I have had so much trouble with scientific writing in English due to grammatical differences between Turkish and English. Nevertheless, he has never left me alone. He has sat next to me and helped me to examine each sentence word-by-word. I can never forget these moments throughout my life. We have so much memories with him, but there is one that I will remember. When I heard that my mother had got cancer, the whole world fell upon me. He has sat in front of me for at least two hours and convinced me that everything would be fine, and yes. He was right. My mother is getting better each day. Again, I would like to thank him for being such a wonderful person and for every word that I have learned from him. I will always be in touch with him throughout my life. In addition, Mustafa Dursun is one of the greatest person and team leader in my company. He was my mentor and coordinate everything between the company and the doctorate program. It is impossible to express my feeling for him. He was such a wonderful brain and big-hearted person. If I had a brother, he would definitely be him. I wish the best for him and his family. As my second mentor, I would like to thank Cihan for sharing his wide knowledge with me. I wish one day I would be "the guru of radar systems" like him. It is impossible to forget two of the most significant supporter of this program. They are Hakime Koç and Gürsel Şahin, who are my manager and director in my company, respectively. They made the bureaucratic processes much easier for me. I would like to thank them for all.

Secondly, I would like to mention about my parents. Although we have stayed apart from each other, we have been as close as one-telephone distance. Almost each day, I have called my parents via video call. Although it is not the same like sitting and chatting together in the same place, this is the only way that I could talk and see them from more than 3000

km distance away. Before coming to the Netherlands, I had always lived together with my family. This was really difficult for me once I started to live here. At the end of the first year, I got used to live by myself. However, nothing decreased my longing towards my parents. They have always supported me for each decision that I had made either correct or wrong. They have always showed me what are ethical and correct and what are not. I always feel myself lucky for having such wonderful and extraordinary parents. Thank you, my parents! If I am standing here and getting this degree, it is just because of them. It is impossible to complete this part without mentioning about my grandpa. He was my first mentor in primary school. Thanks to him, he made me like Math and Science. Now, He is 96. I am afraid of even the feeling of losing him someday. He is not different than my parents. Sometimes, he meant more than my parents in my life. He encouraged me to get this degree, and for this reason, I will show my respect to him until the death tears us apart. Although he may not be with me, I always believe that he will eternally be with me and watch me above the clouds.

At the end of the third year, someone entered my life. At that time, I was not aware that she would be the center of my life. At the beginning, my first impression was she had been talking too much compared to me (I do not want to be misunderstood. She never lost anything from being a talkative person :)). After knowing her better and understanding her unique way of thinking, I really like her a lot and started to think that she was the right person with whom I can live all my life. Unlike me, she is always cheerful and funny, and has a really big heart. She was the childish person who I have looked inside of me to find all the time, but could never find. I do believe that couples should complete the missing parts of each other for great and eternal happiness. Therefore, we got engaged on 22nd of July, 2019. Now, I think that it was the one of the correct decisions that I have made for the whole life. In near future, I will definitely take one step further and marry her. I am not a romantic guy that she has expected from me. Maybe, it is because I am a computer engineer, but I can promise one thing with my heart: I do and always will love her. I hope we will accumulate so much happy and unforgettable memories together. I love you so much!

Let's talk about my Turkish friends in the Netherlands. Maybe, I should call them as "Turkish mafia" as other colleagues call. I would like to say thank them by saying their name in my thesis. I would like to say at least one word for each of my friend. Muharrem (başgan) is the leader of our Turkish mafia :). You are really kind and helpful. As one of my paranymph, Akın is the one of the wonderful person I have met here. The only thing that I can say is he is and always will be more than just a friend for me. As the second paranymph, Oguzhan (Ramazan) is my ahiretlik (I don't know an English word for it). We have shared too many things (I do not want to talk about everything that we shared. I think this is not the right place :)). For instance, we have moved lots of houses and made constructions with his songs from Black sea district of Turkey. We have played basketball, swim, walked, talked, eat together. I will never forget each moment that we had together. Devrim (hoca) is the lecturer at UT. We had many fruitful political discussion. To be honest, I always envy on his romantic part. He knows how to touch the hearth of the girls. I will really miss the

days when we played basketball together. Deniz! You are such a perfect person that your ex-supervisor has never deserved. I wish you will find the life that you have dreamed of in Italy. Seda and I will visit you in your new home as soon as possible. Moreover, I would like to thank you all: Burcu, Cihan, Yiğitcan, Pelin, Derya Demirtaş, Derya Ataç, Atıl, Armağan and Nurcan. Thank you all for all the memories.

Finally, I would like to thank to all of my colleagues in my group, Formal Methods and Tools. I would like to mention about some of them specifically. Tom, Jeroen and Stefano were my office mates that I came here. We had so many discussion in various topics. Sir Marcus! I remember the funny conversations we had in the corridor in front of the coffee machine. Our secretary Ida, I cannot forget her sudden entrance to my office while she is talking. You helped me a lot to proceed my group duty as a hardware guy. In addition, I would like to thank to my ex-daily supervisor Pim and ex-co-supervisor Arend for helping me in the beginning of this journey.

I may forget to mention about the ones that our ways were crossed. Please accept my apologizes for not giving your name in this section.

To my beloved family and my fiancée..

Abstract

Scheduling processes have been applied to a wide range of application areas, such as scheduling tasks in operating systems, scheduling facilities at airports, scheduling assembly lines in production, scheduling resources in project management, timetabling in public transportation, and scheduling activities in cyber-physical systems.

In general, scheduling problems are not trivial to solve effectively and efficiently. Design and implementation of scheduling software is expensive and time-consuming. This dissertation makes three main contributions:

First, we have adopted a feature-oriented Software Product Line Engineering (SPLE) approach. If applied correctly, the SPLE approach provides more economic solutions in case a family of products is developed instead of one product. Companies that develop scheduling systems generally implement product families. After an extensive domain analysis of the theory, the common and variable "features" have been defined. By choosing and configuring the right set of "features", the designers can efficiently define scheduling systems according to their needs.

As a second contribution, to convert the abstract definitions into executable programs, we have designed and implemented an "application framework" called First Scheduling Framework (FSF). We consider reusability and dynamic extensibility as two quality attributes of the framework. To this end, generic "constraint-solvers" are adopted to translate the predefined "scheduling constraints" into the "constraint-language" of the corresponding solver and to use them to solve the translated problem. We have extended our implementation using MDE (Model-Driven Engineering) techniques so that "feature-oriented" models can be easily converted to the abstractions of the "application framework".

As a third contribution, we have expanded our "feature-oriented" approach to a general "model optimization framework". Scheduling systems can be influenced by many contextual factors, such as the desire for lower energy consumption, more precise computation, and dealing with certain hardware limitations. Due to contextual factors, it can be very difficult to define an optimal system that gives the best compromise for such multiple concerns. For this purpose, we propose OptML Framework, which accepts different models representing the contextual factors in the language of ECORE in the MDE environment and calculates the optimal models that meet user-defined requirements.

Contents

CHAPTER 1

Introduction

1.1 Application of Software to Products and Businesses

Software today is applied to a large category of products. Almost all products contain software or are produced through processes controlled by software [4]. From business perspective, the main motivation of adopting software is to increase *efficiency* and *effectiveness* [107]. Efficiency is defined as to do more work per unit of money, and effectiveness is defined as getting closer to the business objectives. Within this context, one can consider, for example, two kinds of businesses: i) Software development for business; and ii) Business for software development [6]. The first one refers to a business where software is deployed. The second one refers to a business where software is developed.

Due to a large variety of products and processes, software systems are in general very diverse. Depending on the requirements and domain of application, software systems can be also very complex. Complexity is defined as "the state of being hard to separate, analyze and/or solve" [97]. In Computer Science literature, the term complexity refers to specific challenges related to the different phases of software development. Examples are *problem complexity*, *model complexity*, *structural complexity* and *algorithmic complexity*[40, 45]. Naturally, complexity can negatively influence the desired quality attributes of software systems. Historically, the term *software crisis* has been used to denote a large category of challenges associated with developing software systems [102].

1.2 Functional Correctness, Timeliness, Reusability and Evolvability of Software

The term quality is defined as "the degree to which a set of inherent characteristics fulfills requirements" [75].

Consider the definitions of the following quality attributes:

- *Functional Correctness.* "Degree to which a product or system provides the correct results with the needed degree of precision." [74]
- *Timeliness.* "The ability of a software system to complete its execution in the specified time."
- *Reusability.* "Degree to which an asset can be used in more than one system, or in building other assets." [74]
- *Evolvability.* "The ability of a software system to adapt in response to changes in its environment, requirements and technologies that may have impact on the software system in terms of structural and/or functional requirements, while still taking the architectural integrity into consideration." [21]

If the corresponding requirement specification supports the desired business objectives, a correctly implemented system will be *effective* as desired. Although functional correctness is the essential property of every software system, other quality attributes such as *timeliness*, *reusability* and *evolvability* can be considered equally important.

In practice, if a system does not satisfy its timeliness requirement, it can, for example, lead to unsatisfied users. Even worse, in safety-critical applications, it can result in catastrophic outcomes, such as damage of properties or loss of life. In the literature, to express the combined effect of the quality attributes correctness and timeliness, the terms *soft real-time* and hard real-time are used:

- *Soft Real-time System.* The system in which a missed deadline does not result in system failure, but the effectiveness can drop in proportion to the delay, depending on the application. [91]
- *Hard Real-time System.* "The system in which the failure of a component to meet its timing deadline can result in an unacceptable failure of the whole system" [73]

The quality attributes *reusability* and *evolvability* are important to decrease the costs of software development. Instead of designing each software system from scratch, providing reusable software libraries, for example, can help decreasing the costs considerably. In addition, reuse of well-proven code can also help assuring software correctness. Similarly, since software requirements continuously change, evolvability can help creating new versions of software in a cost-effective way.

In the software-engineering literature, definitions of these terms are specialized depending on the artifacts of software development. For example, the term correctness may be associated with requirement specifications, models of software, algorithms and/or programs [124]. Similarly, the term reusability can be specialized as model reusability, code reusability, etc. The term evolvability can be classified, for example, as static program evolvability or run time evolvability.

1.3 Software Engineering Methods and Techniques

In software engineering literature, a large number of publications have been written to address *software crisis*. We will now elaborate on some of the methods and techniques that are considered relevant for this thesis.

If a problem to be solved is inherently complex, naturally, the designed software to solve the problem can be also complex. The intention in software development activity is not to introduce unnecessary complexity to the resulting software system [130]. To this aim, it is generally considered that the most powerful instrument in software engineering is to apply the notion of abstraction, which can be defined as "a selective emphasis on detail" [121]. To manage complexity, various techniques have been proposed, such as modular programming languages, domain-specific languages, software libraries such as application frameworks, model-driven engineering and product-line engineering.

General-purpose programming languages offer various mechanisms to group, abstract and encapsulate related program code using the constructs such as procedures, functions, predicates, modules, objects and components.

Domain-specific languages are tailored and as such they offer specific constructs to gain more expressiveness in the corresponding domains [96].

There have been a considerable number of publications addressing how to verify the correctness of programs. Commonly used techniques are type checking [29], assertions [98] and run time verification [35]. There are also attempts to verify programs by using mathematical models of programming languages [67]. Despite all these developments, proving the correctness of programs is still a very challenging task.

Reusability of programs can be improved by building software libraries. An application framework is a reusable, "semi-complete" application that can be specialized to produce custom applications [47]. The interfaces of an application framework enhance the reuse of generic components to create new applications. Application frameworks generally refer to object-oriented implementations of software libraries. In this approach, the user can instantiate, extend or modify a library when needed, for example, by using object-creation, inheritance and/or aggregation mechanisms of the language adopted [78]. Application frameworks, therefore, support reusability and evolvability in dedicated application domains. Application frameworks may help creating correct software if the reused framework is correct, and if the extensions do not violate the previously defined invariants.

Model-driven engineering (MDE) is a model-based approach, where software is represented using abstract models in contrast to programming-language constructs [86]. In general, three types of models are used: Application model, meta model, meta meta model. In addition, model-transformation techniques are used to convert models between each other if needed. Model transformation techniques are also expressed as models using the same principles. Code can be generated from models using dedicated transformation operations. Model-driven approaches can help creating correct programs, if programs are generated from models which are correct. To define expressive models, one needs to carry out a thorough domain analysis. Various checks can be performed at the model level to assure that

models are used in the right way, for example, by type checking of model parameters and causal dependency checking among tasks. If more than one model is used for the same application, it is important that models are consistent with each other. Model-driven engineering supports reusability and evolvability at the model-level through instantiation and transformation of models. Timeliness of programs generated from models depends on the complexity of the domain, definitions of models and effectiveness of code generation.

More and more companies develop and market families of products instead of a single product. This creates additional complexity and reusability challenges. Software Product-Line (SPL) engineering techniques are defined to reuse the common software assets and maintain the product families [135]. A software product-line design method consists of two main phases: *domain engineering* and *application engineering*. These phases are distinguished as development *for reuse* and *with reuse*, respectively. In the first phase, the domain is explored and analyzed, and the components, generators and reuse infrastructures are implemented *for reuse*. In general, feature models are defined to represent common and variable software components and relationships among these. Later, in application engineering phase, a particular software product is created *with reuse* by configuring feature-models according to the requirements. Thorough domain analysis is necessary to define expressive feature models. Commonality and variability aspects of feature models must cover the whole product space. It is important that product configurations obtained from feature models are correct. This means there exists at least one configuration in which the constraints of the feature model are satisfied. Reusability and evolvability are supported with the variability characteristics of feature models.

1.4 Scheduling of Tasks

A software system incorporates one or more tasks that executes on computing resources to fulfill its desired functionality. If the resources that are utilized in the system are considered explicitly as design concerns, mapping of execution of tasks to resources becomes important. In the literature, such problems are studied under the field of scheduling theory.

Scheduling is a decision-making process in which the resources are allocated to the tasks over time [109]. A program that carries out a scheduling task is called a *scheduler*. The outcome of a scheduling task is a *schedule*. Specification of a *scheduling requirement*, which is also termed as the *scheduling problem* must incorporate the description of the tasks and resources, and their properties associated with the desired timing constraints. If a scheduler can compute a schedule, the problem is defined as *schedulable*. There can be more than one schedule for a given scheduling problem.

1.5 Application of Schedulers in Software Systems

Historically, scheduling has been applied in a large category of software systems [28], for example, processor scheduling in operating systems [123], car scheduling in elevator systems [103], work-force scheduling in project management [84], facility scheduling at airports

[115], antenna scheduling in radar systems [68] and assembly line balancing (scheduling) in factories [18].

Due to the emergence of new application domains, adoption of schedulers in software tends to increase. For example, scheduling of events, control signals and data in cyber-physical systems [132], sensor networks [138] and big-data architectures [59] play a crucial role in the overall functioning of these systems.

1.6 Challenges in Designing Scheduling Software

Implementing software systems that incorporate schedulers can be experienced as a time consuming process. In addition to dealing with well-known challenges in designing software systems, the software engineer has to define and implement the required tasks, resources, associated parameters, objectives, strategies and the constraints, and/or algorithms. Due to the inter-dependencies, this can be a complex process. For example, the constraints must be considered in a very precise and robust manner: The tasks have to be scheduled within their *life-scope*; the periodic tasks have to be spawned at each inter-arrival time; the resource requirements of the allocation have to be realized for each task; the precedence relations have to be satisfied for each allocation; the capacity constraints of resources have to be satisfied; the preemption capability is supposed to be realized; the migration capability has to be satisfied; the mutual exclusion constraint among resources have to be satisfied, etc.

Such systems are generally large and complex. For example, facility scheduling systems at airports are very large systems, since they contain many interrelated scheduling parameters such as planes, runways, gates, staff members, passengers, etc. Moreover, such systems are also safety-critical systems, where the correctness of software must be assured. In addition, determining the schedulability of activities [134] is essential. Due to the complexity of such systems, it is generally very costly to develop them from scratch. Therefore, reusability of software is considered very beneficial for these kinds of software systems.

These systems are, in general, long-living as well. Systems, therefore, must be evolvable to cope with the continuous change of user requirements. Since many of scheduling systems, such as airport systems and production systems must be continuously operational, solutions to the new requirements must be introduced to the systems at run time without discontinuing their operations.

A company may be obliged to develop a family of products. For example, different airport facility scheduling systems may be required depending on the characteristics of airports.

In this thesis, these challenges are addressed at three complementary levels:

i *Application Frameworks.* To ease the development of a software system that incorporates schedulers, the concept of application frameworks is adopted. The scheduling framework can incorporate domain-specific class hierarchies with the necessary operations and attributes so that the desired schedulers can be instantiated with the necessary parameters if needed. In the application framework approach, the software engineer has the full freedom to extend, modify, and discard parts of the software li-

brary. As a disadvantage, the software engineer must have detailed knowledge about the library and the programming language used.

ii *Model-Driven Engineering.* This approach provides a higher-level abstraction of the scheduling domain. Dedicated tools for checking the consistency of the parameters are supplied. The advantage is that domain experts on schedulers can conveniently define the desired schedulers since the models are assumed to be closer to the experts' perception with respect to low-level programs. However, the experts can only define schedulers that can be expressed by models.

iii *Software Product-Line.* This is an extension of MDE approaches with the concepts of product families. The advantages and disadvantages are similar to the ones of the MDE approach.

Unfortunately, to the best of our knowledge, there are no publications in the literature that propose an application framework, a model-driven engineering approach and/or a software product-line engineering approach to develop scheduling systems. As such, while designing scheduling systems, the software engineers are not effectively supported with the state of the art software engineering techniques.

1.7 Challenges in Designing Optimal Models in Model-Driven Engineering

In general, when designing software systems, multiple quality attributes are considered together. In addition to schedulability of tasks as discussed in Section 1.4, for example, energy reduction and improved precision can be considered as additional quality attributes. Energy reduction can be accomplished by lowering the execution speed of tasks and/or by allocating tasks to low-level energy resources. However, this may negatively affect the schedulability of tasks. Precision of tasks can be improved by incorporating algorithms that produce more precise results. Again, this may negatively affect schedulability of tasks. Of course, there may be many other quality attributes that one must consider. It is clear that in addition to schedulability of tasks, one may need to search for optimal models that satisfy various quality concerns.

We think that, within the context of trade-off analysis of multiple quality attributes in an MDE approach, the following aspects must be taken into consideration:

i *Large configuration spaces of models*: It is a common practice that multiple related models are used in MDE environments for a given system. Each of these models may define different kinds of variations. The valid combinations of all variations may potentially enable many possible instantiations of models, which can be difficult for the MDE expert to comprehend. It is, furthermore, difficult to determine the invalid combinations of these variations for the MDE expert.

ii *Introduction of new quality attributes.* New quality models must be introduced if necessary.

iii *Optimization of configurations.* Software engineers generally have to trade-off different quality attributes to configure the most suitable model for a given application

setting. For example, a particular model configuration may improve the quality attribute "reducing energy consumption" while decreasing the quality attribute "time performance". MDE environments must provide means to optimize model configurations based on multiple quality attributes.

To compute the "optimal" architectural decomposition for a software system, there exist various approaches [64, 127]. However, to the best of our knowledge, within the MDE context, optimization of models with respect to various quality considerations has not been studied yet.

1.8 Research Questions

To address the problems discussed in Section 1.7, the following research questions are formulated:

RQ1. What are the most relevant concepts of scheduling systems and accordingly how to define an expressive domain model for scheduling systems?

RQ2. How to define an expressive feature model for scheduling systems so that a large category of families of scheduling systems can be expressed?

RQ3. How to ensure the invariants of feature models and check if a valid configuration can be generated accordingly?

RQ4. How to design and implement an object-oriented application framework library for scheduling systems with **i)** a high degree of reusability and **ii)** evolvability?

RQ5. How to design an MDE environment so that new models and meta models, model pruning techniques, quality attributes, quality optimization criteria and search methods can be introduced in a convenient manner?

RQ6. Within the MDE environment, how can the following activities be performed and computed, effectively?

- **i)** Checking consistency among various models,
- **ii)** Generating the model-configuration space,
- **iii)** Annotating various quality attributes to model configurations,
- **iv)** Assigning relative priority to the predefined quality attributes,
- **v)** Analyzing schedulability of models,
- **vi)** Optimizing models based on multiple quality values.

1.9 The Research Methodology

The adopted research methodology can be summarized as follows:

- *Model-based.* By using thorough domain analysis, models are defined to represent the related domains as accurate as possible.
- *Framework-based.* Instead of searching for dedicated solutions to the addressed problems, generic frameworks are developed. The term generic, here, refers to reusability of the methods, techniques and tools by a large category of relevant applications. The term framework refers to an extensible tooling environment that can be used by

software engineers.

- *Incorporation of the state-of-the-art tools.* Instead of building from scratch, state-of-the-art tools such as model-checkers, constraint solvers, modeling environments, transformation languages are incorporated in the realization of the framework, where possible.
- *Mathematical analysis.* Formal techniques are adopted as much as possible to validate the claims.
- *Practically verified.* Developed techniques are implemented and tested in practice.
- *Scenario-based analysis.* Scenarios are defined to verify the claims that cannot be formally assured. These scenarios are generated as much as possible from the developed canonical models so that the coverage of the scenarios can be determined.

1.10 Thesis Outline and Contributions

Chapters 2 and 3 give the necessary background of the thesis so that readers can become familiar with the adopted terminology and the underlying domains. To this aim, the adopted concepts of scheduling theory, application frameworks, model-driven engineering and product-line engineering are presented.

Motivated by the research question **RQ1**, in Chapter 4, an expressive domain-model is defined in the scheduling domain.

As answers to the research questions **RQ2** and **RQ3**, in Chapter 4, a feature model for schedulers and associated tools are described. This framework enables the product-line engineer to conveniently configure the products as desired.

As an answer to the research question **RQ4**, in Chapter 5, a *reusable* and *run time evolvable* application framework called First-Scheduling-Framework (FSF) is introduced. This framework enables the software engineer to create programs for a large category of schedulers by reusing the library.

As answers to the research questions **RQ5** and **RQ6**, in Chapter 6, an MDE-based programming environment called Optimal Modeling Language (OptML) framework is described. This framework allows the software engineers to introduce various quality models, check the consistency among models, analyze the schedulability of the tasks defined by the models over the resources and optimize the models with respect to the given quality attributes and optimization criteria.

Chapter 7 gives our concluding remarks and identifies topics for future work.

CHAPTER 2

Scheduling

This chapter introduces the scheduling domain as background for the thesis. For this goal, a notation which is commonly used in the literature is described in Section 2.1. This notation is extended in Section 2.2 by the attributes necessary to specify real-time systems.

2.1 Scheduling Theory

In the literature, various definitions for the term *Scheduling* exist. In Pinedo [109], the following definition is used:

"*Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries.*",

Similarly, Baker defines the scheduling problem as allocating resources to activities over time [15]. The resources and the activities might be in any form according to the application area. For instance, in airports, runways are reserved for planes for taking-off and landing activities; in project management, employees are assigned to projects for working activities; and in computing systems, hardware components are reserved to processes for computing activities.

In practice, there can be many factors that determine allocation. One can also define scheduling as a decision-making problem in which goals are formalized as explicit objective functions. From this perspective, a scheduling problem can be seen as an optimization problem. In [15], three decision-making goals are defined: *turnaround*, *timeliness* and *throughput*. *Turnaround* refers to the time required by an activity to complete, and *timeliness* corresponds to the latest allowed completion time for an activity. Finally, *throughput* is the number of performed activities within a unit of time.

Baptiste adopts the same definition of Baker, and divides the scheduling problems into two categories: *decision problem* and *optimization problem* [16]. In the former, the number of schedules that satisfy all the constraints are determined; whereas, in the latter, the most suitable schedule that minimizes the objective function is decided. While the predefined constraints restrict the boundary of the solution space, the objectives are the optimization criteria that decide a solution (*schedule*) among many solutions within the solution space. For instance, as an example of constraints, an activity may require the other to be completed before starting. Therefore, it is blocked unless the other activity completes. In contrast, fair allocation of resources is an example of objectives. The optimal schedule among valid schedules is determined according to how much it satisfies the aimed objective. From this perspective, scheduling problems are grouped as a specific subset of optimization problems.

Buttazzo [28] explains the necessity of scheduling from the perspective of real-time computing systems. Here, it is claimed that the effective behavior of any real-time computing system does not only depend on the precise computation of the processes, but also the reaction of the system within a certain amount of time.

To define a scheduling problem, the following triplet notation of Graham [61] is commonly adopted [16, 23]:

$$\alpha|\beta|\gamma, \tag{2.1}$$

where α represents the machine environment; β defines the scheduling constraints; and γ corresponds to the scheduling objective.

Different terminologies for the activities exist. For instance, they are termed as *tasks* in [28], *jobs* and *operations* in [61, 109]. Terminology differences can also be observed for *resources* and *machines*. Throughout the thesis, we adopted the terminology in the book

[28], where a *task* may consist of an activity or a sequence of activities. These activities are called *instances* or *jobs*. From this perspective if a task has an activity, it is called as *job*.

The following attributes are used to define jobs:

- r_j: the *release time* of the job j, which specifies the earliest start time.
- c_{ij}: the *worst-case execution time (wcet)* of the job j, which is the maximum amount of time required for an instance of a task to complete on the machine i. It can also be denoted as c_j according to the attributes of the resources which will be explained later. In the literature, the terms *processing time*, *execution time*, *computation time* are also used instead of the term *wcet*.
- d_j: the *deadline* of the job j, which determines the latest finish time. This attribute can also be termed as the *due date*.
- p_τ: the *period* of the task τ which is the inter-arrival time of the jobs unless it has only one instance.
- w_τ: the *priority* of the task τ, which denotes the relative importance of the task among other tasks. It is also termed as the *weight* in the literature.

To clarify these attributes, we refer to Figure 2.1.

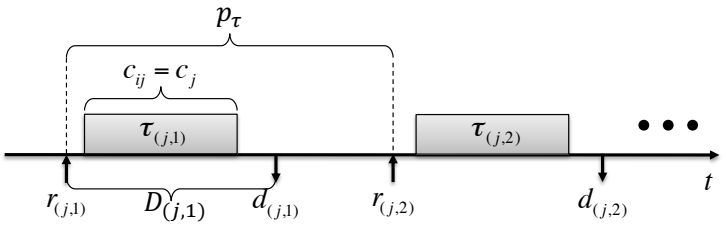


Figure 2.1: Fundamental attributes of the periodic task. Only two consecutive jobs are shown.

We define the time interval between release time and the deadline of a job as *time-scope*.

In addition to these attributes, some derived attributes exist. For instance, the *relative deadline* is the time distance between the actual deadline $D_{(j,k)} = d_{(j,k)} - r_{(j,k)}$ and the release time of the job, and the *laxity* is the residual time within the duration of the relative deadline after extracting the execution time of a job $X_{(j,k)} = D_{(j,k)} - c_{ij}$.

2.1.1 Machine Environment

The machine environment α is composed of two parts $\alpha_1\alpha_2$, namely *machine identifier* and *number of the machines*, respectively. According to the literature, there exist 7 machine identifiers (α_1):

- 1: It refers to the single machine environment on which the jobs can execute sequentially.
- P: All the machines are identical and in parallel, meaning that the execution speed of the jobs does not vary from one machine to the other and the jobs can be executed in parallel unless there is some restriction.
- Q: Unlike identical machines, the resources have different execution speed, therefore the execution time of the jobs depends on the speed of the machine. The execution

time of the resource is calculated according to the operational load of the jobs and the speed of the utilized resource.

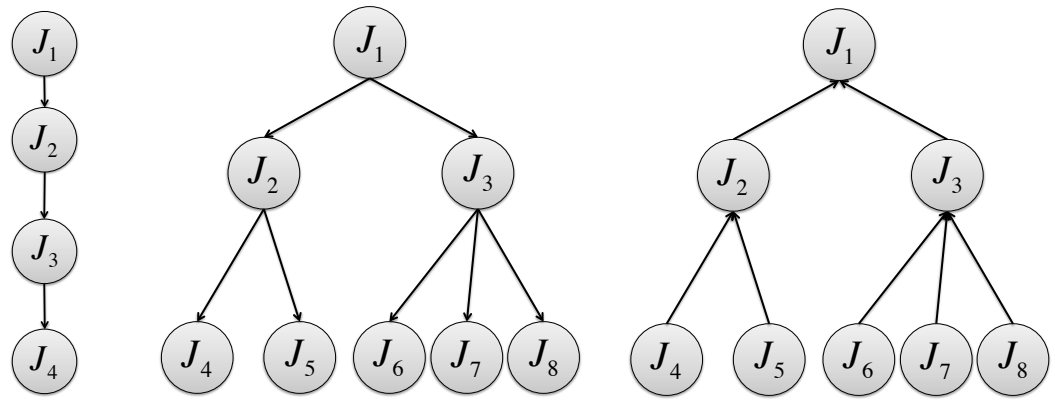
- R: These machines are unrelated to each other and the execution speed may change according to the job. Therefore, the speed of the machine is denoted as v_{ij}.
- O: This machine environment refers to the *Open Shop*. In this environment, there is no restriction to the execution order of the jobs, but they can not be processed in parallel. In addition, a one-to-one relation between the jobs and the resources exists, meaning that the jobs are supposed to execute on each machine.
- F: The *Flow Shop* environment differs from the *Open Shop* in one aspect. Each job has a predetermined path on each one of the machines.
- J: Like the *Flow Shop* environment, the jobs have a route of execution, but they do not need to visit each machine.

We express the number of the machines (α_2) using positive natural numbers $\mathbb{N}^+$. For instance, 3-machine Flow Shop environment is denoted as $\alpha = F3$.

2.1.2 Job Characteristics

In this category, the scheduling process constraints and restrictions are defined as follows:

- $\beta_1 = \{pmtn, \epsilon\}$
  This expresses the preemption ability (preemptability) of a job. In this case, a job may be suspended by another job without completing the execution. The preempted job is restarted to complete the execution when the resource available again.
- $\beta_2 = \{r_j, \epsilon\}$
  This indicates that a job may have a specific release time, before which it cannot start to execute. Otherwise, a job may start at anytime.
- $\beta_3 = \{prec, \epsilon\}$
  The *precedence relation* of jobs blocks the start of one job if it requires the completion of another job. This is commonly represented by directed graphs in which the nodes and the edges represent the jobs and the precedence relations, respectively (see Figure 2.2). If all jobs have at most one predecessor and successor, the relation is referred to as *chains*. If each job has at most one successor, it is denoted as *in-tree*. If a job has at most one predecessor, then the precedence relation is defined as *out-tree*.
- $\beta_4 = \{M_j, \epsilon\}$
  The *Machine Eligibility* constraint obliges jobs to run only on a specific subset of all resources. It is only for machine environment with more than one resource.
- $\beta_5 = \{p_j = p, \epsilon\}$
  It represents that all jobs have fixed execution time p.
- $\beta_6 = \{d_j = d, \epsilon\}$
  In this case, each job is supposed to complete before the fixed deadline d.
- $\beta_7 = \{s_{jk}, \epsilon\}$
  This is defined as the *sequence dependent setup time*, which represents the required amount of time for a machine to start to execute the job k after finishing the job j. Unless it is not specified, all the setup times are initially assumed to be zero.



(a) *chains* (b) *outtree* (c) *intree*

Figure 2.2: The graph representation of precedence relations.

- $\beta_8 = \{batch(b), \epsilon\}$
  A resource may execute b jobs simultaneously if this constraint is defined. The entire batch is finished when the last job of the batch is completed.

The character ϵ above means the corresponding constraint is not applied to the configuration.

2.1.3 Objectives

The objectives are also called the *optimality criteria* [23]. As shown in the following, 7 objectives are commonly formulated using the completion times of the jobs, which is denoted as C_j:

Objectives	Formula
Lateness	$L_j = C_j - d_j$
Earliness	$E_j = max\{0, -L_j\}$
Tardiness	$T_j = max\{0, L_j\}$
Absolute Deviation	$D_j = \lvert L_j \rvert$
Squared Deviation	$S_j = (L_j)^2$
Unit Penalty	$U_j = \begin{cases} 0 & \text{if } C_j \leq d_j \\ 1 & \text{deadline missed} \end{cases}$
Makespan	$C_{max} = \max_j C_j$

Table 2.1: The commonly known objective functions defined in [23]

The objective function that is chosen to be optimized determines the quality of the calculated schedule. From this perspective, the aim of minimizing the objective function *Lateness*

is to complete the tasks *immediately* after their release time, whereas the objective function *Earliness* is the complement of *Lateness*. Minimizing *Tardiness* objective is exactly the same with minimizing *Lateness* when the completion fo a job exceeds its deadline. The quality of the schedule is fixed if the task is scheduled and completed before its deadline. The remaining items in the list (Table 2.1) are derived from these functions and are considered self-explanatory.

In addition, the common objective functions are formulated as *maximum*, *summation* and *weighted summation* over the tasks. For instance, the *maximum* $L_{max} = \max_j L_j$, the *summation* $L_{sum} = \sum^j L_j$ and the *weighted summation* $L^w_{sum} = \sum^j w_j L_j$ are the versions of the *Lateness* objective. The linear combinations of these formulas are also considered.

2.2 Scheduling in Real-Time Systems

In real-time systems, it is crucial for a task to complete its execution not only in a logically correct way but also within the specified time. Not being able to satisfy the timing constraints may result in irreversible system-wide failures. Naturally, real-time scheduling problems have additional constraints over jobs and resources.

2.2.1 Task-related Constraints

In real-time systems, there exists some task which needs to be scheduled regularly. Each activation of the task is called the *instance* (*job*) of that task and has its own release time and absolute deadline. The other types of tasks are taken into consideration when it is requested by applications and they have only one instance and possible future arrival of this kind of tasks is unknown to the scheduler. From this perspective, the tasks are classified as *periodic* and *aperiodic* in terms of their period instantiations.

For periodic tasks, the release time of the first instance is specially termed as *phase* and denoted by ϕ_j. Therefore, the release time of the k^{th} instance is calculated as follows:

$$r_{(j,k)} = \phi_j + (k-1)p_j \tag{2.2}$$

On the other hand, an *aperiodic* task has only one instance and it is requested in case of necessity by the system. An intermediate class between periodic and aperiodic tasks also exists, namely *sporadic*. Unlike aperiodic tasks, they have sequential instances to execute, and the minimum time interval between two consecutive jobs are known. Knowing the earliest next instance of a sporadic task gives the freedom and certainty to the scheduler to schedule another task instance to the time duration after the completion time of this task, by ignoring its existence. From this point, they are considered and scheduled as periodic tasks. Since they do not have fixed periods and are requested each time, they are considered as *aperiodic* tasks.

During the scheduling process, there may be more than one task whose time-scopes overlap in time. In such a case, there are three possible reactions to take: i. the tasks are tested whether they can be scheduled without missing any deadline. It is only possible when the duration between the earliest release time and the latest deadline of the overlapping jobs

must be as much as the total execution time of the tasks. ii. if there are more than one resource, other available resources are requested. iii. ordering tasks with respect to their *priorities* and reserving resources to the tasks one by one. The priority value of a task is determined according to the *scheduling policy*. In general, the scheduling policies are related with the timing properties of the task. For example, a policy with respect to the periods of tasks is *Shortest Period First*; this is also known as *Rate Monotonic* policy [90]. Assignment of a priority may be either fixed or flexible. In fixed-priority assignment, tasks have static priorities that do not change in time, whereas in the case of flexible priority, they are re-calculated for each instance of a task.

Another task-related constraint that affects the utility of the system is related to the natural outcome of the missing deadlines of tasks. Depending on the consequences of not being able to satisfy a deadline constraint, the tasks are classified under three categories:

- **Hard.** Especially in real-time systems, the tasks with a hard deadline constraint must be treated strictly since missing one deadline could lead to the failure of the system
- **Soft.** A task is said to be *soft* if missing its deadline does not cause any harm to the system, but decreases the benefit obtained from it.
- **Firm.** The *firm* tasks might be completed after the deadline like the *soft* tasks, but finishing the task after deadline has neither utility nor harm to the system.

2.2.2 Resource-related Constraints

To represent resource-related constraints, the notation introduced in the previous section has to be extended. For instance, Holenderski [69] classified resources based on their capacities as *single-* and *multi-unit* resources. The latter allows more than one job to execute on the same resource if the demanded total capacity of jobs do not exceed the total capacity of the shared resource. In case multiple tasks accesses the same resource, they have to be synchronized [28]. it is known as *race condition* in concurrent programming that multiple read/write or write/write accesses to a shared resource must be synchronized. This kind of resources is commonly termed as *mutually exclusive* resources.

A task may commonly require multiple resources at the same time such as memory, bus and peripheral devices. One may classify resources as passive and active [139]. Active resources are for example CPU's and they are always needed to execute tasks. Passive resources can be described in terms of the capacities of "resource units". There may be constraints among passive and active resources as well such as "a memory may belong to a certain processor". It must be possible to express such cases adequately.

Mobile devices bring additional challenges since they aim to provide long operation times before batteries are exhausted. Since the mobile devices operate on battery power, they should provide long lasting battery life to users. To accomplish this objective, *Dynamic Voltage Scaling* can be adopted [108, 119, 112, 122, 30, 95], which is used to decrease energy consumption with the price of decreasing processing power. This requires extensions to the model that has been given in Section 2.1.1.

2.2.3 Scheduling Constraints

Some application-specific scheduling constraints may have to be considered. Consider the following two examples:

- **Migration** is a process of suspending a running task on a certain machine and moving the task and its contextual environment to another machine. Two kinds of migration exist, namely *task-level* and *job-level*. *Task-level migration* allows the instances of a task to run on different machines, but each instance (job) has to be completed on the same resource where it has started to execute. In *job-level migration*, there is, nevertheless, no such a limitation on instances. This kind of processing can be seen as a specialization of preemption of resources.
- **Conditional Preemption** is the mechanism to reduce the run time overhead due to preemption. There are various techniques published in the literature:
  - *Preemption Threshold* gives freedom to the tasks that have higher priority values than the specified threshold [136].
  - *Deferred Preemptions* defines the longest time interval in which a task may execute.
  - *Task Splitting (Cooperative Scheduling)* is only allowed at pre-defined *preemption points* of each task [26].

CHAPTER 3

Model-Based Software Engineering

We consider model-based software engineering as an important category of software-engineering methods, where models are used as the fundamental artifacts of engineering. Model is defined by OMG as [100]: "A model of a software system is a description or specification of that system and its environment for some certain purpose." Adopting models is an alternative to using general-purpose programming languages; if defined properly, the abstractions of models can be considered semantically closer to the domain of interest whereas general-purpose programming languages are in general defined from the perspective of computing machines. Advantages of model-based software engineering are claimed to be reducing *complexity*; and enhancing *expressivity, reusability, adaptability*, *correctness*, etc. [4].

In the sub-sections 3.1 to 3.5, as examples of model-based engineering practices, respectively, we describe software product-line engineering, application frameworks, model-driven engineering, search-based software engineering, and model-based verification.

3.1 Software Product-Line Engineering

Software Product-Line Engineering (SPLE) is a paradigm to develop software using *platforms* and *mass customization* [110].

A *platform* is an asset base that consists of reusable software artifacts designed for a given domain. *Mass customization* is a large-scale of mass production process, which is diversified by the desires of customers. The software products belonging to the same product family differ from each other in detail but naturally they share the common aspects of the family. To define expressive models for a product-line, one needs to represent the *commonalities* and *variabilities* of the product family, adequately. Domain expertise is one of the most crucial prerequisite to derive such models.

A software development method based on SPLE, in general, consists of two phases: *domain engineering* and *application engineering*. In *domain engineering*, the asset base is built, which consists of reusable software components; these are defined according to the product portfolio based on certain marketing strategy. In *application engineering*, products are configured from the asset base according to the product requirements. To gain more understanding of the practical application of these phases, the reader may refer to [110].

This *reuse-oriented* software development approach has various advantages compared to developing dedicated software per product:

i *Reducing the development costs.* Setting up an asset base requires initial investment. Nevertheless, since products of a given product family share certain properties, reduction of development costs can be achieved in due time, when a sufficient number of products are produced.

ii *Facilitating correctness of software.* It is assumed that the reusable components in an asset base are well-tested. Naturally, this simplifies testing of the configured products because in this case only the configurations are required to be tested.

iii *Reducing time to market.* After the asset base is built, time to market can be shorter since configuration of products from the asset base generally takes less time than developing products from scratch.

iv *Increasing adaptability.* If designed accordingly, configuration mechanisms can enhance adaptability since products can be re-configured after the product release.

To model the common and variable parts in an asset base, in general, feature models are used [89, 79]. As the name implies, a feature in a feature model corresponds to a basic element that represents some concern in the asset base. A feature can be detailed by relating a sub-feature to it. A sub-feature can be also extended by its sub-feature, where a tree of features can be formed. In such a model, one may term features as super-features and sub-features, where sub-features represent the detailed characteristics of their super-features.

Four basic relations are defined between features, which are shown in Figure 3.1. The root-feature is the super-feature shown at the top of the figure. If a mandatory relation is used, the sub-feature is a necessary feature of its super-feature. The commonalities in an asset base are expressed using mandatory features.

An *optional* relation indicates that during configuration it must be decided if a sub-feature is included or not. An *or* relation between a super-feature and one or more sub-features indicates that during configuration at least one sub-feature must be included. In Figure 3.1, the possible configurations of *or* relations are PC_1, PC_2, and PC_1C_2. An *alternative* (*XOR*) relation between a super-feature and one or more sub-features indicates that during configuration at most one sub-feature must be included. In Figure 3.1, the possible configurations of *alternative* relations are either PC_1 or PC_2.

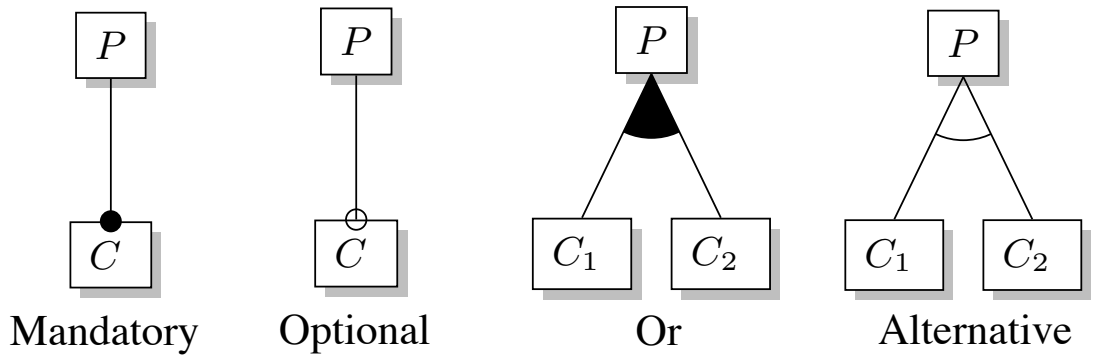


Figure 3.1: Possible relations between features.

Possible configurations can be further limited by using so-called *cross-tree constraints*. Two commonly used cross-tree constraints are:

- A relation that is specified as $a \xrightarrow{\text{req}} b$, where a and b are the features under consideration, if the feature a is selected, the feature b must be selected as well.
- A relation that is specified as $a \xrightarrow{\text{exc}} b$, where a and b are the features under consideration, if the feature a is selected, the feature b must not be selected.

One may also define the cardinality properties of relations [37]. A cardinality property can be expressed by a pair of integers $[l..u]$, where l and u represent the lower and upper bounds of the corresponding sub-features, respectively.

3.2 Model-Driven Engineering

In Model-Driven Engineering (MDE), a software system to be developed is expressed using one or more models, which are also called views. Each view represents a different aspect of the corresponding software system [81, 116]. In MDE, in addition to defining models in a certain notation such as UML, meta models, meta meta models, and model transformation modules are used. Model transformation modules are specified in a similar way as models. This terminology is explained in the following sections.

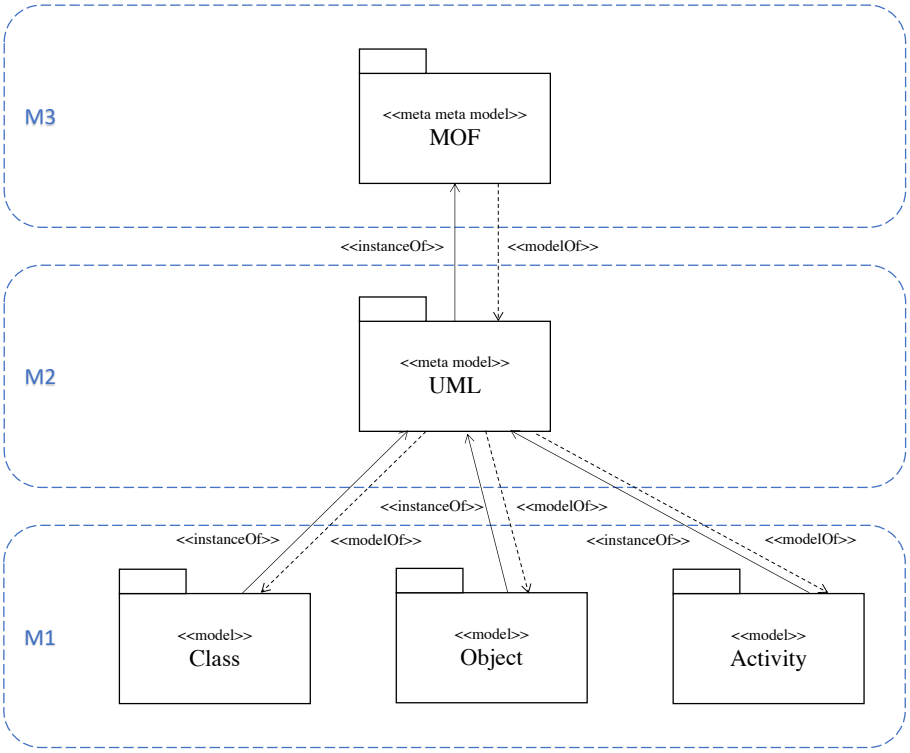


Figure 3.2: A hierarchical structure of meta meta model, meta model and model of a Unified Modeling Language (UML).

3.2.1 Models and meta models

According to the level of abstraction, the models are divided into two groups, *Platform Independent Models* (PIM) and *Platform Specific Models* (PSM). The former is a view of a system regardless of an underlying platform; whereas, the latter encompasses the information about a part of the platform that is relevant for the system. To instantiate a model, there exists a set of rules that is defined in the corresponding meta model. In this sense, a *meta model* is a description of the syntactic and semantic rules from which the model is instantiated.

In Figure 3.2, a UML example is used to show the relation between meta models and models. At the top layer (M3) in the hierarchy, Meta Object Facility (MOF) provides an open-source meta model specification language to define meta models. In this example, UML is an instance of MOF, and MOF is a meta model of UML. To define a new meta model at the level M2, one needs to specify its own rules using MOF. At the bottom layer (M1), class, object and activity models are instantiated from UML.

With the help of meta meta models, meta models, and model transformation models, a more expressive modeling technique than traditional model-based approaches is provided.

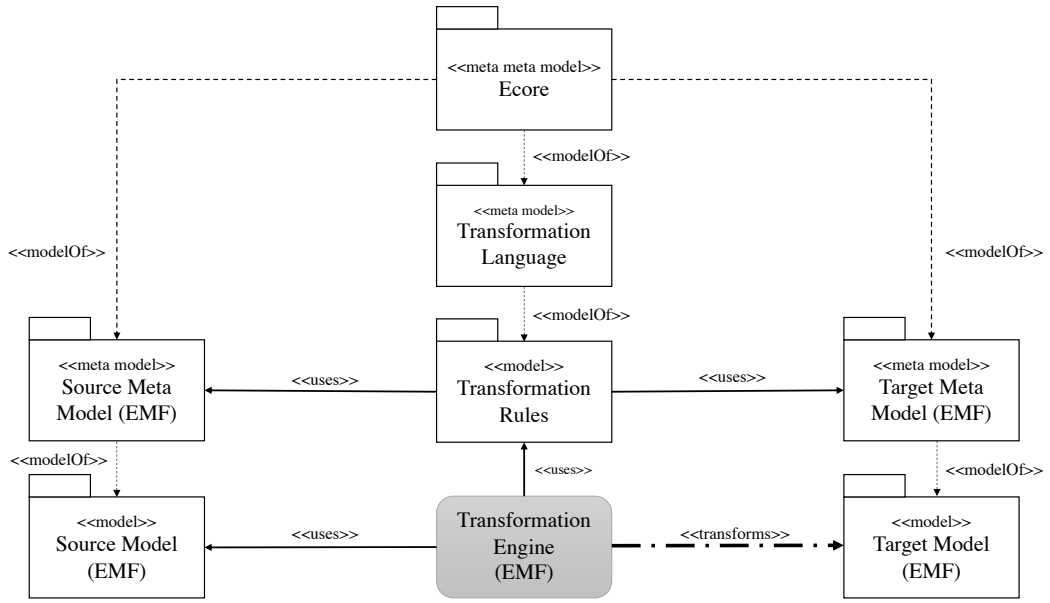


Figure 3.3: Model transformation schema for EMF.

For example, various meta models and model transformation models can be integrated by using a common meta meta model. In addition, considering transformation specifications also as a model enables reuse of model transformation techniques across different modeling approaches.

3.2.2 Model Transformations

Eclipse Modeling Framework (EMF) [129] is a well-known MDE framework that provides modeling and model transformation capabilities for Eclipse applications. As symbolically depicted in Figure 3.3, the Model Transformation Engine is a module that gathers *Transformation Rules* to transform a source model to a target model. The model *Transformation Rules* is defined at the meta-level and includes a set of rules. This "modelOf" relation continues until the meta meta model *Ecore* is reached. Such an organization of models facilitates reuse of transformation models within the context of source and target models.

3.3 Application Frameworks

In traditional library-based reuse, generic functions/procedures are offered that can be called from applications. The main difference between traditional library-based reuse and application frameworks is that application frameworks do not only offer functions, but they also define the main parts of the software applications in an abstract manner. With the use of frameworks, applications are instantiated, configured and/or extended.

Naturally, application frameworks are defined within specific domains. To this aim, based on a thorough domain analysis, application framework designers must identify the essential concepts in the domains of interest and represent these as the abstract components of the framework. In a way, an application framework defines a skeleton of the applications that can be created from the framework. Variations are generally implemented using the adaptability mechanisms of object-oriented programming (OOP) languages.

OOP languages provide three important design features that support the development of application frameworks: *encapsulation*, *polymorphism*, and *inheritance* [47, 78]. *Encapsulation* restricts direct access to certain internal components and helps software developers focus on the interfaces of components only; this increases *modularity*. In this way, framework designers can hide the unnecessary details of their frameworks from the users. *Polymorphism* enables framework users to flexibly configure their applications by using the polymorphic interfaces of the components. By using the *Inheritance* mechanisms of languages, framework users can reuse and/or extend certain parts of the adopted framework.

3.4 Search-Based Software Engineering

One of the main challenges in software engineering practices is to find an optimal solution to a design problem constrained by possibly conflicting requirements. In general, traditional design methods provide a large number of informal design heuristics. Based on intuition and experience, software engineers consider these heuristics and design software systems that offer the best compromise according to their perception. To determine if software satisfies the requirements or not, various techniques are used, such as scenario-based analysis, metrics and dynamic testing.

During the last decades, several researchers have investigated how to compute the best software design using computational methods [63, 126, 64, 40, 127]. In these approaches, a software design problem is converted to an optimization problem and solved computationally. To this aim, first a model of the software system is defined that allows various design configurations along with the constraints and design objectives. The best design alternative is searched using a suitable computational method. This way of designing software systems can give more accurate results. However, defining suitable models causes an additional overhead. Moreover, depending on the optimization problem to be solved, finding an optimal design solution can demand large computing resources.

A considerable number of publications on search-based approaches are available. For example, to find a trade-off among quality attributes availability, reduced memory usage and time performance, in [126], an optimal architectural decomposition is computed. In [40], multi-objective techniques are used to optimize software architectures with respect to qualities production speed, reduced energy usage and print quality. In [20], by using genetic algorithms, model re-factoring is performed to maximize cohesion and minimize coupling.

3.5 Model-Based Verification

As the name implies, *Model-Based Verification* aims to check the correctness of software systems based on models [46]. This is in contrast to dynamic testing of software where checking is carried out on the actual software. Since most models are not specified in a formal language, it is necessary to translate the models to a language that allows formal verification. There are a number of verification systems available. Example categories of verification systems are static analyzers, logic-based analyzers, state-based model checkers, constraint solvers, etc. Depending on the type of a verification problem and the type of models used, an appropriate verification system has to be selected.

There are several challenges in model-based verification. Firstly, accurate models must be defined for the software system to be verified. Secondly, depending on the complexity of a problem, the verification process can be computationally very intensive. Finally, the models must be kept consistent with the software system that is represented.

There have been a considerable number of publication demonstrating the applicability of model-based verification in practice. For example, in [46], a software system is verified to guarantee deadlock freedom and the fulfillment of timing constraints and memory usage. In [7], a graph-based model checking approach is used to detect interference among software components. An interested reader can refer to [55].

CHAPTER 4

A Formal Product-Line Engineering Approach for Schedulers

Scheduling is a decision-making process in which resources are allocated to the activities over time [109]. In general, companies, which market scheduling systems, have to design and implement a family of products instead of a single product.

Designing and implementing robust scheduling systems, however, are not trivial. Firstly, due to many operational constraints in the requirements, software that implements scheduling processes can be very complex. For example, the multi-car elevator scheduling problem is known to be NP-hard [83]. Secondly, details in the requirements may result in a very large scheduling system. For example, software systems that implement facility scheduling at airports are very large systems, since they contain many interrelated scheduling parameters such as planes, runways, gates, staff members, passengers, etc.

To overcome these challenges, the following measures can be adopted. Firstly, complexity can be addressed by defining an expressive domain model for scheduling techniques. Secondly, the cost of developing a family of products can be reduced by adopting Software Product Line Engineering (SPLE) methods. In SPLE, instead of developing from scratch through ad-hoc reuse, products are configured from predefined reusable software components. Reusable components are derived from domain models which are generally expressed in the feature model notation [79]. Unfortunately, to the best of our knowledge, apart from our work, there has not been a comprehensive work in the literature along this line.

This chapter introduces a novel domain model, which represents the scheduling domain. The model is expressed using the *feature model* notation so that it can be integrated with existing SPLE approaches. In addition, the specification language of Clafer [9] is used to formalize the model. This allows verification of the configured products against the invariants of the domain model.

This chapter is organized as follows: The related work is given in the next section. Section 4.2 summarizes the aimed objectives of this chapter. Our SPLE approach is introduced in Section 4.3. In Section 4.4, a feature model is derived from the domain knowledge. In Section 4.5, a number of canonical examples are presented to demonstrate the expressivity of the feature model. An assessment of our approach is presented in Section 4.6. The last section concludes the chapter.

4.1 Related Work

There have been a considerable number of publications that report on the practical application of SPLE approaches [93]. To the best of our knowledge, none of them has been applied in the domain of scheduling. This chapter extends our earlier conference paper [105], with a detailed description of the configured feature models.

In the literature, there has been a considerable amount of publications in solving scheduling problems [61, 28]. Accordingly, a large category of algorithms has been developed. In addition, different kinds of solver-based solutions have been studied and presented in the literature to address planning and scheduling problems [54, 66, 71]. There exists also a study [82] which presents a formal framework to implement reusable schedulers. However, these publications do not aim at creating SPLE based solutions to define a family of products which incorporates scheduling software.

There have been various publications on the formal verification of feature models [17]. In general, these publications do not integrate the verification process in a specific design environment. Moreover, verification of product configurations in the area of scheduling domain has not been studied before.

4.2 Objectives

The objective of our work is to define an SPLE method, and associated tools and techniques to create a family of scheduling applications with the following characteristics:

- ***Reduced complexity:*** To deal with complexity, it is important to identify and define the necessary abstractions in the scheduling domain that encapsulate the unnecessary details.
- ***Expressive domain models:*** To utilize an effective SPLE method in the scheduling domain, it is important to define an expressive domain model that can cover a large category of scheduling applications. To represent a family of applications, special attention must be paid to explicitly deal with the variations in the scheduling domain. The terms applications and products are used interchangeably in this context.
- ***Verification:*** In the process of configuring applications from the corresponding feature models, the created product must not violate the constraints of the scheduling domain.

4.3 Our SPLE Approach for Schedulers

We have developed a novel SPLE approach for configuring applications that incorporate schedulers. Figure 4.1 depicts our application-engineering process.

A product is defined as a dedicated configuration of the corresponding feature model. A description of the feature model is given in the next section. By the help of *Configuration Tool for Feature Model*, the software engineer binds the variabilities defined in the feature model to specify a dedicated product. As symbolically shown in the figure, *Verification Tool*

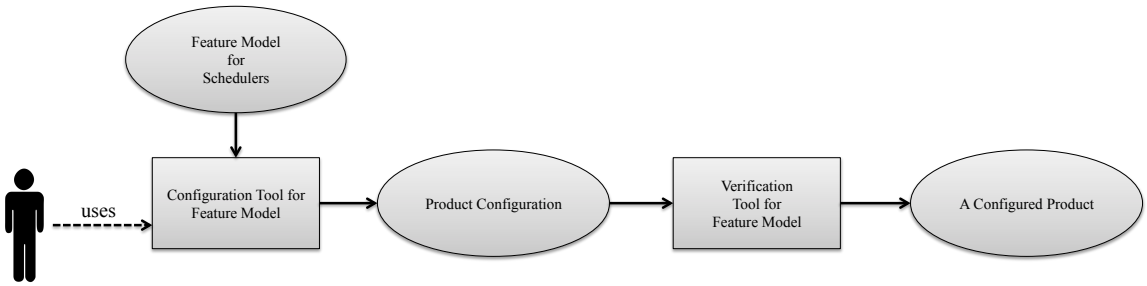


Figure 4.1: An application of our SPLE method. Rectangles and ellipses represent the operations and the data, respectively.

for Configuration takes the data *configuration of products* (also called partial configuration in the literature) and checks the number of configurations. The binding process may result in zero or more configured feature models. A zero configuration indicates that the binding process does not result in any product, possibly because the specification is too restrictive. In this case, the software engineer must reconsider the configuration. A single configuration refers to a desired specification, in which all the variabilities are bound. If there is more than one configuration, not all variabilities are bound and as such the configuration refers to multiple products. In this case, the software engineer needs to complete the binding specifications.

4.4 A Feature Model of Schedulers

In this section, we explain how an expressive feature model can be derived from scheduling theory given in Section 2.1. Most studies in the scheduling domain offer dedicated algorithms that are claimed to be beneficial in certain context [23, 26, 28, 39, 43, 70, 94, 109, 120, 128]. The aim in the domain-engineering phase, however, is to seek for a generic model, which can represent the common aspects of most relevant scheduling techniques. To realize this, we aim to identify commonalities and variabilities in the scheduling and related domains; these refer to mandatory and optional features in the feature model, respectively. In the application-engineering phase, it must be possible to conveniently configure and instantiate a dedicated scheduler derived from the feature model.

After an extensive comparison of various approaches, we have found out that the following model is expressive and generic enough to represent a wide range of scheduling solutions:

$$\mathbb{S} = (\mathcal{T}, \mathcal{R}, \mathcal{C}, \mathcal{S}), \tag{4.1}$$

where $\mathbb{S}$ is the scheduler; $\mathcal{T}$ is the task; $\mathcal{R}$ is the resource; $\mathcal{C}$ is the scheduling characteristics; and $\mathcal{S}$ is the scheduling strategy. The corresponding feature model is depicted in Fig. 4.2. The *feature cardinality* $[a..b]$, defined in [37], is shown above the features. It means that

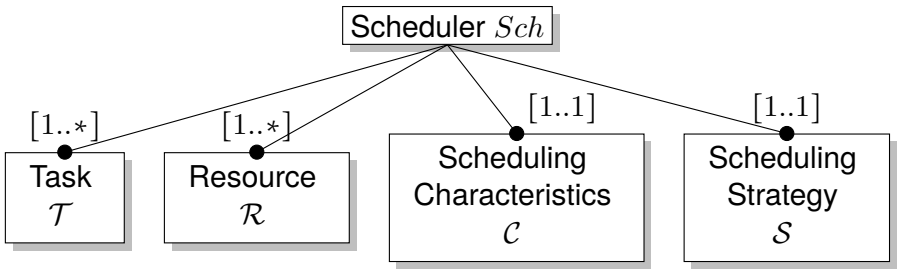


Figure 4.2: A feature model of schedulers.

the parent feature can be configured for at least a and at most b instances of the feature. If the upper bound is defined with wild-card character $(*)$, there may be infinite number of instances of the feature. In the chapter, each mandatory and optional feature without cardinality information is assumed to have the boundaries $[1..1]$ and $[0..1]$, respectively.

The concept $\mathcal{R}$ in Definition 4.1 has a similar meaning as α used in Definition 2.1. After analyzing the related literature in detail, we have observed that the definitions of the terms β and γ strongly relate to all the fields of Definition 2.1. For example, β may define both the characteristics of tasks and of scheduling processes. Similarly, γ relates to the objectives related to both tasks and resources. Although we believe that Definition 2.1 is useful for expressing the domain of scheduling problems in general, it is less optimal for reuse purposes. Moreover, variations in the implementation of schedulers tend to be classified as variations in the definitions of tasks, resources, characteristics of the scheduling processes and scheduling strategies. The fields of Definition 4.1, therefore, are considered to be more suitable to encapsulate these variations.

There are also various cross-tree constraints between inter- or intra-models. These constraints are necessary because naturally sub-models are not orthogonal to each other. Such constraints limit the number of scheduling applications that can be configured. Elaboration on the cross-tree constraints are considered outside the scope of this chapter. A more interested reader can refer to the feature model definition in the repository.

4.4.1 A Feature Model of Tasks

Most publications in the scheduling domain come along with their own task models. Nevertheless, they largely share common terminology. Based on our literature analysis, we generalize a task model with the following formula where a feature model representation of it is shown in Fig. 4.3:

$$\mathcal{T} = (\mathcal{T}_{time}, \mathcal{T}_{prio}, \mathcal{T}_{dep}, \mathcal{T}_{pmtn}, \mathcal{T}_{req}, \mathcal{T}_{gran}, \mathcal{T}_{obj}), \tag{4.2}$$

In the definition above, the symbols are explained as follows:

- $\mathcal{T}_{time}$: It corresponds to the fundamental time-related attributes of a task, which iden-

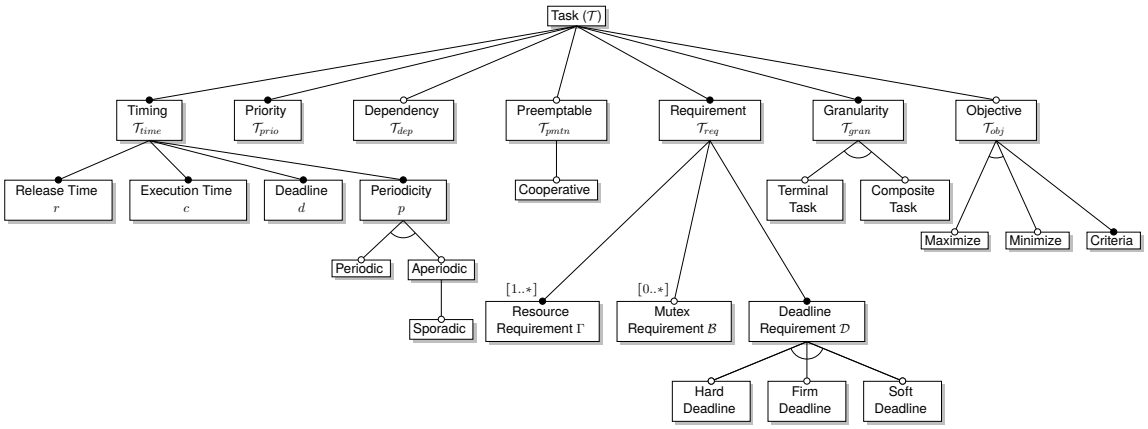


Figure 4.3: A feature model representation of tasks.

tify the task. These are:

$$\mathcal{T}_{time} = (r, c, d, p), \tag{4.3}$$

where the parameters above correspond to the task attributes explained in Section 2.1. In addition, we defined the time interval between release time and deadline of a task as *life-scope*. More information can be found in [28].

- $\mathcal{T}_{prio}$: It represents the *priority* of a task, which is also termed as *weight* in Section 2.1.
- $\mathcal{T}_{dep}$: It refers to the concept of *data dependency* between tasks. We adopted the token-based dependency approach explained in [3, 88]. It blocks the execution of a task until the dependent data (*token*) is fired even if its release time has already passed. In addition, we have defined *sequence dependent setup time* as explained in [109]; This means that after a dependent token is fired, the corresponding task may either immediately start to execute or wait according to the amount of setup time.
- $\mathcal{T}_{pmtn}$: It refers to the *preemption*-ability (*preemtability*) of a task. Preemptable tasks can be interrupted in time and continue afterwards.
- $\mathcal{T}_{req}$: It represents the *resource requirement*, *mutex requirement* and *deadline requirement* of a task. On the contrary to the traditional notation in Definition 2.1, a task may have more complex resource requirements. For instance, expressing a system with *shared* [28] and *multi-unit* resources [69] is impossible using the traditional notation. To be more expressive, we have extended the specification of *resource requirement*. In case of shared resources, the *mutex requirement* is a necessary feature to be adopted. If there is a task, which modifies the internal structure of a resource, such as writing to a memory space, the other tasks have to wait until it is completed. For instance, assume that there exist three tasks using the same-shared resource and τ has mutex requirement. This means that the scheduler should produce a schedule in which the life-scope of the other two tasks cannot overlap with the *life-scope* of the task τ. The *deadline requirement* of a task represents the consequences of missing the deadline on

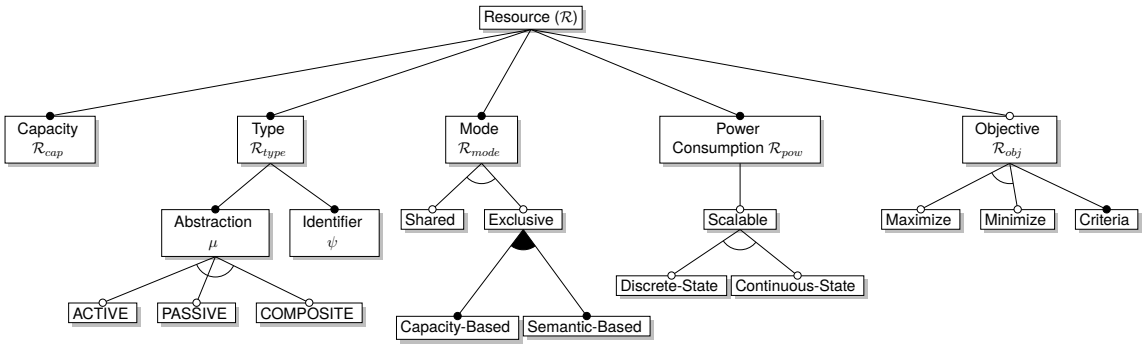


Figure 4.4: A feature model representation for resources.

an underlying system where tasks are running. These are called *hard*, *firm* and *soft* which have been introduced and explained in [25, 28, 128].

- $\mathcal{T}_{gran}$: It represents the granularity of a task. It can be either *terminal* or *composite*. The granularity of tasks as presented in the literature can be classified as terminal. A composite task can be decomposed into terminal and/or composite tasks, which are *contextually* related to each other.
- $\mathcal{T}_{obj}$: A *performance measure* is defined in the literature as an overall system constraint to be enforced [28]. On the other hand, a task may also have its own objective. For this reason, we have introduced the attribute *task-related objectives*.

4.4.2 A Feature Model of Resources

For resource models, although there are some differences, most publications in the scheduling domain adopt a similar terminology. For the sake of expressivity and generality, we propose the following model, shown in Fig. 4.4:

$$\mathcal{R} = (\mathcal{R}_{cap}, \mathcal{R}_{type}, \mathcal{R}_{mode}, \mathcal{R}_{pow}, \mathcal{R}_{obj}), \tag{4.4}$$

- $\mathcal{R}_{cap}$: It represents the *capacity* of a resource. With respect to the attribute *capacity*, *single-* and *multi-unit* are the two kinds of resources [69]. Since any *single-unit* resource can also be defined as a *multi-unit* resource with single capacity, in our model, we only adopt the term *multi-unit*.
- $\mathcal{R}_{type}$: It represents the *type* of a resource. In [139], resources are categorized in two groups: *active* and *passive*. In addition to an active resource, such as central-processing-unit, a task may also require passive resources to perform some supplementary operations, such as sensing the environment, storing and transferring data, etc. Bus, memory, peripheral devices are the examples of passive resources. A passive resource may be associated with an active resource. For example, a processor cache memory as a passive resource may only be accessed through its processor. To this aim, we introduce two additional abstractions called *composite resource* and *accessibility tree*.

A task running on an active resource R_a has access to the passive resources, which are the siblings of R_a, the terminal resources of its ancestors or of each sibling composite resources. Consider the example resource tree depicted in Fig. 4.5. Now assume that the task τ is running on cpu, meaning that it can utilize the terminal resources of the sibling composite resource, namely cache_cpu-1 and cache_cpu-2; and the terminal passive resources of all the ancestors, i.e all passive resources except the cache memories under GPU composite sub-tree.
The structure of a resource μ is expressed as follows:

$$\mu \in \{ACTIVE, PASSIVE, COMPOSITE\}. \tag{4.5}$$

In addition, we define the identifier ψ to refer to a human-understandable name for any type of resource.
Resource type is therefore expressed as follows:

$$\mathcal{R}_{type} = (\mu, \psi), \tag{4.6}$$

There may be more than one resource with the same type, but each resource has only one type.

- $\mathcal{R}_{mode}$: It corresponds to the mode of a resource. In [28, 139], resources are grouped in two groups based on share-ability among tasks. A *shared resource* can be required by multiple tasks but the capacity demand of a task must not exceed the maximum capacity of the corresponding resource. On the other hand, *exclusive resources* are divided into two sub-categories: *capacity-based* and *semantic-based*. For a resource running in *capacity-based exclusive* mode, the total capacity utilized by tasks cannot be more than the capacity of a resource; whereas the mode *semantic-based exclusive* allows only one of mutually exclusive resources to run at the same time, still preserving the capacity constraint.
- $\mathcal{R}_{pow}$: It represents the power consumption of a resource. *Dynamic Voltage Scaling* (DVS) has been introduced in [30, 95, 108, 112, 119, 122]. For the resources with *discrete-state power consumption*, a resource can execute tasks at one of a predefined set of power scales. On the other hand, *continuous-state power consumption* indicates that a resource is able to run at a power scale within the range of predefined maximum and minimum power scales.
- $\mathcal{R}_{obj}$: Because of the individual performance requirements of resources, we define the *resource-related* objectives such as minimizing power consumption, maximizing utilization, etc.

4.4.3 A Feature Model of Scheduling Characteristics

In this section, we focus on *scheduling characteristics*, which is expressed as $\mathcal{C}$ in Definition 4.1. The following is an extension of the model presented in [28, 39, 120]. A sub-feature model is depicted in Fig. 4.6.

$$\mathcal{C} = (\mathcal{C}_{type}, \mathcal{C}_{pmtn}, \mathcal{C}_{mig}, \mathcal{C}_{pol}, \mathcal{C}_{tres}, \mathcal{C}_{prio}, \mathcal{C}_{window}, \mathcal{C}_{obj}), \tag{4.7}$$

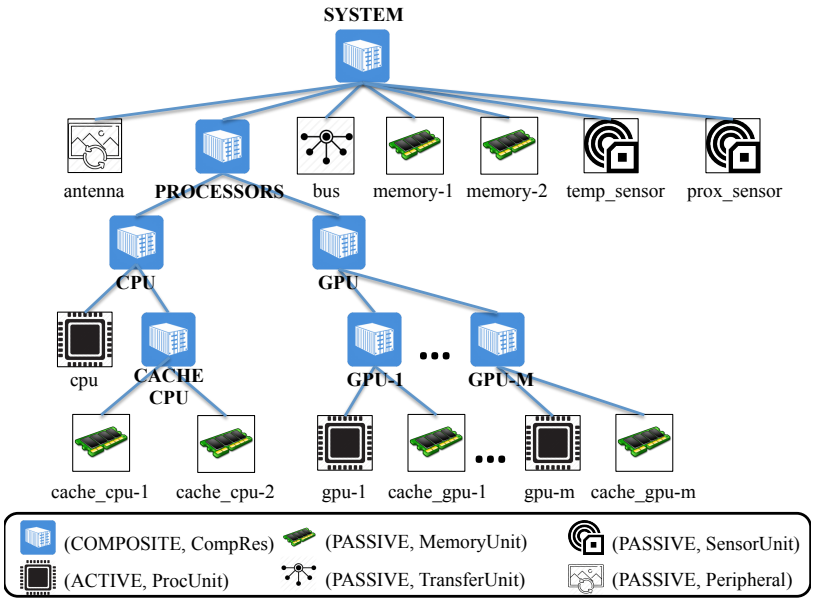


Figure 4.5: An example *resource tree* to specify the accessibility of resources.

- $\mathcal{C}_{type}$: It refers to the type of the scheduling process. It can be either *offline* or *online*. In offline scheduling, resources are allocated to the tasks within a predefined time interval. In online scheduling, the most convenient tasks are chosen for each resource at an instant of time. In both types, the constraints and the objectives as explained in Section 2.1.3 must be satisfied and optimized, respectively.
- $\mathcal{C}_{pmtn}$: It defines whether a scheduling process is *preemptive* or not. Even in cases where tasks are specified as *preemptable*, the scheduler produces non-preemptive solutions unless it is set to *preemptive*.
- $\mathcal{C}_{mig}$: It specifies whether a scheduling process supports *migration* or not. The attribute *migration* indicates the capability of a scheduler to suspend a running task on a resource and move its execution context to another resource. *Task-level* and *job-level* are the two migration strategies. The former supports the migration of only the instances of tasks; each instance must be executed only on the same resource. The latter does not have such a restriction if it is *preemptable*.
- $\mathcal{C}_{pol}$: It represents the *scheduling policy*, which is a criterion to decide on which task takes the permission to utilize a resource at an instant of time. The sub-feature *grouping* is an optional, aimed to specify the importance to a specific set of tasks according to their credentials. After priority-based clustering among tasks with respect to the grouping criteria, the ranking policy is performed to the tasks separately within the same group.
- $\mathcal{C}_{tres}$: It refers to the *time resolution* of the requested scheduling process. It is the minimum time interval within which resources are allocated to tasks. In *online scheduling*,

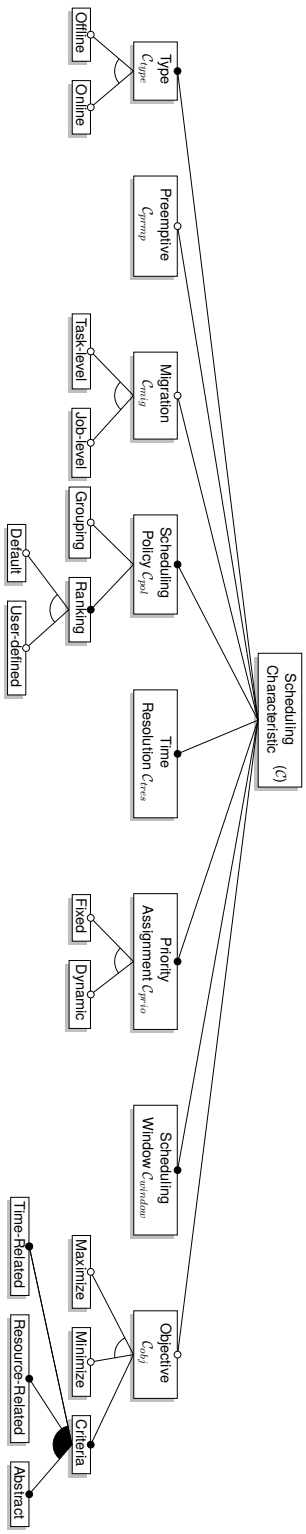


Figure 4.6: A feature model for the characteristics of schedulers.

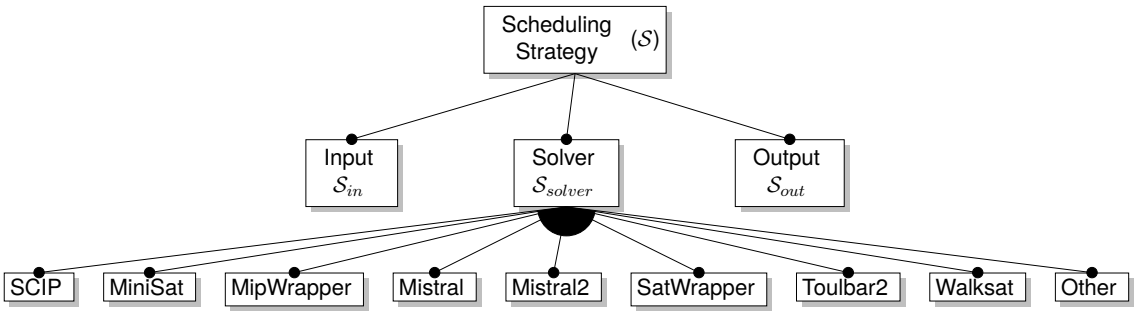


Figure 4.7: A feature model for representing scheduling strategies.

it means *time quantum* [123]. In *offline scheduling*, it determines the *context switch* (*preemption*) points in time.

- $\mathcal{C}_{prio}$: The scheduling algorithms in the literature are also classified in terms of the *priority assignment*. If the scheduling process is classified as *fixed-priority*, the priority values of the tasks are determined when they are appended to the taskset and remain the same; whereas in *dynamic-priority* scheduling process, the *scheduling policy* depends on the attributes which vary with respect to time. Therefore, the priorities are recalculated even during the same scheduling process.
- $\mathcal{C}_{windows}$: It corresponds to the *scheduling window* whose boundaries determine the duration of the time interval when the tasks are scheduled. In *online* and *offline* scheduling, it equals to the *time resolution* and multiple of the *time resolution*, respectively.
- $\mathcal{C}_{obj}$: It corresponds to the *objectives* of the scheduling process. It directly relates to the objectives defined in Section 2.1.3.

4.4.4 A Feature Model of the Scheduling Strategy

In our domain model, a scheduling strategy is responsible for the approach to solve the scheduling problem defined using the sub-models presented in Sections 4.4.1, 4.4.2, and 4.4.3. Our design framework consists of generic constraint solvers for the sake of fostering reuse. Therefore, the model for scheduling strategy depends on solvers. A scheduling problem is expressed in a given format as *input*, fed into a particular solver and the solver produces a *schedule* in a desired format as *output*.

A feature model representation for scheduling strategy is shown in Fig. 4.7.

We formulate a model of scheduling strategy as:

$$\mathcal{S} = (\mathcal{S}_{in}, \mathcal{S}_{solver}, \mathcal{S}_{out}), \tag{4.8}$$

where $\mathcal{S}_{solver}$ corresponds to the set of constraint solvers; $\mathcal{S}_{in}$ and $\mathcal{S}_{out}$ are the specifications of the scheduling problem and the desired format of the schedule, respectively.

The sub-features of $\mathcal{S}_{solver}$ from left to right SCIP to Walksat represent the solvers supported in our framework. *Other* corresponds to a solver implementation that can be plugged

into the system. The dedicated sub-features of $\mathcal{S}_{in}$ and $\mathcal{S}_{out}$ which correspond to the adopted solvers are not shown in the figure. This is, because, in our framework a generic input/output language is used for each solver. Translation to a specific language is encapsulated.

4.5 Model Validation through Experiments

This section demonstrates how the feature model can be configured to express 7 different scheduling solution examples and one optimization example. Each example is presented in the following order: (i) Explaining the problem; (ii) Expressing the problem using the traditional notation formalized in Definition 2.1. The motivation for using the traditional notation is due to its common usage in the field of scheduling theory; (iii) Presenting the example as a configured feature model; and (vi) for the sake of readability, partially visualizing the representations of the corresponding configurations are given. In the figures, the mandatory features that are not directly crucial for the problem definition are omitted and denoted by three dots; and the optional features shown in the figures are all utilized in the configurations of the products.

4.5.1 Rate-Monotonic Scheduling (RMS) Problem

According to the articles [90, 94], in the problem RMS, the tasks are assumed to be preemptable and have various release time requirements. The priorities of the instances of tasks are determined with respect to the period values. The shorter the period of a task, the higher its priority.

Furthermore, there exists only one resource with single-unit capacity and the optimality criteria is to minimize the lateness penalty for each task. Based on this definition, the 3-field notation of this problem is as follows:

$$1|r_j, pmtn|L_j \tag{4.9}$$

In a nutshell, the RMS can be defined as “a fixed-priority online algorithm for scheduling independent, preemptable, periodic tasks on a single processor by minimizing lateness objective”.

This example can be expressed directly by our feature model: The partial visual representations of the configuration can be found in Figure 4.8. There exists only one task instance, which a taskset may include many copies derived from, with different run time values. We term a copy of a task instance as *run time task*. According to the problem domain, the tasks are defined as *preemptable* and any run time task may have different *release time*, the *periodicity* feature is set to be *periodic* under the task instance and *preemptive* feature is included in the configuration under the sub-feature model *Scheduling Characteristic*. According to Definition 4.9, there exists a single machine. Therefore, the *abstraction* of the resource instance is defined as *active* by selecting the corresponding feature under the sub-feature tree rooted by *Resource*. Since the period values of run time tasks do not vary, the *Priority*

Assignment feature is classified as *Fixed* and *Scheduling Policy* is set to *Default* to enable predefined policies. As stated optimality criteria in Definition 4.9, *Minimize* and *Time-Related* under the feature *Objective* are selected. From the perspective of scheduling strategy, we have selected SCIP solver for this problem as well as for the following examples due to its robustness and transcending performance over other open-source constraint solvers.

4.5.2 Multiple-Resource Scheduling Problem (MRSP)

In this problem, unlike the previous example, we have two tasks and two resources. The tasks are either *periodic* or *aperiodic*. While the periodic tasks are not *preemptable*, the aperiodic tasks are *preemptable*. In addition, the *aperiodic* tasks depend on the periodic tasks with predefined *sequence dependent setup time*. We assume that there are two *active* and two *passive* resources. The *active* resources have *discrete-state* scalable power consumption; whereas the power consumption of the passive resources is assumed to be constant per capacity. Furthermore, the passive resources can execute more than one task at a time with respect to a given capacity (batch). Moreover, both of the task instances may need to execute on defined resources.

From the perspective of the scheduling process, the scheduler is *offline* and *preemptive*. Moreover, it has the capability of *job-level* migration. The *scheduling policy* is *FIFO* and the *priority assignment* of the tasks is *fixed*. The objectives are to *minimize* the power consumption and *lateness*, separately. The definition of this problem is:

$$Q2|r_j, d_j, prec, pmtn, M_j, s_{jk}, batch| \sum_j L_j. \tag{4.10}$$

The example can be expressed by the feature model as follows. The partial visual representations of the configuration can be found in Figure 4.9. In this example, there exist two instances of tasks, specified as periodic and aperiodic. Therefore, the instances τ_1 and τ_2 are configured by selecting *Aperiodic* and *Periodic*, respectively. Since the instance τ_1 may have a dependency to the run time tasks of the instance τ_2, the feature *dependency* is included in the configuration. According to the problem definition, the feature *preemptable* is selected only for the instance τ_1. The *resource requirements* of the task instances are the same in terms of configuration and each run time task belonging to any task instance may require utilization on resources integrated into the system. The *deadline requirements* of the instances are *hard*. In this configuration, we instantiate two different resource instances. The instances res_1 and res_2 are specified as *active* and *passive*, respectively. Since the active resource has *discrete-state* power consumption, the corresponding feature is included in the configuration for the instance res_2. As specified, *scheduling characteristic* sub-feature diagram includes *offline* and *preemptive* features. In addition, it consists of the features with the same name *job-level migration* capability. The *scheduling policy* is predefined in our design environment like in the case of the RMS example, hence *default* feature is selected under the feature *ranking*. Since the policy is static and time-invariant, the feature *fixed* is chosen for *priority assignment* mechanism. Finally, the aim of the scheduling process is

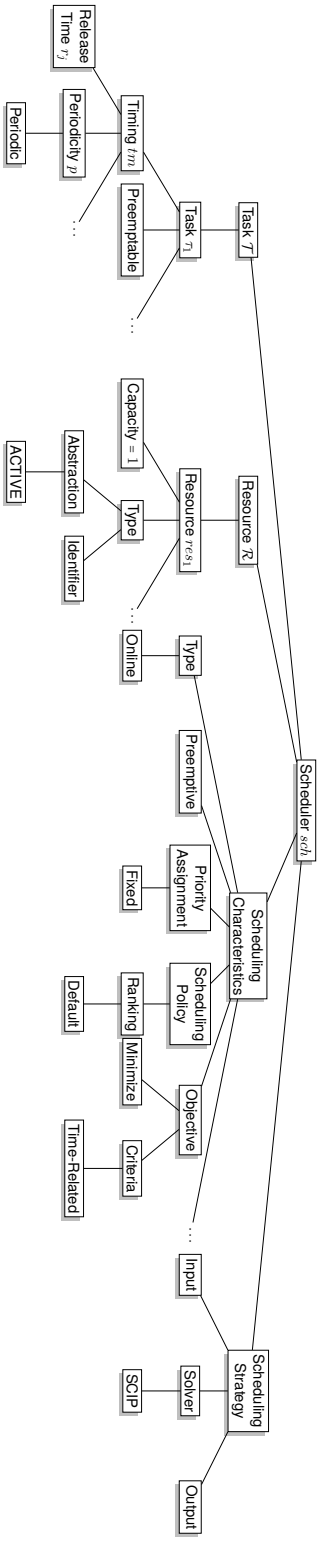


Figure 4.8: A configuration of the feature model expresses the RMS problem as defined in Section 5.4.1

to minimize the power consumption and lateness, separately. Therefore, we have included both *time-* and *resource-related* features as well as the feature *minimize* in the configuration. The *scheduling strategy* is constructed using the solver SCIP.

4.5.3 Elevator Scheduling Problem

In this example, we assume that we have two cars (elevator cabins). Therefore, the scheduling process consists of two phases: (i) dispatching the passengers to the cars (passenger-to-car assignment) and (ii) stopping cars on the corresponding floors according to the requests of passengers [83, 117]. Tasks are categorized into two groups, namely *Car-* and *Hall-Call*. The former is the request of a passenger inside a car to go to the desired destination floor index; whereas the latter is the request to indicate the desired travel direction of a passenger on the floors using UP and DOWN button calls. To ease the computation, we assume that moving between successive floors and the *time resolution* are unit time. Since the elevator should visit the closest destination floor first, the scheduling policy is set to *Shortest Job First* and the *priority assignment* is *dynamic*.

In this example, we have defined our objective as to minimize the waiting time of the passengers. Therefore, minimizing the total lateness objective is chosen. According to the specifications above, this problem can be formalized as follows:

$$P2|r_j, d_j|\sum_j L_j. \tag{4.11}$$

The partial visual representations of the configuration can be found in Figure 4.10. Although we have two different run time tasks, namely *Car-Call* and *Hall-Call*, they share the common properties in terms of scheduling. Therefore, in the configuration, there exists only one instance of a task. Since the request time of any call in elevator systems is in general unpredictable, the *periodicity* is chosen *sporadic*. Due to the policy, a task in operation may be suspended in case of another call with shortest execution time, i.e. a call from or to a floor which is closer, which results in the inclusion of the feature *preemptable*. Therefore, the tasks have *soft deadline requirement*, and the *resource requirement* is only to a *car*. In our scenario, there may exist more than one run time resource termed as *multi-car elevator scheduling problem* in the literature. However, the characteristics of cars are the same according to our problem definition. The instance car is an *active* resource and it is impossible to adjust the *power consumption*. Since the choice of a task that executes first is instantaneous, the scheduling type is selected *online* in the configuration. In addition, the feature *preemptive* is included. The corresponding feature to *Shortest Job First* scheduling policy is selected, and we specified the features *ranking* and *default*. *Priority assignment* is set to *dynamic* as moving one floor to another may change the priorities of tasks with respect to the selected policy. In addition, we assume that after assigning any task to a car, it is impossible to switch it to an another car. Due to this assumption, the *scheduling characteristics* excludes the feature *migration*. The objective shown in Definition 4.11 is to minimize the lateness and the feature *objective* is specified accordingly. For this case, we have deployed the solver Mistral2.

Figure 4.9: A configuration of the feature model expresses the MRSP problem as defined in Section 5.4.2.

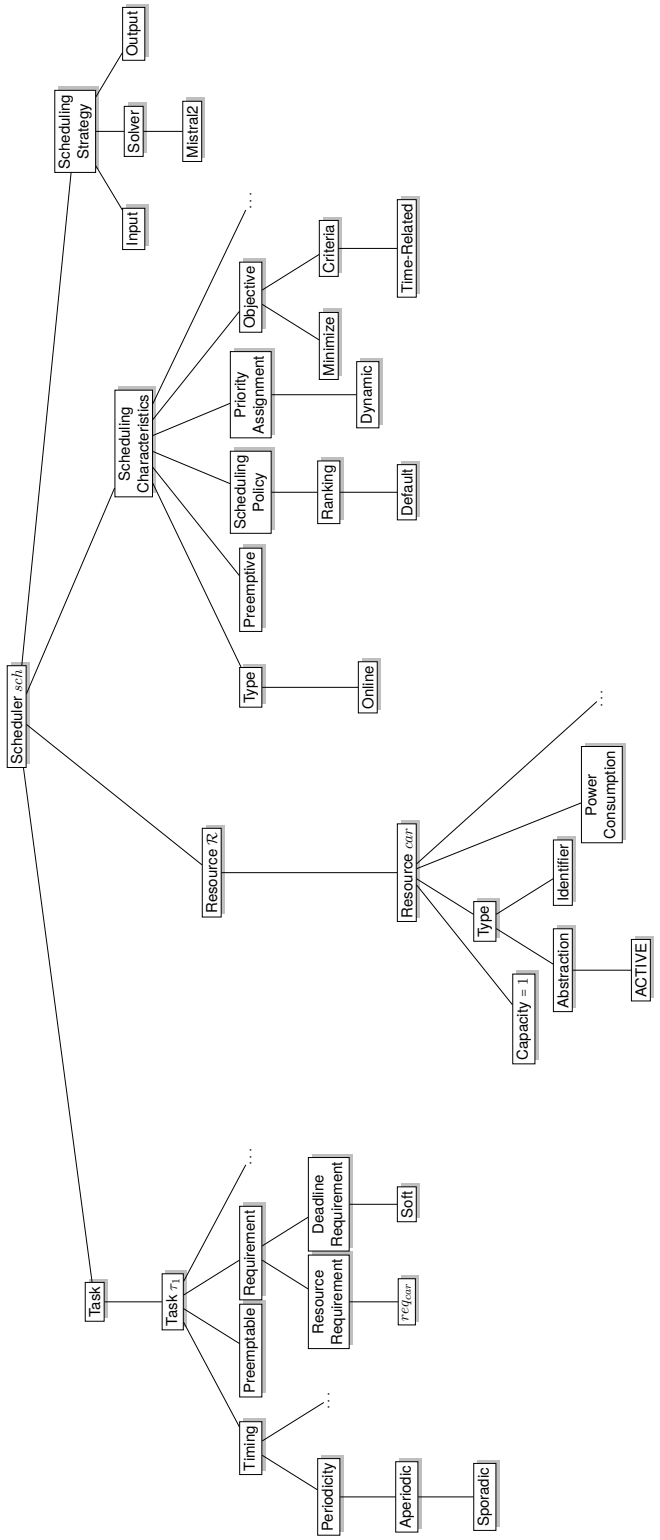


Figure 4.10: A configuration of the feature model expresses the ESP problem as defined in Section 4.5.3.

4.5.4 Flow-shop Scheduling Problem (FSP) with Permutation

In Section 2.1.1, the flow-shop machine has been defined. Unlike the flow-shop problem, the permutation constraint inhibits the sequence changes of tasks among machines [109], so that the execution orders of tasks are the same on different machines.

The corresponding definition of the problem is as follows:

$$F4|prec, prmu|C_{max}. \tag{4.12}$$

The partial visual representations of the configuration can be found in Figure 4.11. There is only one instance of a task. The run time tasks have dependencies to be satisfied, so the feature *dependency* is bound for the configuration. Since the shop problems are specifically realized for factory production assembly lines, the preemption of a task is not possible. For this reason, the configuration does not consist of *preemptable* in the configuration of a task and *preemptive* in the configuration of *scheduling characteristic*. Each run time task requires exactly one run time resource even if there are many copies of the instances of the resource, and has no strict deadline. Consequently, the feature *Soft* is included in the configuration. The shop problems have standard resources, which are multiple conveyor belts to carry the products. Therefore, in our example, we have one instance of a resource that is defined as *active* resource with constant *power consumption*. Therefore, the feature *scalable* under the feature *power consumption* is not bounded to the product configuration. In terms of *scheduling characteristic*, the type of the scheduling is *offline*. The scheduling policy in this example is not crucial since it is impossible to experience any case such that there exist two ready-to-execute task at the same time due to the strict dependency relation among them, hence we set the *ranking* policy to *default*. Again for the same reason, the *priority assignment* is specified as *fixed*. The objective in Definition 4.12 is realized by integrating the features *minimize* and *time-related* into the configuration. In the *scheduling strategy* sub-configuration, we selected again the solver SCIP.

4.5.5 Job-shop Scheduling Problem (JSP)

JSP is one of the well-known combinatorial optimization problems [31, 58]. We have defined four machines and three tasks with 3, 4, 3 jobs each of which requires one of the machines. Since the execution order of the task instances on each machine is predefined, there exists natural precedence relation between tasks. The aim of the scheduling process is to minimize the *makespan*. According to this scenario, the problem definition is as follows:

$$J4|M_j, prec|C_{max}. \tag{4.13}$$

From the perspective of the shop scheduling problems, the variations emerge from the strictness of the dependency among tasks. Especially, for FSP and JSP problems, there is no difference in terms of configuration, but run time initialization. Therefore, they have the same product configuration that is shown in Figure 4.11. However, their data dependencies between the jobs are different. This property is defined in Section 2.1.1.

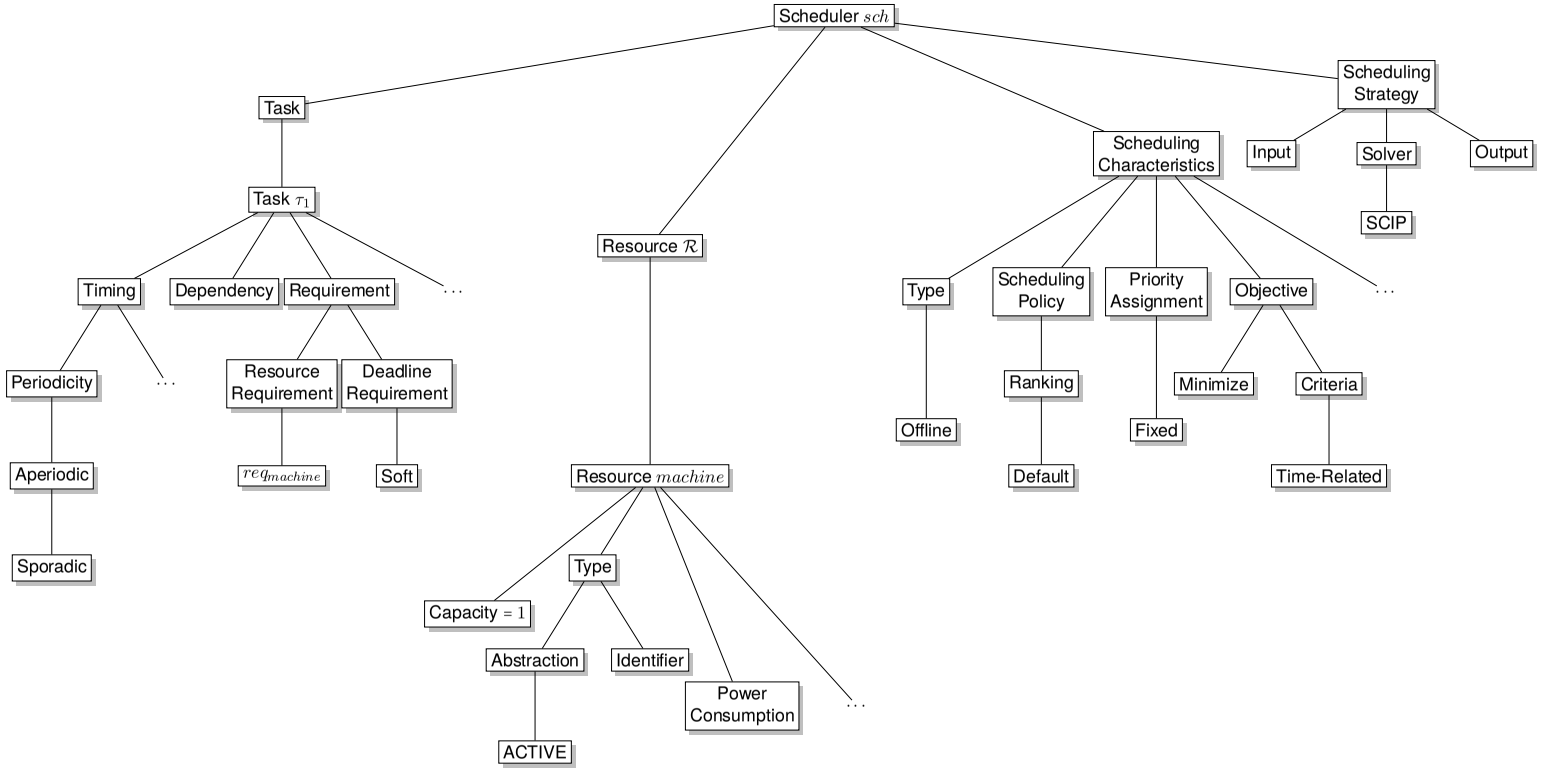


Figure 4.11: A configuration of the feature model expresses the FSP and the JSP problem as defined in Sections 4.5.4 and 4.5.5, respectively.

4.5.6 Open-shop Scheduling Problem with Preemption (OSP/PMTN)

Since the *open-shop* machine environment has been explained in Section 2.1.1, there is no need to mention its explanation. In this example, we assumed that the tasks are preemptable, and so the scheduling strategy, and we have three resources and 5 tasks each of which has 3 jobs assigned to the machines one-by-one. Each task instance has its own release time and deadline. The objective is to minimize the *makespan*.

The definition of this problem is:

$$O3|r_j, d_j, M_j, pmtn|C_{max}. \tag{4.14}$$

The partial visual representations of the configuration can be found in Figure 4.12. As we stated earlier, the shop problems share almost the same configuration in terms of modeling. One of the differences in this example, which can be understood from the title of the section is the *preemption* capability of the tasks and *preemptive* feature of the *scheduling characteristic*. Further, since the strength of the dependency among tasks is alleviated in the context of shop scheduling problems with the order of FSP, JSP, OSP; to avoid contextually interrelated tasks assigned to different run time resources to execute in parallel, it is necessary to define the *mutex requirement* under the feature *requirements*.

4.5.7 Open-shop Scheduling Problem without Preemption (OSP)

Only the difference of this example from the previous one is the preemption capability of the tasks. Therefore, the definition of the problem using the traditional notation is as follows:

$$O3|r_j, d_j, M_j|C_{max}. \tag{4.15}$$

Since the tasks are non-preemptable unlike the example in Section 4.5.6, the feature *preemptable* in Figure 4.12 does not exist in the configuration, which is the only difference.

4.5.8 Travelling Salesman Problem (TSP) as an Optimization Problem

TSP is an optimization problem [50] (also known as a *path planning problem*). The aim is to find the minimum traveling path for a salesman who is visiting each city on a route and finally arriving at the departure city. We need to map the concepts of this problem to the concepts in the scheduling domain. We model the traveling cost between two cities as *sequence dependent setup time*. Furthermore, the salesman's worst case staying time in a city is represented as the execution time of the corresponding task. Finally, for simplicity, we assume that all tasks have the same execution time.

The objective is to minimize the criteria *Makespan* [28]. The corresponding problem definition is as follows:

$$1|s_{jk}, p_j = 1|C_{max}. \tag{4.16}$$

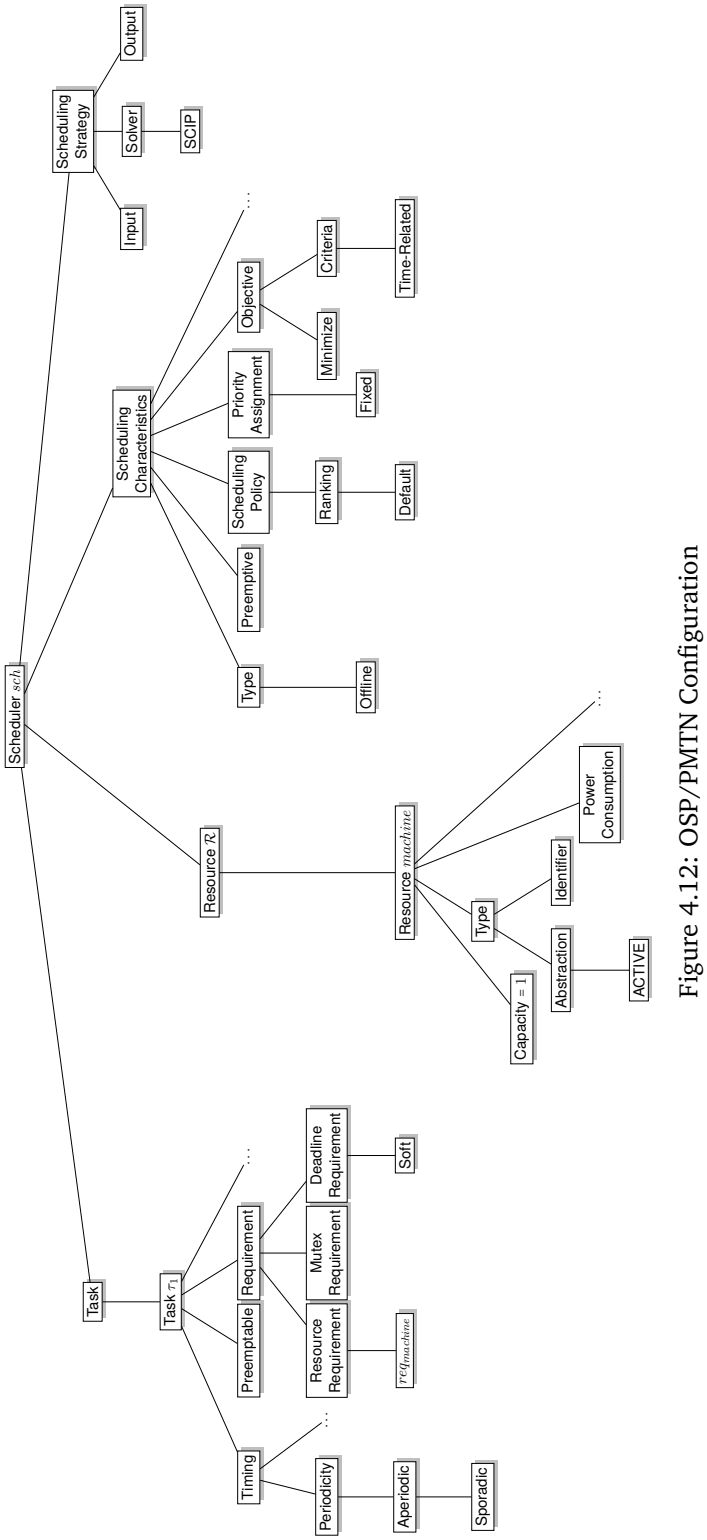


Figure 4.12: OSP/PMTN Configuration

The partial visual representations of the configuration can be found in Figure 4.13. To travel from one city to another, the dependency constraint between two corresponding cities is supposed to be satisfied. Therefore, the configuration of the task has to include the feature *dependency*. Besides, the problem is defined as visiting each city once and arriving to destination city. For this reason, run time tasks should be *aperiodic* without *sporadic* property. In addition, the execution time of run time tasks are equal to a symbolic value which is also the same for *time resolution* value, meaning that it is impossible to interrupt tasks before completion. Therefore, the instance of the task is configured by excluding the feature *preemptable*. For this example, the deadline of the task is set to the maximum time which is enough to travel even if a salesman takes the longest path, which makes the *deadline requirement* of a task *soft*. The problem is modeled with a single run time resource instantiated from the instance of a resource named $city$. The run time resource has a single-unit capacity without *scalable power consumption* capability. TSP is an *offline* scheduling problem in which the priorities of the run time tasks are unique. Because of this reason, we designate the *ranking* of *scheduling policy* as *default* and the *priority assignment* as *fixed*. As shown in Definition 4.16, the *objective* of the problem is selected accordingly by including the features *time-related* of the feature *criteria* and the feature *minimize*.

4.6 Evaluation

In this section, we assess our approach explained in three previous sections with respect to the objectives described in Section 4.2.

4.6.1 Assessment Method

In general, models that represent "commonalities/variabilities" form the backbone of any PLE process, as they offer the necessary abstractions that derive the various implementations in the related domain. In our case, the feature model notation is used for this purpose. From this perspective, we firstly assume that the "quality of the feature model" is one of the most important factors that determine the "quality of the PLE approach". Secondly, the "quality of a PLE approach" is also determined by the configurability of the feature model.

Our method to evaluate our feature model is as follows: Firstly, based on a thorough domain analysis, a feature model is defined. Secondly, as presented previously, the feature model is configured with respect to a set of well-known canonical applications defined in the literature. Thirdly, based on these examples, our approach is evaluated with respect to the following three requirements: reduced complexity, expressivity and verification of the feature model.

Reduced Complexity

A company that develops a family of software systems that incorporates possibly different kinds of scheduling techniques have to deal with two important challenges: (1) Well-known software engineering challenges such as correctness, timeliness, adaptability, reusability, etc.; and (2) A product family in which applications have many common features but still they are different in some significant ways. Model-based development is crucial in guiding

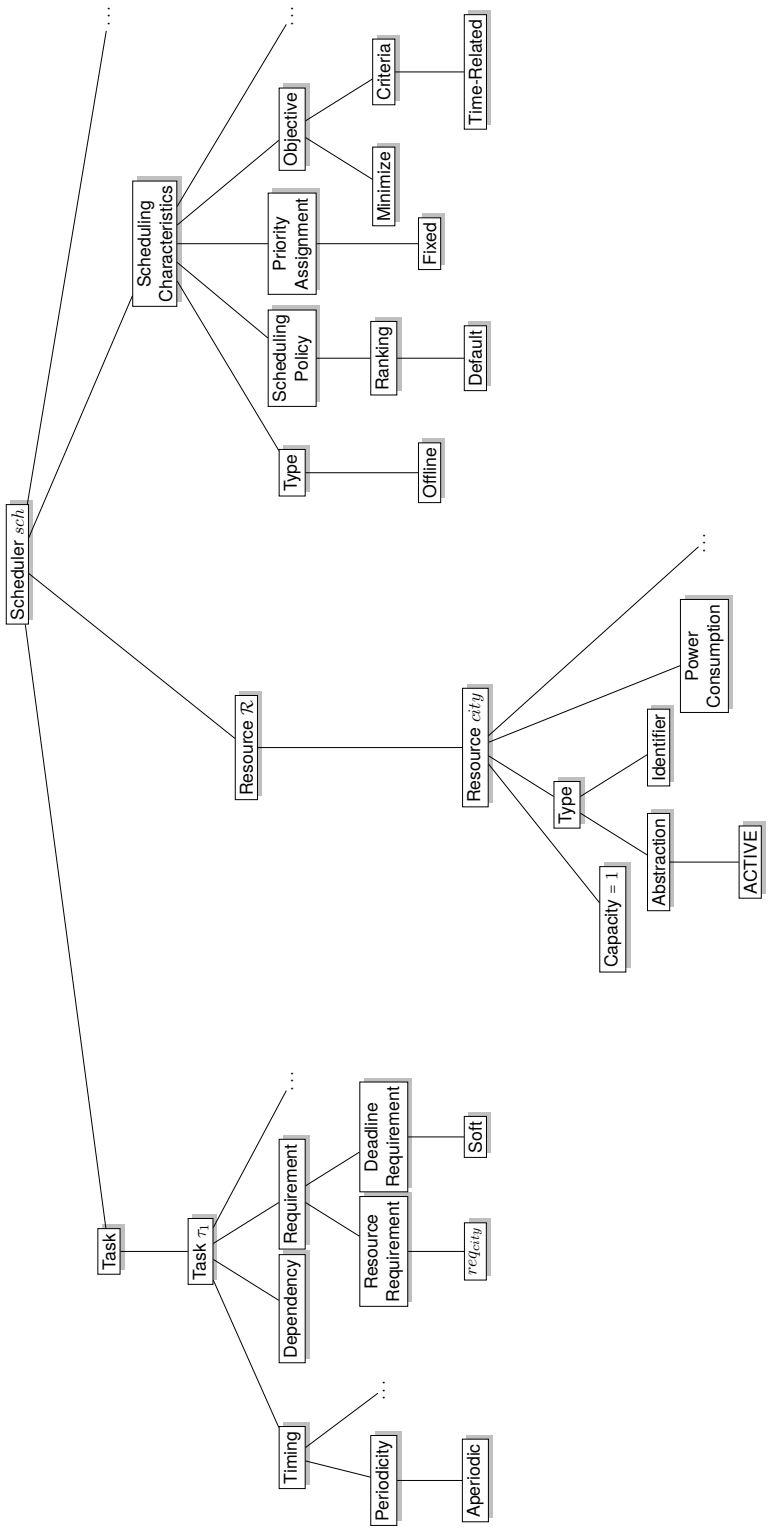


Figure 4.13: A configuration of the feature model expresses the TSP problem as defined in Section 4.5.8.

the software engineers in dealing with both problems. If effective models in the related domain are available, the modeling abstractions, attributes and relations can function as an effective guideline for the software engineer to shape the software system to be designed. Additionally, to address the second problem, the domain models must explicitly specify the commonalities and variabilities so that different versions of the family can be created with less effort. In particular, designing software systems that incorporate scheduling techniques can be experienced as a complex process due to the involvement of many parameters: tasks, resources, objectives, strategies, constraints, etc. While specifying these parameters, for example, the software engineer may need to define the following constraints in a very precise and robust manner: life-scope of tasks, periodic/aperiodic tasks, resource requirements, precedence relations, capacity constraints of resources, mutual exclusion constraints among resources and tasks, preemption and migration capability of resources, etc. Obviously, compared to software development from scratch, an effective feature model as presented in this chapter with all the necessary modeling abstractions can help reducing the complexity of designing such systems. A key property to accomplish effectiveness is the expressivity of the domain model. This property is evaluated in the following subsection.

Naturally, the adopted feature model notation and the associated tools support variability in the domain model. In the previous sections, we derive a generic model that is capable of expressing a large category of different schedulers and present a set of illustrative examples from the scheduling domain to demonstrate the expressivity of our model.

Expressivity of the Domain Model

To evaluate the expressivity of our feature model, we first refer to the 3-field notation introduced in Definition 2.1, which is used in the literature as a common notation to express scheduling problems. Each example given in this chapter is explained using the 3-field notation.

		α_1					α_2	
		P	Q	F	O	J	1	M
β	r_j	ESP	MRSP		OSP/PMTN, OSP		RMS	MRSP, ESP, OSP/PMTN, OSP
	$p_j = p$						TSP	
	$prec$		MRSP	FSP		JSP		MRSP, FSP, JSP
	$pmtn$		MRSP		OSP/PMTN		RMS	MRSP, OSP/PMTN
	$d_j = d$	ESP	MRSP		OSP/PMTN, OSP			MRSP, ESP, OSP/PMTN, OSP
	M_j		MRSP		OSP/PMTN, OSP	JSP		MRSP, JSP, OSP/PMTN, OSP
	s_{jk}		MRSP				TSP	MRSP
	$batch$		MRSP					MRSP
	$prmu$				FSP			FSP

Table 4.1: Coverage of examples in Section 4.5.

Table 4.1 depicts the overall scheduling domain. The letters shown in the cells correspond to the abbreviations of the examples presented in this chapter. The parameters in the rows of the table corresponds to the *scheduling characteristics* and the set of all these parameters

are denoted by β; whereas the columns of the table are grouped into two categories, namely the *machine identifier* and the number of machines which are denoted by α_1 and α_2, respectively. Within these categories, the parameters *P, Q, F, O, J, 1, M* are explained in detail in Section 2.1.1. As also argued in the link[1], the scheduling domain can be represented using these parameters as shown in Table 4.1. As it can be seen, in each column and row from top to bottom and left to right, respectively, at least one example resides. This illustrates that the examples cover at least one case of the parameters in the scheduling domain. Since we manage to define the examples using our feature model, we claim that our feature model is expressive enough to cover a large category of scheduling applications.

Verification

In Section 4.2 it was stated that the configured models must not violate the constraints of the domain. To detect whether the configured application violates the domain constraints or not, the configured feature model, which is the model of the application, is checked using Clafer [9]. The complexity of the model-checking process mainly depends on the characteristics of the feature model and as such it is more or less independent of the examples given in Section 4.5. We have verified the configurations of all the examples and in case all variabilities are bound. Approximately 0.25 seconds are needed for the verification of the configured applications using the MacBook Pro computer on 2.6 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 memory with MacOS X 10.9.5.

4.7 Conclusion

Scheduling techniques have been applied to a large category of software systems. Companies that manufacture such systems generally design and implement family of products. Designing and implementing robust scheduling systems, however, are not trivial. The cost of developing family of scheduling products can be reduced by adopting SPLE methods. There has not been a comprehensive work in the literature along this line.

As stated in Section 4.2, however, to be effective, an SPLE approach for designing a family of scheduling software must fulfill three requirements: reduced complexity, expressive domain models, and verification of the configurations of the domain models.

To address these challenges, in Section 4.4, a feature model is presented by using the scheduling theory. To this aim, a large set of relevant literature is studied and compared. The invariant characteristics in the existing literature were used to identify the commonalities, whereas the variations were used to derive the variabilities of the future model.

To exemplify the feature model, in Section 4.5, a set of canonical examples have been selected and defined. Each example is configured by using the abstractions as defined by the feature model.

To evaluate the reduced complexity and expressivity, in Section 4.6, first the domain of scheduling is defined as a matrix. Second, to evaluate the coverage of the examples, each canonical example presented in Section 4.5 is placed in the matrix. By this way,

[1] http://www2.informatik.uni-osnabrueck.de/knust/class

we showed that the examples cover a large cases in the scheduling domain, and as such the feature model is expressive. To verify the feature models, we adopt standard model-checking techniques where each configured model is translated to the language of the model checker.

In this chapter, we show that it is possible to adopt SPLE approaches in the scheduling domain, and effective domain models based on the feature model notation can be used for this purpose.

CHAPTER 5

Designing Reusable and Run Time Evolvable Scheduling Software

Implementing software systems that incorporate scheduling systems can be a time consuming process. In addition to dealing with well-known challenges in designing software systems, the software engineer has to define and implement the required tasks, resources, associated parameters, objectives, strategies, and the constraints, and/or algorithms. Dealing with all these constraints can be a very time consuming and error-prone tasks. For example, the constraints must be considered in a very precise and robust manner: Tasks have to be scheduled within their *life-scope*; periodic tasks have to be spawned at each inter-arrival time; the resource requirements of the allocation have to be realized for each task; the precedence relations have to be satisfied for each allocation; the capacity constraints of resources have to be satisfied; the preemption capability is supposed to be realized; the migration capability has to be satisfied; the mutual exclusion constraint among resources have to be satisfied. A highly *reusable* and *run time evolvable* framework designed specifically in the scheduling domain can ease this burden [53, 99].

In the literature, to the best our knowledge, studies on schedulers do not aim to develop a reusable and run time evolvable scheduling software implementation; they rather concentrate on specific application-dependent solutions. Since there exists hardly any generic and expressive library, framework or design environment, the software engineer has to implement all the necessary scheduling abstraction by herself, which increases the complexity and effort. In addition, due to lack of run time evolution support, maintenance of continu-

ously operating scheduling systems becomes a challenge.

This chapter introduces an object-oriented application framework called **FSF** which can be utilized to implement schedulers with a high-degree of reusability and run time evolvability. The utility of the framework is demonstrated with a set of canonical examples and evolution scenarios. The framework is fully implemented and tested.

This chapter focuses on the actual implementation of schedulers and as such it deals with the software architecture, run time environment and the result of execution of schedulers. Whereas Chapter 4 focuses on the early phases of product-line software development processes, as such it mainly deals with feature models and product (scheduler) configuration.

The remaining sections of this chapter are organized as follows: the next section presents the problem statement and objectives that are addressed in this chapter. The related work is summarized in Section 5.2. The software architecture of the framework is described in Section 5.3. In Section 5.4, as case studies, a set of canonical examples is introduced to evaluate the proposed framework. Finally, the evaluation of the framework and concluding remarks are presented in Section 5.5.

5.1 Problem Statement and Objectives

A considerable number of publications have been presented in the literature to guide software engineers in designing software systems [124]. It is generally agreed that certain quality attributes play an important role along this line. Although many proposals have been presented to enhance *reusability* and *run time evolvability* quality attributes in software development practices, there has been hardly any publication aiming at designing scheduling systems.

Our focus on these software quality attributes in this chapter is limited to the scheduling domain.

We adopt the term *reusability* as *ease of use* of a dedicated software library and associated tools to create a large category of scheduling systems. To this aim, to create a particular scheduling system, the code written from scratch must be much shorter than the code of the library that is reused. To fulfil the reusability requirement, the concept of *application frameworks* [78] can be utilized. An object-oriented application framework is defined within a context of an application domain and consists of a set of dedicated class hierarchies that can be instantiated and/or sub-classed to create a specific application in that domain. An important motivation for using this approach is two fold: to provide a reusable programming library for the programmer within the scheduling domain, and to give flexibility to the programmer to alter the library if needed.

We adopt the term *run time evolvability* as an ease of modification of an existing scheduling software with respect to a new *meaningful* set of user requirements. Since many scheduling systems, such as airport systems and production systems must be continuously operational, solutions to the new requirements must be introduced to the system at run time. The term *meaningful* here refers to the fact that requirements are natural and defined within a single application context. It is assumed, for example, that an airport scheduling system is not expected to evolve into an elevator scheduling system.

Within the context of this chapter, *run time evolvability* must be supported for the following cases:

A. Changing (adding, removing or modifying) resources and/or tasks;
B. Changing the optimization criteria based on the number of existing tasks.
C. Changing the timing constraints of the tasks.
D. Changing the dependency specifications among existing tasks.
E. Changing the attributes of existing tasks.

5.2 Related Work

There have been a considerable number of publications that report on the practical applications of frameworks [93, 78, 1]. To the best of our knowledge, none of them has been applied to the domain of scheduling.

Many researchers focus on scheduling problems, and much research work has been published in this area [61, 28]. Accordingly, a large category of algorithms has been developed.

In addition, different kinds of solver-based solutions have been studied and presented in the literature to address planning and scheduling problems [54, 66, 71]. There exists also a study [82] which presents a formal framework to implement reusable schedulers. However, these publications do not aim at creating a framework satisfying *reusability* and *run time evolvability* as defined in this chapter.

5.3 Framework Architecture and Configuration

We define software architecture as an abstract (blue-print) representation of a software system [5]. Diagrams describing software architecture can ease understanding the essential elements of software systems. In addition, software architecture plays an important role in determining the software quality of systems. In the following subsections, both static and dynamic models of the architecture of the framework are presented. The static model is expressed in the UML component diagram notation, which shows the logical structure of our framework. It represents the important abstractions, called components, which have well-defined interfaces. Each component corresponds to a piece of object-oriented program that implements a logical concern. A component can only be invoked through its interface functions. Subsystems group related components together. The dynamic model is expressed in the UML sequence diagram notation. It shows how components interact with each other to perform a system-wide behavior. In our example, we utilize sequence diagrams to express the instantiation processes to create scheduling software.

5.3.1 Component Diagram of the Framework Architecture of FSF

FSF software architecture has been designed after an extensive study of the scheduling theory. The components and the relationships of the architecture are derived from the essential concepts of the theory. Furthermore, the architecture is justified by considering the applications of the theory to a large category of scheduling problems. It is implemented using an object-oriented library and supported by a set of open-source software and own developed tools.

FSF software architecture is symbolically shown in Figure 5.1. The overall architecture is depicted as a large rectangle with thick lines and denoted by the UML stereotype (≪*System*≫). The architecture consists of three subsystems: *Resource*, *Task* and *Scheduler*. These are shown as dark gray rectangles and are denoted by the UML stereotype (≪*Subsystem*≫). The components are shown as light gray rectangles and placed in the subsystems. The users of the system are shown using the UML actor notation (). These indicate the roles and named as *Resource Designer, Task Designer* and *Scheduler Designer*. In practice, one or more person can fulfill the roles. Interfaces are shown using specific symbols. Components exchange information among each other through interfaces. These are classified under provided () and required () interfaces. The direction of an arrow indicates a dependency relationship from a required to a provided interface.

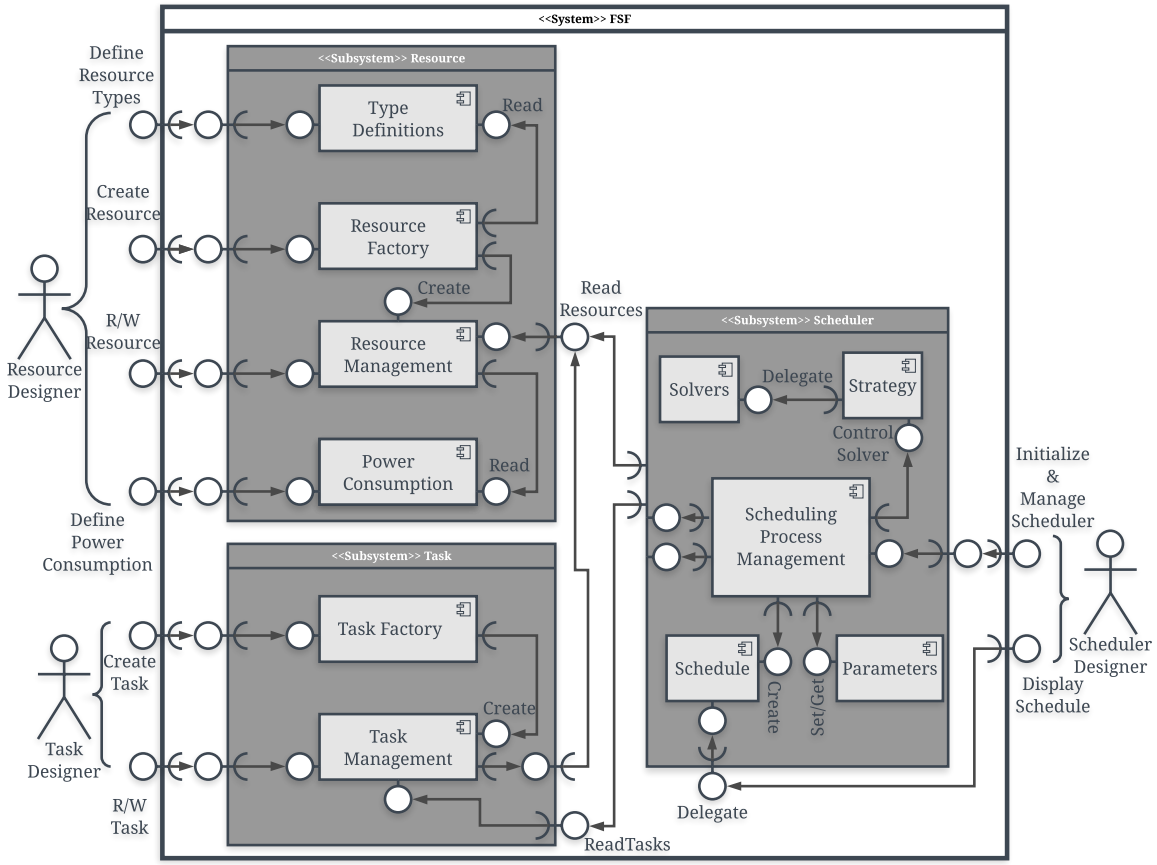


Figure 5.1: **FSF** software architecture depicted as a UML component diagram notation.

Resource Subsystem

The subsystem *Resource* contains four components: *Type Definitions*, *Resource Factory*, *Resource Management* and *Power Consumption*. The interfaces of these components are exported to the user *Resource Designer*.

The component *Type Definitions* keeps tracks of all resource types defined. Currently, three kinds of abstract resource types are provided: *Active*, *Passive* and *Composite*. Abstract resource types are parameterized to create the concrete resource types such such as *Memory, CPU, Machine, Antenna, Bus, GPU, Sensor,* etc.

The user *Resource Designer* interacts with the component *Resource Factory* to create instances of the desired resources by "using" the predefined concrete resource types.

The component *Resource Management* contains all the instances and provides an interface to the user to read and write their properties. Furthermore, a read interface is provided to the subsystem *Scheduler*.

The component *Power Consumption* is used to read and/or write the power consumption related properties of instances. The motivation for defining a separate component is to provide sharing: different kinds of instances may share similar power consumption characteristics. In this case, these instances can simply denote to the same power consumption definition.

The rationale to define the subsystem *Resources* in this way is to create a hierarchically organized resource structures, which is motivated in the following: According to [139], the resources in computing systems are classified as either *active* or *passive*. While a task can only be executed on an *active* resource, it may also require one or more *passive* resources. The traditional resource model introduced in [61] does only support active resource. This makes it cumbersome to express tasks that require also passive resources.

Recently, general purpose computing on graphic cards (GP-GPU) has gained importance to solve the problems which can be divided into sub-problems such as rendering images, performing audio operations, etc., where each of them can be executed in parallel. While defining resources, it may be necessary to consider the hardware architecture of GPU's. This requires hierarchical resource models where each resource element in hierarchy may define its own rules of accessibility. To represent complex resource structures such as the ones adopted in GPU's, we define both active, passive and composite resources. Active and passive resources are represented as the terminal nodes in the hierarchy.

A composite resource may embody one or more active, passive and/or composite resources. The accessibility of resources is defined as follows: A task running on an active resource has access to the terminal resources of each of its ancestors, and all the terminal resources of each of its sibling composite resources. For example, assume that a task executes on cpu shown in Figure 5.2. It has access to the terminal resources of its ancestors, antenna, bus, memory-1 memory-2, temp_sensor and prox_sensor; and the terminal resources of its sibling composite resources, cache_cpu-1 and cache_cpu-2.

Recently, reducing energy consumption has become more and more important. For this reason, for example, *Dynamic Voltage Scaling (DVS)* has been introduced to reduce power consumption of processing units [30, 95, 108]. This requires dedicated task scheduling.

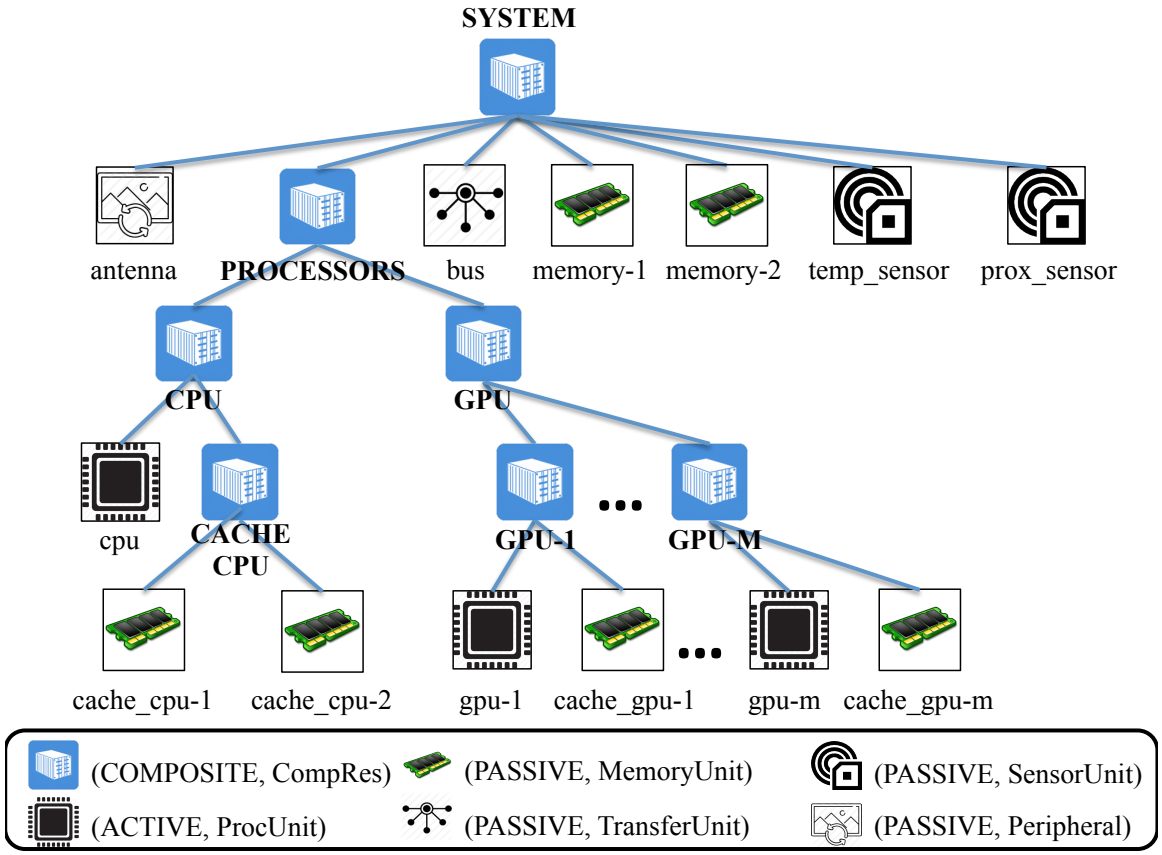


Figure 5.2: An example *resource tree* defines the accessibility relation among the resources.

To reduce energy consumption within the timing constraints, the scheduler has to consider both voltage levels and corresponding executing speeds.

To express these scheduling problems, in **FSF** power consumption is explicitly modeled in two options: *discrete-* and *continuous-state power consumption* options. For the resources specified as the former, there exist power states, each of which is the pair of the value sc ($0 < sc \leq 1.0$) and the power consumption value pc; whereas for the resources belonging to the latter, any value between minimum and maximum power scales can be chosen.

In [69], resources are categorized as *single-* or *multi-unit* capacity. A resource with *multiple units*, each of which is serially accessible entity is reserved partially to more than one tasks; whereas a *single-unit* resource can only be accessed by a task at a time. Therefore, we defined the numeric attribute *capacity* representing the number of units for resources with both *single-* or *multi-unit* capacities.

To express simultaneous access in utilizing the capacity of a resource as explained in [28], we defined the term *mode*. If a resource operates in a *shared* mode, each capacity unit can be accessed by tasks, simultaneously. Otherwise, the resource can either work in *capacity-*

based exclusive mode in which each capacity unit can be accessed by at most one task at a time or *semantic-based* exclusive mode in which the utilization of any capacity unit of a resource is blocked in case a task is executing on one of its exclusive resources.

Task Subsystem

The subsystem *Task* includes two components: *Task Factory* and *Task Management*.

The user *Task Designer* utilizes the component *Task Factory* to create instances of tasks.

The component *Task Management* contains all the instances and provides an interface to the user *Task Designer* for reading and writing their properties. Furthermore, a read interface is provided to the subsystem *Scheduler*.

Like resources, tasks can also have composite structure. A task can be classified either *composite* or *terminal*. The tasks assigned to a composite resource are recursively dispatched to resources within its life-cycle until there are no composite tasks left. The design rationale for this way of allocation of tasks is to ease the scheduling process since it divides a scheduling problem into simpler sub-problems and deals with each of them in its own time scope. Since the computational complexity of each sub-system is supposed to decrease, this process should reduce the execution time of the scheduler. In addition, it groups the relatively similar sub-tasks that have the same resource requirements.

A number of task attributes are not shown in Figure 5.1 for brevity reasons. Some essential ones are described in the following:

- Class *Time* has been defined to provide system-wide consistency for time-related attributes, and it is used within the definition of tasks. It has class variables such as *resolution of time*, and *unit of time*.
- The attribute *precedence constraint* is also used in the definition of tasks. It ensures tasks to execute in certain order [28]. The predecessor task has to complete its execution to let the successor task of it start. In [3], a precedence constraint has been expressed by *data dependency*; a predecessor task fires a token when it finishes and the successor task has to consume this token in order to start. We have adopted a more expressive constraint specification, which extends the token-based dependency with the relational operators AND, OR and the temporal operator AFTER.
- The attribute *resource requirements* is defined for each task to express capacity requirements for a set of actual resources belonging to the same concrete resource type.

For brevity, the attributes that are used in tasks such as *time*, *precedence* and *resource requirements* are not shown in Figure 5.1. These are set in the component *Task Management*.

The other timing related attributes of tasks will be handled in Section 6.3.

Scheduler Subsystem

The subsystem *Scheduler* consists of five components: *Scheduling Process Management*, *Parameters*, *Strategy*, *Solvers* and *Schedule*.

The component *Scheduling Process Management* functions as the coordinator. To this aim, it first interacts with the user *Scheduler Designer* to set the scheduler-related properties and store them in the component *Parameters*, then retrieves the information about the resources and tasks from the corresponding subsystems, determines the strategy to be used, and

activates the solver. Finally, it stores the result of the solver in the component *Schedule*.

The component *Strategy* determines which solver algorithm should be utilized. To accomplish this, it interacts with the *Scheduling Process Management* and makes a request to the component *Solver* to select a particular solver algorithm.

The component *Solvers* incorporates a set of solver algorithms.

In addition to contain the schedule, the component *Schedule* provides the scheduling execution context and offers various utilities to display the results in different output format.

To realize the token-based data dependency, each scheduler instance includes an attribute corresponding to a token pool.

We consider the scheduling problem as an optimization problem; This requires the definition of the optimization criteria. In Section 2.1.3, various criteria are shown in Table 2.1. The *purpose* of the optimizer is to either *minimize* or *maximize* the selected criterion. In our framework, all the criteria that are shown in the table have been implemented.

The scheduling policy is used to determine the relative importance of tasks for an underlying system. Currently, our framework supports the policies FIFO (*First-In-First-Out*), EDF (*Earliest Deadline First*), SJF (*Shortest Job First*), LJF (*Longest Job First*), RM (*Rate-Monotonic*, i.e. Shortest Period First), ERT (*Earliest Release Time*). It is also possible to define new policies by modifying and extending the related parts in the framework.

Since our framework adopts a solver-based approach, the solver has to be specified as well. The available solvers are SCIP, *MiniSat*, *MipWrapper*, *Mistral*, *Mistral2*, *SatWrapper*, *Toulbar2* and Walksat.

5.3.2 Instantiation of the Framework to Create a Scheduler

Application frameworks [78] offer a reusable library in a certain domain which must be instantiated and if necessary extended to create a particular application. In our approach, to create a dedicated scheduler, the framework must be instantiated according to the requirements of the desired scheduler. Since run time evolvability is one of the key objectives, an instantiation process is realized at run time.

In the following subsections we illustrate an instantiation process for a particular scheduling example in three steps: instantiation of resources, tasks, and the scheduler.

Instantiation of Resources

Assume that we would like to create an implementation of a resource model with two elements: cpu and mem. In **FSF**, as shown in Figure 5.3, this can be realized at run time by calling the necessary operations on the corresponding components/objects.

Figure 5.3 shows a sequence diagram in UML to illustrate the creation process. On the top left of the figure, as an UML actor notation, the role *Resource Designer* is shown who is in charge of defining the resource model. The vertical bars on the most left side of the figure depict the actions that are initiated by *Resource Designer*. The other vertical bars correspond to the instances which are involved in the interaction process. The types of these components/objects, namely *SystemCompositeResource, ResourceFactory* and *PowerFactory*, are represented at the top of the picture and linked to their instances by dashed-lines: The sequence of interactions are from top to bottom. In our framework, *system* incorporates all

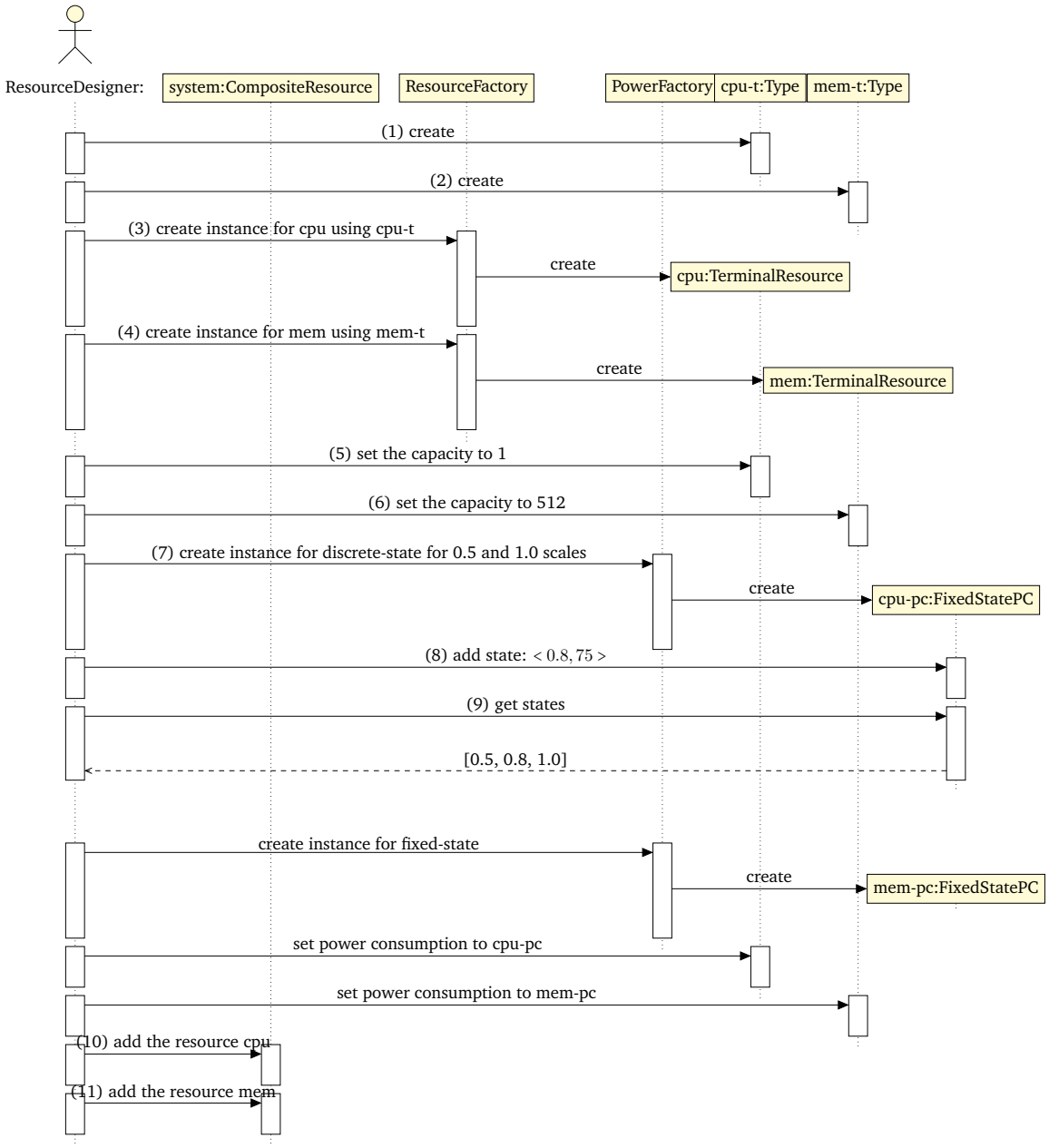


Figure 5.3: Sequence diagram to create and configure cpu and mem resources; and add them to the composite resource system.

the resources that are created. Initially, *system* is empty.

As shown in Figure 5.3, the sequence of calls has the following meaning: In the first two calls (1) and (2), the *Resource Designer* creates the identities of cpu and mem types. The text on the call arrows illustrates their meaning informally. In calls (3) and (4), by calling on *ResourceFactory* with the identities as parameters, actual resource objects are created. In calls (5) and (6), the capacities of cpu and mem are defined as 1 and 512, respectively. In calls (7), (8) and (9), the power consumption characteristics of the resources are defined as *discrete states*. For illustration purposes, the call (9) is defined as a read operation. The dashed line from right to left illustrates the response for this call. Finally, in calls (10) and (11) are used to add the created resources to the system.

Instantiation of Tasks

Assume that we would like to create 3 tasks called t1, t2, and t3. The actor *Task Designer* represents the role who creates, instantiates and configures the tasks. In Figure 5.4, call (1) is used to create an instance of the task t1. Calls (2), (3), and (4) are used to set the timing parameters of this task. The calls (5) and (6) are used to set the resource requirements of t1. Similarly, calls (7) and (8), and (9) and (10) symbolize the creation and definition of the tasks t2 and t3. For brevity the details are not shown. Calls (11), (12), and (13) are used to illustrate possible definitions of dependencies between tasks. An interested reader should refer to the **FSF** website (previously called LFOS [1]) for the details.

Instantiation of the Scheduler

This part illustrates how a scheduler can be created at run time based on the tasks and resources defined in the previous sections. The actor *Scheduler Designer* represents the role who creates, instantiates and configures the scheduler. In Figure 5.5, call (1) symbolizes the initial creation operation of the necessary objects. Call (2) initializes the scheduler with the previously defined tasks. Call (3) is responsible for setting the resource tree as presented in Section 5.3.2. Calls (4) to (10) symbolize how the important parameters of the schedulers are set. Where necessary, the component scheduler delegates the calls to the responsible components. Finally, call (11) starts the scheduling process. This call sets the parameters of the solver, instantiates it with the given type and starts the algorithm of the solver. The solver returns a schedule, which can be in turn utilized to execute the tasks accordingly. Eventually, the obtained schedule can be plotted. Call (12) represents such an action.

5.4 Case Studies

In Section 5.1, design and implementation of a scheduler framework which has a high-degree of *reusability* and *run time evolvability* are defined as the two key research objectives. To obtain these quality attributes, a common practice today is to design an application framework. As a design method, we have adopted the method of *deriving the key abstractions of the framework from the corresponding theories* [8]. In application frameworks, to

[1] `https://github.com/gorhan/LFOS/tree/master/LFOS`

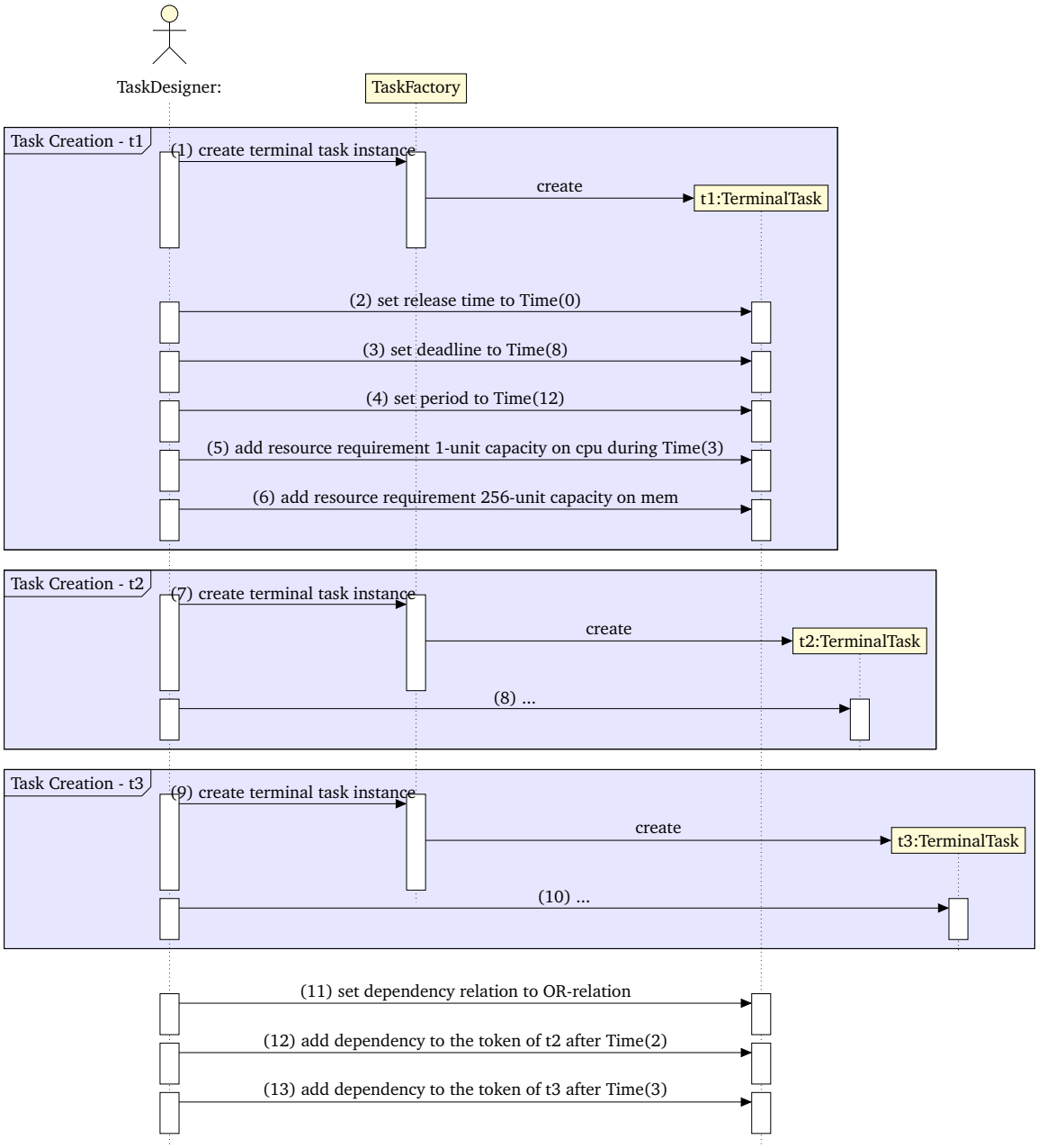


Figure 5.4: Sequence diagram to create and configure the tasks t1, t2 and t3.

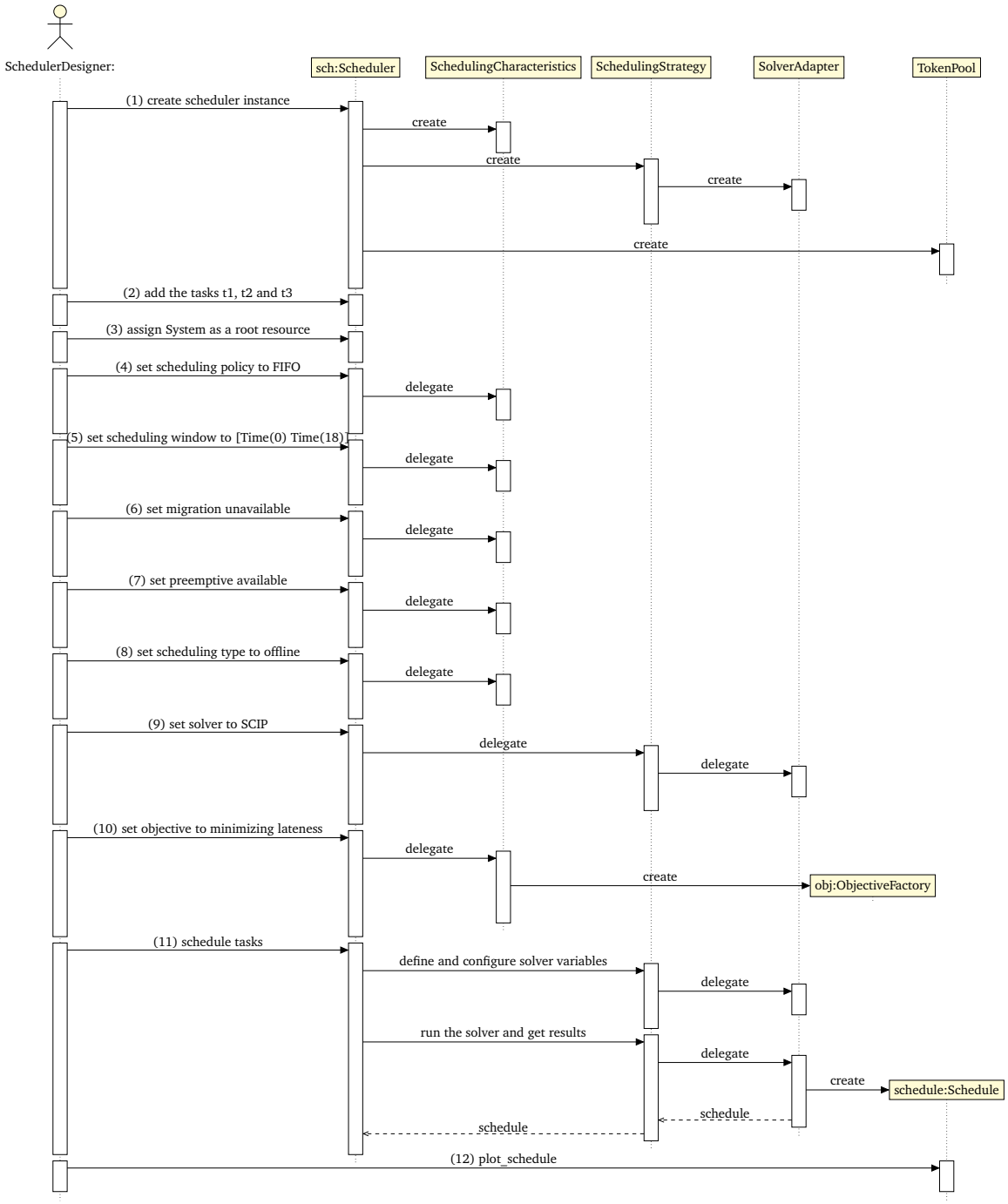


Figure 5.5: Sequence diagram to create, configure scheduler sch and get the optimized schedule as a graph.

provide a high-degree of *reusability* in a given domain, the framework library must be expressive enough to implement the well-known examples of that domain. To this aim, to demonstrate *reusability* of **FSF**, this section presents a set of canonical examples from the scheduling domain which are instantiated from the framework. To demonstrate the quality attribute *run time evolvability*, each example is extended with a set of evolution scenarios.

5.4.1 Rate Monotonic Scheduling (RMS)

Rate Monotonic Scheduling (RMS) is a scheduling method deployed in real-time operating systems. Although in real-time systems tasks can be defined both as periodic and aperiodic tasks, in RMS only *periodic* tasks are assumed. The priority values of the instances of the tasks are *fixed* and determined with respect to inter-arrival time (period) of instances. The shorter period a task has, the more privileged it becomes. Unlike aperiodic tasks, the periodic ones have hard deadline requirements and these are equal to the beginning of the next request of the task. The scheduling process is preemptive. As a consequence, a task can never be in a waiting state for a less privileged task [94].

Initial Requirement

Assume that the following RMS is desired, which is expressed using Definition 2.1 which is presented in Section 2.1:

$$1|pmtn, r_j|\sum_j w_j L_j. \tag{5.1}$$

In fact, an RMS is a "*a fixed-priority online scheduling problem for scheduling independent, preemptable, periodic tasks on a single processing unit aiming at minimizing the total weighted lateness objective*".

Implementation of the Initial Requirement

We assume that our taskset consists of four tasks, and a single resource named cpu. We specify the end of the scheduling window as the completion time of the latest second instance of a task in the taskset[2].

As for the required parameters, we have generated the timing attributes of the tasks randomly; the values are shown in Table 5.1.

Tasks	Release Times	Execution Times	Period	Priorities
τ_1	$r_{(\tau_1,0)} = 0$	$c_{(\tau_1,\text{cpu})} = 7$	$p_{\tau_1} = 29$	$\star$
τ_2	$r_{(\tau_2,0)} = 3$	$c_{(\tau_2,\text{cpu})} = 3$	$p_{\tau_2} = 28$	$\star\star$
τ_3	$r_{(\tau_3,0)} = 2$	$c_{(\tau_3,\text{cpu})} = 2$	$p_{\tau_3} = 22$	$\star\star\star$
τ_4	$r_{(\tau_4,0)} = 1$	$c_{(\tau_4,\text{cpu})} = 4$	$p_{\tau_4} = 25$	$\star\star\star$

Table 5.1: Randomly generated values of the timing attributes for the taskset. The number of stars in the column priority corresponds to the relative importance of the corresponding task in the taskset.

[2]An interested reader can refer to our repository: `https://github.com/gorhan/LFOS/blob/master/Tests/RMS.py`

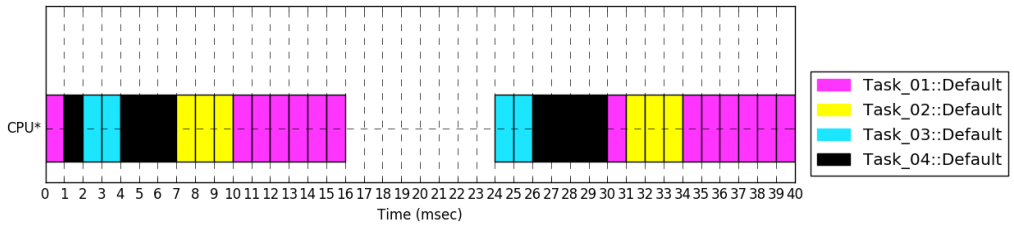


Figure 5.6: A graphical output of the example RMS instantiated with the initial requirements.

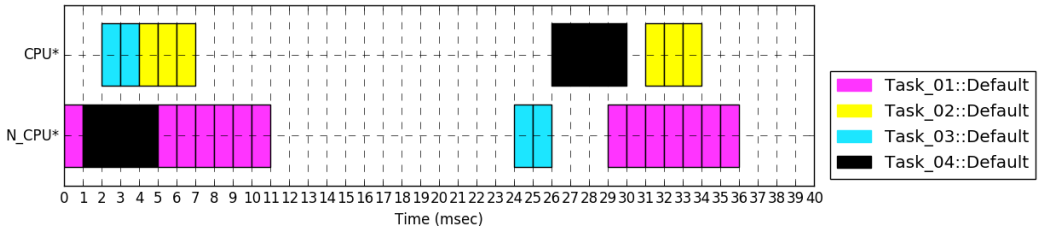


Figure 5.7: A graphical output of the example RMS according to the new requirements.

Based on this initial requirement, a scheduler is instantiated and executed. The result is displayed in Figure 5.6. Since τ_4 is more privileged than τ_1, the resource is reserved to τ_4 at $t = 1$. Again for the same reason, τ_3 and τ_2 take the permission of utilization of the resource cpu at $t = 2$ and $t = 31$, respectively.

Evolution of the requirement: Platform is extended with an additional CPU

To demonstrate *run time evolvability* of the scheduler, we assume the following change in the requirements: The system is migrated to a new platform where 2 CPU's are utilized. The additional CPU has the same characteristics as the initial one.

The evolved requirement can be defined as follows:

$$P2|pmtn, r_j|\sum_j w_j L_j. \tag{5.2}$$

Since both resources are assumed to be identical, the execution times of the tasks do not change.

Implementation of the new requirement

We will now extend the implementation of the previously instantiated scheduler at run time in 3 steps: (i) creating an additional resource; (ii) setting the resource properties, which are equal to the ones of the first resource; and (iii) introducing this new resource instance to the system.

After run time evolution, a new scheduler is configured. The output of it is shown in Figure 5.7. Since there are no available resources and the task τ_4 is ready to execute, the scheduler preempts the task τ_1 at 1. Due to the additional resource, the second instances of the tasks have completed their executions in 4 units of time earlier than the initial case.

5.4.2 Multiple Resource Scheduling (MRS)

This example is defined to demonstrate the implementation of a complex scheduling problem involving multiple tasks and resources.

Initial Requirement

Assume that there are 2 tasks and 2 resources each with different characteristics. There are periodic and aperiodic tasks, which are referred to as *system-level* and *application-level* tasks, respectively. A system-level task has a higher priority than an application-level task. Unlike an application-level task, a system-level task cannot be suspended. It is also assumed that an application-level task requires a system-level task to complete. Therefore, aperiodic tasks *depend* on periodic tasks. The resources in this example are classified as active and passive resources and named as processing unit and memory, respectively. The active resources can process the tasks with different speeds by adjusting the exerted power. This is not possible with the passive resource. The scheduling process is defined as *offline*. It is also assumed that some tasks are preemptable. As a design choice, the tasks are prioritized with respect to their release times. Therefore, the scheduling policy is defined as Earliest-Release-Time-First.

Instances of a task are re-prioritized after the completion of each instance for the following 2 reasons: (1) the release time of each instance can be different; and (2) an instance with earliest release time compared to the other instances may not have the same release time characteristics in the next scheduling window with respect to its period.

Finally, the overall objective of the scheduler is defined as minimizing the consumed power on the resources while executing the tasks.

The requirement is expressed using Definition 2.1:

$$Q2|r_j, d_j, prec, pmtn, M_j, s_{jk}, batch|\sum_{i,j} PW(i,j), \tag{5.3}$$

where $PW(i,j)$ is the total exerted power of the resource i on running the task j.

Implementation of the Initial Requirement

The framework is configured in the following way[3]: There are two instances for each task: $\tau_{(1,1)}$ and $\tau_{(1,2)}$, and $\tau_{(2,1)}$ and $\tau_{(2,2)}$ are defined as instances of periodic tasks and aperiodic tasks, respectively. There are also two instances for each resource: cpu1 and cpu2, memory1 and memory2 are instances of active resources with single-unit capacities, and as passive resources with 512-unit capacities, respectively. In addition, in terms of power consumption, the active resources have two modes, half-scale (0.5) and full-scale (1.0). If a resource is running in half-scale mode, the execution time of any task on that resource becomes two times longer than its actual execution time.

In Table 5.2, the timing attributes of instances of tasks are shown. Since the attributes used for periods are irrelevant for aperiodic tasks they are shown as NA (not applicable).

[3]An interested reader can refer to our repository: `https://github.com/gorhan/LFOS/blob/master/Tests/MRSP.py`

Tasks	Release Times	Execution Times	Deadlines	Period
$\tau_{(1,1)}$	$r_{(\tau_{(1,1)},0)} = 0$	$c_{(\tau_{(1,1)},\text{cpu1})} = c_{(\tau_{(1,1)},\text{cpu2})} = 3$	$d_{(\tau_{(1,1)},0)} = 6$	$p_{\tau_{(1,1)}} = 6$
$\tau_{(1,2)}$	$r_{(\tau_{(1,2)},0)} = 2$	$c_{(\tau_{(1,2)},\text{cpu1})} = c_{(\tau_{(1,2)},\text{cpu2})} = 1$	$d_{(\tau_{(1,2)},0)} = 4$	$p_{\tau_{(1,2)}} = 4$
$\tau_{(2,1)}$	$r_{(\tau_{(2,1)},0)} = 3$	$c_{(\tau_{(2,1)},\text{cpu1})} = c_{(\tau_{(2,1)},\text{cpu2})} = 2$	$d_{(\tau_{(2,1)},0)} = 14$	$p_{\tau_{(2,1)}} = NA$
$\tau_{(2,2)}$	$r_{(\tau_{(2,2)},0)} = 8$	$c_{(\tau_{(2,2)},\text{cpu1})} = c_{(\tau_{(2,2)},\text{cpu2})} = 1$	$d_{(\tau_{(2,2)},0)} = 11$	$p_{\tau_{(2,2)}} = NA$

Table 5.2: The timing attributes of tasks. (NA = Not Applicable)

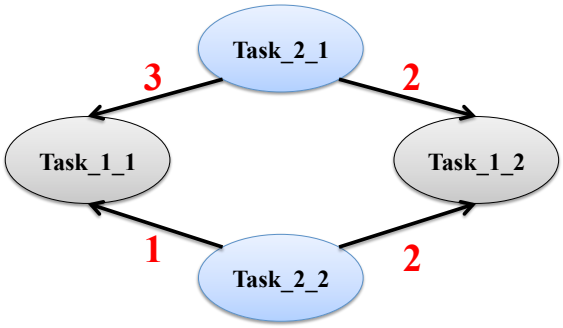


Figure 5.8: Data dependency graph.

Data dependencies between tasks are shown in Figure 5.8. The numbers above the edges represent the *sequence dependent setup times* explained in [109]. The direction of an arrow indicates the dependency of tasks. The target tasks $\tau_{(1,1)}$ and $\tau_{(1,2)}$ depend on the source tasks $\tau_{(2,1)}$ and $\tau_{(2,2)}$. An instance of a task is eligible to execute at any time after at least one of the dependency relations is satisfied. We term this type of dependency as *OR-dependency*.

In addition, the instances of tasks $\tau_{(1,1)}$ and $\tau_{(2,1)}$ are instantiated with 650- and 140-unit capacities on the passive resources, respectively.

With these settings, the framework is instantiated and executed. The result is shown in Figure 5.9. As it can be seen from the figure, to lower the power consumption, the scheduler utilizes the resource cpu2 at half-scale mode. Due to the dependency relation defined for $\tau_{(2,1)}$, this task cannot start immediately after its release time, and consequently its completion is deferred. The resource cpu1, therefore, has to operate at a full-scale mode at $t = [9, 11)$ so that the tasks $\tau_{(2,2)}$ and $\tau_{(1,2)}$ can be completed within their deadlines.

Run time evolution of the requirement: change of the objective due to increasing task demand

To evaluate the run time evolvability of the example, we introduce the following new requirement. Assume that the number of instances of tasks is becoming more than the underlying system can support. In this case, the overall objective of the scheduling process is changed to minimize the total weighted lateness. The justification of this change is to avoid missing deadline. Since the evolution is not correlated with the scheduling parameters but

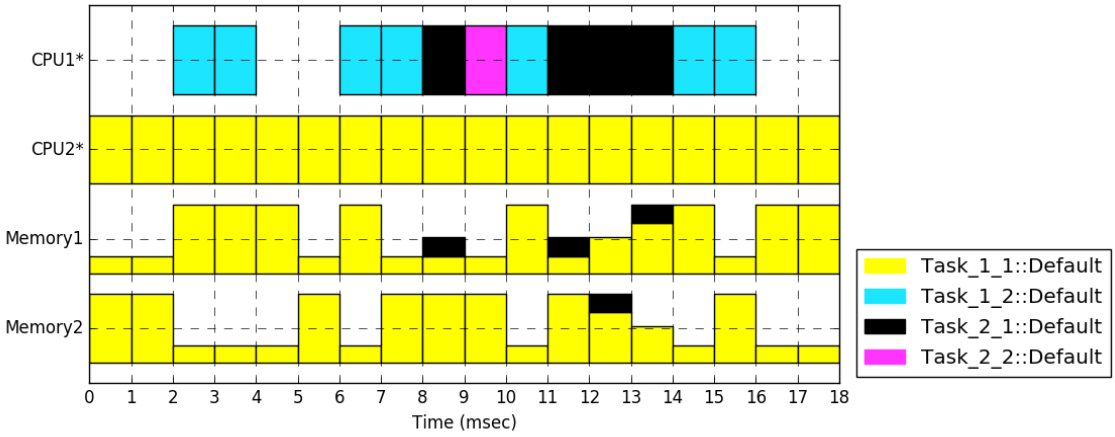


Figure 5.9: A schedule aimed to minimize power consumption of the resources.

the objective, the specification of the evolved scheduler is expressed as follows:

$$Q2|r_j, d_j, prec, pmtn, M_j, s_{jk}, batch| \sum_j w_j.L_j. \tag{5.4}$$

To instantiate the scheduler at run time with respect to the new requirements, we execute the following calls: (i) altering the objective to minimizing the total weighted lateness; and (ii) calling the method schedule on *Scheduler*.

Implementation of the new requirement

To realize the new requirement, the following modifications are carried out at run time: (i) before each scheduling process, a conditional statement is added to check the number of instances. (ii) if there exist more than five task instances, the overall objective is set to the total weighted lateness. After the re-instantiation, the schedule is computed. The result is shown in Figure 5.10. The scheduler chooses to run the resources at a full-scale mode and complete the tasks as soon as possible to realize the desired objective.

5.4.3 Job-shop Scheduling (JS)

In Job-shop scheduling, the resources are identical. They show differences based on the dependency relation between tasks and the machine eligibility of them. An interested reader can refer to Section 2.1.1. In JS, each job has a predetermined path of execution and it is not necessary for a job to visit each resource during its execution [16]. In addition, the execution orders of jobs on resources may be different. For instance, the job k_1 and k_2 may have the execution orders (3,2,4,1) and (1,4,2,3) on the resources, respectively. The eligible resources for the third activities of the jobs are denoted by $\mu_{(3,k_1)} = 4$ and $\mu_{(3,k_2)} = 2$.

In the following sub-section, we present a specific JS example.

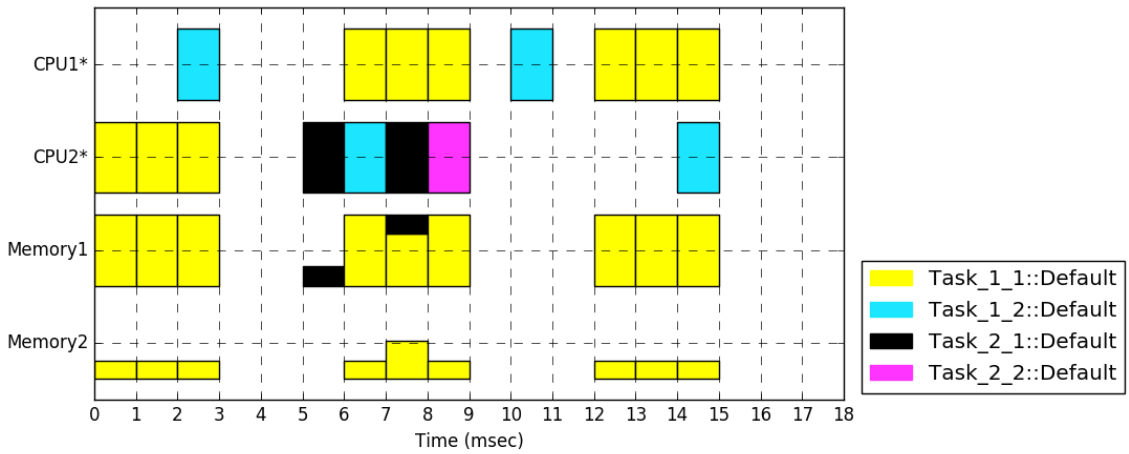


Figure 5.10: A schedule of the example which aims at minimizing total weighted lateness.

Initial Requirements

Assume that there are three tasks with 3, 4 and 3 instances, respectively. The machine environment consists of four resources with single-unit capacity.

From the perspective of scheduling process, the priority is determined with respect to the execution duration of a job, which is constant. Due to this condition, the priorities are assigned to jobs once and remain constant unless they are modified. The scheduling algorithm is chosen to be *non-preemptive*. The objective is to minimize the makespan.

The initial requirement is expressed as follows:

$$J4|M_j, prec|C_{max}. \tag{5.5}$$

Implementation of the Initial Requirement

In our implementation, jobs and machines are modeled as instances of tasks and resources, respectively, We have adopted the parameters defined in Example 7.1.1 in [109][4]. Each task is defined *aperiodic* and *non-preemptable*. To oblige each instance of a task to execute on a specific resource, the *machine eligibility* constraint is defined. In addition, we define the *dependency* relation among instances of tasks to specify the execution path of a task on different resources. Since the tasks have no release time or deadline constraints, a task may start to execute if the dependency constraint is satisfied. The eligible resources and execution times of the instances of tasks are given in Table 5.3.

The execution paths of jobs as defined in JS are expressed as dependency relations of tasks. These are shown in Figure 5.11.

The framework is instantiated and executed. The obtained schedule can be seen in Figure 5.12. The tasks $\tau_{(4,2)}$ and $\tau_{(3,1)}$ are ready to execute at time 18 on the resource Re-

[4]The details about the implementation can be found in our repository `https://github.com/gorhan/LFOS/blob/master/Tests/JSP.py`

Run time Tasks	$\mu_{(i,j)}$	$c_{(i,j)}$
$\tau_{(1,1)}$	1	10
$\tau_{(2,1)}$	2	8
$\tau_{(3,1)}$	3	4
$\tau_{(1,2)}$	2	8
$\tau_{(2,2)}$	1	3
$\tau_{(3,2)}$	4	5
$\tau_{(4,2)}$	3	6
$\tau_{(1,3)}$	1	4
$\tau_{(2,3)}$	2	7
$\tau_{(3,3)}$	4	3

Table 5.3: Execution times of the instances of tasks.

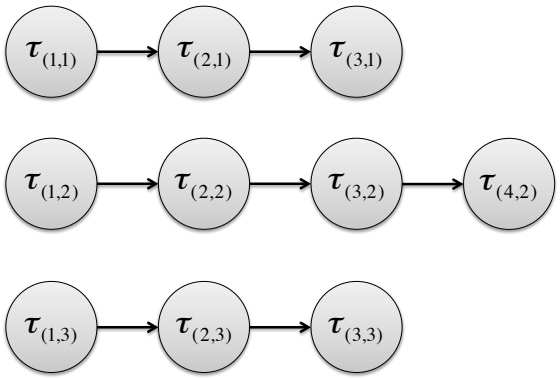


Figure 5.11: Dependency graph of tasks in the example.

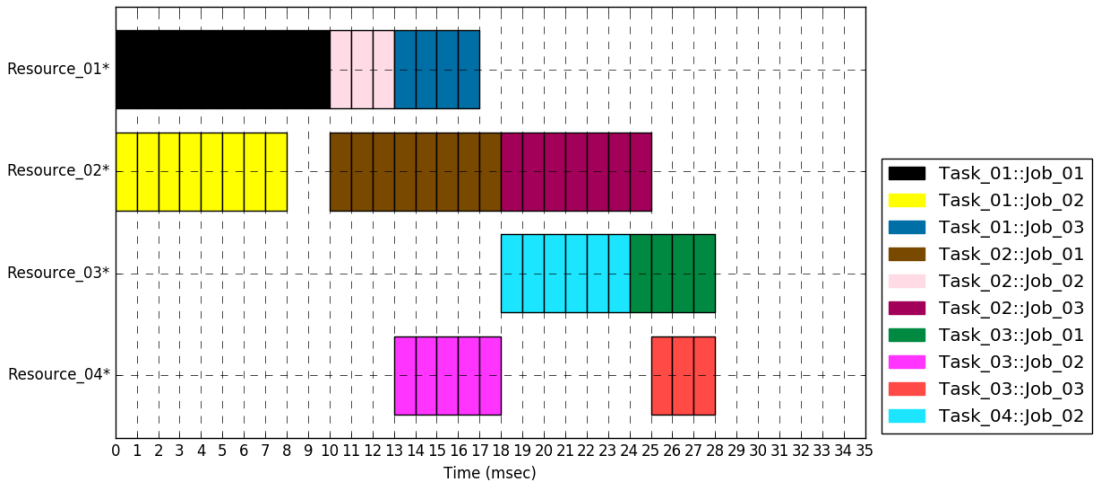


Figure 5.12: An optimized schedule by minimizing the makespan.

source_03. Since the task with the longer execution time ($\tau_{(4,2)}$) has higher priority than the others, it executes first.

Run time evolution of the requirement: adding release time constraint

As a run time evolution of the previous example, now assume that some of the tasks cannot execute immediately after requesting a schedule. The new problem definition is accordingly expressed as follows:

$$J4|M_j, prec, r_j|C_{max}. \tag{5.6}$$

Implementation of the new requirement

To implement this evolution request, a new release time for the targeted instances of tasks must be set. To this aim, we define the release times of the tasks $\tau_{(1,1)}$ and $\tau_{(2,3)}$ as 2 and 21, respectively, so that, to this aim, the corresponding method to set the release time of each instance of tasks is called.

As shown in Figure 5.13, due to the restriction on release times, the completion time of the latest task $\tau_{(3,3)}$ is delayed to 31.

5.4.4 Flow-shop Scheduling (FS)

Flow-shop Scheduling is commonly utilized in industrial production. It is a kind of shop problem where jobs have the same execution order on each machine, $\forall_{i,j}\ \mu_{(i,j)} = i$. Therefore, the execution path of jobs in the first machine have to be preserved on other resources, which is defined as *permutation* in [109]. Since the process of jobs in an assembly line should not be altered, the scheduling algorithm is not preemptive. Furthermore, all the queues are postulated to operate under the policy *First-In-First-Out (FIFO)*.

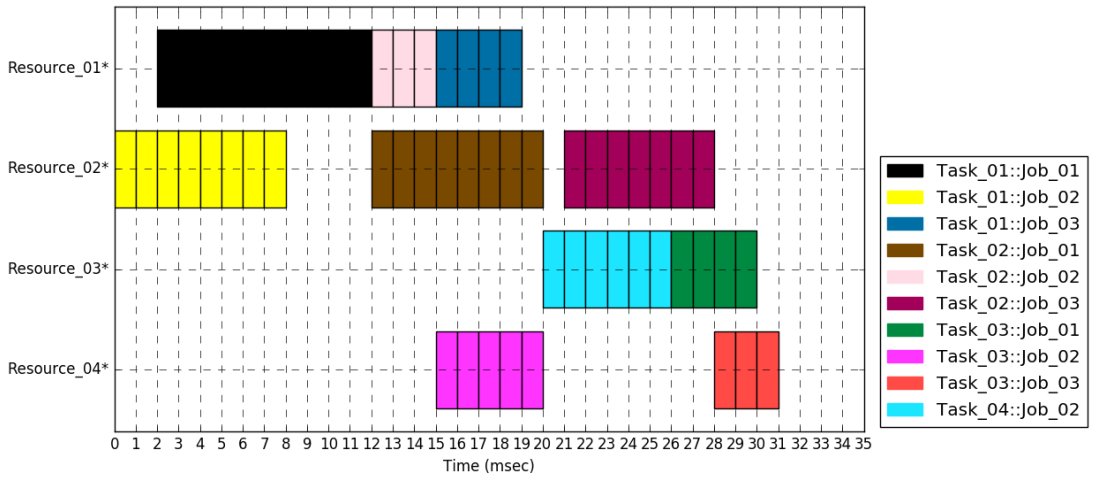


Figure 5.13: A graphical representation of the schedule of the evolved JS example.

Initial Requirements

In our framework, jobs and machines are represented as instances of tasks and resources, respectively. Each instance of a task is assigned to exactly one resource. This is specified as *machine eligibility*. Since each instance of a task has to execute on each instance of a resource once, tasks are defined as aperiodic. There exist dependency relations among tasks to ensure the order of executions on each instance of a resource. According to the requirements:

- The tasks are not *preemptable*.
- The resources are active and have single-unit capacities.
- The speed of resources is assumed to be constant.
- The scheduling process is defined as *offline*.
- The scheduling policy is determined as *FIFO*.
- The priority assignment is defined as *dynamic*.
- The overall objective is chosen as *minimizing* the *makespan*.

This requirement can be represented as follows:

$$F4|prec, prmu, M_j|C_{max}. \tag{5.7}$$

Implementation of the Initial Requirement

Example 6.1.1 in the book of Pinedo [109] is adopted in the implementation of our FJ[5]. There are five instances of tasks and four instances of resources. Since each job is supposed to execute on each resource, the taskset consists of 20 instances (5 *jobs* x 4 *resources*). The

[5]The implementation using our design environment and details can be found in our repository `https://github.com/gorhan/LFOS/blob/master/Tests/FSP.py`

	j_1	j_2	j_3	j_4	j_5
$\tau_{(1,j)}$	5	5	3	6	3
$\tau_{(2,j)}$	4	4	2	4	4
$\tau_{(3,j)}$	4	4	3	4	1
$\tau_{(4,j)}$	3	6	3	2	5

Table 5.4: Execution times for the tasks.

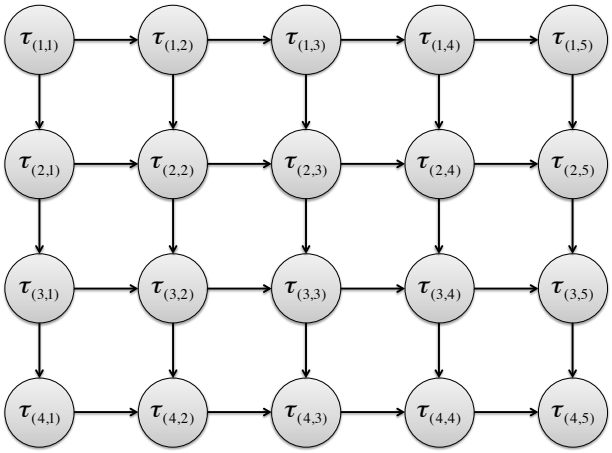


Figure 5.14: Dependency graph of the flow-shop scheduling example.

execution times of the tasks are shown in Table 5.4. The columns correspond to the jobs. The row $\tau_{(i,j)}$ corresponds to the execution time of the job j on the resource i.

The dependency relations of the tasks are shown in Figure 5.14. Here, the tasks (nodes) with two incoming edges have to wait until both dependency constraints are satisfied. We call this constraint as *AND-dependency*.

The example is implemented and executed. The obtained schedule is given in Figure 5.15. Since the objective is minimizing the makespan, it is not required to schedule the tasks as soon as possible. For this reason, some instances of tasks starts later than its earliest start time., such as $\tau_{(2,3)}$ and $\tau_{(3,3)}$. As it can be seen in the figure, the makespan of the schedule is 34.

Evolution of the requirement: relaxation of dependencies

Assume that dependencies among the tasks have to be relaxed at run time by removing the dependencies shown as vertical edges in Figure 5.14. Since there is no change in general scheduling attributes, the problem definition given in Equation 5.7 is still valid.

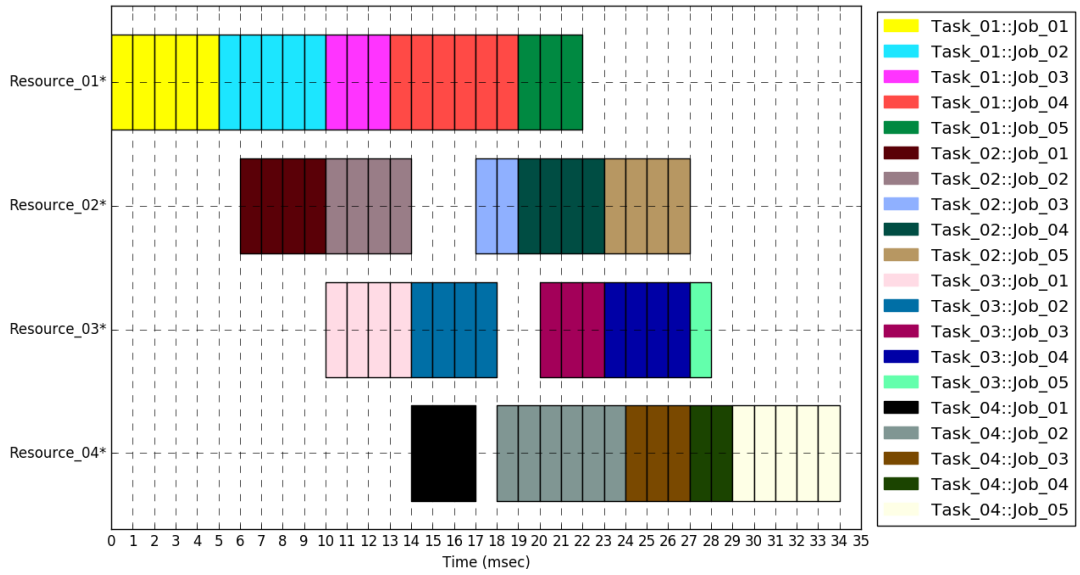


Figure 5.15: A graphical representation of the schedule.

Implementation of the new requirement

There are two ways to implement the required evolution: Data dependency relations of each task are abandoned and the new ones are defined, or the undesired dependencies (shown as vertical edges) for each task is removed.

This evolution requirement is instantiated at run time and executed. The resulting output is shown in Figure 5.16.

5.4.5 Open-shop Scheduling (OS)

In open-shop scheduling, like the previous shop examples, a job is assigned to one machine. On the other hand, the dependencies among jobs are relaxed. A job can be freely allocated the corresponding machine when it is available. However, any two activities of a job cannot execute in parallel and therefore they cannot be active at the same time; an activity should finish its execution before another activity of the same job starts to execute. Like all shop problems, the jobs are not preemptable.

Initial Requirements

Assume that there are five jobs and three active resources. Each job is supposed to execute on each resource. Therefore, there are 15 instances (5 *jobs* x 3 *resources*) according to our model. The details of the example are adopted from Example 8.4.1 of [109][6] and modified

[6]The implementation using our design environment and details can be found in our repository of `https://github.com/gorhan/LFOS/blob/master/Tests/OSP.py`

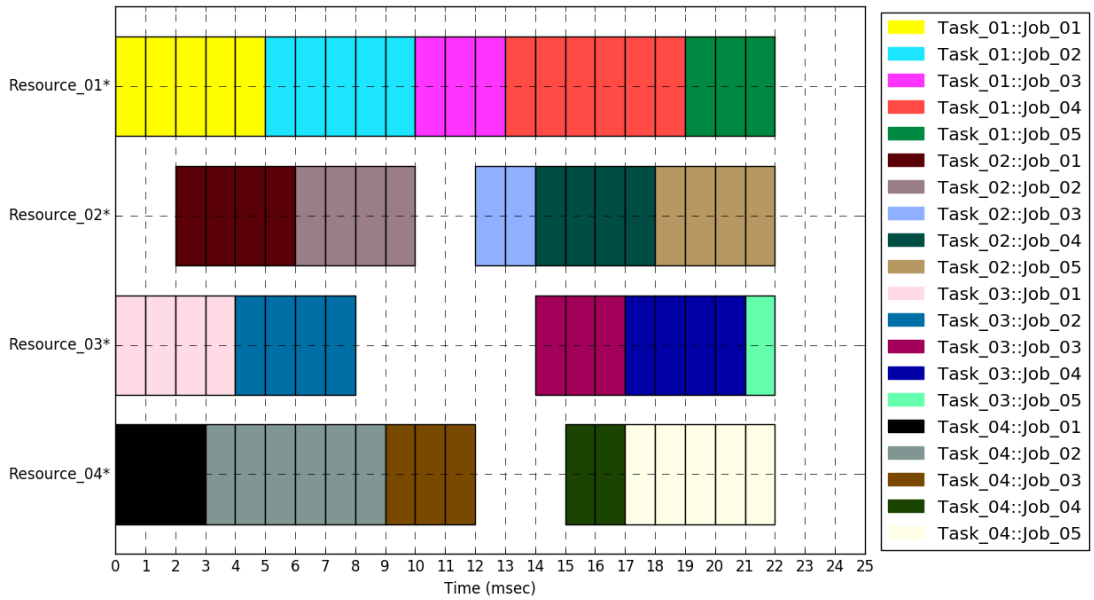


Figure 5.16: A graphical representation of the output of the evolved FS example.

according to our model.

The adapted problem definition of this example is as follows:

$$O3|r_j, d_j, M_j|L_{max}. \tag{5.8}$$

Implementation of the Initial Requirement

Similar to other shop examples, jobs and machines are defined as tasks and resources, respectively. Each instance of a task is assigned to exactly one resource using the *machine eligibility* specification. Instances of tasks in the same subset as *mutually-exclusive*.

The execution times of the tasks on each resource, their release times and deadlines are shown in Table 5.5.

This example was instantiated and executed. The resulting schedule is displayed in Figure 5.17. There exists only one solution to this scheduling problem. The duration between release times and deadlines of the job task j_3 and j_4 is equal to their execution times. Therefore, there is no any other scheduling possibility. Since the execution time of task $\tau_{(1,5)}$ is 3 and its deadline is 11, it is supposed to be scheduled immediately after task $\tau_{(1,4)}$. Due to the non-preemptable tasks, although task $\tau_{(1,1)}$ has the highest priority among the tasks, task $\tau_{(1,2)}$ has to be scheduled at the time when its release time starts.

Evolution of the requirement: preemptable tasks

As an evolution step, now assume that the tasks are defined to be preemptable. This new requirement can be expressed as:

	j_1	j_2	j_3	j_4	j_5
$\tau_{(1,j)}$	1	2	2	2	3
$\tau_{(2,j)}$	3	1	2	2	1
$\tau_{(3,j)}$	2	1	1	2	1
r_j	1	1	3	3	3
d_j	11	9	8	9	11

Table 5.5: The execution times, release times and deadlines for the taskset.

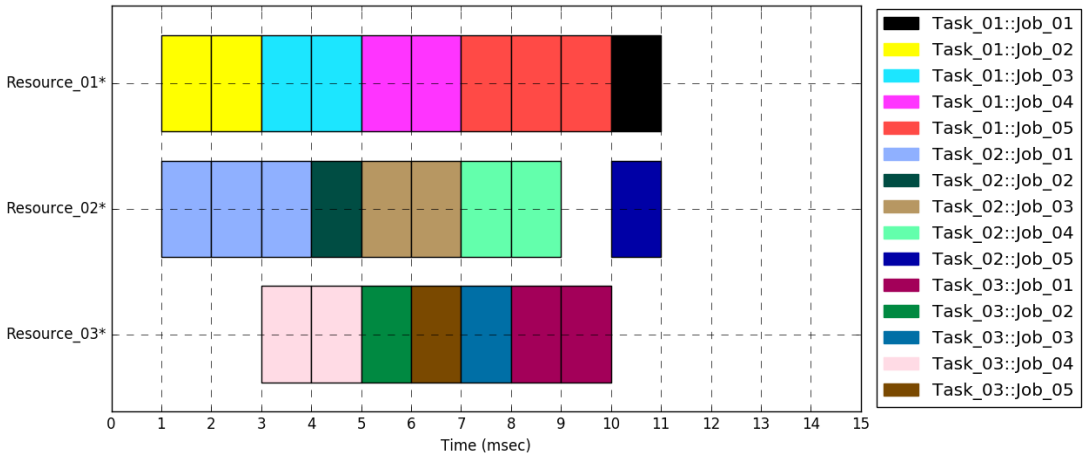


Figure 5.17: A graphical representation of a schedule of the OS example.

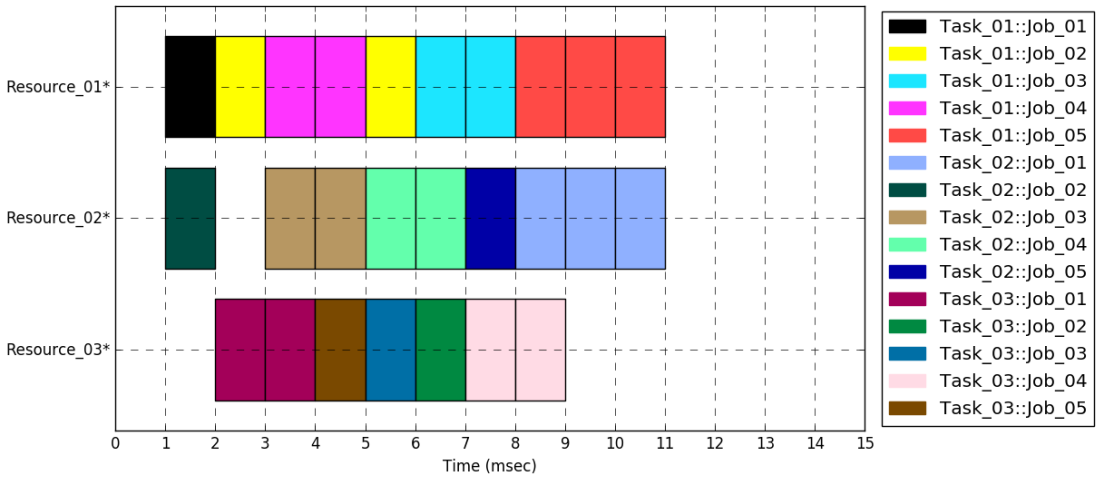


Figure 5.18: A graphical representation of the output of the evolved example.

$$O3|r_j, d_j, M_j, pmtn|L_{max}. \tag{5.9}$$

Implementation of the new requirement

To implement this evolution, the taskset must be traversed and the tasks must be re-specified as preemptable. The previously defined schedule is modified at run time accordingly and executed. The output schedule is shown in Figure 5.18. Here, the task $\tau_{(1,4)}$ is able to start immediately after its release time as the task $\tau_{(1,2)}$ is preempted by the task $\tau_{(1,4)}$.

5.5 Evaluation and Conclusions

In this section, the framework is evaluated against the objectives described in Section 5.1.

5.5.1 Assessment Method

Our framework is evaluated against the two required quality attributes *reusability* and *run time evolvability*.

From the perspective of *reusability*, it is stated that to create a particular scheduling system, the code written from scratch must be shorter than the code of the library that is reused. This definition refers to the Lines-of-Code (LoC) metric [52]. There is much debate on the preciseness of the metric, because it may not accurately express the effort spent. The metric may be influenced from many factors, such as the characteristics of the adopted programming language, the formatting styles used in coding etc. Therefore, the validity of the LoC metric in a particular measurement context must be considered carefully. In addition, the definition of *reusability* within the context of application frameworks implies that the

framework must be expressive enough to instantiate a large category of implementations in the domain of the framework.

Run time evolvability is defined as ease of modification of an existing scheduling software with respect to a new *meaningful* set of user requirements during the operational phase of the software. This implies that all relevant parameters of a system must be set by invoking operations on the corresponding objects. To validate this quality attribute, one can define evolution scenarios for each possible parameter change. One disadvantage of this evaluation is that there may be too many possible evolution scenarios. Nevertheless, within the domain of scheduling, the number of relevant attributes is limited. For example, in Section 5.1, run time evolution support is required for five cases.

		α_1				α_2	
		Q	F	O	J	1	M
β	r_j	MRS		OS		RMS	MRS, OS
	$prec$	MRS	FS		JS		MRS, FS, JS
	$pmtn$	MRS		OS		RMS	MRS, OS
	$d_j = d$	MRS		OS			MRS, OS
	M_j	MRS		OS	JS		MRS, JS, OS
	s_{jk}	MRS					MRS
	$batch$	MRS					MRS
	$prmu$		FS				FS

Table 5.6: Domain coverage of examples used. Abbreviations represent the scheduling examples referred to in this chapter.

Reusability of the Framework

To evaluate the expressivity, in Section 5.4, five canonical examples from the scheduling domain are presented.

It is argued that the scheduling domain can be represented using Table 5.6[7]. The cells refer to the abbreviations of the example schedulers. The parameters in the rows of the table corresponds to the *scheduling characteristics* and are denoted by β; whereas the columns of the table are grouped into two categories, namely the *machine identifier* and the number of machines which are denoted by α_1 and α_2, respectively. Within these categories, the parameters *Q, F, O, J, 1, M* are explained in detail in Section 2.1.1. As it can be seen, in each column and row from top to bottom and left to right, respectively, at least one example resides. This illustrates that the examples cover at least one case of the parameters in the scheduling domain.

As previously shown in Figure 5.1, the framework can be divided into three subsystems: *Resources, Tasks* and *Scheduler*. As for implementation languages, Python and C/C++ are used. The third-party software that is adopted in the architecture are Numberjack [65],

[7] http://www2.informatik.uni-osnabrueck.de/knust/class

SCIP [57], MiniSat [125], MipWrapper [65], Mistral [44], Mistral2 [65], SatWrapper [65], Toulbar2 [36] and Walksat [118]. To implement the supporting functions, we have integrated the following third-party tools: Clafer [9] and MatplotLib [72].

The framework library including third-party software contains 18071 and 927904 LoC written in Python and C/C++ programming languages, respectively. The LOC of the supporting software is not included in this count.

	RMS	MRS	FS	JS	OS
Additional LoC (Python)	67	94	58	67	57

Table 5.7: LoC for each example in Section 5.4.

In Table 5.7, the columns refer to the examples presented in this chapter. The row indicates the LoC of the examples.

The LOC metric is not very precise and not all the code of the library is used in each example. Nevertheless, the amount of reuse of the library code is so much higher than the metrics shown in the row of Table 5.7 that the impreciseness in this context is considered negligible. Therefore, it is assumed the framework satisfies the *reusability* requirement.

Run Time Evolvability of the Framework

Scenario	Example
Adding new resource	RMS
Changing the objective	MRS
Adding different release times to the tasks	JS
Removing some dependency relations	FS
Setting the tasks preemptable	OS

Table 5.8: Scenarios to evaluate *run time evolvability* of the framework.

	Parameter	Symbol	RMS	MRS	FS	JS	OS
BrE	**# Constraints**	n_c^i	4067	738	13980	2818	545
	Time for constraint definition (sec)	t_d^i	1.3961	0.46911	11.36172	5.1782	0.50092
	Time for solver execution (sec)	t_s^i	268.93	107.41	22342.83	274.71	7.71
	Overhead	$t_o^i = t_d^i/(t_d^i + t_s^i)$	0.0052	0.00435	0.00051	0.0185	0.06101
ArE	**# Constraints**	n_c^e	4629	738	12322	2417	754
	Time for constraint definition (sec)	t_d^e	1.91886	0.5057	12.22426	4.92931	0.5021
	Time for solver execution (sec)	t_s^e	251.65	107.18	14955.39	167.76	11.76
	Overhead	$t_o^e = t_d^e/(t_d^e + t_s^e)$	0.00757	0.0047	0.00082	0.02854	0.04095

Table 5.9: FSF time performance overhead with respect to a *bare solver* alternative.

In Section 5.1, five cases for *run time evolvability* are given. Here, the capital letters in the first column correspond to the cases. The second column *Scenario* describes briefly the evolution scenarios presented for each example in Section 5.4. The last column lists the abbreviations of each corresponding example. It is clear from the table that these evolution

scenarios can be realized at run time. Therefore, it is assumed that the framework satisfies the *run time evolvability* requirement.

Evolution of the Time-Performance Overhead

As demonstrated in the previous two subsections, **FSF** provides a high-degree of *reusability* and *run time evolvability*. A legitimate question one may ask is *what is the cost of enhancing these quality attributes in terms of time performance?* To answer this question, the time performance of **FSF** is compared with *bare* solver-based solutions. Consider Table 5.9. Here, two tables are integrated into one. The upper and lower tables, which are named as **BrE** and **ArE** refer to the examples before and after evolution scenarios, respectively. The columns **Parameter** and **Symbol** refer to the relevant parameters for our evaluation. The columns **RMS, MRS, FS, JS**, and **OS** represent the measured parameters of the examples presented in Section 5.4. The row # **Constraints** indicates the number of constraints generated; these are to be considered by the solver. Obviously, the number of constraints gives an indication about the complexity of the problem and the required time-delay caused by the solver. The actual time performance of the solver also depends on the nature of the constraints and how they are related to each other. The parameter **Time for constraint definition** refers to the time spent for the generation and transformation of constraints realized by **FSF**. The parameter **Time for solver execution** refers to the time required by the *bare solver*. In this case, it is assumed that no time is spent in the preparation of the constraints since they are readily expressed in the specification language of the solver. A ratio of these two parameters is defined as **Overhead**.

As it can be seen from the table, **Overhead** varies between 0.00051 and 0.06101. It is clear that the execution time caused by **FSF** is comparably much lower than the execution time of the solver adopted.

5.5.2 Conclusions

In this chapter, *reusability* and *run time evolvability* are defined as the two key requirements of an object-oriented framework aimed at creating scheduling software. To this aim, a framework called **FSF** has been implemented. With the help of canonical examples, it is shown that **FSF** satisfies the *reusability* requirement. The *run time evolvability* of the framework is demonstrated with a set of evolution scenarios. To the best of our knowledge, **FSF** is the first framework that provides a scheduling design framework with these quality attributes.

CHAPTER 6

OptML Framework and its Application to Model Optimization

During the last decade, there has been an increasing emphasis on model-driven engineering (MDE) [27]. There has been a considerable effort in definition and implementation of models in a large category of application domains and as such many useful models are readily available for use.

Availability of models in the domains of interest, however, creates its own problems to deal with:

First, due to complexity of the domain of interest, complexity and size of models can be very large [19]. Although there have been some approaches, such as model splitting/merging/transforming [24], which can be used to deal with model complexity, generally they must be "hand-tailored" and their effects in reducing complexity can be rather limited. A number of model complexity reduction approaches has been proposed. However, as it is stated in Babur's article on models [14], this is an active research area and the problem of model complexity has not been solved yet satisfactorily.

Second, due to built-in variation mechanisms, models may be configured in many different ways. In addition to functional requirements, selection and configuration of models may largely depend on certain *quality attributes* and *contextual parameters*, which may not be explicitly specified as parts of models. Examples of *quality attributes* are for example, time-performance, energy reduction, precision in computations, etc. Examples of contextual parameters are software and hardware architectural styles of adopted platforms, their

characteristics, etc.

Last but not least, since the number of model configurations can be very large, given a set of requirements, it may be very hard for software engineers to derive the most suitable configurations in a convenient manner. Optimizing a model-configuration for a single purpose is generally not satisfactory. Software engineers generally have to trade-off different objectives to configure the most suitable model for a given application setting.

This chapter proposes a novel tool workbench called Optimal Modeling Language (OptML)[1] framework to represent certain *quality attributes* and *contextual parameters*, explicitly. This approach is supported by Optimal Modeling Process (OptMP) to guide the software engineer in selecting and configuring models, according to the desired optimization criteria. The framework incorporates a dedicated set of tools to compute the desired optimal model configurations. Examples of currently supported *quality attributes* are time performance, energy reduction and precision. Furthermore, as contextual parameters, single- and multi-core platforms and various distributed and/or parallel system architectures are supported. The software engineer can define new quality attributes by using the *value* meta model of the framework. The utility of the model, the associated process and tools are demonstrated by a set of examples. A prototype implementation is realized using the Eclipse framework [129] and FSF application framework. FSF is a dedicated software library to implement a large category of scheduling systems [105].

This chapter is organized as follows. The following section introduces an illustrative example and explains the addressed problems. Section 6.2 presents the architecture of the framework. Section 6.3 gives a set of examples models based on various architectural views. *Model Processing Subsystem* and *Model Optimization Subsystem* of the framework are described in Sections 6.4 and 6.5 respectively. Section 6.6 briefly summarizes the related work. Section 6.7 evaluates the approach. Finally, Section 6.8 concludes the chapter.

[1]The term Optimal Modeling Language is selected for the following reason. The purpose of this framework is to compute the optimal configuration from a set of models utilized by the software engineer. We assume that each model is based on a dedicated modeling language (meta model) in Ecore Modeling Language tradition. In addition, to compute the optimal model, appropriate models must be introduced to specify the optimization constraints.

6.1 Illustrative Example, Problem Statement and Requirements

In this section, an illustrative example from the image processing domain is given which will be used throughout the chapter to demonstrate the problems and the proposed solutions. This example is considered illustrative for the following reasons. First, this chapter addresses the concerns where model size and complexity are considerably high. There is a comprehensive and fully-implemented software library of the example, which is considered representative for the purpose of the chapter. Second, this software library can be configured in many ways. This chapter aims at dealing with models with a large number of configurations. Third, while reusing this software library for a particular application, the software engineers are typically concerned with various quality attributes, such as timeliness, energy consumption, and precision. This chapter aims to select the optimal model configuration that satisfy multiple quality constraints.

Registration is a problem of reconstructing an image output by matching two or more related images captured in different environmental conditions [22], so that the obtained image is more expressive for a particular purpose than the individual input images. This may be needed in systems where multiple sensors are used with different resolutions, positions and imaging characteristics.

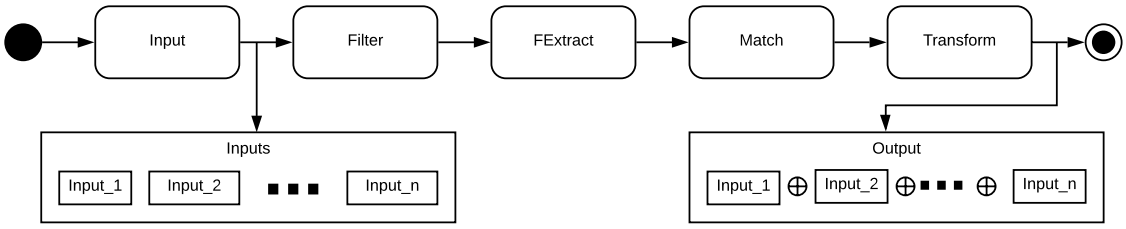


Figure 6.1: The process of a Registration system to reconstruct an image from multiple inputs.

Consider, for example, the following pipeline architecture for a Registration system, which is represented in five consecutive states, depicted in Figure 6.1. This architecture is inspired from the Point Cloud Library (PCL) [114]. From the left, the state *Input* represents the data acquisition loop which gathers image data from one or more sensors. The second state *Filter* aims to reduce the data size if necessary so that only the relevant information is used for further processing. In addition, the original images are preserved. The third state *FExtract* is responsible to compute the predefined key features from the data to reason about the geometric characteristics of the images. The fourth state *Match* is used to correlate the extracted features with each other. The state *Transform* is used to transform the original images into a common image based on the matching process. Here, the sign $\oplus$ represents the transformation operator.

Assume that a version of the PCL library is instantiated in an Ecore MDE environment with the following models, which represent the system from different architectural views [34]. More detailed information about these models can be found in Section 6.3:

- A class model, which describes the logical structure of the system.
- A feature model, which defines the variations to configure different versions of registration systems.
- A platform model, which describes the underlying computational resources of the registration system.
- A process model, which illustrates the execution flow of the processes and the necessary synchronization points among them.

From the perspective of this chapter, the following potential problems can be observed:

1. *Large configuration spaces of models*: It is a common practice that multiple related models are used in MDE environments for a given system. Each of these models may define different kinds of variations. The possible combination of all variations may potentially enable many possible instantiations of models, which can be difficult for the MDE expert to comprehend. Consider, for example, the registration system which is described by four different kinds of models. Due to the variations of each model, the design space of the registration system can be very large. In our example case, for instance, the number of variations of the defined feature model is computed as 6144 (see Section 6.4). There have been a number of proposals, such as model splitting, merging or transforming [24, 14], which can be used to deal with model complexity. However, many of these proposals provide dedicated solutions, for example, through the application of predefined rules.
2. *Lack of quality concerns in model configurations*: This problem is a natural consequence of the previous problem. An important set of criteria for creating a particular configuration from a model space is to select the configuration that fulfills the desired quality attributes.
   For example, while configuring a particular Registration system, it may be necessary to check whether the tasks in the process model can be completed on a given platform configuration within a given time. To this aim, it must be possible i) to decorate the process model with the desired attributes, such as the execution times of processes, and ii) to check if the process can be completed on time (schedulability analysis [49]). Obviously, new models can be introduced aiming at different architectural views if necessary. Assume that we would like to extend the set of models in the MDE environment with two additional models:
   Energy model: it is a model to define the energy demanded by processes (also called operations) to complete them on the configuration of the underlying platform in certain time. The factors that affect the completion time of processes depend on the demanded energy by them and the offered energy by the platform configuration.
   Precision model: It is a model to define the quality of the resulting image accuracy in registration process. In the implementation, alternative algorithmic solutions are defined with different precision. Depending on the requirements, a low-precision algorithm may be preferred to a high-precision one for the sake of time performance.
3. Optimization of configurations: Software engineers generally have to trade-off different quality attributes to configure the most suitable model for a given application

setting. For example, a particular model configuration may improve the quality attribute "reducing energy consumption" while decreasing the quality attribute "time performance". The MDE environment must provide means to optimize model configurations by considering multiple quality attributes.

Based on these observations, OptML Framework should support at least the following requirements.

It must be possible to evaluate:

1. configurations of models whether at least one configuration exists that can be mapped on a specified platform architecture while satisfying the timing and resource constraints; and/or
2. find out the optimal model among configurations based on certain optimization criteria and objectives. Along this line, for example, it must be possible to
   (a) introduce a model for each quality attribute;
   (b) normalize the quality attributes;
   (c) relatively prioritize the quality attributes with respect to each other;
   (d) apply the comparison operators on the values of attributes, such as "$<, >, <>, =$";
   (e) select the models with respect to minimization or maximization of the quality attributes;

   and/or
3. find out the optimal model among configurations based on certain optimization criteria and objectives, and/or
4. whether the introduced models are consistent with each other with respect to the predefined consistency rules.

The software engineers may demand many different kinds of models when developing their applications. The facilities provided by OptML Framework may need to be extended accordingly. The effort that is spent in realizing OptML Framework can only be justified if the facilities of OptML Framework are demanded by multiple software engineers. OptML Framework, therefore, must offer solutions to the recurring problems of software engineers. If, however, OptML Framework is required to be extended to satisfy the emerging needs of software engineers, it must be extended correctly. I addition, enhancements to the implementation of OptML may be necessary from time to time, for instance, to improve performance. Based on these assumptions, the following extensions are considered foreseeable:

5. supporting new models defined in the ECORE environment;
6. introducing new pruning mechanisms while extracting models from the model-base;
7. introducing new value-based quality attributes;
8. introducing new value optimization algorithms where necessary;
9. adopting new search strategies for the schedulability analysis and optimization techniques;

6.2 Framework Architecture

As shown in Figure 6.2, the architecture of OptML Framework consists of three sub-systems. The *Model Editing Subsystem*, which is symbolically shown on the left side of the figure, can be used to define various models representing different architectural views based on the corresponding meta models. If necessary, new meta models can be introduced to the system using MDE facilities. We assume that this subsystem corresponds to a standard MDE editing facilities such as the Eclipse Modeling Framework. The second process in the figure, *Model Processing Subsystem* is used to transform the introduced models into a representation that can be processed by *Model Optimization Subsystem*. The *Model Optimization Subsystem* part of the framework, which is shown symbolically at the right-hand side of the figure, automatically process the transformed models and computes the optimal model based on the criteria provided by the model driven (MD) engineer. The last two processing subsystems form the essential components of OptML Framework. In our approach, we adopt Ecore Modeling language and the Eclipse platform for *Model Editing Subsystem*. Since this framework is well-known, we do not explain it further in this chapter. Nevertheless, in sections 6.4 and 6.5, we describe *Model Processing Subsystem* and *Model Optimization Subsystem* in detail, respectively.

6.3 Examples of Models for Registration Systems based on Various Architectural Views

We will now introduce six models subsequently as running examples to show how the framework works and to deal with the complexity of the example design problem.

The modeling paradigm adopted in this chapter follows the MDE ECore tradition, which means that first a meta model is to be defined that conforms the Ecore meta meta model (called ECore EMF format) [129]. A model is an instantiation of its meta model. In the following subsections, the described meta models are:

- UML Class diagram meta model: To depict the *logical view* [85] of the example,
- Feature meta model: To specify the possible configurations of the example [80].
- Platform meta model: To specify the *physical view* and the *deployment view* [85] of the example.
- Process meta model: To specify the *process view* [85] of the example.

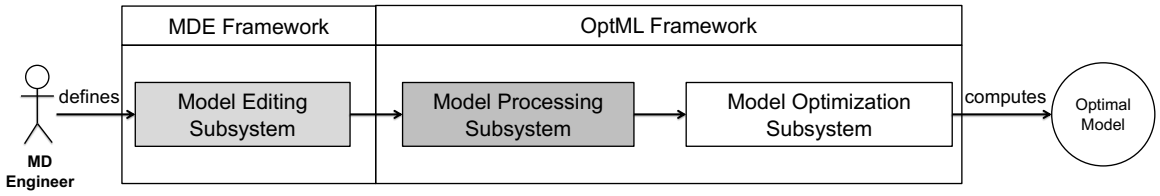


Figure 6.2: A Software architecture of OptML Framework.

- Value meta model: To specify the quality concerns of the example.

These models are selected because they are considered as fundamental models required by many applications as published by Kruchten [85]. In addition, to address the requirements of this chapter, the value meta model is introduced so that the optimal model can be computed accordingly.

6.3.1 UML Class Model

Class diagram is used for specifying the logical building blocks of a software system. We do not define a new meta model for classes, but rather we adopt the standard UML Class meta model, which is one of the registered packages[2] of Epsilon Modeling Framework in Eclipse IDE.

Figure 6.3 shows a class model for the registration system, which is introduced in Section 6.1. Here, for brevity, the attributes and operations are not shown in the figure.

The names of classes Input, Filter, FExtract, Match and Transform are indicated in bold in the figure, and they correspond to the subsystems of a registration system presented our example in Figure 6.1 of Section 6.1.

6.3.2 Feature Meta Model

The meta model representing feature models is shown in Figure 6.4. The aim with the feature model is to express *commonalities* and *variabilities* in a family of software systems. A feature model enables the model-driven (MD) engineer to express various configurations of the system. Due to various options, configuring a feature model may refer to more than one software system. In a traditional model-driven engineering approach, the MD engineer is supposed to evaluate each configuration and choose the most suitable one based on some criteria.

Fundamentally, any feature model has exactly one root from which sub-feature models are originated. Each feature may have zero or more child features and attributes. Each attribute has a type and defaultValue, which may belong to the primitive types Boolean, String, Integer or Object. In addition, each feature has a type as optional, alternative or or if it is a variability; or mandatory if it is a *common* asset for the product family. All the features are placed into a Group with upper and lower. The number of bound features inside the same group has to be between these values in any configuration instantiated from one feature model. Finally, a feature model may have cross-tree constraints that are defined as rules in any constraint-based language such as OCL.

A feature model must be consistent with the process model and the class diagram. Therefore, we assume that each feature defined in a feature model must correspond to a class in the class diagram.

A feature model of the registration system, which is instantiated from the Feature meta model, is given in Appendix A.1.

[2]EMF-based implementation of the Unified Modeling Language (UML)

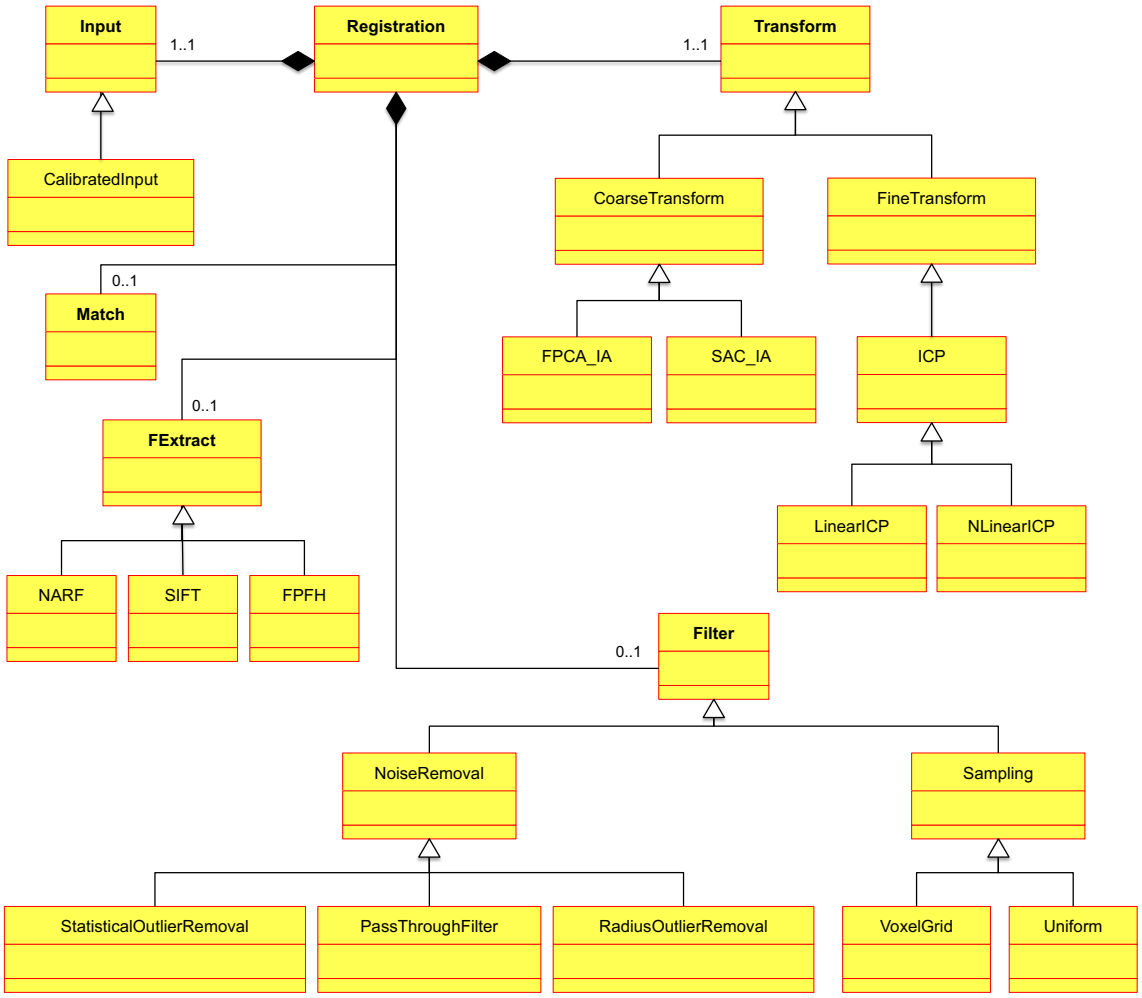


Figure 6.3: Class diagram for the registration system.

6.3.3 Platform Meta Model

We will present the Platform meta model, as adopted in OptML Framework. A platform model, which is also termed as deployment model [104], is an instance of this meta model, and enables the designer to express the components of the computational system. If a platform model is not specified, it is assumed that the underlying computational system is transparent and as such mapping of software modules to computer architecture is to be handled by the operating system in some way. In many system design problems, however, it may be necessary to consider the platform into account, for example, when designing software architecture over distributed and/or multi-core systems, in Internet of Things (IoT) applications to map software modules to the underlying architecture, etc.

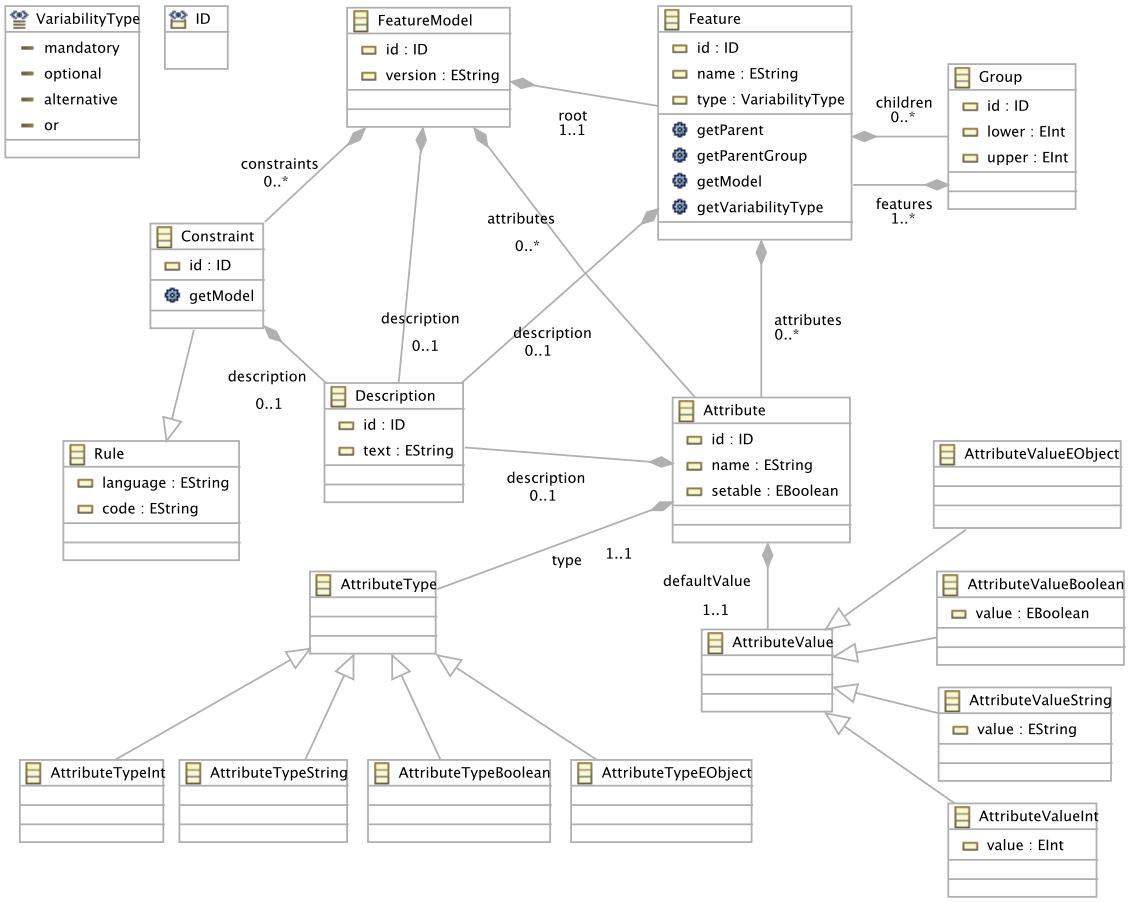


Figure 6.4: Feature meta model.

The *Platform meta model* is shown in Figure 6.5. It expresses hierarchically nested software/hardware architectures, which can be composed of various types of architectural components. The Platform meta model is represented as class PlatformDiagram, which aggregates classes ResourceType and CompositeResource. Class ResourceType specifies the characteristics of the corresponding resources with a unique string-type identifier and an enumeration of the literals ACTIVE, PASSIVE and COMPOSITE. We aim to create a uniform model by considering all possible architectures as a special configuration of a composite object. To this aim class PlatformDiagram aggregates class CompositeResource. To create a hierarchical platform organization, class CompositeResource uses the composite pattern format [56]. Class Resource here corresponds to an abstract representation of an architectural component, since every resource inherits its properties. Class CompositeResource may encompass zero or more terminal and/or composite resources, where composite resources may further aggregate resources and so on. The aggregation relation from class CompositeResource to class Resource enables to create nested instances of classes Com-

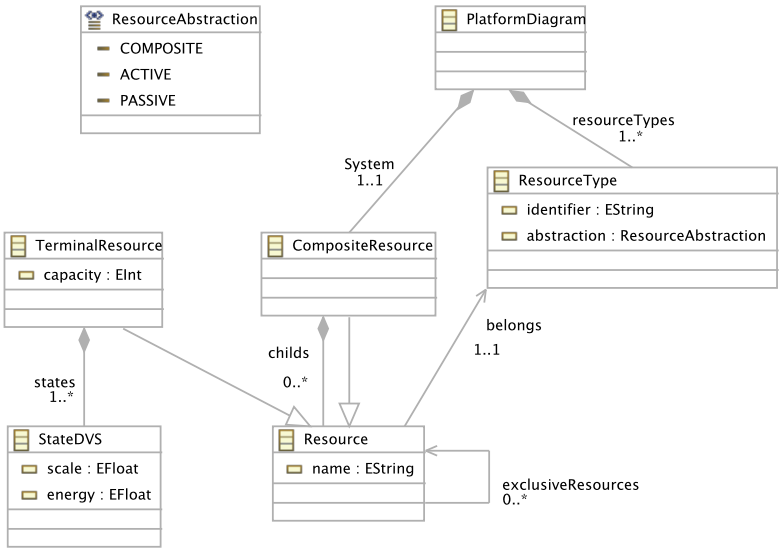


Figure 6.5: Platform meta model.

positeResource and/or TerminalResource. Class TerminalResource, as the name implies, is the representation of the resources that cannot be decomposed any further. The attribute capacity of class TerminalResource defines the maximum utilization unit that a resource can provide [69].

In recent years, mobile devices are increasingly used as a computing platform. Due to limited operational time of batteries, reducing power consumption of mobile device has become important. To this aim, for example, the *Dynamic Voltage and Frequency Scaling* (DVFS) technique is introduced [30, 92]. In Lin's article [92], the concept of operating frequency levels has been defined. The levels correspond to the frequency-scaling factors varying between 0 and 1. A higher value means higher energy consumption. Due to the popularity of this approach in practice, we adopt this technique in our platform model as well; Class StateDVS is introduced for this purpose. Each level has its scaling factor and corresponding power consumption value. Each level has its scaling factor and corresponding power consumption value, which are represented by the attributes scale and energy of class StateDVS, respectively.

To avoid race conditions and simultaneous access to shared resources, the self-reference relation over abstract class Resource is defined. It avoids that multiple resources run at the same time.

The reference relation from Resource to ResourceType is used to denote the type of the corresponding resource.

The platform model as an instantiation of this meta model for the registration system is given in Appendix A.2.

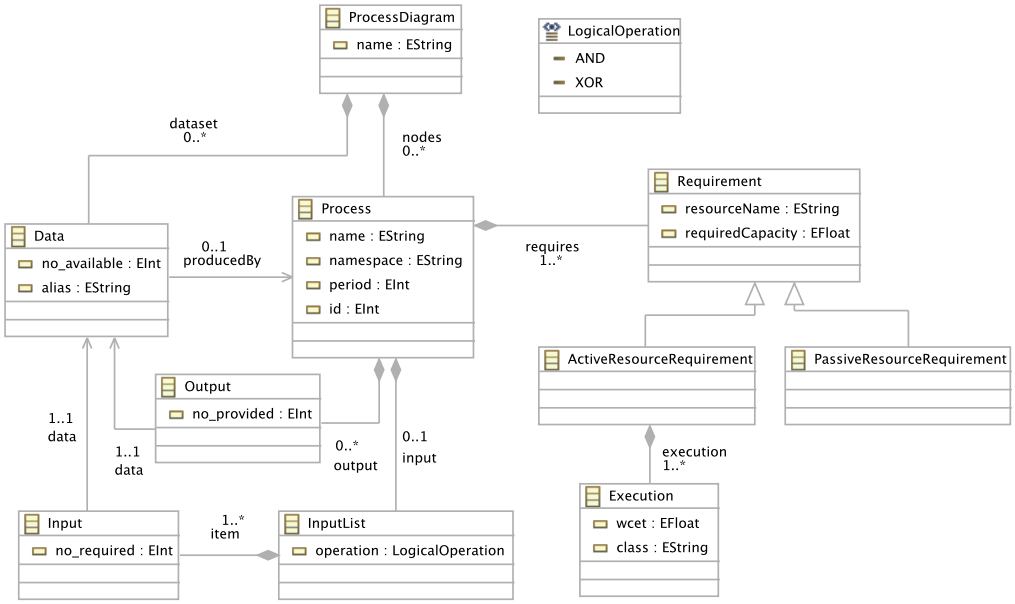


Figure 6.6: Process meta model.

6.3.4 Process Meta model

Our framework allows the MD engineer to understand the dynamic behavior of systems, for example, when allocating software to the underlying architecture, verifying the operational semantics of software, determining the time performance, etc. In the literature, various kinds of models have been presented such as state diagrams, process diagrams, collaboration diagrams and activity diagrams. If the timing-constraints are to be considered when mapping the functionality to a particular platform, we assume that a process model is defined which represents the processes and their execution flow, input-output data dependencies and resource requirements. Consider, for example, the following Process meta model which is shown in Figure 6.6.

A common practice to represent a process model is to consider it as a graph where each process is a node and the dependencies among processes are the edges of the graph [13]. In our approach, in Figure 6.6, class ProcessDiagram represents the root of the graph. Since there may be multiple independent processes in the system, the attribute *name* of this class is used to denote a particular process diagram.

Class ProcessDiagram aggregates zero or more nodes, where each node is represented by class Process. The attribute *name* of this class is used to identify a particular process in a process diagram. In pure object-oriented programs, a process is associated with an object of a class. To represent this property, the attribute *namespace* is used to denote the corresponding class. In the literature, *periodic* processes are executed repeatedly at each time interval [28]. To represent this characteristic of a process, the attribute period is

defined. In case of multiple instances of a process, the attribute *id* can be used to distinguish the instances from each other.

The application semantics why requires processes to execute in a certain order [28, 109]. To specify such conditions, we adopt the concept of *data dependency* constraint explained in [88, 3]. A data dependency constraint specifies the data that are required and/or provided by a process. If, say the process P_1 provides the data d_1 which is demanded by the process P_2, then P_2 is eligible to execute only after the completion of P_1. In Figure 6.6, the required and provided data for a process are represented by classes Input and Output, respectively. Both classes refer to only one instance of class Data. The availability of a particular data item is indicated by the attribute no-available of class Data. The attributes no-required and no-provided of classes Input and Output refer to the *required* and *provided* number of available data items, respectively. Class InputList specifies the input dependency constraints of a process. If a process requires more than one data item, and it is eligible to start when only one of the data items is available, the attribute operation of class InputList should be defined as XOR. However, if the process requires the availability of all data items, then the attribute should be defined as AND, instead.

Resource requirements of a process are expressed by class *Requirement*. The attributes resourceName and requiredCapacity refer to the necessary resource type and its capacity, respectively. To increase the utilization factor of resources, it may be profitable to divide resource types in active and passive resources [105, 139]. To this aim, classes ActiveResourceRequirement and PassiveResourceRequirement are defined which inherit from class Requirement. For allocating active resources to processes that complete in a timely manner, worst-case execution time of a process is an important factor to consider. The attribute WCET of class ActiveResourceRequirement is used for this purpose.

The process model of our example case defined by instantiating the meta model shown in Figure 6.6 is presented in Appendix A.3.

6.3.5 Value Meta Model

OptML Framework aims to optimize software models based on various quality attributes. Value meta model assumes that these attributes are expressed in *numeric values* associated with the models[3]. If there are multiple attributes, the associated values must be differentiated by the types of the qualities used. We define nevertheless a common Value meta model, which can be instantiated for different quality attributes if needed. The *Value meta model* is shown in Figure 6.7. Here, class ValueModel has an attribute name that is used to specify the type of a quality attribute. The value of a *quality attribute* is assigned to each operation within a class. As short-hand notation, it is also possible to assign a value to a class. This means however that the values of the operations defined in that class are equal.

[3] According to the principals of measurement theory, each quality attribute must be expressed in an appropriate value system [48]. However, the computation of values of quality attributes and their association with modeling elements can be defined in various ways. If different value model is required than the presented one in this chapter, the tool designer must extend OptML Framework accordingly. This is briefly discussed in Section 4.6.

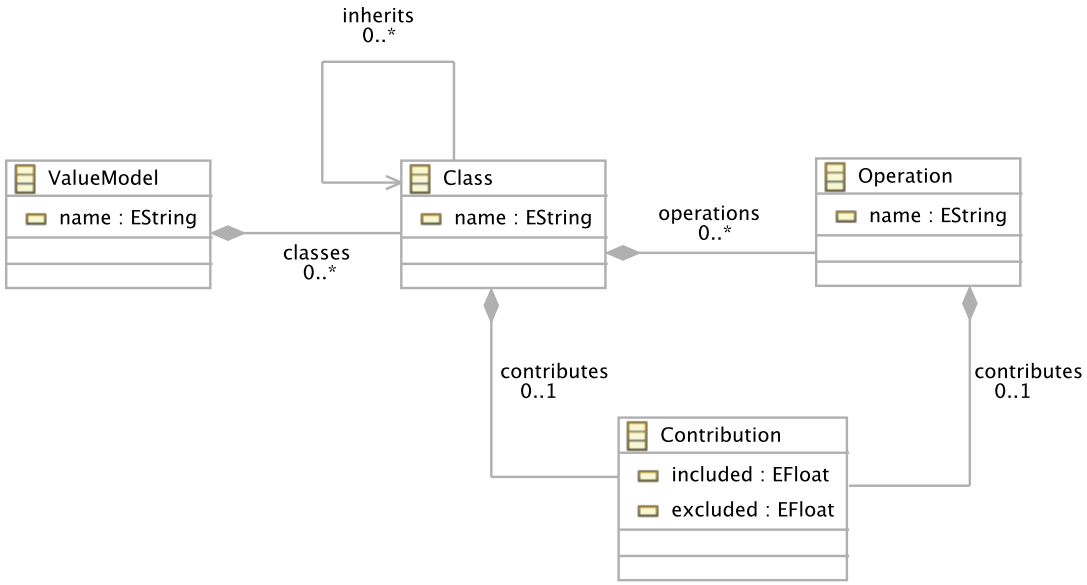


Figure 6.7: Value meta model.

If two different values are assigned to a class and an operation of that class, the value of the operation overrides the value of the class. It is also possible to override values through the inheritance relation. To specify these values, class *Contribution* is defined which is associated with Class and Operation. Here, the attributes included and excluded indicate the positive and the negative contribution values depending on the inclusion or exclusion of the corresponding element in the configuration, respectively.

The instantiations of the Value meta model for energy consumption and computation accuracy are given in Appendix A.4.

6.4 Model Processing Subsystem

Model Processing Subsystem is used to combine the models with each other. The output of this subsystem is expressed as a dynamic data structure called *pipeline data* which will be explained in the following subsections. Figure 6.8 symbolically depicts the processing steps of this subsystem.

As shown in the figure, for each model that is defined, there exists a corresponding transformation unit (TU). TU's are organized in a pipeline structure and carry out the following two operations: *inputModel* and *transform*. The input and output formats of each TU conform to the *pipeline data* type. The operation *inputModel* retrieves the corresponding model from the model base.

The operation *transform* is specialized with respect to the characteristics of the corre-

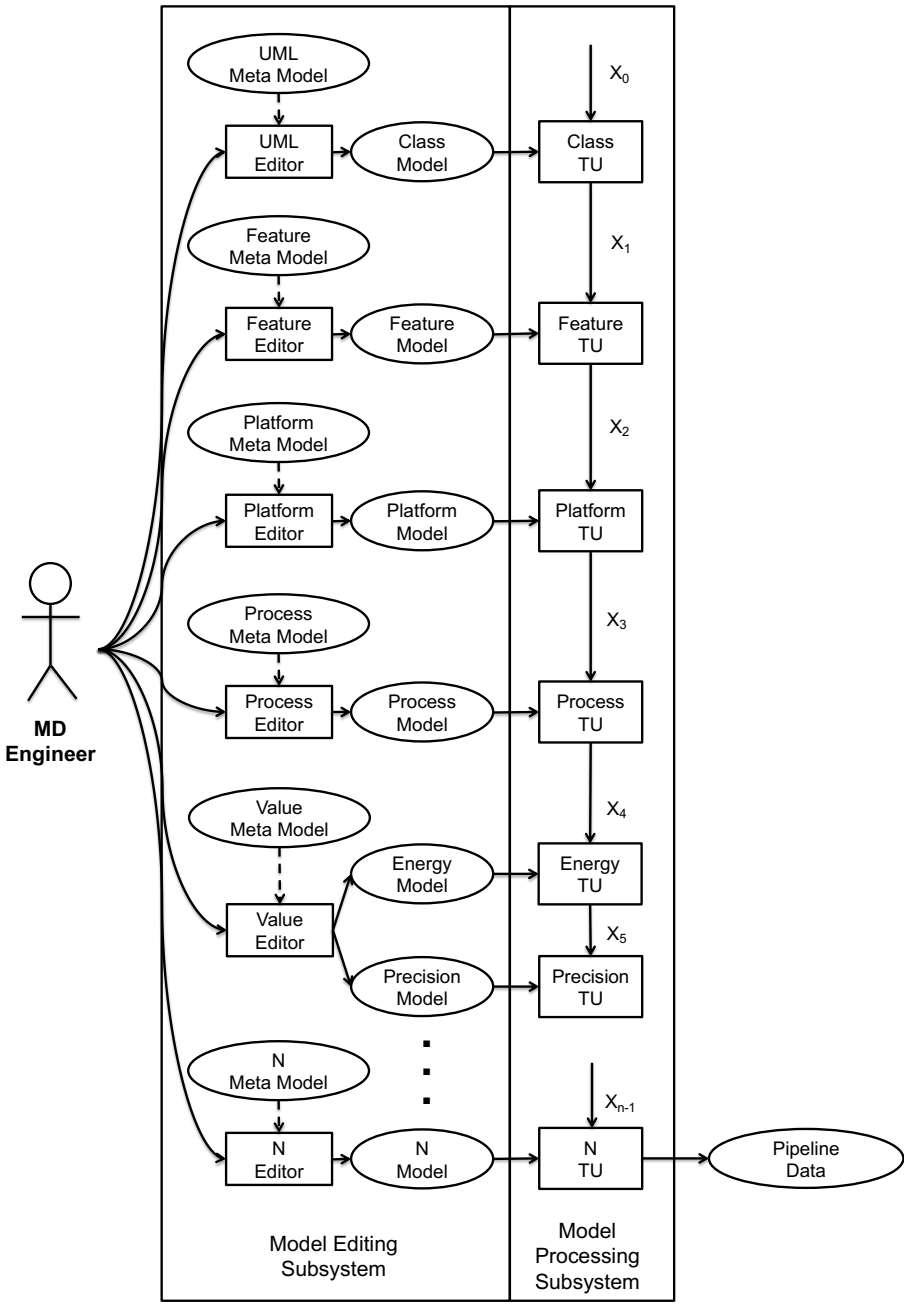


Figure 6.8: Symbolic representation of the *Model Editing* and *Model Processing* subsystems. Here, the ellipses represent data; whereas, the rectangles represent the processes.

sponding model. A typical transformation operation consists of the following steps:

1. it retrieves the incoming data from the pipeline.
2. it checks the consistency between the incoming data and the corresponding model. An error message is generated in case of inconsistency.
3. it transforms the structure of the corresponding model to the format of the *pipeline data*.
4. it concatenates the incoming pipeline data with the transformed data and place it at the output. The next processing unit takes it as an incoming *pipeline data*, and so on.

An example *pipeline data* is represented in Listing 6.1.

```
[ ...,
  {
        "owner": <aTU>,
        "size": <aSize>,
        "data": <aData>
  },
  ...
]
```

Listing 6.1: Structure of *pipeline data*.

Pipeline data consist of zero or more entities. Each TU adds its own data as an entity to the pipeline data. The three dots before and after the brackets { and } indicate a possible existence of more entities. The key owner denotes the identity of the TU which outputs this data. Here, the item <aTU> must be replaced with the name of a concrete identity of the corresponding TU. The key size indicates the number of entries in the corresponding data. The key data holds the instance of the data type that represents the corresponding model.
As defined in Figure 6.8, now assume that the first TU in the pipeline corresponds to a class model. Since this is the first TU in the pipeline, it creates the first entity, which is shown in Listing 6.2. Here, owner is defined as ClassTU, and the data size is computed as 24. A class model is defined as a dictionary where each key indicates an operation and the corresponding value indicates one or more classes that incorporate the operation. This data format is accepted by *Model Optimization Subsystem*. In our example class model given in Section 6.3, there are 24 operations with unique names. In the figure, this is represented in the following way. After the key data, first the operation getData and classes Input and CalibratedInput which incorporate this operation are specified. Following this, 23 more operations must be included in the list. For brevity, in the figure, only the last operation is shown. Here, the operation transform, is associated with four classes: FineTransform, ICP, LinearICP, NLinearICP.

```
[
  {
    "owner": "Class␣TU",
    "size": 24,
    "data":
    {
```

```
      "getData": [ "Input", "CalibratedInput" ],
      ...
      "transform": [ "FineTransform", "ICP", "LinearICP", "
         NLinearICP" ]
    }
  }
]
```

Listing 6.2: Representation of the *Class TU* output data.

The second TU corresponds to the *Feature model*. In the description of the feature model, we assumed that each feature corresponds to a class in the class model. For this reason, *Feature TU* checks whether for each feature there exists a matching class in the class model. In case of a mismatch, an error condition is raised. In Listing 6.3, the output of our example *Feature TU* is shown. Here, we will only focus on the keys size and data. The value of size is calculated as 6144, meaning that with the current specification of the feature model 6144 configurations are possible. *Feature TU* computes the configurations and adds these as the entries of the instance of the data type list, as required by *Model Optimization Subsystem*, and stores it at data. For example, the first configuration in the list includes the features Registration, Input, Filter, Sampling, VoxelGrid, Match, Transform, ICP, and LinearICP. For brevity, the remaining configurations are not shown in the figure.

```
[ ...,
  {
    "owner": "Feature␣TU",
    "size": 6144,
    "data": [
        ["Registration", "Input", "Filter", "Sampling",
        "VoxelGrid", "Match", "Transform", "ICP",
        "LinearICP"],
        ...
      ]
  }
]
```

Listing 6.3: Representation of the *Feature TU* output data.

Platform TU is the third unit in the pipeline. The output of this TU is shown in Listing 6.4. The key size is set to value one, meaning that there is only a single entry in the model and that is the platform model. The key data refers to the instance object of the platform model, denoted by the variable name System, which is defined in the model base. *Model Optimization Subsystem* directly accepts the models (objects) that conform to the platform meta model.

```
[ ...,
  {
    "owner": "Platform␣TU",
    "size": 1,
```

```
    "data": System
  }
]
```

Listing 6.4: Representation of the *Platform TU* output data.

Process TU is defined as the fourth unit in the pipeline. In the process model, we assumed that each process corresponds to an operation in the class model. Furthermore, there is a consistency relation between the process model and the feature model, since there is one-to-one relation between features and classes. Therefore, *Process TU* first checks whether these conditions are satisfied. It is possible that while configuring *Feature model* some of the optional features are not included. The operations corresponding to these excluded features must be excluded from the process model as well. Listing 6.5 shows the output data of *Process TU*. From the figure, it can be seen that the value of size is dropped from 6144 to 966 due to elimination of the irrelevant configurations in our example. The data are organized as a dictionary type where the keys are the configurations and the values are the corresponding instances that represent the relevant portions of the process model. *Model Optimization Subsystem* requires this dictionary data type. In Listing 6.5, only the first entry of the dictionary is shown. Here, the features Registration, Input, Filter, Sampling, VoxelGrid, Match, Transform, ICP and LinearICP correspond to a relevant configuration of the feature model. This configuration is associated with the instance Process that includes the portion of the corresponding processes. In the figure, for brevity, the remaining 965 instances are not shown.

```
[ ...,
  {
    "owner": "Process TU",
    "size": 966,
    "data":
      { ["Registration", "Input", "Filter", "Sampling",
         "VoxelGrid", "Match", "Transform", "ICP",
         "LinearICP"] : Process,
         ...
      }
  }
]
```

Listing 6.5: Representation of the *Process TU* output data.

Energy TU is defined as the fifth unit in the pipeline. First, *Energy TU* checks if the *Energy model* and the models retrieved from the *pipeline data* are consistent with each other. In this context, the consistency is specified as follows: Every class and operation defined in the *Energy model* must conform to the class model. Second, the configurations of processes are taken from the *pipeline data* and the total energy value per configuration is computed. Third, *Energy TU* creates a dictionary where the keys are the relevant configurations, and the values are the total energy value of the processes that are utilized in the corresponding

configuration. Finally, this dictionary is concatenated with the incoming pipeline data and placed in the output. This data representation is required by *Model Optimization Subsystem*. An example output pipeline data is shown in Listing 6.6. Consider now the key data. Here, only the first configuration is shown, which consists of Registration, Input, Filter, Sampling, VoxelGrid, Match, Transform, ICP and LinearICP. In this example, the total energy value consumed by this configuration is computed as 80.0.

```
[ ...,
  {
    "owner": "Energy␣TU",
    "size": 966,
    "data": {["Registration", "Input", "Filter", "Sampling",
      "VoxelGrid", "Match", "Transform", "ICP",
      "LinearICP"] : 80.0],
      ...
      }
  }
]
```

Listing 6.6: Representation of the *Energy TU* output data.

The last unit in the pipeline is *Precision TU*. The steps carried out in this *TU* are the same of the previous one, namely, checking consistency, extracting configurations from the process model, computing the value of each configuration and concatenate the obtained data with the incoming pipeline data. In this context, the values correspond to the precision values. The output pipeline data is shown in Listing 6.7. The first configuration associated withdata is the same but the associated value of this configuration is computed as 320.0.

```
[ ...,
  {
    "owner": "Precision␣TU",
    "size": 966,
    "data": {["Registration", "Input", "Filter", "Sampling",
      "VoxelGrid", "Match", "Transform", "ICP",
      "LinearICP"] : 320.0,
      ...
      }
  }
]
```

Listing 6.7: Representation of the *Precision TU* output data.

6.5 Model Optimization Subsystem

In this section, we first define the adopted optimization process. Second, we shortly describe the architecture of *Model Optimization Subsystem*. Finally, we give three example

scenarios to illustrate how the subsystem computes the optimal model according to the given constraints.

6.5.1 Optimization Process

OptML Framework is used in this example to optimize *software models*. According to our definition, the essential property of every software system is the *execution of operations* according to a certain program. The *Process meta model* is defined to express the dynamic behavior of a software system. The MD engineer must specialize this meta model to describe the dynamic behavior of the system being designed. This model is particularly useful to compute the timeliness and energy consumption properties of the models.
A *process configuration* corresponds to a program, which is defined as a *valid set of* processes conforming to its process model. In general, more than one process configuration can be derived from a *process model*. We assume that if there are inconsistent definition, they are detected by *Model Processing Subsystem* before the *pipeline data* reaches *Model Optimization Subsystem*.
Model Optimization Subsystem searches for a solution of a process configuration, where each process is allocated to the appropriate elements of the *Platform model*, while satisfying the constraints[4] defined by the MD engineer of each process. Currently, the following constraints are supported:

- Capacity. This is specified based on the units of the relevant platform elements. Examples are memory size, processing power, etc.
- Worst-case execution time.
- Release time.
- Deadline.
- Dependency constraints.
- Preemption constraint.
- Migration constraint.
- Mutual-exclusion constraint.

The Platform model specifies the resources and their characteristics so that they can be matched to the constraint defined in the process model.
While searching for an optimal configuration, *Model Optimization Subsystem* may adopt various strategies among the set of candidate configurations, such as first-fit, nth-fit and first-n searches. In the first-fit search approach, the first configuration that satisfies the requirements is selected and the search process is terminated. In the nth-fit search, as the name implies, the first n valid configurations are selected if they exist. Finally, in the first-n search, the first n configurations are selected even if some of them is invalid.
In addition to the process requirements, the optimization process can be extended by defining constraints that can be derived from the *Value meta model* similarly to the examples of the energy and precision models presented in Appendix A.4. These additional constraints

[4]These are canonical constraints taken from [28]

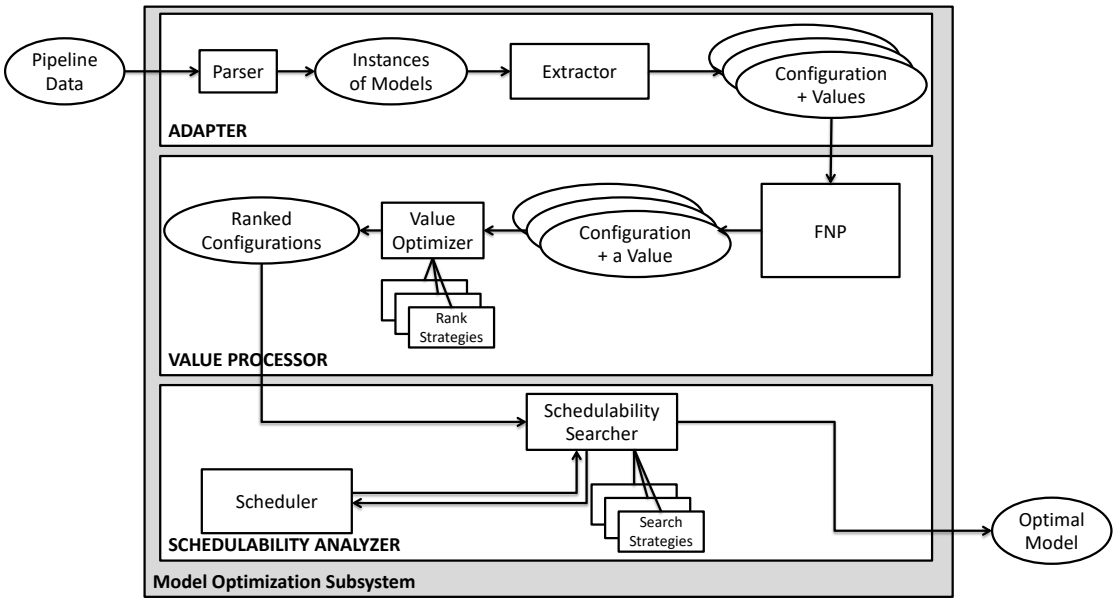


Figure 6.9: Representation of a *Model Optimization Subsystem* architecture.

are only meaningful if more than one configuration is considered. The configurations that satisfy the process requirements and are within the boundaries of the desired value constraints, ranked according to the optimal required value. The optimal value may be either minimum or maximum of the values of the considered configurations.
A *Class model* is a static representation of a program, and as such it defines the bindings of processes to the operations of classes which can be overridden through inheritance. Therefore, the Class model restricts the definition of configurations derived from the Process meta model. Similarly, the *Feature model* can be seen as a restriction over the *Class model*.

6.5.2 A Model Optimization Subsystem Architecture

Model Optimization Subsystem consists of three components: *Adapter*, *Value Processor* and *Schedulability Analyzer* as shown in Figure 6.9.
Adapter has two subcomponents:

- *Parser* accepts the *pipeline data* as input and extracts the instances of models that are generated by the *transformation units*. These are a dictionary representing the class model, a list of configurations obtained from the feature model, an instance object that represents the platform, a dictionary of process configurations, and two dictionaries representing energy and precision models, respectively.
- *Extractor* processes the instances of models and generates a list of *process configurations* associated with the values to be considered for the optimization process.

The module *Value Processor* includes two subcomponents:

- *FNP*, implements three operations *Filter*, *Normalize* and *Prioritize*. First, the operation

Filter, eliminates the *process configurations* that have associated values out of the desired boundaries. For example, the MD engineer may indicate that the precision value must be above a certain number, and/or the energy value must be less than a certain number, etc. Second, the operation *FNP* normalizes each value between 0 and 1. Finally, based on the input given by the MD engineer, the operation *Prioritize* computes a single value using the following formula:

$$P(\vec{v}_{m_1}, \vec{v}_{m_2}, \ldots, \vec{v}_{m_n}) = \sum_{i=1}^{n} \alpha_i \times N \circ F(\vec{v}_{m_i}), \tag{6.1}$$

where α_i represents the priority of the corresponding model m_i in the total equation; $N \circ F$ indicates the composition of functions *normalize* and *filter*; and $\vec{v}_{m_i}$ represents the list of values of the process configurations computed according to the value model m_i.

- *Value Optimizer* ranks the list with respect to the selected *Rank Strategy* in an ascending or a descending order. According to the choice of the strategy, the MD engineer may request for the best n number of process configurations rather than delegating all of them to the following subcomponent *Schedulability Searcher*. The best configurations are computed by using an appropriate optimization algorithm. In this way, only the most promising configurations are considered first. If no quality attributes are defined, this component has no effect.

The module *Schedulability Analyzer* incorporates two subcomponents:

- *Scheduler* is based on an application framework called First Scheduling Framework (FSF) [106], which provides the necessary abstractions and mechanisms to implement schedulers and explained in Chapter 5 in detail.
  Since the models defined in this chapter are designed in accordance with the models in FSF, the transformation of the models is realized in the following way: Firstly, the platform model in OptML is translated to the resource model in FSF. Secondly, the *Process model* that defines the execution flow of the operations in the *Class model* converted to the task model in FSF. Thirdly, the scheduling characteristics are defined with respect to the requirements of each process defined in Section 6.5.1, and scheduling strategy is defined as *minimizing the makespan*. Since the execution time of a process in the *Process model* changes with respect to the class in which it is defined, a separate scheduling problem is generated for each configuration. In the sense of schedulability, any feasible solution (schedule) computed by *Scheduler* makes the corresponding process configuration *valid*.
- *Schedulability Searcher* implements the search algorithm based on a certain strategy. To this aim, it retrieves the top element of the ranked configurations from the subcomponent *Value Processor* and calls *Scheduler* to evaluate it. Depending on the result of this evaluation, this process may iterate over the remaining configurations in the ranked list based on the selected strategy. For example, if the first-fit strategy is used, *Schedulability Searcher* terminates the search as soon as *Scheduler* finds a solution that satisfies the constraints. This configuration is considered to be the *optimal model*.

Other related models, such as the *Class model*, can be reconstructed based on this result.

6.5.3 Example Scenarios

In the following subsections, we will give a set of model optimization scenarios to illustrate the utility of OptML Framework with respect to the requirements defined in Section 6.1. In practice, it is not possible to validate the correctness of a software system with the help of user-defined scenarios since the number of scenarios in any practical system can be extremely large. Therefore, we categorize the scenarios in the following way: (i) time analysis on a single- and multi-processor architecture; (ii) model optimization based on time analysis combined with single quality attribute; (iii) model optimization based on time analysis on multiple quality attributes; (iv) three different search strategies to find an optimal model. We assume that these four categories represent a large number of scenarios that can be experienced in practice.
With respect to the requirements given in Section 6.1, the fulfillment of the first three requirements, finding the schedulable optimal model while satisfying the quality requirements, is demonstrated by the three categories of the scenarios that will be given in the following subsections.

Scenario 1: Finding out the schedulability of the model with respect to a platform model

This scenario aims at demonstrating the fulfillment of the first requirement: schedulability of processes on platforms. To this aim, we have made the following assumptions about the models:

- The class, feature and process models are shown in Appendix.
- The platform model has the following characteristics: It has two terminal resources CPU0101 and MEM0101, belonging to ACTIVE and PASSIVE resource types, respectively. The object diagram for the corresponding platform model is shown in Figure 6.10. The capacity of the active terminal resource is 1; whereas, the passive terminal resource has 1024-unit capacity. Each of these terminal resources has one running state. In this scenario, we choose a more restricted platform model (with a single processing unit) than the one given in Appendix A.2 to demonstrate the effect of platform capacities on the schedulability process.
- The criteria of the scheduling objective is set to "minimizing the maximum makespan" [109].
- The scheduling window for the processes is set to 50.
- The search strategy is chosen as *first-fit*.
- The process configurations are not ranked by the subcomponent *Value Optimizer*.

With these given assumptions, *Scheduler* returns a solution that is depicted in Figure 6.11. The figure consists of two rectangles. The top horizontal rectangle depicts the schedule computed by *Scheduler*. This figure only includes the processes that are selected by the optimizer. The horizontal axis corresponds to *Time*, and the vertical axis corresponds to the capacities of the two *terminal resources*, CPU0101 and MEM0101. Each unit in the vertical

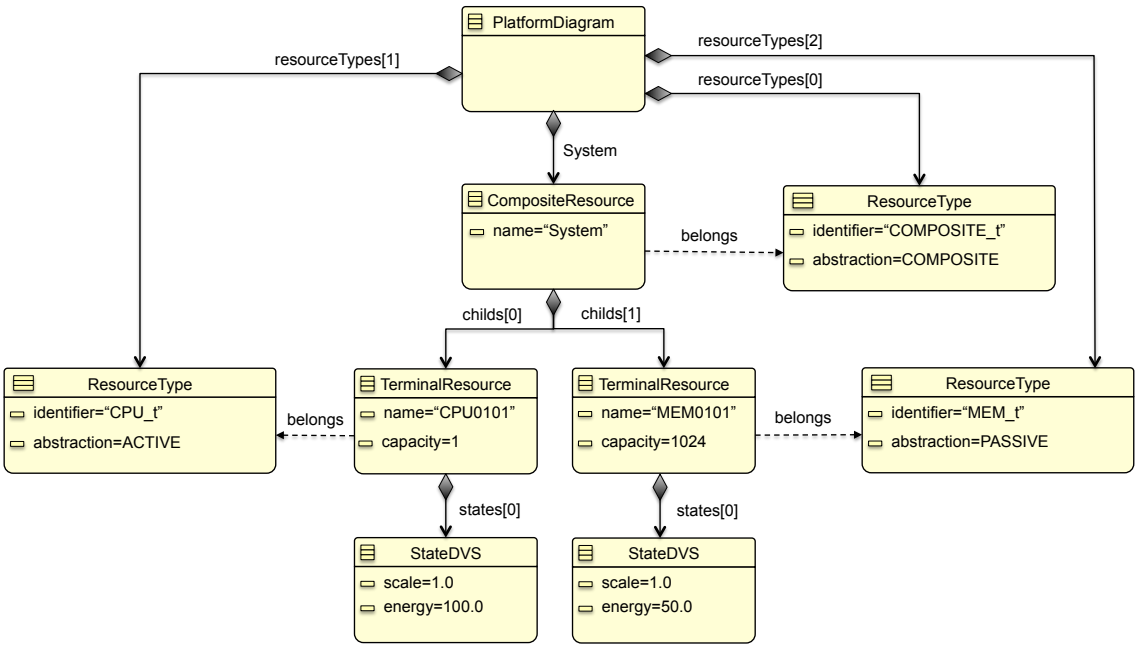


Figure 6.10: An object diagram of the platform model.

axis corresponds to total amount of capacity for each resource. The larger rectangle shows all the processes, although some of them may not be included in the schedule. Each process is indicated by a different color as shown in the legend.

Scenario 2: Finding the schedulable optimal model with respect to a single quality attribute

The second scenario is defined to illustrate the first three requirements in Section 6.1: introducing new quality attributes and finding out the optimal model that satisfies both the schedulability and the quality requirements.
Along this line, the following assumptions are made:

- The class, feature, platform and process models are taken from Appendix.
- A single quality model *energy consumption* is introduced, which is defined in Appendix A.4.
- The criteria of the scheduling objective is set to "minimizing the maximum makespan".
- The scheduling window for the processes is set to 23.
- The search strategy is chosen as *3rd-fit*.
- The process configurations are ranked by the subcomponent *Value Optimizer* with respect to the ascending energy values.

In this scenario, the multi-processor architecture is selected. The result of the *optimization process* based on the given assumptions is shown in Figure 6.12. This figure consists of three subfigures where each subfigure shows the evaluation of a particular process configuration.

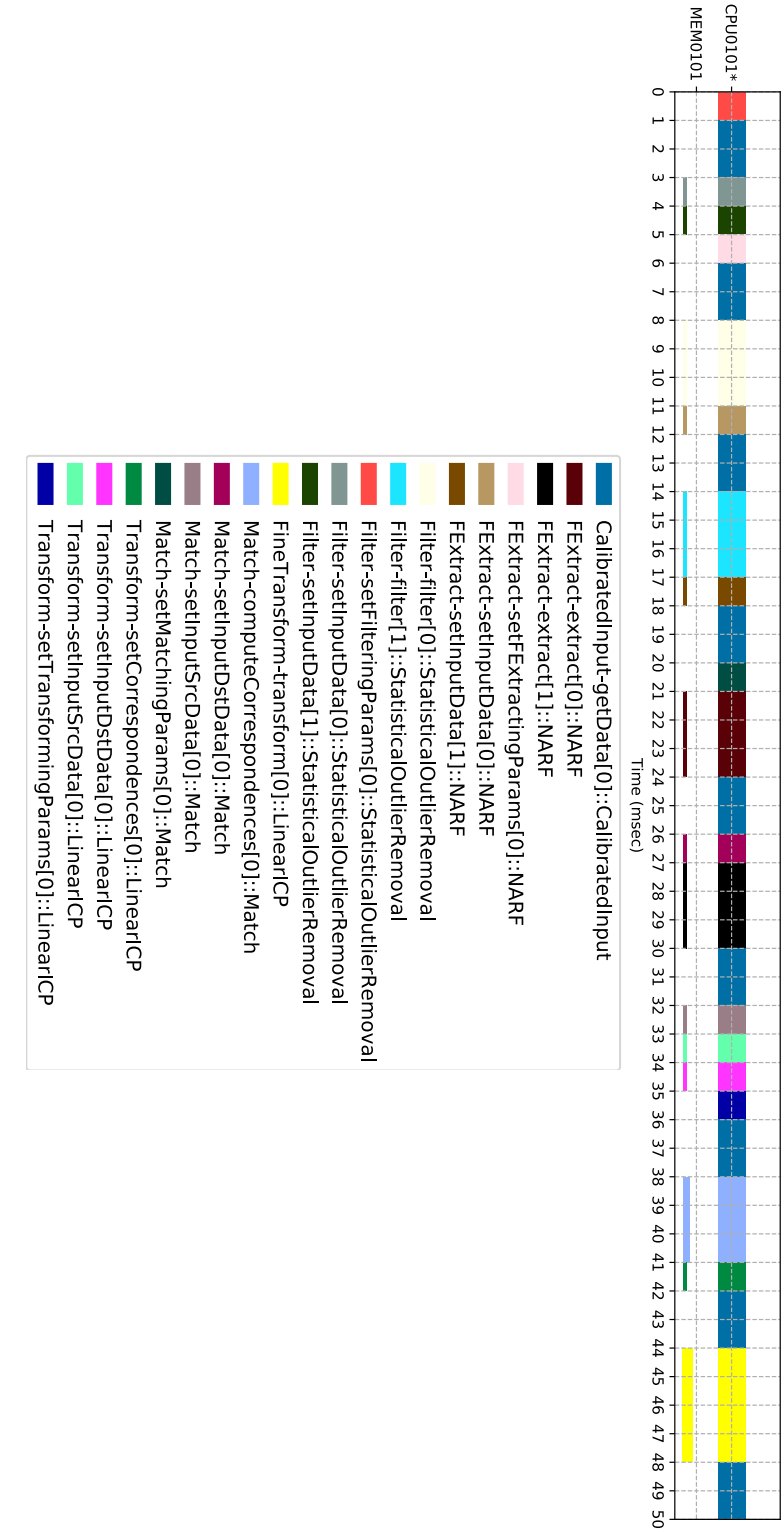


Figure 6.11: Schedule produced by *Scheduler* using the first-fit strategy.

The top subfigure shows a schedule with the least energy consumption; whereas, the bottom subfigure shows the most among the ones that are evaluated. Therefore, the top subfigure is considered as the optimal model.

Scenario 3: Finding the schedulable optimal model with respect to multiple quality attributes

This scenario extends the previous one with an additional *precision* quality attribute. To this aim, the assumptions are the same as the previous scenario except the followings:

- The scheduling window size is set to 21.
- The search strategy is set to first-25.
- A new *precision* quality attribute is defined in Appendix A.4.
- The priorities of the quality attributes *energy* and *precision* are defined as 0.2 and 0.8, respectively.
- The process configurations are ranked by the subcomponent *Value Optimizer* with respect to the ascending both energy and precision values. To preserve uniformity, normalized value 1 corresponds to lowest precision; whereas 0 is the highest.

The result is given in Figure 6.13. Here only two configurations are shown. The second configuration found in the ranked configurations happens to be not schedulable. This is, because, the operation transform exceeds the specified scheduling window size if it is implemented with a high precision value.
The figure consists of two subfigures, where the top subfigure corresponds to a configuration with a better quality value. This configuration is selected as the optimal model.

6.6 Related Work

Model-driven architecture (MDA) aims at separating platform-independent and platform-dependent models from each other [12]. MDE adds engineering practices to MDA with meta models and model transformations [38]. In MDE, not only models but also meta models and model transformations are the core assets of software development. The research activities over MDE are very broad, including domain specific models, model building, model verification, model reuse, model transformation and code generation [77, 131].
In the literature, the terms model and optimization are used in two ways: (i) models for optimization; and (ii) model optimization. There have been considerable number of works on models for optimization, where researchers investigate mostly mathematical models to define and implement optimization processes. For our approach, such techniques are adopted in the subcomponents *FNP* and *Value Optimizer* for value optimization, and in the subcomponents *Schedulability Searcher* and *Scheduler* for schedulability analysis.
The purpose of model optimization, however, is to search for the models within a model-base that satisfy certain criteria. This is the main focus of the chapter. In contrast with models for optimization, there are hardly any publications that address this problem. As stated by Chenouard [32], a constraint-programming based design synthesis process is presented using MDE techniques. A similar approach is adopted in Joachim's article [42], where an optimal model is searched within the context of certain requirements. The dif-

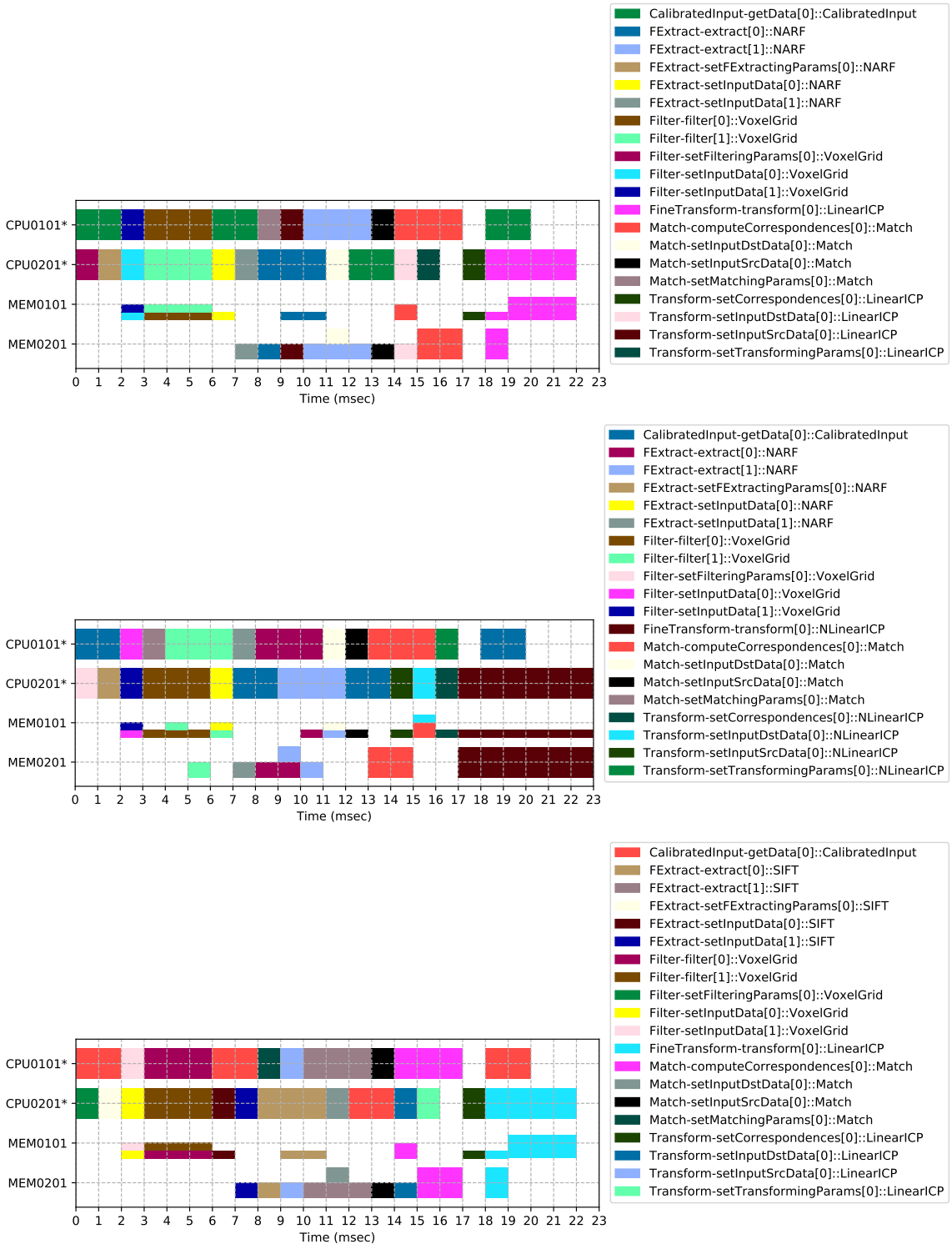


Figure 6.12: Three schedules ordered according to their energy consumption values.

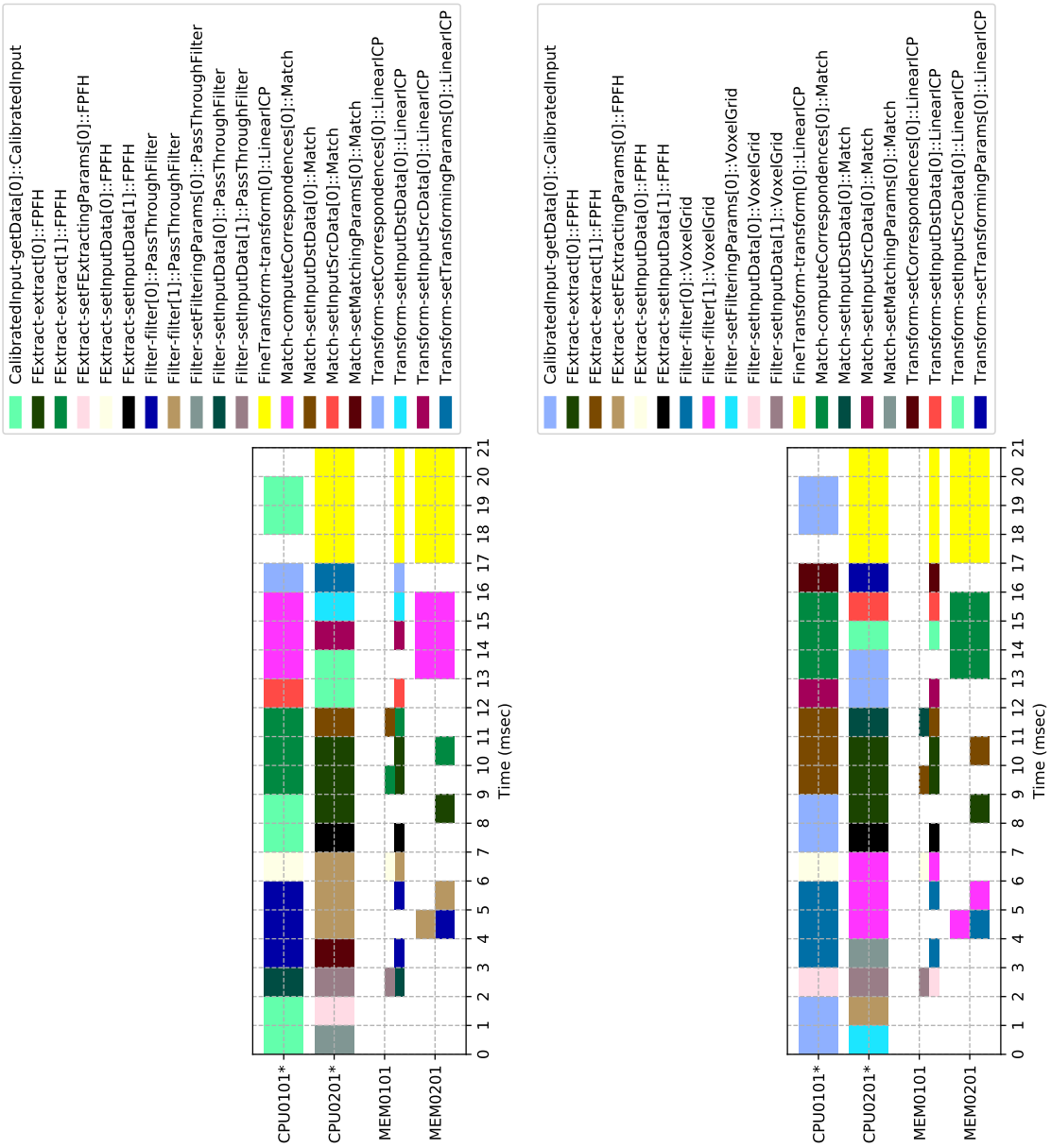


Figure 6.13: Two schedules ordered according to their energy consumption and precision values.

ference between these two articles is that in the former, the optimal model is searched at a model level using constraint programming, whereas in the latter, search is defined as a model transformation. The objectives of both articles are, however, different than ours. The aim in these articles is to synthesize the optimal model that satisfies the constraints, whereas in our work, the aim is to select the optimal model among model configurations that satisfy the constraints. There are a number of research initiatives that aim at verifying models based on certain specifications. With the help of OCL [137], for example, certain properties of models can be formally specified. Various tools have been developed to verify models annotated with such specifications [60]. These tools in general are used for verification and testing purposes but not for model optimization.
To the best of our knowledge, there is no framework proposal that aims at optimizing models defined in the context of MDE, as proposed in this chapter.
There has been a number of research initiatives aiming at optimizing software architectures according to a set of quality attributes. In [127], algorithms are proposed to optimize TV architectures for the qualities availability, reduced memory usage and time performance. Multi-objective optimization techniques are proposed in [41] with respect to certain quality attributes such as production speed, reduced energy usage and print quality. A design method for balancing quality attributes energy reduction and modularity of software is proposed in [133].

6.7 Evaluation

We will now evaluate this chapter with respect to the nine requirements given in Section 6.1. The fulfillment of the first three requirements given in Section 6.1 is explained in Section 6.5.3.
The fourth requirement, checking consistency among models, is realized by the TU's defined in *Model Processing Subsystem* as explained in Section 6.4.
The fifth requirement, supporting new models in EMF, is provided with the following condition: For each new model, the *tool engineer* must define the corresponding TU with the necessary model extraction, consistency checking and model transformation functions.
The sixth requirement, pruning models, is supported in the TU's by re-defining the operation *inputModel* with different retrieval strategies so that only the relevant parts of the model are selected. For example, in the current implementation of the framework, we support the following strategies: (a) retrieve the complete model (default); (b) include only the classes denoted by the *MD engineer*; (c) include all the classes with a query. The architecture allows to introduce new strategies. However, if not all model elements are selected, the pruned model can be inconsistent with the other models. In our framework, for example, if some classes, which are eliminated from the class model, are included in the feature model, the feature TU will give an error message. For brevity, this chapter does not focus on the pruning techniques. An interested reader may refer to the report [87]. As it is explained in Section 6.4, due to the pipeline architecture of *Model Processing Subsystem*, the framework allows the tool designer to introduce such extensions to the existing utilities.

As demonstrated in Section 6.5.3, the seventh requirement, introducing new quality attributes, can be realized by specializing the *Value meta model* with the following restrictions: (a) the quality attributes are expressed in numbers and associated with classes and their operations; (b) if necessary, the quality attributes per configuration are computed by the corresponding TU. If this way of representing the desired quality attribute is not appropriate, a new value meta model and the corresponding TU must be introduced. The pipeline architecture of *Model Processing Subsystem* makes this possible without changing the other TUs.
The eighth requirement, introducing new value optimization algorithms, is supported in the following way: Currently, OptML Framework implements a rather straightforward value optimization based on filtering, normalizing, prioritizing and priority-based ranking with the help of the subcomponents *FNP* and *Value Optimizer*. Different quality values are merged into a single value. As the next step, the schedulability of the ranked configurations is analyzed. One may adopt, however, different value optimization techniques depending on the needs, and the feasibility of time and space complexity of the optimization algorithms. For example, one may aim at reducing the time of optimization process by using heuristic rules. In the literature, various optimization algorithms are presented such as hill-climbing, exhaustive search and pareto-front multi-objective optimization [127]. These changes can be encapsulated in the subcomponents *FNP* and *Value Optimizer*.
The last requirement, adopting new search strategies for scheduling, can be fulfilled by defining a new search strategy for the subcomponent *Schedulability Searcher*.
The time performance of the model optimization process is considered important for the usability of the framework. The current architecture allows performance improvement in the following ways: (a) introducing effective model-pruning strategies in the corresponding TUs; (b) applying different optimization algorithms in *Model Optimization Subsystem*; (c) using efficient schedulability search strategies; and (d) using efficient solvers in the subcomponent *Scheduler*.
The average execution time of the solver per configuration are shown in Table 6.1. In the figure, the bottom row shows the results of the evaluation. The top five rows defines the parameters of the implementations. In none of these scenarios, model pruning is used. Normalization of the quality values are only applied to the second and third scenarios where the values are mapped into the range of 0 to 1. Here, NA means Not Applicable. The definitions of rank and search strategies are self-explanatory. The row Utilized Solver defines the constraint solver that is used in the subcomponent *Scheduler*. The execution time of the solver for the scenario 1 is much higher due to the longer schedulability window size. The time that is required to search among the possible solutions by the subcomponent *Schedulability Searcher* is negligible when compared with the time performance of the solvers. To determine the total time performance, one needs to multiply the last column value with the number of iterations in the search strategy.

Parameter	Scenarios		
	Scenario 1	**Scenario 2**	**Scenario 3**
Model Pruning	No	No	No
Normalization	NA	[0 1]	[0 1]
Rank Strategy	NA	Ascending Order	Ascending Order
Search Strategy	First-fit	3rd-fit	first-25
Utilized Solver	Gurobi [62]	Gurobi	Gurobi
Average Execution Time of the Solver per configuration	305 sec	30 sec	26 sec

Table 6.1: The performance values of the scenarios. The scenarios are implemented on MacBook Pro with 2.6 GHz Intel Core i5 processor and 8 GB 1600 MHz memory.

6.8 Conclusion

This chapter identifies the following problems: (a) model complexity; (b) the lack of quality-based model selection; and (c) the lack of support of multiple quality-based evaluations. To address these problems, OptML Framework is proposed. This framework accepts various models defined in EMF and processes them according to user preferences and model properties; and computes the optimal schedulable models based on value optimization and constraint-based scheduling algorithms. To the best of our knowledge, this is the first generic MDE framework that is suitable for model optimization.
The utility of the framework is demonstrated by implementing three scenarios inspired by *registration systems* used in the area of image processing. The scenarios show that the required objectives of model optimization as defined in this chapter are fulfilled. With the help of design patterns and various architectural styles, the architecture has a modular structure and thereby allows the introduction of new strategies for model extraction and transformation, value optimization and schedulability analysis. The framework is fully implemented and tested. It integrates a number of third-party software such as Eclipse EMF [51], pyecore [10], pyuml2 [11], FSF [106], Numberjack [65], matplotlib [72], Gurobi [62], SCIP [57], Mistral [44].

CHAPTER 7

Conclusions

This chapter concludes the thesis by reflecting on the addressed research challenges in two categories: challenges in designing (a) scheduling software, and (b) optimal models in model-driven engineering. To this aim, first the identified research challenges are presented. Second, the related research questions are summarized related to the identified challenges.

7.1 Designing Scheduling Software

7.1.1 Challenges

In Chapter 1, we stated that implementing software systems that incorporate schedulers can be challenging due to the following concerns:

1. *Time-consuming process:* In addition to dealing with well-known challenges in designing software systems, the software engineer has to define and implement the required tasks, resources, associated parameters, objectives, strategies, and the constraints, and/or algorithms. Due to the inter-dependencies, the software engineer may have to spend a considerable amount of time to complete this process successfully.
2. *Systems can be large and complex:* Scheduling systems may incorporate many objects. In addition to the scheduling related parameters, these objects may also contain complex inter-dependencies.
3. *Safety-critical:* Scheduling systems are in general safety-critical systems. Schedulability of tasks is an important quality attribute to be enforced.
4. *Reusability:* Due to the complexity of such systems, it is generally very costly to develop them from scratch. In addition, companies may aim to develop family of products. Therefore, reusability of software is considered very beneficial.
5. *Run time evolution:* Scheduling systems are in general continuously operational systems. Therefore, to cope with the continual change of user requirements, solutions to the new requirements must be introduced to the systems at run time.

Of course, many of these challenges depend on each other. This makes designing software systems that incorporate schedulers even harder.

7.1.2 The Software Engineering Approach

The challenges stated in Section 7.1.1 are addressed at three complementary levels:

i *Application Framework.* An application framework for schedulers incorporates domain-specific class hierarchies with the necessary operations and attributes so that the desired schedulers can be instantiated with the necessary parameters if needed. In this approach, the software engineer has the full freedom to extend, modify, and discard the parts of software library. As a disadvantage, the software engineer must have a detailed knowledge about the library and the programming language used.

ii *Model-Driven Engineering.* This approach provides a higher-level abstraction of the scheduling domain. Dedicated tools for checking the consistency of the parameters are supplied. The advantage is that domain experts on schedulers can conveniently define the desired schedulers since the models are assumed to be closer to the experts' perception in comparison to expressing schedulers at the level of programming languages. However, the experts can only define schedulers that can be expressed by these specific models.

iii *Software Product-line.* This is an extension of MDE approaches with the concepts of product families. The advantages and disadvantages are similar to the ones of the

MDE approach.

Unfortunately, to the best of our knowledge, there are no publications in the literature that propose an application framework, a Model-Driven Engineering approach and/or a software product-line engineering approach to develop scheduling systems.

7.1.3 Research Questions and Solutions

The identified research questions and the proposed solutions in relation to the challenges to design scheduling software are the following:

RQ1. What are the important concepts of scheduling systems and accordingly how to define an expressive domain model for scheduling systems?

Solution proposal: Chapter 2 gives a comprehensive overview of the scheduling theory, the adopted terminology, and the notations used. Based on Chapters 2 and 5 proposes a domain model for schedulers. To illustrate the expressivity of the model, a set of scheduling problems are presented and configured, which are commonly referred to in the literature as canonical examples. We show in Chapter 4 that the proposed model is expressive enough to represent these examples adequately.

RQ2. How to define an expressive feature model for scheduling systems so that a large category of family of scheduling systems can be expressed?

Solution proposal: This research question is related to the first research question. To derive a domain model for a large category of family of scheduling systems, a commonality and variability analysis is carried out using the available literature in the scheduling theory. The result of this analysis is expressed using the feature model notation. The expressivity of the notation is validated by configuring a set of canonical scheduling examples from the feature model.

RQ3. How to ensure the invariants of feature models and check if a valid configuration can be generated accordingly?

Solution proposal: As a notation, a feature model defines the mandatory and optional features, cardinality rules, cross-tree constraints, etc. Assuring invariants means that the invariants of the model are respected during the configuration phase of the feature model. Furthermore, there should be at least one possible configuration from the feature model otherwise there is no use of adopting a feature model if it cannot express any configuration in the desired domain. In our case, the feature model must express the commonality and the variability of the scheduling theory as desired. As described in Chapter 4 of this thesis, a model checking tool is adopted to generate and verify the valid configurations from the specified feature model.

RQ4. How to design and implement an object-oriented application framework library for scheduling systems with i) a high degree of reusability and ii) evolvability?

Solution proposal: An application framework library is presented in Chapter 5 that conforms to the feature model described in Chapter 4. The framework is an object-oriented implementation of the feature model, where classes and class-inheritance mechanisms are used to express the features. To be able to express the variabilities, design patterns are used. In addition, APIs are defined to alter the instantiations of

the framework at run time, if needed. It is shown in Chapter 5 that the canonical scheduling examples can be configured from the framework through reuse and/or by calling the appropriate functions at run time.

7.1.4 Discussions and Future Work

Justification of the claims for the quality attributes expressivity, reusability and run time evolvability are based on the following assumptions:

1. The domain of the scheduling theory is rather mature and the notations that are adopted are expressive enough to cover different scheduling problems.
2. The canonical scheduling examples used in the literature are representative.
3. The adopted design patterns and run time functions are expressive enough to implement the configurations as required.
4. The feature model notation is sufficient enough to express the invariants of the domain.

To determine the validity of these assumptions in practice, we plan to carry out experiments in real industrial settings.

7.2 Designing Optimal Models in Model Driven Engineering

7.2.1 Challenges

We claim in Chapter 1 of this thesis that in addition to schedulability of tasks, one may need to search for optimal models that satisfy various quality concerns. Within the context of trade-off analysis of multiple quality attributes in an MDE approach, the following aspects must be taken into consideration:

i) *Large configuration spaces of models:* It is common practice to use multiple related models in MDE environments for a given system. Each of these models may define different kinds of variations. The possible combination of all variations may potentially enable many possible instantiations of models, which can be difficult for the software engineer to manage.

ii) *Introducing new quality attributes:* New quality models must be introduced if necessary.

iii) *Optimization of configurations:* Software engineers generally have to trade-off different quality attributes to configure the most suitable model for a given application setting. For example, a particular model configuration may improve the quality attribute “reducing energy consumption” while decreasing the quality attribute “time performance”. It would be very profitable if the MDE environments are equipped with tools and techniques to optimize model configurations based on multiple quality attributes.

7.2.2 The Software Engineering Approach

Our approach to address the challenges stated in Section 7.2.1 is to design a workbench that extends the existing MDE environments with the necessary tools. These tools must enable the designers to check the consistency among the models defined, deal with large configuration spaces, introduce new quality models if needed, define constraints and optimization criteria and select the model configurations that satisfy the best trade-off conditions according to the given criteria.

7.2.3 Research Questions and Solutions

The identified research questions and the proposed solutions in relation to the challenges of configuring optimal models are the following:

RQ5. How to design an MDE environment so that new models and meta models, model pruning techniques, quality attributes, quality optimization criteria and search methods can be introduced in a convenient manner?

Solution proposal: To address these challenges, in Chapter 6, OptML Framework is proposed. Here, we assumed that models and meta models are realized by using the standard MDE environments like MDE EMF environment. Once the models are defined, they are provided to Model Processing Subsystem of OptML Framework. The model processing system adopts a pipeline architecture of transformation units (TU's). Each model defined by the MDE engineer has a corresponding TU, which inputs the previously defined model, checks the consistency rules of the previously defined model and the current model and transforms these to a common pipeline data format. This process continues until the last TU in the pipeline is defined. The output is then provided to the model optimization subsystem. This subsystem ranks the possible configurations of models according to the optimization criteria. Finally, the schedulability of the configurations is checked and if necessary, the configurations are re-ordered. The framework architecture is designed in a modular fashion so that all the desired extensions, such as new model pruning techniques, quality attributes, quality optimization criteria and search methods can be introduced in a convenient manner.

RQ6. Within MDE environment, how can the following aspects of models can be expressed and computed effectively?

- **i)** Checking consistency among various models,
- **ii)** Generating the model-configuration space,
- **iii)** Annotating various quality attributes to model configurations,
- **iv)** Assigning relative priority to the predefined quality attributes,
- **v)** Analyzing schedulability of models,
- **vi)** Optimizing models based on multiple quality values.

Proposed solution: These aspects are computed in the framework in a pipeline fashion in two main subsystems: Model-Processing Subsystem and Model Optimization Subsystem. Each subsystem has an internal pipeline architecture. The above given computations are carried out in order by the dedicated modules in the pipeline. The implementation of each module can be changed if desired, for example, to optimize

performance, provided that input-output data formats are respected.

7.2.4 Discussions and Future Work

The usefulness of OptML Framework is justified based on the following assumptions:

1. The desired models can be defined using the MDE ECore framework.
2. The consistency among the models can be checked by referring to the static properties of the models.
3. For every model, there is a corresponding TU. The tool engineer must configure the framework for this purpose.
4. The pipeline architecture is suitable to process and optimize the models. If there are mutual dependencies among models, then data rerouting techniques may be employed.
5. The desired quality models can be defined by associating quality attributes on model elements.
6. If the current search methods and optimization techniques are not satisfactory, implementations of some modules must be changed. The architecture is prepared to deal with implementation changes through the adoption of design patterns, if the input-output signatures of the corresponding modules are kept.
7. The feature model notation is sufficient to express the invariants of the domain.

As future work, it is possible to verify the current decomposition of the OptML architecture with the use of various quality attributes and optimization criteria within the industrial projects.

Bibliography

[1] Django project. http://www.djangoproject.com/.

[2] Abeni, L. and Buttazzo, G. (2004). Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167.

[3] Ahmad, W., de Groote, R., Holzenspies, P., Stoelinga, M., and van de Pol, J. (2014). Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In *Proceedings of the 14th International Conference on Application of Concurrency to System Design (ACSD 2014)*, pages 72–81, United States. IEEE Computer Society.

[4] Akşit, M. (2018). The role of computer science and software technology in organizing universities for industry 4.0 and beyond. In Ganzha, M., Maciaszek, L., and Paprzycki, M., editors, *Proceedings of the 2018 Federated Conference on Computer Science and Information Systems*, volume 15 of *Annals of Computer Science and Information Systems*, pages 5–11. IEEE.

[5] Aksit, M. (2002). *Software Architectures and Component Technology*. The Springer International Series in Engineering and Computer Science. Springer Verlag.

[6] Aksit, M. (2004). The 7 c's for creating living software: A research perspective for quality oriented software engineering. *Turkish Journal of Electrical Engineering and Computer Sciences*, 12:61–95.

[7] Aksit, M., Rensink, A., and Staijen, T. (2009). A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD '09: Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development*, pages 39–50. Association for Computing Machinery (ACM). Winner of the Best Paper Award.

[8] Aksit, M., Tekinerdogan, B., Marcelloni, F., and Bergmans, L. (1999). *Deriving Object-Oriented Frameworks from Domain Knowledge*, pages 169–198. John Wiley & Sons.

[9] Antkiewicz, M., Bąk, K., Murashkin, A., Olaechea, R., Liang, J., and Czarnecki, K. (2013). Clafer tools for product line engineering. In *Software Product Line Conference*, Tokyo, Japan. Accepted for publication.

[10] Aranega, V. (2016). A python(nic) implementation of emf/ecore (eclipse modeling framework).

[11] Aranega, V. (2018). A python implementation of the uml2 metamodel based on pyecore.

[12] Assmann, U., Aksit, M., and Rensink, A. (2005). *Model Driven Architecture: European*

MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004. Revised Selected Papers.

[13] Azizi, S. and Panahi, V. (2012). Formal specification of semantics of uml 2.0 activity diagrams by using graph transformation systems.

[14] Babur, Ö., Cleophas, L., van den Brand, M., Tekinerdogan, B., and Aksit, M. (2018). Models, more models, and then a lot more. In Seidl, M. and Zschaler, S., editors, *Software Technologies: Applications and Foundations*, pages 129–135, Cham. Springer International Publishing.

[15] Baker, K. R. and Trietsch, D. (2013). *Principles of sequencing and scheduling*. John Wiley & Sons.

[16] Baptiste, P., Le Pape, C., and Nuijten, W. (2012). *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media.

[17] Batory, D. (2005). *Feature Models, Grammars, and Propositional Formulas*, pages 7–20. Springer Berlin Heidelberg, Berlin, Heidelberg.

[18] Becker, C. and Scholl, A. (2006). A survey on problems and methods in generalized assembly line balancing. *European Journal of Operational Research*, 168(3):694 – 715. Balancing Assembly and Transfer lines.

[19] Belady, L. A. and Lehman, M. M. (1976). A model of large program development. *IBM Systems Journal*, 15(3):225–252.

[20] Bodhuin, T., Canfora, G., and Troiano, L. (2007). Sormasa: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. pages 23–24.

[21] Breivold, H. P., Crnkovic, I., and Eriksson, P. J. (2008). Analyzing software evolvability. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 327–330. IEEE.

[22] Brown, L. G. (1992). A survey of image registration techniques. *ACM Comput. Surv.*, 24(4):325–376.

[23] Brucker, P. (2010). *Scheduling Algorithms*. Springer Publishing Company, Incorporated, 5th edition.

[24] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., and Sabetzadeh, M. (2006). A manifesto for model merging. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 5–12. ACM.

[25] Burns, A. (1991). Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116–128.

[26] Burns, A. (1994). Preemptive priority-based scheduling: An appropriate engineering approach. In *Advances in Real-Time Systems, chapter 10*, pages 225–248. Prentice Hall.

[27] Buschmann, F., Henney, K., and Schimdt, D. (2007). *Pattern-oriented Software Architecture: on patterns and pattern language*, volume 5. John wiley & sons.

[28] Buttazzo, G. C. (2011). *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media.

[29] Cardelli, L. (1987). Basic polymorphic typechecking. *Science of Computer Program-*

ming, 8(2):147 – 172.

[30] Chen, J.-J. and Kuo, T.-W. (2005). Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In *2005 International Conference on Parallel Processing (ICPP'05)*, pages 13–20.

[31] Cheng, R., Gen, M., and Tsujimura, Y. (1996). A tutorial survey of job-shop scheduling problems using genetic algorithms—i. representation. *Computers & Industrial Engineering*, 30(4):983 – 997.

[32] Chenouard, R., Hartmann, C., Bernard, A., and Mermoz, E. (2016). Computational design synthesis using model-driven engineering and constraint programming. volume 9946, pages 265–273.

[33] Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., and Little, R. (2002). *Documenting Software Architectures: Views and Beyond*. Pearson Education.

[34] Clements, P., Garlan, D., Little, R., Nord, R., and Stafford, J. (2003). Documenting software architectures: views and beyond. In *Proceedings of the 25th International Conference on Software Engineering*, pages 740–741. IEEE Computer Society.

[35] Colin, S. and Mariani, L. (2005). *18 Run-Time Verification*, pages 525–555. Springer Berlin Heidelberg, Berlin, Heidelberg.

[36] Cooper, M. C., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., and Werner, T. (2010). Soft arc consistency revisited. *Artif. Intell.*, 174(7-8):449–478.

[37] Czarnecki, K., Helsen, S., and Ulrich, E. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10:7 – 29.

[38] da Silva, A. R. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139 – 155.

[39] Davis, R. I. and Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44.

[40] de Roo, A. (2012). *Managing Software Complexity of Adaptive Systems*. PhD thesis, University of Twente, Netherlands. CTIT Ph.D. thesis series no. 12-217.

[41] de Roo, A., Sözer, H., Bergmans, L., and Aksit, M. (2013). Moo: An architectural framework for runtime optimization of multiple system objectives in embedded control software. *Journal of systems and software*, 86(10):2502–2519. eemcs-eprint-24550.

[42] Denil, J., Jukss, M., Verbrugge, C., and Vangheluwe, H. (2014). Search-based model optimization using model transformations. In Amyot, D., Fonseca i Casas, P., and Mussbacher, G., editors, *System Analysis and Modeling: Models and Reusability*, pages 80–95, Cham. Springer International Publishing.

[43] Dertouzos, M. L. and Mok, A. K. (1989). Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506.

[44] Dillig, I., Dillig, T., and Aiken, A. (2009). *Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers*, pages 233–247. Springer Berlin Heidelberg, Berlin, Heidelberg.

[45] Edmonds, B. (1999). Syntactic measures of complexity. Technical report, Department of Philosophy. PhD Thesis.

[46] Engels, G., Küster, J. M., Heckel, R., and Lohmann, M. (2003). Model-based veri-

fication and validation of properties. *Electronic Notes in Theoretical Computer Science*, 82(7):133 – 150. UNIGRA'03, Uniform Approaches to Graphical Process Specification Techniques (Satellite Event for ETAPS 2003).

[47] Fayad, M. and Schmidt, D. (1997). Object-oriented application frameworks. *Communications of the ACM*, 40.

[48] Fenton, N. and Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc., Boca Raton, FL, USA, 3rd edition.

[49] Fersman, E., Mokrushin, L., Pettersson, P., and Yi, W. (2006). Schedulability analysis of fixed-priority systems using timed automata. *Theor. Comput. Sci.*, 354(2):301–317.

[50] Flood, M. M. (1956). The traveling-salesman problem. *Operations Research*, 4(1):61–75.

[51] Foundation, E. (2016). Eclipse modeling framework.

[52] Frakes, W. and Terry, C. (1996). Software reuse: Metrics and models. *ACM Comput. Surv.*, 28(2):415–435.

[53] Frakes, W. B. and Kang, K. (2005). Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536.

[54] Fromherz, M. P. (2001). Constraint-based scheduling. In *American Control Conference, 2001. Proceedings of the 2001*, volume 4, pages 3231–3244. IEEE.

[55] Gabmeyer, S., Kaufmann, P., Seidl, M., Gogolla, M., and Kappel, G. (2019). A feature-based classification of formal verification techniques for software models. *Software & Systems Modeling*, 18(1):473–498.

[56] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[57] Gamrath, G., Fischer, T., Gally, T., Gleixner, A. M., Hendel, G., Koch, T., Maher, S. J., Miltenberger, M., Müller, B., Pfetsch, M. E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Vigerske, S., Weninger, D., Winkler, M., Witt, J. T., and Witzig, J. (2016). The scip optimization suite 3.2. Technical Report 15-60, ZIB, Takustr.7, 14195 Berlin.

[58] Garey, M. R., Johnson, D. S., and Sethi, R. (1976). The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129.

[59] Gautam, J. V., Prajapati, H. B., Dabhi, V. K., and Chaudhary, S. (2015). A survey on job scheduling algorithms in big data processing. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–11.

[60] Gogolla, M. and Cabot, J. (2016). Continuing a benchmark for uml and ocl design and analysis tools. In Milazzo, P., Varró, D., and Wimmer, M., editors, *Software Technologies: Applications and Foundations*, pages 289–302, Cham. Springer International Publishing.

[61] Graham, R., Lawler, E., Lenstra, J., and Kan, A. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In Hammer, P., Johnson, E., and Korte, B., editors, *Discrete Optimization II*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier.

[62] Gurobi Optimization, L. (2018). Gurobi optimizer reference manual.

[63] Harman, M. and Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43(14):833 – 839.

[64] Harman, M., Mansouri, A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45.

[65] Hebrard, E., O'Mahony, E., and O'Sullivan, B. (2010). *Constraint Programming and Combinatorial Optimisation in Numberjack*, pages 181–185. Springer Berlin Heidelberg.

[66] Heinz, S., Ku, W.-Y., and Beck, J. C. (2013). Recent improvements using constraint integer programming for resource allocation and scheduling. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 12–27. Springer.

[67] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[68] Hochwald, B. M., Marzetta, T. L., and Tarokh, V. (2004). Multiple-antenna channel hardening and its implications for rate feedback and scheduling. *IEEE Transactions on Information Theory*, 50(9):1893–1909.

[69] Holenderski, M., Bril, R. J., and Lukkien, J. J. (2012). Parallel-task scheduling on multiple resources. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 233–244. IEEE.

[70] Hong, K. and Leung, J.-T. (1988). On-line scheduling of real-time tasks. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 244–250.

[71] Hooker, J. N. (2007). Planning and scheduling by logic-based benders decomposition. *Oper. Res.*, 55(3):588–602.

[72] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.

[73] ISO 24718:2005 (2005). ISO/IEC TR 24718:2005 - Information technology - Programming languages — Guide for the use of the Ada Ravenscar Profile in high integrity systems. Standard, International Organization for Standardization, Geneva, CH.

[74] ISO 2501n:2019 (2019). ISO/IEC 2501n:2019 - Quality Model Division. Standard, International Organization for Standardization, Geneva, CH.

[75] ISO 9001:2015 (2015). ISO 9001:2015 - Quality management systems - Requirements. Standard, International Organization for Standardization, Geneva, CH.

[76] Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[77] Jacob, F., Wynne, A., Liu, Y., and Gray, J. (2014). Domain-specific languages for developing and deploying signature discovery workflows. *Computing in Science Engineering*, 16(1):52–64.

[78] Johnson, R. E. and Foote, B. (1988). Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35.

[79] Kang, K., Cohen, S., Hess, J., Nowak, W., and Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report.

[80] Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65.

[81] Kent, S. (2002). Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298. Springer-Verlag.

[82] Kim, J. H., Legay, A., Traonouez, L.-M., Acher, M., and Kang, S. (2016). A formal modeling and analysis framework for software product line of preemptive real-time systems. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1562–1565, New York, NY, USA. ACM.

[83] Koehler, J. and Ottiger, D. (2002). An ai-based approach to destination control in elevators. *AI Mag.*, 23(3):59–78.

[84] Kolisch, R. and Hartmann, S. (1999). *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*, pages 147–178. Springer US, Boston, MA.

[85] Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Software*, 12:42–50.

[86] Kumar, S., Aylor, J. H., Johnson, B. W., and Wulf, WM. A., A. A. H. M. (1996). pages 129–159. Springer US, Boston, MA.

[87] Kuyucu, B. (2018). *On the design of a user-interface for Optimal Modeling Language (OptML) Framework*. University of Twente, Drienerlolaan 5, 7522NB, Enschede, The Netherlands. Report on practical training.

[88] Lee, E. A. and Messerschmitt, D. G. (1987). Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35.

[89] Lee, K., Kang, K. C., and Lee, J. (2002). Concepts and guidelines of feature modeling for product line software engineering. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, ICSR-7, pages 62–77, London, UK, UK. Springer-Verlag.

[90] Lehoczky, J., Sha, L., and Ding, Y. (1989). The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium.*, pages 166–171.

[91] Li, Q. and Yao, C. (2003). *Real-Time Concepts for Embedded Systems*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

[92] Lin, X., Wang, Y., Xie, Q., and Pedram, M. (2015). Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment. *IEEE Transactions on Services Computing*, 8(2):175–186.

[93] Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[94] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61.

[95] Matic, S., Goraczko, M., Liu, J., Lymberopoulos, D., Priyantha, B., and Zhao, F. (2006). Resource modeling and scheduling for extensible embedded platforms. Technical report, MSR-TR-2006-176.

[96] Mernik, M., Heering, J., and M. Sloane, A. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37:316–.

[97] Merriam-Webster Online (2019). Merriam-Webster Online Dictionary.

[98] Meyer, B. (1992). Applying 'design by contract'. *Computer*, 25(10):40–51.
[99] Mili, H., Mili, F., and Mili, A. (1995). Reusing software: issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562.
[100] Miller, J. and Mukerji, J. (2003). Mda guide version 1.0.1.
[101] miroha (2015). Watch.
[102] Naur, P. and Randell, B., editors (1969). *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*.
[103] Nikovski, D. and Brand, M. (2003). Decision-theoretic group elevator scheduling. In *ICAPS*, volume 3, pages 9–13.
[104] Odell, J. (2000). Extending uml for agents.
[105] Orhan, G., Akşit, M., and Rensink, A. (2017). A formal product-line engineering approach for schedulers. In *22nd International Conference on Emerging Trends and Technologies in Convergence Solutions*, pages 15–30. The Society for Design and Process Science (SDPS).
[106] Orhan, G., Aksit, M., and Rensink, A. (2018). Designing reusable and run-time evolvable scheduling software. In *Proceedings of the 12th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2018)*, pages 339–373.
[107] P. Haubris, K. and J. Pauli, J. (2013). Improving the efficiency and effectiveness of penetration test automation. pages 387–391.
[108] Pillai, P. and Shin, K. G. (2001). Real-time dynamic voltage scaling for low-power embedded operating systems. *SIGOPS Oper. Syst. Rev.*, 35(5):89–102.
[109] Pinedo, M. L. (2010). *Scheduling: Theory, Algorithms, and Systems*. Springer New York Dordrecht Heidelberg, 4th edition.
[110] Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
[111] Prud'homme, C., Fages, J.-G., and Lorca, X. (2016). *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.
[112] Quan, G. and Hu, X. (2001). Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Design Automation Conference, 2001. Proceedings*, pages 828–833.
[113] Rusu, R. B., Blodow, N., and Beetz, M. (2009). Fast point feature histograms (fpfh) for 3d registration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3212–3217. Citeseer.
[114] Rusu, R. B. and Cousins, S. (2011). 3d is here: Point cloud library (pcl). In *2011 IEEE International Conference on Robotics and Automation*, pages 1–4.
[115] Saraf, A. P. and Slater, G. L. (2006). An efficient combinatorial optimization algorithm for optimal scheduling of aircraft arrivals at congested airports. In *2006 IEEE Aerospace Conference*, pages 11 pp.–.
[116] Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31.

[117] Seckinger, B. (1999). Synthesis of Elevator Controls Based-on Constraint-based Search. Master's thesis, Albert-Luwigs University, Freiburg, Germany.

[118] Selman, B., Kautz, H., and Cohen, B. (1995). Local search strategies for satisfiability testing. In *DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE*, pages 521–532.

[119] Seo, E., Jeong, J., Park, S., and Lee, J. (2008). Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1540–1552.

[120] Sha, L., Abdelzaher, T., årzén, K.-E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., and Mok, A. K. (2004). Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2):101–155.

[121] Shaw, M. (1984). *The Impact of Modelling and Abstraction Concerns on Modern Programming Languages*, pages 49–83. Springer New York, New York, NY.

[122] Shin, Y., Choi, K., and Sakurai, T. (2000). Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '00, pages 365–368, Piscataway, NJ, USA. IEEE Press.

[123] Silberschatz, A., Galvin, P. B., Gagne, G., and Silberschatz, A. (1998). *Operating system concepts*, volume 4. Addison-Wesley Reading.

[124] Sommerville, I. (2015). *Software Engineering*. Pearson, 10th edition.

[125] Sörensson, N. and Een, N. (2002). Minisat v1.13 - a sat solver with conflict-clause minimization. 2005. sat-2005 poster. 1 perhaps under a generous notion of "part-time", but still concurrently taking a statistics course and leading a normal life. Technical report.

[126] Sözer, H. (2009). *Architecting Fault-Tolerant Software Systems*. PhD thesis, University of Twente, Netherlands. IPA Dissertation 2009-05.

[127] Sözer, H., Tekinerdoğan, B., and Akşit, M. (2013). Optimizing decomposition of software architecture for local recovery. *Software Quality Journal*, 21(2):203–240.

[128] Sprunt, B., Sha, L., and Lehoczky, J. (1989). Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60.

[129] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

[130] Suh, N. P. (1998). Axiomatic design theory for systems. *Research in engineering design*, 10(4):189–209.

[131] Sun, Y., Gray, J., and White, J. (2015). A demonstration-based model transformation approach to automate model scalability. *Software & Systems Modeling*, 14(3):1245–1271.

[132] Tabuada, P. (2007). Event-triggered real-time scheduling of stabilizing control tasks. *IEEE Transactions on Automatic Control*, 52(9):1680–1685.

[133] te Brinke, S., Malakuti Khah Olun Abadi, S., Bockisch, C., Bergmans, L., Akşit, M., and Katz, S. (2014). A tool-supported approach for modular design of energy-aware software. In *SAC '14*, pages 1206–1212, United States. Association for Computing Machinery.

[134] Tindell, K. and Hansson, H. (1995). Real time systems and fixed priority scheduling. *Technical Report Tech. rept*.

[135] Urli, S., Blay-Fornarino, M., and Collet, P. (2014). Handling complex configurations in software product lines: A tooled approach. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 112–121, New York, NY, USA. ACM.

[136] Wang, Y. and Saksena, M. (1999). Scheduling fixed-priority tasks with preemption threshold. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*, pages 328–335. IEEE.

[137] Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading, MA, 2 edition.

[138] Xiao, J.-J., Cui, S., Luo, Z.-Q., and Goldsmith, A. J. (2006). Power scheduling of universal decentralized estimation in sensor networks. *IEEE Transactions on Signal Processing*, 54(2):413–422.

[139] Zhao, W., Ramamritham, K., and Stankovic, J. A. (1987). Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, C-36(8):949–960.

APPENDIX A

Appendix

A.1 Feature Model

The tool designer instantiates the feature meta model shown in Figure 6.4. The instantiated model is shown in Figure A.1. The number of systems that can be configured from this model is computed as 1288.
In this section, we present a feature model for the example registration system. This model conforms to the feature meta model presented in Figure 6.4. This model is used in the scenario implementations presented in Section 6.5.3. By definition, in our framework, each feature corresponds to a class in the class model. The descriptions of the names of the adopted features are presented in Section 6.3.2. To avoid repetition, the features are not described here again. The number of systems that can be configured from this model is computed as 1288.

A.2 Platform Model

The meta model shown in Figure 6.5 is instantiated according to the registration system given in Section 6.1. To this aim, we define a platform model depicted in Figure A.2, in which System has two composite resources, each of which consists of one active and one passive resource. Each terminal resource has one state except the active resource of second composite resource. There are three resource types: processing unit (ACTIVE), memory (PASSIVE) and computation node (COMPOSITE). Unlike the other terminal resources, the

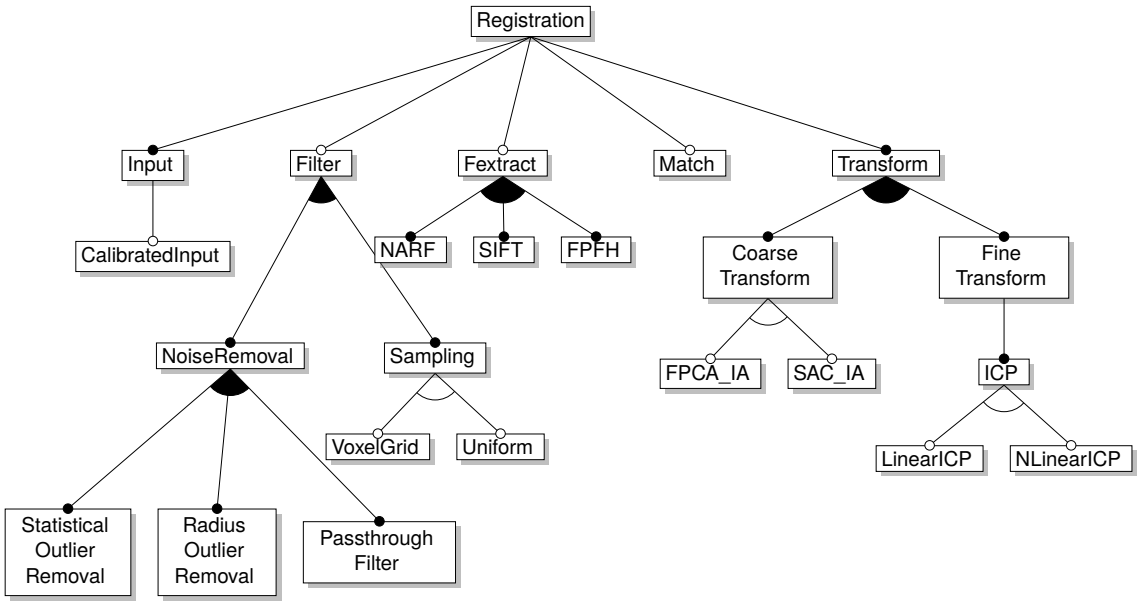


Figure A.1: A feature model of the registration system derived from the feature meta model.

processing unit of the second node has two states: half- and full-speed running modes. Each processing unit has a unit capacity. The memory components have 512- and 256-unit capacity on the first and second computation nodes, respectively.

A.3 Process Model

In Figures A.3 and A.4, the process model is presented which is created from the process meta model presented in Figure 6.6. Since the process model is rather large, in the figure, we only show elaborate a selected set of processes. As explained previously, the input data is acquired by utilizing class Input. For this purpose, the operation getData is defined. In some cases, the accuracy of acquired data may be crucial. To this aim, class CalibratedInput is used instead of using its superclass. The operation calibrate aims to increase the quality of the data if it is called before calling getData.
To reduce the size of the input data, class Filter is defined. The operation setInputData is used to set the interested data. To set the filtering-related parameters, the operation setFilteringParams is defined. The operation filter is finally called to gather the filtered data. Class Filter is specialized further into classes NoiseRemoval and Sampling, which are responsible for eliminating the erroneous data and getting a part of the data to reduce the size, respectively. Classes inheriting from these classes, such as StatisticalOutlierRemoval or VoxelGrid, represent to the different algorithmic approaches.
Class FExtract is responsible for computing predefined key features of the data to reason about geometric properties. Similarly, the operation setInputData of class FExtract is utilized to set the interested data, and the operation setFExtractingParams is responsible for

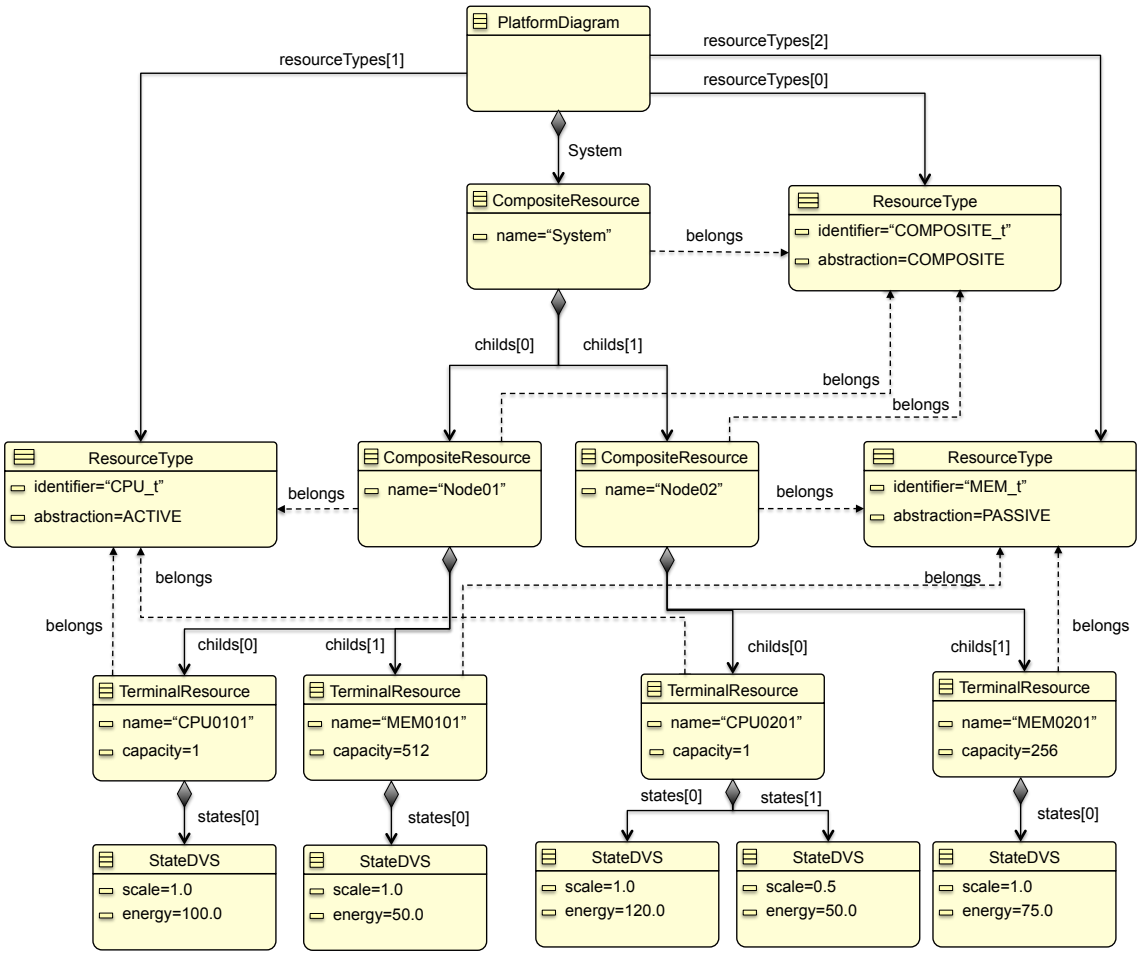


Figure A.2: The platform model derived from the meta model.

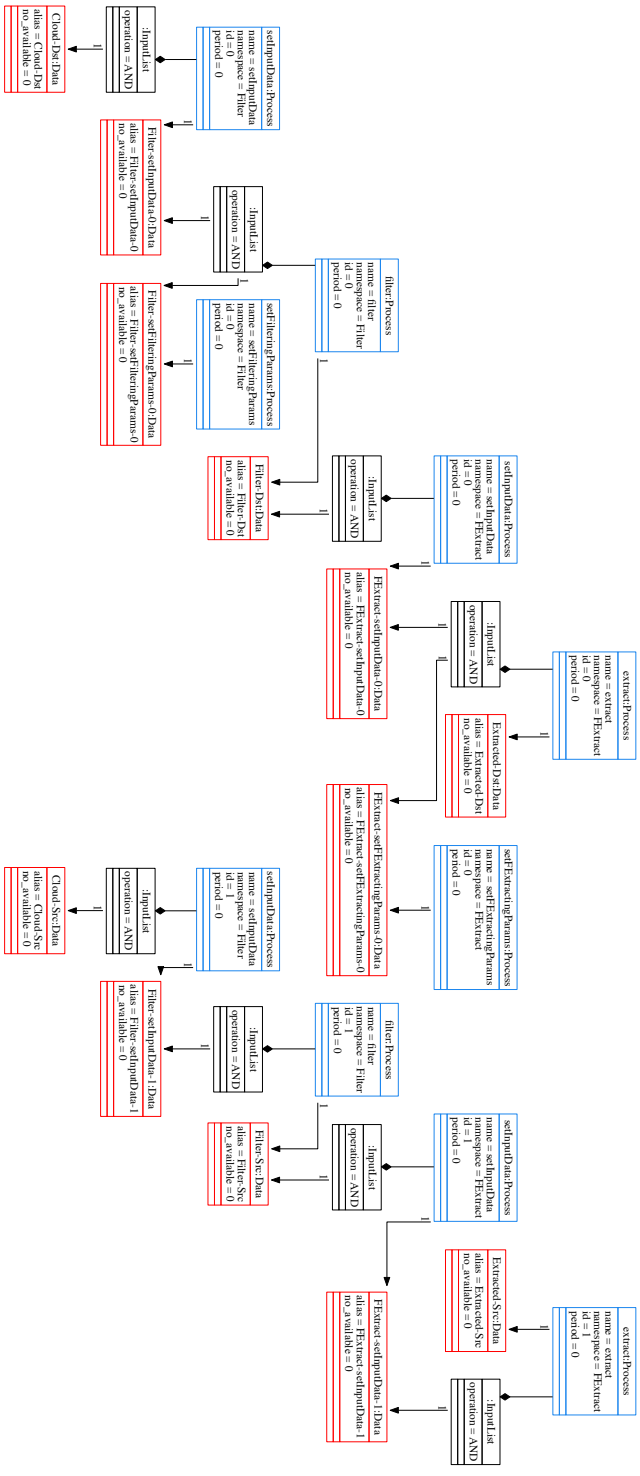


Figure A.3: The partial process model including the functions of classes *Input*, *Filter* and *FExtract*

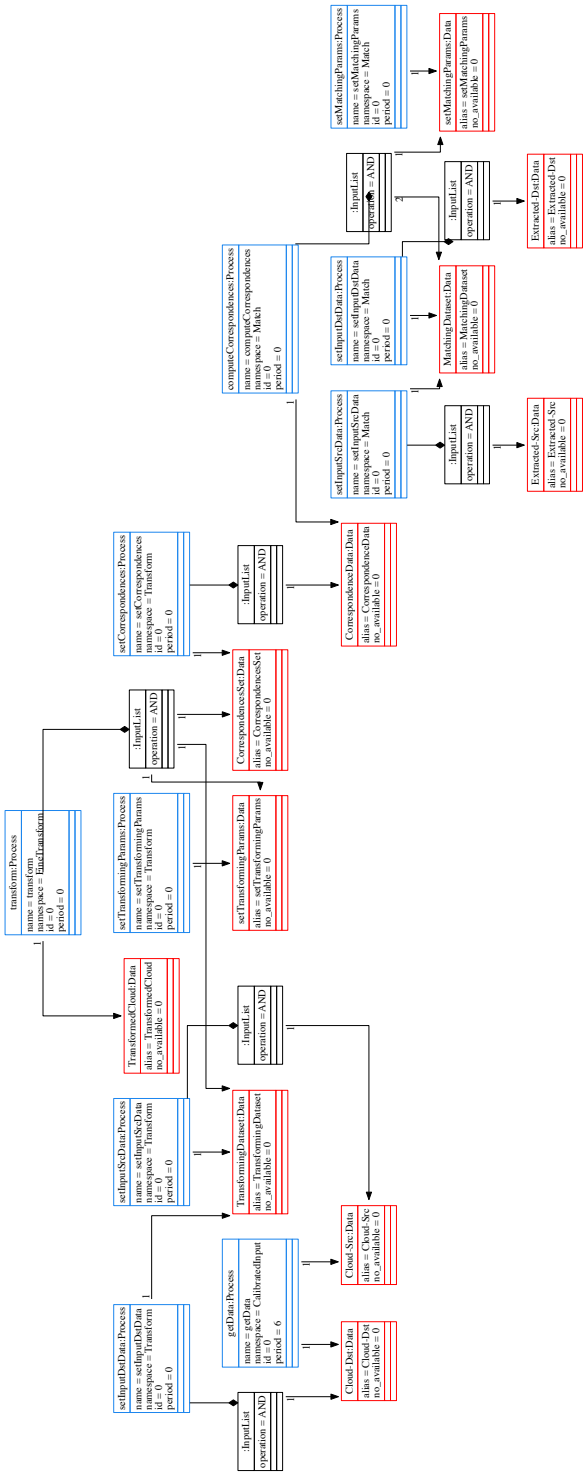


Figure A.4: The partial process model including the functions of classes *Match* and *Transform*

adjust the settings of the class. To gather a combination of computed key features and the given data, the operations extract is called. Classes NARF, SIFT and FPFH represent the definitions for computing various predefined features.
To relate two different input to each other, class Match is defined. One of the important factors that is supposed to be decided is the direction of the relation from *source* to destination data, which are set with the operations setInputSrcData and setInputDstData, respectively. The relevant parameters are adjusted using the operation setMatchingParams. The operation computeCorrespondences is responsible for obtaining the related data in pairs.
To transform the source data into the coordinate frame of the destination data, class Transform is utilized. Similar to class Match, it has two operations setInputSrcData and setInputDstData for setting source and destination data. The operation setTransformingParams is used to configure the parameters. To increase the accuracy of a *transformation* process, the computed correspondences can be given using setCorrespondences. The operation getTransformationMatrix gives the transformation matrix including rotation and translation. This class is specialized into two classes CoarseTransform and FineTransform. In the literature [113], the coarse transform is known as the pre-process *initial alignment* to pre-transforming the data to increase the accuracy of the process. In some cases, there may be no need to further transformation after initial alignment, but mostly to gather accurate results, the process *fine transformation* is applied. Classes CoarseTransform and FineTransform have two different operations, align and transform, respectively, to perform their own computation and set the transformation matrix, accordingly. Class Coarse Transform is specialized further into two different approaches FPCA_IA and SAC_IA. Class ICP, called as *iterative closest point*, is one of the commonly known algorithms. Classes LinearICP and NLinearICP correspond to the different versions of ICP algorithms.

A.4 Instantiation of the Value Meta Model for Energy Consumption and Computation Accuracy

In this section we will illustrate how the value meta model that is presented in Figure 6.7 is instantiated for the two quality attributes energy reduction and precision.
The instantiation parameters are shown in Table A.1, where the rows and columns represent the features defined in the feature model and the values for the quality attributes, respectively. In the figure, each cell includes the positive and negative contributions of the feature to the process configuration. Since the operations are not specifically defined in the value models, the contributions of operations are the same with the contributions of the owner class defined in the class model. The higher values for energy model and precision model mean that the corresponding feature increases the energy consumption and decreases the computation accuracy, respectively.

	Energy Value	**Precision Value**
Registration	0 / 0	0 / 0
Input	0 / 0	0 / 0
CalibratedInput	50 / 30	20 / 50
Filter	5 / 0	5 / 10
NoiseRemoval	10 / 0	10 / 20
PassthroughFilter	15 / 0	15 / 30
RadiusOutlierRemoval	15 / 0	15 / 30
StatisticalOutlierRemoval	15 / 0	15 / 30
Sampling	10 / 0	10 / 20
VoxelGrid	15 / 0	15 / 30
Uniform	15 / 0	15 / 30
FExtract	10 / 0	5 / 10
SIFT	30 / 0	10 / 30
NARF	30 / 0	10 / 30
FPFH	30 / 0	10 / 30
Match	20 / 0	15 / 50
Transform	5 / 0	5 / 5
CoarseTransform	10 / 0	10 / 30
FPCA_IA	12 / 0	5 / 30
SAC_IA	12 / 0	5 / 30
FineTransform	15 / 0	2 / 90
ICP	15 / 0	3 / 90
LinearICP	20 / 0	1 / 110
NLinearICP	20 / 0	1 / 110

Table A.1: Values of energy and precision models for each feature.

Samenvatting

Planningsprocessen ("scheduling"-processen) worden toegepast in een grote categorie gebieden, zoals taakplanning in besturingssystemen, planning van faciliteiten op luchthavens, planning van assemblagelijnen in productie, resourceplanning in projectbeheer, planning in openbaar vervoer en planning in "cyber-physical" systemen.
Over het algemeen is het niet triviaal om planningsproblemen effectief en efficiënt op te lossen. Ontwerp en realisatie van planning-software is kostbaar en tijdrovend. Dit proefschrift levert drie bijdragen om hierin verbetering te brengen:
Ten eerste hebben we een feature-georiënteerde Software Product Line Engineering (SPLE) -aanpak gevolgd. Indien correct toegepast, levert de SPLE-aanpak goedkopere oplossingen voor het geval dat een productfamilie wordt ontwikkeld in plaats van één product. Bedrijven die planningssystemen ontwikkelen implementeren in het algemeen productfamilies. Na een uitgebreide domeinanalyse van de theorie hebben we de stabiele en variabele features gedefinieerd. Door de juiste set features te kiezen en te configureren, kunnen de ontwerpers efficiënt de planningssystemen definiëren.
Als een tweede bijdrage , hebben we een applicatie-framework genaamd First Scheduling Framework (FSF), ontworpen en geïmplementeerd, om de abstracte definities om te zetten in uitvoerbare programma's. We beschouwen herbruikbaarheid en dynamisch-uitbreidbaarheid als twee kwaliteitskenmerken van het framework. Hiertoe worden generieke constraint-oplossers gebruikt om de vooraf gedefinieerde "plannings-constraints" te vertalen naar de constraint-taal van de overeenkomstige oplosser en in te zetten om het vertaalde probleem op te lossen. We hebben onze implementatie uitgebreid met MDE-technieken (Model-Driven Engineering) zodat feature-gerichte modellen gemakkelijk kunnen worden omgezet naar de abstracties van het applicatie-framework.
Als derde bijdrage hebben we onze feature-gerichte aanpak uitgebreid tot een algemeen model-optimalisatie-framework. Planningssystemen kunnen worden beïnvloed door vele contextuele factoren, zoals de wens naar lager energieverbruik, meer precisie en het omgaan met bepaalde hardware-beperkingen. Vanwege zulke factoren kan het zeer moeilijk zijn om een optimaal system te definiëren. Voor dit doel stellen we het OptML Framework voor, dat verschillende modellen van contextuele factoren accepteert (in de ECORE MDE-omgeving) en daaruitoptimale modellen berekent die voldoen aan door de gebruiker gedefinieerde eisen.