Domain-Specific Modelling for Coordination Engineering

Dipl.-Inform. Stefan Gudenkauf

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2012

Kiel Computer Science Series (KCSS) 2012/2 v1.0 dated 2012-10-07

ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

Published by the Department of Computer Science, Christian-Albrechts-Universität zu Kiel

Software Engineering Group

Please cite as:

▻ Stefan Gudenkauf. *Domain-Specific Modelling for Coordination Engineering* Number 2012/2 in Kiel Computer Science Series. Department of Computer Science, 2012. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

```
@book{Gudenkauf2012,
  author = {Stefan Gudenkauf},
  title = {Domain-Specific Modelling for Coordination Engineering},
  publisher = {Department of Computer Science, CAU Kiel},
  year = {2012},
  month = {october},
  number = {2012-2},
  isbn = {9783848228485},
  series = {Kiel Computer Science Series},
  note = {Dissertation, Faculty of Engineering,
  Christian-Albrechts-Universit\"at zu Kiel}
}
```


Herstellung und Verlag: Books on Demand GmbH, Norderstedt

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at the Christian-Albrechts-Universität zu Kiel. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Wilhelm Hasselbring Christian-Albrechts-Universität Kiel
2. Gutachter:	Prof. Dr. Michael Hanus Christian-Albrechts-Universität Kiel

Datum der mündlichen Prüfung: 6. Juli 2012

Preface

Through the advent of multi-core processors in the consumer market, parallel systems became a commodity. The semiconductor industry today is relying on adding cores, introducing hyper-threading, and putting several processors on the motherboard to increase the performance, since physical limitations impede further performance gains based on increasing clock speed. With the spread of multi-core processing units on server and desktop systems, laptops and meanwhile also tablets and smart phones, parallel programming moves from a niche for specialists toward mainstream application programming. This will affect most application programmers. A fundamental question is how to exploit these emerging hardware architectures for software applications.

As a great challenge, we need to make the complexity of parallel programming controllable for application programmers. Parallel programming languages intend to offer the programmer features for explicit parallel programming, while parallelising compilers try to detect implicit concurrency in sequential programs for parallel execution. Automatic parallelisation only works well for very specific domains such as loops in numerical simulations.

In this thesis, Stefan Gudenkauf proposes a new approach to engineering parallel programs, which combines model-driven software engineering with the space-based parallel programming paradigm. This combinations aims at providing both an easy-to-use and an efficient approach to parallel software engineering. Based on identified requirements on application-level parallelism, Stefan designed the SCOPE coordination model (Space-Coordinated Processes) for coarse-grained choreography of parallel processes, while the fine-grained parallelism within these processes is specified with a BPMN-based orchestration (Business Process Modeling Notation). This combination constitutes his new coordination engineering method.

The technical design and the implementation re-uses and integrates

many software components and frameworks from various domains and sources. The re-use of such powerful components and frameworks relieves from building the respective functions, but imposes the challenge to check their fitness for purpose and to integrate diverse architectural styles into a coherent whole. The tool prototype and experiments designed and realised in this thesis constitutes a remarkable engineering achievement. Besides the conceptual and the technical design, this engineering thesis provides an extensive experimental evaluation based on his PROCOL library (Process Coordination Library).

If you are interested in parallel application programming, this is a recommended reading for you.

Wilhelm Hasselbring
Kiel, August 2012

Contents

Chapter 1

Introduction

> The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency
>
> Herb Sutter, 2005

In this chapter we motivate the challenges that led to this thesis. We also present an overview of the overall approach and the resulting contribution. At the end of the chapter we describe the structure of this work, discuss preliminary work, and summarise the previous sections.

1.1 In the Middle of a Major Transition

In 2005, Sutter [2005] published his influential article "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". He argued that concurrency would have a similar fundamental impact on software development as Object-Orientation (OO) had in the 1990s: Software industry would soon have to react to the beginning transition of the hardware industry to establish multiple, and simpler cores as the new performance driver for their future processors. This prospect is grounded by the fact that physical limitations in the design of single core processors make it impossible to further dissipate the generated heat, to limit power consumption, and to prevent current leakages [Sutter, 2005]. Today, industry has indeed turned towards multiple cores per processor as their new performance

drivers. Multi-core processors are a commodity and many-cores are on the verge of consumer market introduction [Marowka, 2007; AMD, 2011; Intel Corporation, 2011b,a].

Marowka [2010] describes this transition as a period of revolution, according to the observations of Kuhn [1996] on the structure of scientific revolutions: Marowka denotes the first three decades of the sequential computing era as the *preparadigm* phase, when there was no consensus on the feasibility and practicality of parallel computing.[1] As an example, he notes that the computer pioneers John von Neumann, Presper Eckert, John Mauchly, and Herman Goldstine claimed that serial operation is preferred over parallel operation in electronic machines, provided that components are fast enough [Marowka, 2010, p. 73]. The second phase, *normal science*, is identified by Marowka to have started two decades ago, when parallel machines and programming paradigms were firstly explored. This phase is characterised by a niche scientific community on High-Performance Computing (HPC), a small number of specialised professional HPC programmers, and very expensive and maintenance-intensive parallel machines. The third phase, *revolutionary science*, started in 2005, when the first affordable multi-core processors were developed. Low-cost parallel machines entered the consumer market and the need arose to exploit concurrency in applications of all kinds and domains.

1.2 Challenges of Concurrent Programming

The shift from sequential to parallel machines marks the beginning of a revolution that imposes several challenges for the engineering of concurrent software systems: the need for a generalised abstraction over concurrent system behaviour, continuity with prevalent technologies, and the dissemination of patterns and practice.

Performance is the main reason for the transition towards multi- and many-core processors. Obtaining the utmost performance speed-ups requires extensive expertise to understand the application as well as the

[1] According to Buyya [2000], the sequential computing era is set from the early 1940s to circa 2010, and the parallel computing era is set from the 1950s to the 2030s.

characteristics of the parallel machine, and extensive effort to find the best solution to exploit them [Marowka, 2010, p. 79]. Portability, on the other side, is the primary concern for development cost reduction. It represents the degree to which applications can be moved between different machines and environments without yielding different results [Marowka, 2010, p. 78]. Its prerequisite is a generalised abstraction between the application and the target environments. Unfortunately, the diversity of parallel machines and programming models makes it harder to find an adequate abstraction than it is for the less diverse sequential ones. However, there are clear indications that considering concurrency on higher levels of abstraction enables software engineers to produce software systems that yield performance speed-ups that are more significant than speed-ups gained from lower abstraction levels [Pankratius et al., 2009; Wegner, 1997].

Current programming languages like Java, C/C++, and C# support concurrency by lock-based programming based on the threading model as the predominant model for general-purpose parallel programming. This model is highly non-deterministic and requires software developers to cut away unwanted non-determinism by means of synchronisation [Lee, 2006]. Libraries such as OpenMP [Ope, 2011] and Message Passing Interface (MPI) [MPI, 2009] can be regarded as a loophole, but they often turn out to be non-portable and platform-specific. As these languages and libraries continue to be used, concurrent software engineering requires to be integrated with them. The need for higher-level abstractions requires the programming languages themselves to incorporate these abstractions as first class language aspects. As long as this is not the case, there is the clear need for means to enforce the compliance of lock-based program code to higher-level concurrency models in a standardised way.

After years of sequential programming practice, the majority of software engineers lack experience in the application of existing parallel software design knowledge [Ortega-Arjona, 2010, p. xiii]. Patterns and pattern languages are an acknowledged way of communicating and preserving such knowledge. Existing pattern languages and pattern-based methods for concurrent software design must be disseminated effectively to software engineers [Mattson et al., 2004; Ortega-Arjona, 2010] in a way that is accepted by them.

1.3 Approach

The goal of this thesis is to improve the quality of concurrent software systems from the software engineer's viewpoint. To attain this goal, we address the above mentioned challenges by finding answers for the following four questions, see Figure 1.1:

1. What is an adequate high-level abstraction on which concurrency should be considered by software engineers?

2. How can the compliance of low level concurrency code to a high-level concurrency abstraction be enforced?

3. How can the acceptance of such an abstraction be fostered?

4. How can such an abstraction be aligned with the existing body of patterns and practices?

We argue that the first question can be addressed by coordination modelling [Gelernter and Carriero, 1992; Papadopoulos and Arbab, 1998], since it provides a high-level abstraction over concurrent systems that defines the interaction of active and independent entities and the coordination laws that specify how the entities coordinate themselves through the given coordination media [Ciancarini, 1996]. Based on the identified requirements for coordination models that address the concurrent software systems engineering challenges, we developed Space-Coordinated Processes (SCOPE)[2], a coordination model that is based on Space-Based Systems (SBS) and distinguishes between the choreography of multiple concurrent processes and the orchestration of fine-grained activities within a single process. To show the eligibility of the model, we developed the Java process coordination framework Process Coordination Library (PROCOL)[3] and a Domain-Specific Language (DSL) workbench as concrete embodiments of SCOPE, and performed several experiments to investigate the operational behaviour of the implementations.

[2] http://scope-dsl.sourceforge.net
[3] http://procol.sourceforge.net/

We address the second question by applying techniques of Model-Driven Software Development (MDSD). While a coordination model provides a high-level abstraction over concurrent systems, MDSD considers such models as first-class artefacts and makes them amenable to explicit modelling by DSLs. Model-to-Text (M2T) transformations enforce the compliance of low level concurrency code to higher-level coordination models.

To address the third question, acceptance, we see it as beneficial to adopt wide-spread standards for the dissemination of coordination modelling. For example, although primarily oriented towards the business domain, the generic nature, the prominence, and the tool support of the widely adopted Business Process Model and Notation (BPMN) [Axway et al., 2010] makes it a promising vehicle for rapid dissemination. Language piggy-backing guarantees the conformity of a coordination model to the BPMN. Consequently, our SCOPE coordination model conforms to the BPMN and uses it as a graphical outline on the behavioural architecture of concurrent software systems. Model-to-Model (M2M) transformations can guarantee this conformance.

The fourth question can be addressed by providing a method to apply model-driven coordination modelling in a way that integrates well with existing patterns and practices. We present Coordination Engineering as such a method. It integrates a refinement of the coordination design phase of the Parallel Software Design method of Ortega-Arjona [2010].

1.4 Contribution

The scientific contribution of this work can be summarised as follows:

SCOPE is a conceptual coordination model that introduces the SBS-based choreography of independent processes, which internally orchestrate fine-grained workflow activities.

PROCOL is a library-based embodiment of the SCOPE coordination model as a proof of feasibility and to ease the development of concurrent space-based programs in Java. It can also be considered as an *internal* DSL for SCOPE.

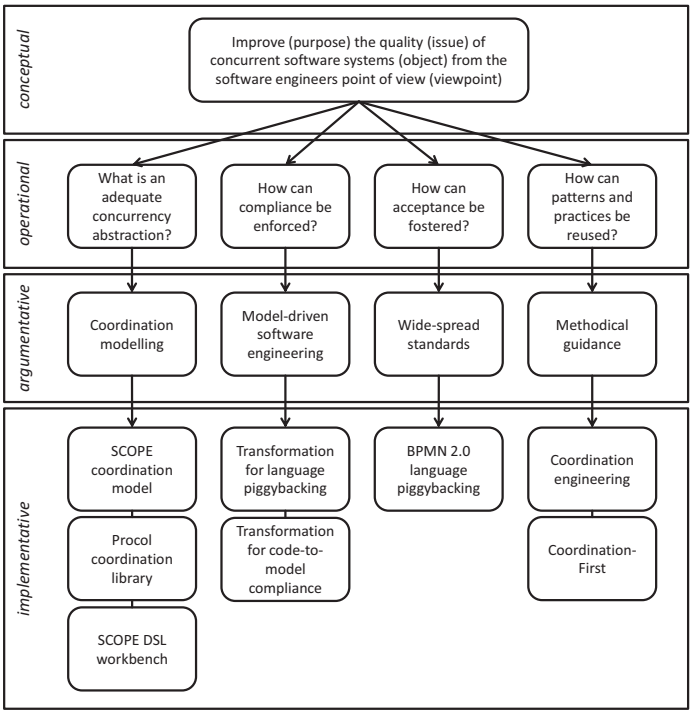

Figure 1.1. Overview of the contents of the thesis.

SCOPE DSL and Workbench is an embodiment of the SCOPE coordination model in an *external* textual DSL that conforms to the BPMN, and a workbench for the SCOPE DSL that is based on the Xtext language framework.

Coordination Engineering is a method that regards the coordination model of a concurrent software system as the first development artefact in the software development process. The method targets concurrent software development as an architecture-centric Model-Driven System (MDS) that emphasises the establishment of concurrent software product lines and families.

1.5 Preliminary Work

This section briefly discusses work we conducted preceding to this thesis, but parts of which are reused or relevant in the following chapters. We summarise the contents and the related goals. Further information on the works can be obtained from the corresponding publications.

In the position paper "A Coordination-Based Model-Driven Method for Parallel Application Development" we discussed the need for higher abstractions in parallel software development [Gudenkauf, 2010], motivated by the inappropriateness of the non-deterministic threading model, the lack of parallel programming experience, and the supposed impact of higher-level abstractions on application performance. We proposed a model-driven method that regards the coordination model of parallel programs as the first development artefact to satisfy this need, and envisioned an adequate coordination language. The proposed method can be regarded as a primitive precursor of the Coordination Engineering method, see Chapter 5.

The conference paper "Space-Based Multi-Core Programming in Java" investigates the use of SBS for multi-core programming in Java [Gudenkauf and Hasselbring, 2011]. Firstly, we argued about appealing properties of space-based programming for multi-core machines and presented the PROCOL programming model that introduces the space-based choreography of active components, which internally orchestrate fine-grained activities. This model is based on extensions to the third-party tuple space framework LighTS [Balzarotti et al., 2007]. Secondly, we evaluated the performance of several tuple space data structures for PROCOL and compared the performance of two equivalent Mandelbrot applications – one implemented with the standard Java thread model, one with our PROCOL programming model. Both, the data structures and the Mandelbrot applications, were evaluated on several machines. The conclusions drawn from these experiments are that, in principle, scalable tuple space implementations can be provided for multi-core architectures, and that the performance overhead that the PROCOL programming model imposes is at least for the considered application a reasonable trade-off for the ease of programming that the model provides. The PROCOL programming model represents an internal DSL for the SCOPE coordination model in the form of a programming library. The results of the

paper are mainly incorporated in Section 8.1, Section 9.1, and Section 10.2

In the paper "Domain-Specific Modelling for Coordination Engineering with SCOPE" we introduced the Coordination-First approach [Gudenkauf et al., 2012]. The separation of collaboration from process definition, and the application of model-driven techniques are essential parts of this approach. To support the approach, we developed a prototype of the space-based coordination language SCOPE and an appropriate workbench prototype. Both, the language and the workbench, conform to the BPMN specification and are interoperable with BPMN-conformant tools. Additionally, SCOPE models can be validated on domain-level rather than only on level of the underlying BPMN, and support iterative software development processes. The paper can be regarded as an in-progress report of this thesis. Topics that are not addressed by the paper include the semantic clearness of SCOPE, a transformational mapping of SCOPE to the PROCOL programming library, and experiments on the application of the approach by considering practical application scenarios. The results of the paper are mainly incorporated in Chapter 6, Section 8.2, and Section 9.2.

1.6 Structure of this Work

This thesis consists of the following four parts:

- Part I contains the foundations and forms the general setting of the thesis. We introduce definitions of various terms that are fundamental for the understanding of the following parts.

  - Chapter 2 presents an overview of concurrent programming. It introduces general terms and definitions and different models of concurrent programming, describes the concurrent programming abstraction, the difference between processes and threads, different factors that affect the performance of concurrent programs, and presents different concurrent programming approaches and techniques.
  - Chapter 3 presents coordination modelling as a solution for the design and implementation of large and complex parallel systems. The

chapter introduces coordination models, along with a categorisation, and presents the SBS model and the BPMN as two interesting examples.

- ▷ Chapter 4 introduces Model-Driven Software Development (MDSD). Firstly, it discusses different types of model transformations. Secondly it introduces the notion of Model-Driven System (MDS) and discusses a variant of MDSD that is concerned with the architecture of software systems. Thirdly, it presents Domain-Specific Languages (DSLs) and their importance for MDSD.

▷ Part II represents the conceptual main part of the thesis. It performs a domain analysis, defines the conceptual SCOPE coordination model, and defines Coordination Engineering as a promising method for concurrent software development that emphasises the establishment of software product lines and families. Furthermore, the requirements and the component design for a software infrastructure for SCOPE are presented.

- ▷ Chapter 5 describes the Coordination Engineering method. We present an overview of the method, the relevant artefacts, its role model, method application processes, and the placement of the method in several contexts.
- ▷ Chapter 6 presents the SCOPE coordination model. First, we describe the requirements that apply to the coordination model. Second, we describe SCOPE as a combination of SBS and the BPMN specification. Third, we identify a subset of the BPMN that can express SCOPE models, and discuss the operational semantics of the subset. Finally, we conclude the chapter with a placement of the SCOPE coordination model in several contexts.
- ▷ Chapter 7 describes the requirement for a software infrastructure for SCOPE. We describe the requirements for a SCOPE coordination library and the requirements for a SCOPE DSL workbench.
- ▷ Chapter 8 describes the conceptual design for a SCOPE software infrastructure. We describe the design of a coordination programming library for SCOPE, as well as the design of an appropriate coordination workbench that considers SCOPE as an external DSL.

- Part III contains the evaluation of the SCOPE coordination model and the Coordination Engineering method.
  - Chapter 9 discusses the implementation of a coordination library for SCOPE that is named PROCOL, and the SCOPE DSL and a corresponding coordination workbench. At the end of the chapter, both implementations are categorised.
  - Chapter 10 documents several experiments that we conducted to illustrate the feasibility of our approach. First, we discuss the relevant performance metrics. Second, we justify the SCOPE coordination model by experiments we conducted with the PROCOL coordination library. Third, we illustrate the Coordination Engineering approach by two experiments using the SCOPE coordination workbench: an experiment on the visualisation of the Mandelbrot set on the complex plane, and one on Point-Feature Label Placement (PFLP) in Enterprise Architecture Visualisation (EAV). The chapter is concluded with a discussion of the threats to validity.
  - Chapter 11 describes the state of art in relevant research areas forming the context of this thesis, and distinguishes the thesis from related work in these areas.
- Part IV provides an outlook and presents the conclusion.
  - Chapter 12 summarises the essential findings and the contribution of this thesis.
  - Chapter 13 presents an outlook on possible future extensions and refinements.

1.7 Summary

Figure 1.2 summarises the previous sections: Physical limitations in the design of single core processors forced the hardware industry to establish multi-core and many-core processors as the new performance drivers. This imposes three challenges to software engineering: Portability and human

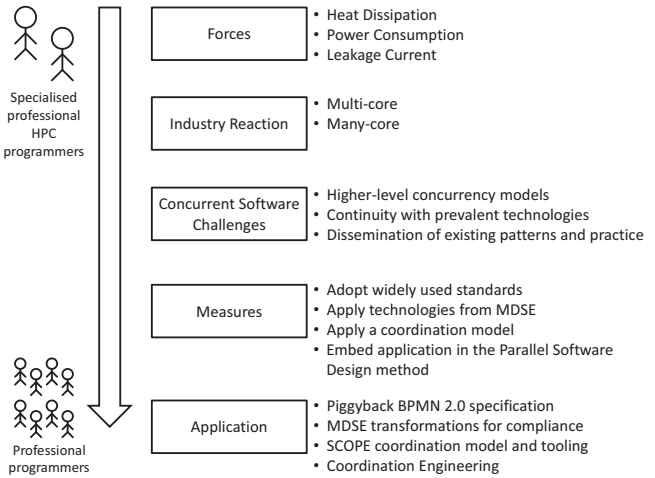

Figure 1.2. The contents of the thesis in the context of the concurrent software engineering revolution.

perception require providing standardised higher-level concurrency models, continuity with prevalent technologies that are widely used must be guaranteed, and existing patterns and practice of parallel software design must be disseminated to software engineers of all kinds and domains. As a solution, we propose to combine coordination models with techniques from MDSD. We propose to use the BPMN specification as a vehicle for dissemination, model transformations as a means to enforce compliance, a combination of BPMN and SBS as a coordination model, and a concrete method that is embedded in the Parallel Software Design method of Ortega-Arjona [2010] as application guidance.

In the following, we provide the SCOPE coordination model, the Coordination Engineering method, and the embodiment of SCOPE in a coordination library and a DSL workbench as a proof of feasibility. The results are enhanced with proto-benchmarks and experiments on applications for massive parallelism and PFLP. Beforehand, we discuss the relevant setting and foundations.

Part I

Setting and Foundations

Chapter 2

Concurrent Programming for Parallel Systems

> The challenge in concurrent programming comes from the need to synchronize the execution of different processes and to enable them to communicate
>
> Mordechai Ben-Ari, 2006

In contrast to sequential processing, parallel processing offers increased performance for the solution of many problems. To exploit the potential performance of a parallel system, one has to formulate the solution of a problem in a way that regards parts of the problem to be computed independently. The development of such solutions is often highly dependent on the underlying programming environment. In the following, we present an overview of the concepts of concurrent programming, starting with the basic definitions for what we consider a (parallel) system and a programming environment.

System A system is a collection of components organised to accomplish a specific function or set of functions [IEEE Architecture Working Group, 2000].

Parallel system A parallel system is a system, where the emphasis is put on accomplishing the specific function or the set

of functions collaboratively and in a coordinated manner, cf. [Rauber and Rünger, 2007, p. 17].

Platform/programming environment A platform/programming environment is a system that represents the entirety of hardware and software, including operating system, programming languages, compilers, runtime libraries etc., that is available to a programmer (cf. *parallele[s] Rechnersystem* [Rauber and Rünger, 2007, p. 113]).

2.1 Models of Parallel Systems

To use hardware-independent programming methods and principles, systems are classified and described in models. These models allow to describe classes of programming environments on a certain level of abstraction, thus allowing to develop and analyse programs for classes of systems instead of individual systems. The *von-Neumann* model, or the Single Instruction Single Data (SISD) model, represents the basic abstraction of sequential systems [Rauber and Rünger, 2007]. Further models for sequential programming are distinguished only by their degree of abstraction from the von-Neumann model. In contrast, there are several models for parallel systems. Regarding their level of abstraction, these models can be differentiated in machine models, architectural models, computational models, and programming models [Rauber and Rünger, 2007, pp. 114-117].

A *machine model* is a model of a programming environment that is close to the hardware and the operating system, and that typically describes the registers and data paths of a processing element, or the I/O-Buffers and their connections of a node in an interconnection network. Typical examples of languages that base upon machine models are assembler languages.

An *architectural model* is an abstraction over machine models that describes the organisation of processing elements and their interconnections, and the resulting data and control flows. Architectural models can be further categorised by applying Flynn's Classification, see [Flynn, 1972; Rauber and Rünger, 2007]. Typical aspects of interest in architectural models are the topology of the interconnection network of processing elements,

the memory organization (shared or distributed memory), the execution mode of the processing elements (synchronous or asynchronous), and the execution mode of instructions (Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), or Multiple Instruction Multiple Data (MIMD)).

A *computational model* is an abstraction of an architectural model that allows to design algorithms for the respective parallel system whose costs, most prominently the execution time on the system, can be measured. Therefore, computational models consist of an operational part that describes the operations that can be executed, and a corresponding analytical part that specifies the corresponding costs of the operations.[1]

A *programming model* is an abstraction of a computational model that represents a view on a system from the viewpoint of a programmer, where the view is based upon a certain programming environment. With respect to different programming environments or the ambiguity of a single programming environment, there may be different programming models for a single system.

We introduce the following definitions to foster a thorough understanding:

> ***Instruction*** An instruction is a discrete statement. More complex operations are built up by combining these statements which are executed either sequentially or directed by control flow instructions.
>
> ***Processing Element*** A Processing Element (PE) is "a hardware component that executes a stream of instructions" [Mattson et al., 2004, p. 17].
>
> ***Unit of Execution*** A Unit of Execution (UE) is "a generic term for one of a collection of possibly [parallel] executing entities" [Mattson et al., 2004, p. 16]. For example, these can be processes or threads from a technical viewpoint.

[1] For example, the Random Access Machine (RAM) model is the corresponding computational model for the *von-Neumann* model.

Viewpoint A viewpoint is "[a] pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis" [IEEE Architecture Working Group, 2000].

View A view is "a representation of a whole system from the perspective of a related set of concerns" [IEEE Architecture Working Group, 2000]. It describes only those concerns that are defined by its viewpoint definition.

2.2 The Concurrent Programming Abstraction

According to Ben-Ari [2006], there are three concrete abstractions from the parallel machine that are used by software engineers: instruction sets, programming languages, and systems and libraries. The Concurrent Programming Abstraction model of Ben-Ari provides a theoretical abstraction over these that is defined as the interleaved execution of atomic statements [Ben-Ari, 2006, p. 39]. It allows to model the behaviour of a wide range of concrete systems, for example, multitasking systems, multiprocessor computers, and distributed systems. A fundamental distinction that is made is to differentiate concurrency from parallel processing. A concurrent program is a set of processes that *can* execute in parallel, whereas the term parallel describes the actual overlapping of the processes during their execution on several processors. Concurrency can thus be regarded as the precondition for parallel processing, or as potential parallelism.

Program (concurrent) A concurrent program is "a finite set of (sequential) processes." [Ben-Ari, 2006, p. 8]. It operates "by executing a sequence of the atomic statements obtained via arbitrarily interleaving the atomic statements from the processes" [Ben-Ari, 2006, pp. 8-9].

Process (theoretical) A process consists of "a finite [ordered] set of atomic statements" [Ben-Ari, 2006, p. 8].

Statement (atomic) An atomic statement is a statement that "is executed to completion without the possibility of interleaving statements from another process" [Ben-Ari, 2006, p. 19].

Scenario A scenario, also denoted as *computation*, is a concrete "execution sequence that can occur as a result of the interleaving" within program execution [Ben-Ari, 2006, p. 9].[2]

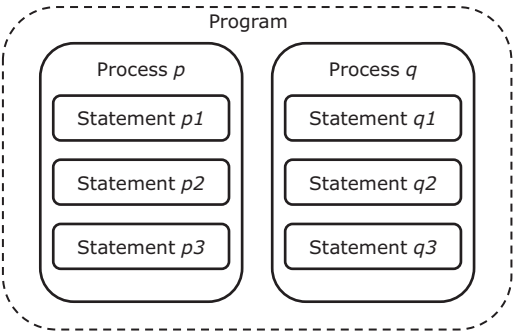

Figure 2.1. A concurrent program.

The correctness of a concurrent program is defined in terms of safety and liveness properties of scenarios [Ben-Ari, 2006, p. 21]. A safety property must *always* be true. It must be true in *every* state of *every* scenario. Contrary, a liveness property must *eventually* be true. It requires that in *every* scenario, there is *some* state in which the property is true. For example, consider a concurrent banking simulation in which two clients update the same bank account, see Figure 2.2.[3] It illustrates the *critical section* problem [Ben-Ari, 2006, pp. 45-48]: Occasionally, data must be accessed that is shared among processes. Let us consider that the following two properties must be true:

[2]For example, for Figure 2.1, a possible scenario is *p1, q1, q2, q3, p2, p3*.
[3]The program is inspired by an example from Oechsle [2007, pp. 26-29].

1. Always, bookings must not get lost (safety property), meaning that the statements of the critical section of a process (*p3* to *p5*, and *q3* to *q5*, respectively) must not be interleaved with statements of another process.

2. If a client updates an account, the account is eventually updated (liveness property). This means that once a progress enters the critical section, it eventually finishes executing the statements.

Banking example	
global variables: Account[] accounts	
p (client1)	q (client2)

p1: int accountNr = 1;
p2: float amount = 500.0;
p3: float oldBalance =
accounts[accountNr].
getBalance();
p4: float newBalance =
oldBalance + amount;
p5: accounts[accountNr].
setBalance(newBalance);
	q1: int accountNr = 1;
	q2: float amount = -300.0;
	q3: float oldBalance =
	accounts[accountNr].
	getBalance();
	q4: float newBalance =
	oldBalance + amount;
	q5: accounts[accountNr].
	setBalance(newBalance);

Figure 2.2. A simple booking program.

The negation of a safety property is a liveness property, and vice versa. This can be used to prove correctness since the negation may be easier to test. For example, when we consider the scenario *p1, q1, p2, q2, p3, q3, p4, p4, p5, q5* we can see that eventually, the booking of process p1 is lost, thus violating the safety property. The liveness property is not violated since the account is always updated but interleaving statements can override the account with an incorrect value. The interleaving can be eliminated by using synchronisation mechanisms. Typically, they consist of additional statements that are placed before (preprotocol) and after the critical section (postprotocol), see Figure 2.3.

Synchronised banking example	
global variables: Account[] accounts	
p (client1)	q (client2)

p1: int accountNr = 1;
p2: float amount = 500.0;
p3: *preprotocol*
p4: float oldBalance =
accounts[accountNr].
getBalance();
p5: float newBalance =
oldBalance + amount;
p6: accounts[accountNr].
setBalance(newBalance);
p7: *postprotocol*
	q1: int accountNr = 1;
	q2: float amount = -300.0;
	q3: *preprotocol*
	q4: float oldBalance =
	accounts[accountNr].
	getBalance();
	q5: float newBalance =
	oldBalance + amount;
	q6: accounts[accountNr].
	setBalance(newBalance);
	q7: *postprotocol*

Figure 2.3. A synchronised booking program.

Synchronisation Synchronisation is the enforcement of ordering constraints of a set of statements within a program execution to ensure the correctness of a program, cf. [Mattson et al., 2004, p. 17] and [Ben-Ari, 2006, p. 46].

2.3 Processes and Threads

From a technical viewpoint, a process can be regarded as a program in execution which comprises, for example, the program's data on the run-time stack or heap, actual register contents, and the program counter [Rauber and Rünger, 2007]. It runs in its own address space which is managed by the operating system. Therefore, in the technical sense, a process can be regarded as a UE consisting of a finite ordered set of atomic statements, where two processes do not share a common address space.

A thread, on the other side, runs within the address space of a single process [Ben-Ari, 2006, p. 4]. The thread concept is an extension of the process concept by allowing a process to consist of *a finite set of* a finite ordered

set of atomic statements, where the individual sets of statements share the address space of the process. Information exchange between threads is much faster than information exchange between processes via sockets, but essentially requires much more effort to synchronise data utilisation. The term *thread* originates from the widely-implemented pthreads (POSIX threads) specification [Nichols et al., 1996].

Today, the thread model – also denoted as *lock-based programming* because of its use of lock mechanisms for synchronisation – can be regarded as the predominant model for general-purpose concurrent programming. As prominently discussed by Lee [2006], it is highly non-deterministic and requires programmers to cut away unwanted non-determinism instead of inserting it where it is needed. The reason is the immense number of the possible interleaving of thread instructions that makes it extremely difficult to reason about the actual behaviour of a program.

> ***Process (technical)*** A process is "a collection of resources that enables the execution of program instructions. The resources can include virtual memory, I/O descriptors, a runtime stack, signal handlers, user and group IDs, and access control tokens" [Mattson et al., 2004, p. 16]. A process is typically regarded as a heavyweight UE that has an own address space.

> ***Thread*** "A thread is associated with a [technical] process and shares the process's environment." [Mattson et al., 2004, p. 16]. It is regarded as a lightweight UE that possibly shares its address space with other threads.

2.4 Performance Factors

The performance of a parallel system is influenced by various factors, such as the memory organisation of the architectural model, the programming language, and the problem to be solved. We briefly discuss these topics in the following.

2.4.1 Memory Organisation

The memory organisation of parallel systems can be separated in shared memory and distributed memory architectures. Other types of parallel systems regarding the memory architecture are hybrid systems. In a Shared Memory Machine (SMM), a single address space is shared by all processes/UEs. The processes communicate with each other indirectly by writing and reading shared variables. SMMs can be further classified:

In a *Symmetric Multiprocessor (SMP)* system, all processors share a common memory and can access all memory locations in the same time [Mattson et al., 2004, p. 10]. SMP systems are typically limited by processor/memory bandwidth, since an increasing number of processors also increases memory contention. Figure 2.4 illustrates the SMP architecture.

Non-Uniform Memory Access (NUMA) systems share memory that is addressed uniformly by all processors, but that cannot be accessed uniformly since some parts of the memory are physically more closely associated with some processors than others [Mattson et al., 2004, p. 10]. This increases the available memory bandwith, thus allowing to build systems with more processors than SMP systems. The downside is significantly different memory access times. Figure 2.4 illustrates the NUMA architecture.

Cache-Coherent Non-Uniform Memory Access (ccNUMA) is a further development of the NUMA architecture that exploits the locality of references within memory access. In ccNUMA systems, processors each have a small amount of non-shared cache memory along with hardware support to guarantee cache coherence, meaning that for all processors, the value of any variable is guaranteed to have a consistent value.

According to Ortega-Arjona [2010, p. 14], SMMs have the following advantages and disadvantages: First, the global address space simplifies concurrent programming since memory can be accessed similarly to sequential programming. Second, data sharing across processors is in general fast and uniform. On the other side, the amount of memory and processors is hard to scale since adding more processors increases the traffic between memory and processors geometrically. Finally, the programmers are responsible for proper synchronisation to ensure the correctness of their programs.

In a Distributed Memory Machine (DMM), each processor has its own

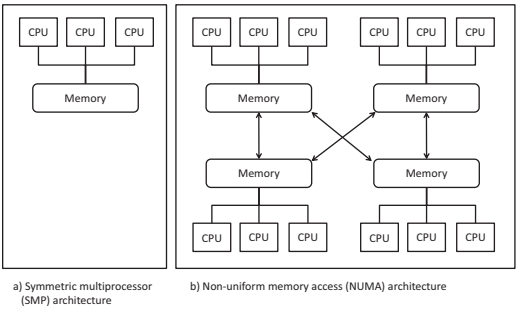

Figure 2.4. The SMP and NUMA shared memory architecture according to [Mattson et al., 2004].

local memory and there is no shared memory [Ortega-Arjona, 2010, p. 14]. Processors can communicate with other processes by explicit input/output operations over an interconnection network. Therefore, *message passing* is the prevalent communication model. The speed of communication in a DMM highly depends on the topology and the technology used for processor interconnection, and programmers must explicitly address communication and data distribution [Mattson et al., 2004, p. 11]. Figure 2.5 illustrates the DMM architecture. DMMs can be further classified:

Massively Parallel Processors (MPP) are specialised systems in which processors and the network infrastructure are tightly coupled [Mattson et al., 2004, p. 11]. They are extremely scalable since they can support the use of many thousands of processors in a single system [Mattson et al., 2004, p. 11], [Adiga et al., 2002].

Clusters are inexpensive parallel systems that are composed of commodity hardware: off-the-shelf computers connected by an off-the-shelf network infrastructure [Mattson et al., 2004, p. 11]. Beowulf Clusters are a special category of clusters which have standard personal computers as nodes that run a Linux operating system.

DMMs have the following advantages and disadvantages, see [Ortega-Arjona, 2010, p. 15]: First, memory scales with the number of processors. Second, processors can access their own memory rapidly without inter-

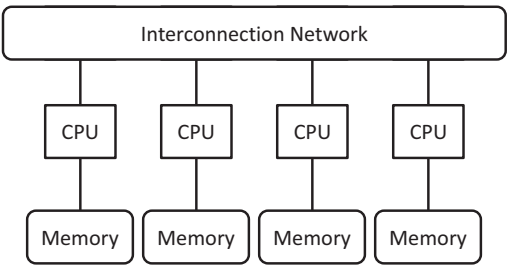

Figure 2.5. The distributed memory architecture according to [Mattson et al., 2004].

ference from other processors. Unfortunately, the communication of data between processors is in the responsibility of the programmers, and often very detailed. Additionally, common data structures that are based on shared memory can not easily be mapped to DMMs.

2.4.2 Programming Languages

Programming languages affect the performance of a parallel system in various ways, as well as the effort needed to develop a concurrent program [Ortega-Arjona, 2010, p. 16-22]:

- Language Infrastructure: Variation in the compiler and runtime support environments can constrain the exploitation of potential parallelism.
- Library Support: The types and the number of libraries that can be used by/within a concurrent program can have a significant effect on the performance of the parallel system and the effort to develop the program.
- Language Features: A programming language can be evaluated by its capacity to express concurrency, synchronisation, and the control of non-determinism between processes.
  - Concurrency and Sequencing: Otherwise sequential programming languages require a separate construct to express the concurrent ex-

ecution of statements. Examples are parallel composition (e. g., the `parbegin/parend` proposal of Dijkstra in ALOGOL60 [Dijkstra, 1968]), parallel construction (e. g., the $\|$ operator of CSP [Hoare, 1978, 1985]), interprocess concurrency (e. g., the `PAR` instruction and channels in Occam [Pountain, 1987]), and intraprocess concurrency (e. g., threads in the Java programming language [Lea, 1999; Goetz, 2009]). On the other side, concurrent programming languages require a separate construct to express the sequential execution of statements. Typically, this is achieved with the sequential construction, which is mostly expressed with the ; symbol between individual sequential instructions.[4]

- Synchronisation: There are several mechanisms for synchronisation of concurrent processes. They can be differentiated by the targeted memory organisation. For example, synchronisation mechanisms for languages that target SMMs comprise semaphores [Dijkstra, 1968], monitors [Hoare, 1974], or keywords to denote blocks of code as a critical section [Lea, 1999; Goetz, 2009]. Synchronisation mechanisms for languages that target DMMs, for instance, comprise blocking the processes during inter-process communication, as it is the case with the send and receive instructions in Communicating Sequential Processes (CSP) [Hoare, 1978, 1985].
- Non-Determinism: Non-Determinism is a characteristic of concurrent programs that states that the order of interleaving between the statements performed by a set of concurrent processes is arbitrary. It can be controlled with boolean expressions that *guard* the execution of a set of statements denoted as *guarded commands* [Ortega-Arjona, 2010, p. 21]. An example of guards is the `ALT` statement in Occam [Pountain, 1987].

2.4.3 The Problem Domain

The performance of a parallel system is heavily dependent on how a problem can be expressed as a set of concurrent processes. The reason are

[4] The Occam language represents an exception since it uses a `SEQ` construct instead of the otherwise prevalent semicolon [Pountain, 1987]

the *orthogonal dimensions* of coordination to computation of a concurrent program [Wegner, 1997]. This means that a concurrent program does not only consist of the correct computation statements of the problem, but also of the coordination that is employed to organise the statements into concurrent processes. Coordination modelling (Chapter 3) and Parallel Software Design (Section 3.1) represent two combinatoric approaches that encompass different solutions to implement this kind of organisation, for example, by conceptual patterns [Mattson et al., 2004; Ortega-Arjona, 2010].

2.5 High-Level Concurrent Programming

There are several software engineering techniques for high-level concurrent programming. They share the goal to find a reasonable compromise between high performance and portability. Amongst these techniques are the following:

Auto-tuners are adaptive libraries that use "automatically calibrated statistical models to select from among several implementations and to optimize the parameters of each implementation" for a given parallel system [Marowka, 2010, p. 80]. For instance, the Intel Software Autotuning Tool (ISAT) supports automatic searching for near-optimal values of program parameters such as cache-blocking factors in matrix computations, the task granularity in the Intel Threading Building Blocks (TBB) library, and the scheduling policy of OpenMP parallel constructs [Luk, 2010; Luk et al., 2011]. Another example is the Perpetuum auto-tuner which tunes applications cooperatively at run-time [Karcher and Pankratius, 2011]. It represents the first Operating System (OS)-based auto-tuning approach for *system-wide* performance improvement of simultaneously executing multi-threaded applications in contrast to the prevalent application-wide approaches.

In the context of software engineering, patterns capture successful experiences and techniques from software development by describing recurring successful solutions to common software problems, cf. [Ortega-Arjona, 2010, p. 2]. A *pattern system* is a collection of patterns that is oriented towards a domain, "together with guidelines for their implementation, combination and practical use in software development" [Buschmann et al., 1996, p. 361].

Pattern languages for concurrent program development are provided by Mattson et al. [2004] and Ortega-Arjona [2010].

Parallel Algorithmic Skeletons, are a form of recurring patterns of computation and communication in concurrent programs that have been formalised to the extend that they can be expressed as concrete language constructs, so-called *templates* [Cole, 1989; Falcou, 2009]. The benefit of skeletons is threefold: Templates encapsulate all low-level communication implementation, application programmers only have to know the operational semantics of the skeletons to implement concurrent programs, and sequential functions can be reused directly since they are decoupled from communication. There are several skeleton implementations for various programming languages. Examples comprise the Eskel library that resemble Message Passing Interface (MPI) primitives in C code [Benoit et al., 2005], the C++ Muenster Skeleton Library (Muesli) for multi-node, multi-core cluster computers [Ciechanowicz et al., 2009], and the Quaff library by Falcou [2009] to abstract over MPI code in C++ programs by using the C++ template mechanism.

Prototyping refers to the development of a system model in an early state of software development to explore the essential features of a system to be developed through experimentation before it is implemented [Floyd, 1984; Hasselbring, 2000]. There are many languages and approaches that can be considered for prototyping of concurrent programs. For example, the set-oriented prototyping language ProSet-Linda considers ProSet-Linda models as executable programs to examine parts of the functionality of the desired system [Hasselbring, 1998]. The language regards parallel system prototypes as dynamically created concurrent processes that are coordinated indirectly through virtual shared data spaces.

Complementary approaches to parallel system development can be bundled into *Integrated Development Environments (IDEs)*. For example, PARSEDAT provides a graphical design editor environment that uses the PARSE Process Graph Notation for software design and subsequent transformation of PARSE models to π-Calculus expressions for analysis and verification [Liu and Gorton, 1998]. A more recent example is the Eclipse Parallel Tools Platform (PTP)[5] that intends to provide a portable and extensible IDE

[5] `http://www.eclipse.org/ptp/`

for a wide range of parallel architectures and runtime systems. Features include means for coding and analysis, performance tuning, launching and monitoring, and debugging.

Model-Driven Software Development (MDSD) emphasises to develop software by generating code from formally specified models as the main development artefacts [Völter and Stahl, 2006]. In this way, quality and reuse is increased by allowing to correct mistakes in the model or in the generator templates once, instead of having to do multiple corrections in source code. Regarding the development of concurrent programs, there are few examples for the application of MDSD. Hsiung et al. [2009] present the model-driven development of multi-core embedded software based on SysML models as an input that generates C++ source code. The code architecture consists of an OS, the TBB library [Reinders, 2007], a framework for executing concurrent state machines, and the application code. Pllana et al. [2009] propose an IDE that targets multi-core systems. The IDE is envisioned to combine model-driven development with software agents and high-level parallel building blocks to automate time-consuming tasks such as performance tuning. Unified Modeling Language (UML) extensions are proposed for graphical program composition.

In his survey on prototyping approaches, Hasselbring [2000] presents a taxonomy for the surveyed approaches. It is abstract enough to be used as a foundation for a general taxonomy for high-level concurrent software engineering approaches, and comprises the following types of approaches:

High-level *libraries* provide simple interfaces for concurrent programming but often support only unstructured programming when appropriate use cannot be guaranteed at design time. Recent examples for libraries that abstract from thread-based programming are the LighTS framework which provides a minimalistic interface for space-based programming in Java [Balzarotti et al., 2007], the TBB, a portable C++ library targeted towards multi-core processor parallelism [Intel Corporation, 2011c], and Microsoft Task Parallel Library (TPL), a .NET Framework library that intends to increase the productivity of application programmers by taking over low-level details such as work partitioning, thread scheduling, cancellation, and state management from the application programmer [Leijen and Hall, 2007].

Data parallelism refers to the execution of simultaneous operations on

data sets instead of multiple concurrent threads of control [Hillis and Steele, 1986]. Thus, expressing the concurrent execution of many interrelated but semantically different operations is not supported. Example for data parallelism are OpenMP [Ope, 2011], a popular library that supports loop-level data parallelism, and Ateji(R) PX for Java [Viry, 2010], a language extension for concurrent programming on multi-core processors, Graphics Processing Units (GPUs), Grid, and Cloud for Java that supports both data and task parallelism.

Functional programming considers a program as the evaluation of mathematical functions. Functional programs are implicitly parallel since they are guaranteed to yield the same result irrespective of the order of computation. Although strictly functional programs are often considered as impractical for concurrent programming because of their limited abilities to manage mutable state and to support decisions (non-deterministic alternatives), there are sound concepts for functional concurrent programming, for example, Composable Software Memory Transactions [Harris et al., 2005]. Examples for general purpose functional languages comprise Clojure, Erlang, Haskell, and F# [Halloway, 2009; Armstrong, 2007; Hutton, 2007; Syme, 2010]. A particular domain-specific example is the R language for statistics [Horton and Kleinman, 2010].

Logic programming refers to the expression of a program as a set of clauses. The probably best known example is PROLOG [Clocksin and Mellish, 1994], which supports concurrency implicitly: Clauses can be evaluated separately, meaning that only one of the clause must be evaluated successfully (OR-parallelism, non-deterministic alternative). Additionally, so-called subgoals of clauses can be evaluated in parallel, meaning that all of them must be evaluated successfully to evaluate the clause (AND-parallelism).

Functional logic programming is a declarative conservative combination of functional and logic programming [Hanus, 2007; Antoy and Hanus, 2010]. Programs that fall into this category and do not use functional features are also logic programs, and functional logic programs that do not use logic features are functional. The approach can be characterised by the distinction between the construction of data and defined operations that manipulate data, by the ability to deal with incomplete knowledge by narrowing, and by demand-driven operations. Functional logic programming can be re-

garded as a high-level concurrent programming approach by the notion of non-deterministic choice. First, *don't care choice* refers to the irrelevance of the order of *argument evaluation* when two or more sub-expressions must be evaluated in a function call, meaning that the arguments can be evaluated concurrently. Second, *don't know choice* refers to the irrelevance of the order of *alternatives evaluation* when two or more independent alternatives must be computed for a given function call in which narrowing is applied, meaning that the alternative can also be computed concurrently. As Antoy and Hanus [2010] notice, the degree of parallelism offered by functional logic programming is potentially higher than that of corresponding imperative programs, but future research is necessary to investigate the true potential. Recent examples of functional logic programming languages include CURRY, TOY, and Mercury [Hanus, 2005; Arenas Sánchez et al., 2011; Henderson et al., 2012; Wang, 2006].

Concurrent Object-Oriented Programming (COOP) combines concurrency with Object-Orientation (OO) [Agha, 1990]. While concurrency provides an abstraction over the particular order of execution of statements, OO provides an abstraction that encapsulates data with means for their manipulation, usually in the form of methods [Meyer, 2000]. A popular example for COOP is Erlang, a concurrency-oriented programming language that originates from telecommunication industry and whose sequential subset is functional [Armstrong, 2007]. The language's features for concurrency are based on the actor model [Agha, 1985, 1990].

Coordination-based approaches separate computation and coordination [Gelernter and Carriero, 1992]. While coordination denotes the organisation of independent active entities, computation denotes the activity of the individual entities [Malone and Crowston, 1994; Wegner, 1996; Omicini and Papadopoulos, 2001]. As such, Coordination-based programming addresses the need for high-level abstractions for concurrent programs, usually in the form of separate coordination languages that are significantly higher than low-level communication primitives and libraries. The Linda/Tuple Space coordination language by David Gelernter can be regarded as the origin of coordination-based approaches [Gelernter, 1985]. It allows communication partners to be uncoupled from each other by the introduction of a Tuple Space, a shared virtual associative data structure that manages data in

the form of tuples. Today, most modern coordination approaches such as the Business Process Model and Notation (BPMN) coordinate fine-grained components denoted as *activities* within single composite components denoted as *workflows* [Axway et al., 2010]. Coordination is thereby specified by control flows that define the order of activities by small pieces of data regarded as control information.

Graph-based approaches rely on graphs to describe the data and control flows of concurrent programs. Typical examples are Petri nets [Reisig, 2010], Statecharts [Harel, 1987], and data flow diagrams [DeMarco, 1979]. The Intel Threading Building Blocks Flow Graph is a library-based variant that expresses data dependencies as messages passed between nodes. Nodes represent either custom function objects provided by the application programmer, or predefined objects to buffer, filter, split/join, broadcast, or order data as it flows through the graph [Intel Corporation, 2011c; Voss, 2011].

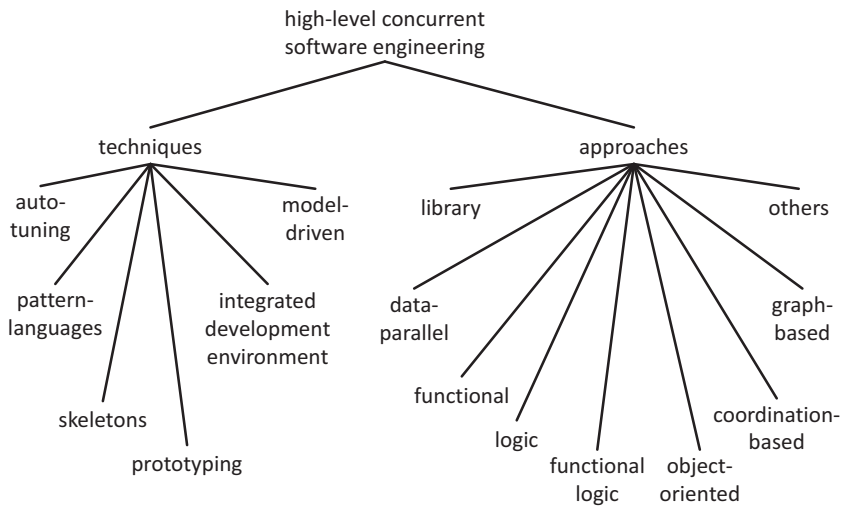

Figure 2.6. A taxonomy for high-level concurrent software engineering.

We combine the Hasselbring taxonomy with the list of high-level concurrent programming techniques described in this section to obtain a taxonomy

for high-level concurrent software engineering. We omitted the differentiation between linguistic and graphical approaches since we consider both as linguistic. Figure 2.6 illustrates the resulting taxonomy.

2.6 Summary of Concurrent Programming

Parallel processing offers an increased performance for the solution of many problems in contrast to sequential processing. To exploit the potential performance of a parallel system, one has to formulate the solution of a problem in a way that regards parts of the problem to be computed independently.

There are several models for parallel systems. Regarding their level of abstraction, these models can be differentiated in machine models, architectural models, computational models, and programming models.

Concurrency provides us an abstraction over the behaviour of a wide range of parallel systems that allows us to ignore the concrete order of execution of individual program statements.

Synchronisation can be required to ensure the correctness of a concurrent program.

A Process is a finite ordered set of sequentially executed atomic statements that runs in its own address space. Threads are an extension that allow a process to consist of more that one set of statements. These can run in parallel and share the address space of the process.

The performance of a parallel system is influenced by the memory organisation of the parallel system, the programming language, and the problem domain.

There are different approaches and techniques for concurrent programming that can be used as a taxonomy for classification.

Chapter 3

Coordination Models and Languages

> The concept of coordination is by no means limited to Computer Science
>
> George A. Papadopoulos and Farhad Arbab, 1998

Coordination models and languages provide a high-level abstraction for the design and implementation of large and complex parallel systems. They focus on the interaction patterns between the independent active components of the respective systems while the domain-specific logic of the individual components is neglected. As such, they can be regarded as tools for software integration [Ciancarini, 1996]. The overall goal of coordination modelling is to provide a framework that facilitates the modularity of the systems to be developed, the reuse of existing components, portability, and language interoperability [Papadopoulos and Arbab, 1998]. Multilinguality is supported by either considering component coordination in a separate super-language above the heterogeneous programming environment of the parallel system, or by providing an interface between the separate parts of the programming environment. The heterogeneity of parallel systems is further supported by providing an abstraction that can integrate different models of parallel systems.

The separation between computation and coordination was first presented by Gelernter and Carriero [1992] in the context of parallel and distributed systems. They described coordination as "the glue that binds

separate activities into an ensemble" while they considered an individual activity as computation. Since then, coordination has been considered in various domains, for example, programming languages, concurrent and distributed systems, artificial intelligence, multi-agent systems, and software engineering [Omicini and Papadopoulos, 2001]. These areas each have their own meaningful names for the activities to be coordinated (e. g., services, agents, individuals) and their coordination models (e. g., workflows, processes).

In their seminal paper "The interdisciplinary study of coordination", Crowston et al. [2006] address the multi-disciplinary nature of coordination and present Coordination Theory (CT) as an approach to unify the research on coordination as a separate research area. The main claim of CT is that dependencies and the mechanisms to managing them are *general* among domains. An important contribution of CT is a concise definition of coordination:

> ***Coordination*** Coordination is "the management of dependencies between activities" [Malone and Crowston, 1994].

Crowston et al. [2006] define dependencies as arising between activities instead of actors or roles. This simplifies the reassignment of activities to different actors – itself an activity that is common in process redesign efforts, as they notice. The definition focuses on the dependencies among activities instead of their respective outcomes. This has the advantage that coordination is not reduced to resource management but can address all kinds of dependencies that constrain how activities can be performed. Common types of dependencies are access to shared resources, the assignment of activities to actors, producer-consumer relationships, simultaneity constraints, and the decomposition of activities in sub-activities.

The conceptual distinction between computation and coordination is shown by Wegner [1997] by discussing their different expressiveness: computation and coordination are two orthogonal dimensions for programming languages, see Figure 3.1. The figure illustrates that a program can be seen as a composite of (sequential) computation and coordination, where the program code that refers to computation is organised by the program code that refers to coordination. While the former represents the domain-specific

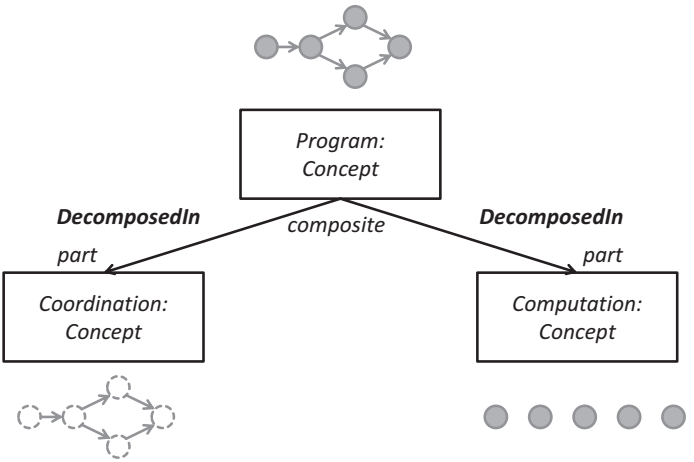

Figure 3.1. The distinction between computation and coordination.

logic of the program, the coordination code is completely problem domain-agnostic and only refers to the temporal and organisational dependencies of the individual parts of the computation and in which order they can be executed.

3.1 Models of Coordination

Concurrent programming languages basically support interactions through shared variables or message passing [Ciancarini, 1996]. Coordination models are system abstractions that are significantly higher-level than the shared variables or message passing paradigm.

> ***Coordination model (conceptual)*** A coordination model defines the interaction of active and independent components by defining the coordination laws that specify how the components coordinate themselves through the given coordination media (cf. [Ciancarini, 1996]).

In accordance with Ciancarini [1996] we interpret the definition as follows:

- *Coordination components* are the entity types to be coordinated. Depending on the domain these could be processes, threads, objects, actors, or users. They typically embody the domain logic of an application. The components are regarded as *active* since they show a behaviour such as producing or modifying passive data structures. They are *independent* insofar that they can execute at the same time and do not necessarily exist at the same location.

- *Coordination media* are domain-agnostic components that enable the communication among components. This can include the aggregation of components to form a higher-level component. Examples of coordination media are concurrent programming constructs such as semaphores, monitors, and channels, or architectural analogies such as tuple spaces, blackboards, and pipelines.

- *Coordination laws* specify how component coordinate themselves through the given coordination media. They can be represented by a set of well-defined coordination *primitives* that are defined on the coordination media and can be called by the components. Examples are object-oriented methods that enact synchronous or asynchronous behaviour among components.

The work of Wegner [1997] suggests that models of coordination have a significant impact on the engineering of complex systems. This is shown today with coordination-based programming-in-the-large approaches that are employed in industry and academia, see for example [Scherp et al., 2009]. Most modern coordination models coordinate fine-grained components denoted as *activities* within single composite components denoted as *workflows*. Coordination is thereby specified by control flows that define the order of activities by small pieces of data regarded as control information. This kind of coordination modelling is called *orchestration* [Melzer, 2007].

There are several ways to realise a coordination model. According to Ciancarini, a coordination model can be realised either as a coordination

language, or a coordination software architecture. We add libraries and explicit models:

A *library* can provide a high-level interface for a coordination model. They support various styles of programming, but appropriate use often cannot be guaranteed at design time. We consider them as particularly useful for coordination model prototyping. Recent examples for libraries that can be considered as coordination model realisations are the LighTS framework which provides a minimalistic interface for space-based programming in Java [Balzarotti et al., 2007], the Intel Threading Building Blocks (TBB), a portable C++ library targeted towards multi-core processor parallelism [Intel Corporation, 2011c], and Microsoft Task Parallel Library (TPL), a .NET Framework library that intends to increase the productivity of application programmers by taking over low-level details such as work partitioning, thread scheduling, cancellation, and state management from the application programmer [Leijen and Hall, 2007].

A *software architecture* is "[t]he fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [IEEE Architecture Working Group, 2000]. As such, software architectures can be regarded as implementations of coordination models from a behavioural viewpoint. However, they are often implemented using low-level concurrent or distributed programming techniques instead of being modelled on a higher level of abstraction. Examples of coordination software architecture comprise Parallel Pipes and Filters, Parallel Layers, Communicating Sequential Elements, Manager-Workers, and Shared Resource [Ortega-Arjona, 2010].

A *coordination language* is "the linguistic embodiment of a coordination model" [Gelernter and Carriero, 1992]. An example is the Linda coordination language developed by Gelernter and Carriero [Gelernter, 1985; Carriero and Gelernter, 1989] whose invention can be regarded as the cornerstone of CT in computer science. A recent example is the Business Process Model and Notation (BPMN), a specification that is targeted towards the coordination of business process in the business domain [Axway et al., 2010].

> ***Coordination language*** A coordination language is "the linguistic embodiment of a coordination model" [Gelernter and Carriero, 1992].

In the context of Model-Driven Software Development (MDSD), a *coordination model* is a platform-independent model of the software architecture of a system to be developed. The focus of this model lies on the behavioural viewpoint [IEEE Architecture Working Group, 2000]. It is explicit, meaning that it is an *artefact* that can be manipulated by computer systems or users. For example, a coordination model can be realised as an Ecore model file conforming to the Ecore Metamodel from the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009].

> ***Coordination model (explicit)*** A coordination model is an artefact that represents the software architecture of a system to be developed from a behavioural viewpoint.

3.2 Categorisation

A coordination model can be categorised by the types of components being coordinated, the types of the coordination media, the mechanisms and semantics of the coordination laws, the realisation of the coordination model, and its target platforms [Papadopoulos and Arbab, 1998; Wegner, 1996; Ciancarini, 1996]. More generally, Papadopoulos and Arbab [1998] argue that coordination models can be categorised according to what is focussed to be coordinated, the data or the components. Consequently, they classify coordination models as either mostly data-driven or control-driven. In the following, we explain the two categories and subsume their implications. The descriptions are based on the characterisations provided by Papadopoulos and Arbab [1998] and our own comprehension.

Data-driven coordination models define the state of the computation as both the values of the data being received or sent and the values of the data that is handled within the coordinated components. The characteristics are:

▻ Data coordination is emphasised before component coordination. The

relevant data is mostly considered as *products* (pre- or postconditions of component activity).

- *Shared Dataspaces* are commonly used as the coordination media [Roman and Cunningham, 1990]. Components are loosely coupled.

- Realisations of data-driven coordination models are usually *endogenous*: Coordination is considered as a set of coordination primitives to be invoked by components within a computational host language/model. As a consequence, libraries and architectures are an obvious choice for realisation.

- There is at least one coordinating component that examines and manipulates data, and that coordinates itself and/or other components by the coordination primitives.

- A clear separation between computational and coordination-related aspects is not enforced. Uses of coordination primitives are intermingled with computational code. It is in the responsibility of the programmer to provide a clear separation between the coordination-related and the computational aspects among and within the components of a program to be developed.

Control-driven coordination models define the state of the computation by the actual configuration of the interactions between the components. The actual values of the data being manipulated are of minor interest. The characteristics are:

- Component coordination is emphasised before data coordination. The relevant data for component coordination is considered as *information* (evaluation of expressions to control component activity).

- *Streams* or *channels* are commonly used as the coordination media. The overall communication paradigm is that of *message passing*, bound to point-to-point and limited broadcasting capabilities.

- ▻ Computational components are considered as black boxes with clearly defined interfaces. Except for these interfaces, coordination is almost completely separated from computation, but components are more closely coupled to each other than in data-driven coordination.

- ▻ Realisations of control-driven coordination models are usually *exogenous*: Separate languages/models are used to coordinate components programmed with computational languages.

With the success of Service-Oriented Architectures (SOA) [Melzer, 2007], a new class of coordination models showed up that gained widespread adoption in the business domain: *Workflow languages* are mostly control-driven and target the coordination of fine-grained closely coupled components denoted as *activities* within single composite components denoted as *workflows* or *processes*. Examples of such languages are the XML Process Definition Language (XPDL) and the Web Services Business Process Execution Language (WS-BPEL) [XPD, 2008; Alves et al., 2007]. Soon, the need emerged to coordinate the processes itself, and subsequent development efforts produced either extensions to existing languages or new complementary solution, for example, the Web Services Choreography Description Language (WS-CDL) and BPMN [Kavantzas et al.; Axway et al., 2010]. As a consequence, a second dimension for the categorisation of coordination models is the level of detail (granularity) on which coordination is considered [Melzer, 2007]:

Choreography-oriented models specify the behaviour of a system as the coordination of two or more coarse-grained components denoted as workflows/processes. They define inter-workflow/process behaviour.

Orchestration-oriented models specify the behaviour of a system as the coordination of fine-grained activities within a single workflow/process. They define intra-workflow/process behaviour.

3.3 Space-Based Systems

Space-Based Systems (SBS) are a class of data-driven coordination models that employ a data-sharing approach based upon the concepts of

Object-Orientation (OO) and generative communication [Freisleben and Kielmann, 1997; Hasselbring, 1998]. *Object-Orientation (OO)* provides composable, self-contained components that are protected by their interfaces [Meyer, 2000]. *Generative communication* – an idea originally introduced by the Linda coordination model – allows communicating components to be uncoupled from each other [Gelernter, 1985]. The overall analogy is that of a blackboard on which data can be written, or from which data can be read or removed.

3.3.1 Overview

In SBS, data-sharing is realised via so-called *spaces*, coordination-specific data management components that represent the coordination media. Client components communicate with each other indirectly by publishing data objects into the spaces and by reading or consuming data objects from them. Also, components can wait until a data object to be read or consumed actually has been inserted into the respective space. This means that operations that read or consume data objects can have either blocking or non-blocking behaviour. The decision what kind of data object is to be read or consumed is made by a *template* specification on the side of the reading or consuming component. The component sends this template as a query to a space. Matching that template to data objects is performed by the space. The data objects are usually realised in the form of *tuples*, ordered collections of data items that consist of *actual fields* (i. e., typed values). Templates, on the other hand, can consist of actual and *formal fields* (i. e., place-holders for typed values).

Spaces employ a matching algorithm to match a template specification to one or more data objects. These are, in turn, returned to the querying component. The standard matching algorithm can be described as follows: A tuple is regarded as matching a given template if (1) the tuple has no fields at all, or (2) the arities of the tuple and the template are equal and (2a) each actual field in the tuple has the same type and value as the corresponding field of the template, given it is also an actual field, or (2b) each actual field in the tuple has the same type as the corresponding field of the template, given it is a formal field. The following listing clarifies the algorithm in

pseudocode:[1]

```
ALGORITHM matches (Array A, Array F)
  IF A.size == 0
    RETURN true
  Boolean match =  A.size == F.size
  Integer i = 0
  WHILE match && i < A.size
    match = match && matches(A[i], F[i])
    i++
  ENDWHILE
  RETURN match
END

ALGORITHM matches (Field a, Field f)
  IF a INSTANCEOF FormalField
    RETURN a.getType() == f.getType()
  ELSE
    RETURN a.getType() == f.getType() &&
           a.getValue() == f.getValue()
END
```

3.3.2 History

SBS had their culmination in the late 1990s and the early 2000s in the form of IBM TSpaces, an SBS for Java,[2] and JavaSpaces, a service specification that provides an exchange mechanism for distributed Java objects [Freeman et al., 1999]. Especially the latter is of interest, since it attracted much attention upon its announcement but turned out a niche technology, for example in financial services and telecommunications. A possible reason is the fact that JavaSpaces is part of the Java Jini technology, a commercially unsuccessful framework of co-operating services for distributed systems [Edwards, 2000]. Sun contributed Jini to the Apache Software Foundation in 2007. The Jini project is continued under the name Apache River.[3]

Further advances of SBS can be described in contrast to the original Linda model. These developments can be related to one of two categories: increase of the *degree of distribution* of the coordination media, or increase of the *expressiveness* of the coordination laws. A recent example of the first category

[1] The algorithm was derived from the implementation of the classes `Tuple` and `Field` of the LighTS tuple space framework [Balzarotti et al., 2007].

[2] http://www.almaden.ibm.com/cs/TSpaces/

[3] http://river.apache.org/

is GigaSpaces, an in-memory SBS data grid for business applications in the context of Cloud Computing [Shalom, 2006]. The SBS is based on JavaSpaces. An example for the latter is TuCSoN, an SBS whose spaces (denoted as tuple centres) have a programmable behaviour that allows them to support application-specific coordination laws [Omicini and Zambonelli, 1998, 1999]. Today, there exists a wide range of SBS implementations, for example Fly Object Space, Blitz JavaSpaces (Pure Java Edition) 2.1, PyLinda, Rinda, Gruple, LinuxTuples, SemiSpace, and LighTS.[4]

3.3.3 Use for Concurrent Programming

As discussed by Gudenkauf and Hasselbring [2011], SBS show appealing features for concurrent programming:

SBS assume *explicit indirect coordination* amongst components. Instead of requiring software developers to cut away unwanted non-determinism by means of fine-grained synchronisation, they introduce non-determinism on a higher level of abstraction by the space operations.

While residing in spaces, data objects are *immutable*. To modify a data object, components must explicitly remove it from the space, modify it, and reinsert it. Data objects can never encounter conflicts or inconsistencies when multiple components attempt to modify them, thus eliminating undesirable situations such as lost updates.

SBS ease concurrent programming by abstracting from the location of components *in space and time*. Components do not necessarily have to exist at the same time, and always remain anonymous to each other.

SBS abstract from the computational model and are *orthogonal* to widespread General-Purpose Programming Languages (GPLs) such as C++, C#, and Java, since most GPLs are based on the computational model in principle.

[4] See http://www.flyobjectspace.com/,
http://www.dancres.org/blitz/,
http://code.google.com/p/pylinda/,
http://www.ruby-doc.org/stdlib/libdoc/rinda/rdoc/index.html,
http://gruple.codehaus.org/,
http://linuxtuples.sourceforge.net/,
http://www.semispace.org/semispace/, and
http://lights.sourceforge.net/

We summarise the use of SBS for concurrent programming by the following formula:

Shared spaces + Immutability of space-residing
data objects = Manageable concurrency

3.3.4 Example

In this section, we present a small example of an SBS, the LighTS tuple space framework [Balzarotti et al., 2007]. LighTS is a small extensible open source Java implementation of an SBS for non-distributed concurrent programs. Security, data persistence, and remote access/distribution are not addressed. Its interface can be used as a front-end for third-party tuple space implementations. Additionally, LighTS provides fuzzy-logic extensions for context-aware systems since conventional matching mechanisms that are based on exact values are largely insufficient for context-aware systems [Balzarotti et al., 2007]. LighTS was originally invented as the core implementation of the LIME SBS middleware [Murphy et al., 2006]. The following listing illustrates the tuple space interface of LighTS, along with comments that explain the individual coordination primitives defined on spaces:

```
public interface ITupleSpace {
  String getName();

  void out(ITuple tuple);        // publish a tuple
  void outg(ITuple[] tuples);    // publish a group of tuples

  ITuple in(ITuple template);    // blocking tuple consumption
  ITuple inp(ITuple template);   // non-blocking tuple consumption (probe)
  ITuple[] ing(ITuple template); // non-blocking atomic consumption of a set of tuples

  ITuple rd(ITuple template);    // blocking tuple read
  ITuple rdp(ITuple template);   // non-blocking tuple read
  ITuple[] rdg(ITuple template); // non-blocking atomic read of a set of tuples

  int count(ITuple template);    // number of available matching tuples
}
```

The following self-explaining example illustrates how a space can be created and how a tuple can be published by a client.

```
// create tuple space
ITupleSpace ts= new TupleSpace("Space");

// create a tuple
ITuple tuple= new Tuple()
.add(new Field().setValue("ID"))              // actual field
.add(new Field().setValue(newInteger(12345))) // actual field
.add(new Field().setType(Integer.class));     // formal field

// publish the tuple into the tuple space
try{ ts.out(tuple); }
catch(TupleSpaceException e) {e.printStackTrace(); }
```

Additionally, LighTS supports OO by finding a reasonable compromise between object encapsulation and the requirement to publish the internals of a data object stemming from generative communication: The interface `ITuplable` allows to flatten objects into tuples with the method `toTuple`, while `setFromTuple` allows to recreate an object from a tuple. This is supported by the class `ObjectTuple`, which can remember the type of the object a tuple was created from.

```
// writing component
Complex complex= new Complex (-1.0, 1.0);
try{ ts.out( complex.toTuple() ); }
catch (TupleSpaceException e) {...}

//reading component
ITuple template = new Tuple()...;
ObjectTuple objTuple= null;
try { objTuple= (ObjectTuple) ts.rd(template); }
catch (TupleSpaceExceptione) {...}
this.complex = (Complex) objTuple.getObject();

// data object definition
public class Complex implements ITuplable, Serializable {
  private double a; private double b;

  @Override
  public ITuple toTuple() {
    ObjectTuple objTuple = new ObjectTuple(Complex.class)
    objTuple.add(newField().setValue(new Double(a)))
            .add(newField().setValue(new Double(b)))
    return objTuple;
  }

  @Override
  public void setFromTuple(ITuple tuple) {
    this.a = (Double) ((Field) tuple.get(0).getValue();
    this.b= (Double) ((Field) tuple.get(1).getValue();
```

```
  }
  ...
}
```

However, LighTS suffers from the limitation of being a library-based internal domain-specific language for SBS (see Section 4.5.2): coordination code is compromised with Java language syntax, proper usage cannot be enforced, and domain-specific validation is not possible. Additionally, the platform-specific implementation uses serialisation to guarantee deep copy behaviour for data object hierarchies.

3.4 Business Process Model and Notation 2.0

The Business Process Model and Notation (BPMN) is an example for control-driven coordination models. Its goal is the provision of a language targeted towards the coordination of business processes that is understandable by a wide range of stakeholders, ranging from business analysts that draft processes, to technical developers that implement the facilities to execute the processes, and business process monitors that manage the processes during execution. In its current version (2.0), the specification represents a full-fledged coordination language, cf. [Axway et al., 2010, p. 22]:

- The BPMN specification comprises a graphical notation as its concrete syntax and an abstract syntax in the form of a metamodel.
- BPMN provides informal execution semantics that describe the execution behaviour of the individual elements of the language.[5]
- It provides an extensibility mechanism for both metamodel extensions and graphical notation extensions.
- It supports the definition of human participation in business processes.

[5] In fact, the specification states that it "formalizes the execution semantics for all BPMN elements" [Axway et al., 2010, p. 22], but only descriptions in natural language are provided: "The execution semantics are described informally (textually), and this is based on prior research involving the formalization of execution semantics using mathematical formalisms." [Axway et al., 2010, p. 425]. Also, only control flow execution semantics are described [Axway et al., 2010, pp. 425-444].

BPMN also provides machine readable representations of the specifications (XML Schema Definition (XSD), XML Metadata Interchange (XMI), Extensible Stylesheet Language Transformations (XSLT), and Meta Object Facility (MOF) compatible) that allow the serialisation and transformation of BPMN models.

3.4.1 Overview

BPMN supports the following viewpoints on coordination in the business domain, cf. [Axway et al., 2010, p. 23]:

- Processes: Components whose internal activities are *orchestrated*
  - Private non-executable Processes: Processes that are internal to a specific participant and were modelled for the purpose of documentation
  - Private executable Processes: Processes that are internal to a specific participant and were modelled for the purpose of being executed by appropriate execution environments
  - Public Processes: Processes that represent views on private processes that show only those activities that are used to communicate with other processes
- Collaborations: A view on two or more processes that models both processes as well as the interactions between the processes as the exchange of messages.
- Choreographies: A view on the interactions between participants that models communication as the ordered exchange of messages, thus representing the *communication protocols* between the participants
- Conversations: A view on the interactions between participants that models the ways messages can be sent, thus representing the *communication channels* between the participants

The elements of the specification are categorised as follows, see Figure 3.2. *Flow Objects* are the main elements that define the behaviour of a

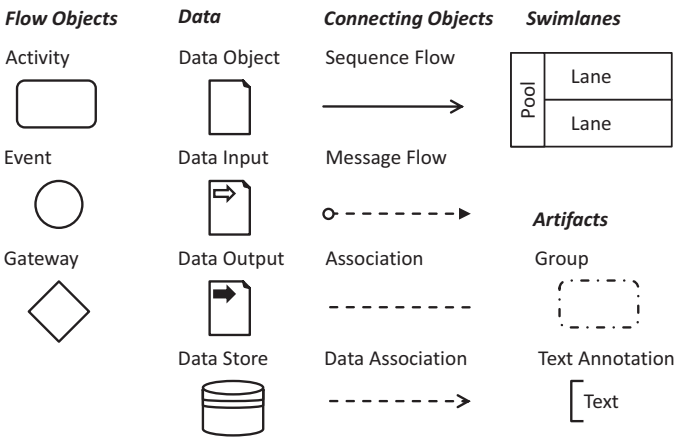

Figure 3.2. Overview of the elements of the BPMN 2.0 specification.

Process: *Events* are foreseen to occur during the execution of a Process to affect its execution. They usually have a cause or an impact and require or allow for a reaction. It is differentiated between start, end and intermediate events. Examples are Message Intermediate Events to either send or receive a Message. *Activities* represent work to be performed within a Process. They can be either atomic or compound (hierarchically composed). Examples are Task (an atomic Activity that cannot be broken down to a finer level of detail) and Sub-Process (a composite Activity whose internal behaviour has been modelled using Activities, Gateways, Events, and Sequence Flows). Activities can be decorated with Loop Characteristics to model looping behaviour, or sequential or parallel instantiation. *Gateways* represent decisions to be made or followed to control the execution of a process. Examples are Exclusive Gateway to create or merge alternative paths within the execution of a Process, and Parallel Gateway to create or synchronise concurrent execution paths within a Process.

Data is represented by the following elements: *Data Objects* are generic representations of data that can represent states and collections of data. The lifecycle of a Data Object is bound to the lifecycle of its parent Process or

Sub-Process. *Data Inputs* and *Data Outputs* represent data that is explicitly required as a precondition, or provided as a postcondition for the execution of Activities or Processes. *Data Stores* represents an information system to store data that will persist beyond the lifecycle of a Process. *Properties*, like Data Objects, are generic representations of data. However, they are not visually presented on a Process diagram. Only Processes, Activities, and Events can contain Properties. The lifecycle of a Property is bound to the lifecycle of its parent Flow Object.

Flow Objects are connected by so-called *Connecting Objects*: *Sequence Flows* specify the (sequential) order of the execution of Flow Elements in a Process or a Choreography. Each Sequence Flow has only one source and only one target element. *Message Flows* specify the message exchange between two participants. They represent the message passing-based communication facility between individual Processes. *Associations* are mostly used to associate Artifacts with Flow Objects. *Data Associations* specify how data is pushed into or pulled from model elements. They show a copy-behaviour, meaning that the source of the association is copied to the target instead of transferred.

Swimlanes represent elements for grouping: A *Pool* is the representation of a Participant in a Collaboration and can include a Process to be executed by the Participant. *Lanes* are sub-partitions of a Process.

Artifacts provide additional information about BPMN model elements: *Groups* represent an informal mechanism to informally group elements without any effects on the behaviour of the BPMN model or its elements. Similarly, *Text Annotations* add informal information to annotated model elements without affecting the behaviour of the model or its elements.

3.4.2 History

BPMN was invented by Stephen A. White (IBM) and published in 2004 by the Business Process Management Initiative (BPMI). In 2005, the BPMI merged into the Object Management Group (OMG), and the maintenance of the BPMN specification was continued by the OMG [Freund and Rücker, 2010]. BPMN is since an official OMG standard.

Version 1.0 was accepted in 2006 as an official OMG standard [Allweyer, 2009]. Versions 1.1 (2008) and 1.2 (2009) were maintenance releases

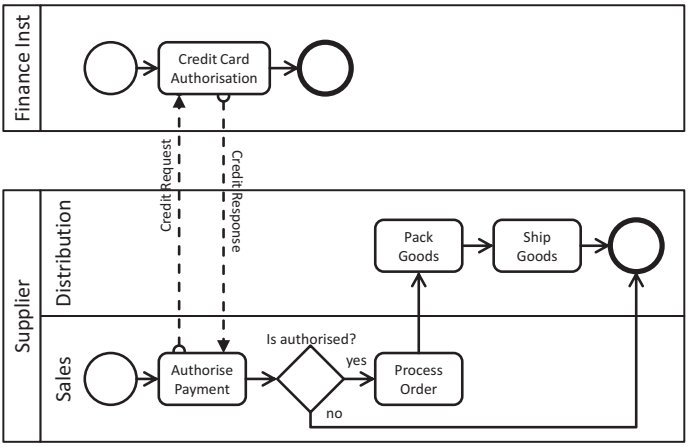

Figure 3.3. A BPMN collaboration example diagram illustrating a supplier and a financial institution collaborating to ship goods.

that mainly targeted issues related to graphical notation elements. The current version 2.0 was released in January 2011 and represents a major release. Amongst others it contributes an abstract syntax in the form of a metamodel and informal execution semantics, thus evolving BPMN from a mere notation into a full-fledged coordination language.

3.4.3 Example

Figure 3.3 shows a simple BPMN example that illustrates the collaboration of a supplier and a financial institution to ship goods. Both processes start concurrently with their respective Start Events (the Events with the thin border on the left sides). Firstly, the supplier authorises a current payment by contacting the financial institution. This is modelled as a Message Flow from the "Authorise Payment" Activity of the supplier to the "Credit Card Authorisation" Activity of the financial institution. The financial institution checks the credit card and sends a response that contain information about the outcome of the authorisation ("Credit Response" Message Flow). After-

wards, the process of the financial institution is terminated as modelled with an End Event (the Event with the thick border on the right side of the financial institution's Pool). If the payment is authorised (Exclusive Gateway), the supplier processes the order, packs the goods and ships them. Then, the process of the supplier is also terminated.

3.5 Summary of Coordination Models and Languages

Coordination models and languages provide a high-level abstraction for the design and implementation of large complex parallel systems that focusses on the interaction patterns between the independent components of the respective systems. They embrace the multilinguality and heterogeneity of these systems.

Coordination is the management of dependencies between activities, and as such, interdisciplinary.

Conceptually, coordination models are system abstractions that are significantly higher-level than the shared variables or message passing paradigm. They define the interactions of active and independent components in terms of the coordination laws, the coordination components, and the coordination media. Coordination components are the entity types to be coordinated, coordination media are domain-agnostic components that enable the communication among components, and coordination laws specify how component coordinate themselves through the given coordination media.

A coordination model can be realised as a library, a software architecture, a separate language, or, in the context of MDSD, as an explicit platform-independent model artefact that can be manipulated by computer systems or users.

Coordination models can be categorised in two dimensions: data-driven or control-driven (coordination focus), and choreography or orchestration-oriented (granularity). Data-driven coordination models emphasise data coordination before component coordination and do not enforce a clear separation between computational and coordination-related aspects. Control-

driven coordination models emphasise component coordination before data coordination and consider computational components as black boxes with clearly defined interfaces. Choreography-oriented models specify the behaviour of a system as the coordination of two or more coarse-grained components denoted as workflows/processes (inter-workflow/process behaviour). Orchestration-oriented models specify the behaviour of a system as the coordination of fine-grained activities within a single workflow/process (intra-workflow/process behaviour).

SBS are data-driven coordination models that employ a data-sharing approach. SBS show appealing features for concurrent programming: decoupled components, explicit indirect coordination amongst components, immutability of space-residing data objects, and orthogonality to widespread general purpose programming languages.

The BPMN is a control-driven coordination language targeted towards the coordination of business processes. It is intended to be understandable by a wide range of stakeholders, ranging from domain experts to technical developers.

Chapter 4

Model-Driven Software Development

> Abstraction is the key to progress in software engineering
>
> Eelco Visser, 2008

Model-Driven Software Development (MDSD) emphasises to develop software by generating code from formally specified models as the main development artefacts [Völter and Stahl, 2006]. Quality and reuse is increased by allowing to correct mistakes in the model or in the generator templates once, instead of having to do multiple corrections in source code. These models are significantly more abstract and domain-oriented than executable program code, and often cannot be executed. Instead, models have to be translated into executable code for a specific platform by using model transformations and model transformation infrastructures. Thereby, a series of model transformations typically ends with a model-to-code transformation that results in the executable code. Domain-oriented models improve managing complexity by focussing on long-living domain-specific aspects, leaving short-living aspects for Platform-Specific Models (PSMs) or directly for code generation. Domain-Specific Languages (DSLs) allow domain experts to participate directly in software development. In the following, we present an overview of MDSD concepts, starting with definitions for the terms model and metamodel.

> ***Model*** A model is a representation of a system under study, where the model is (1) itself a system, (2) is simpler than the

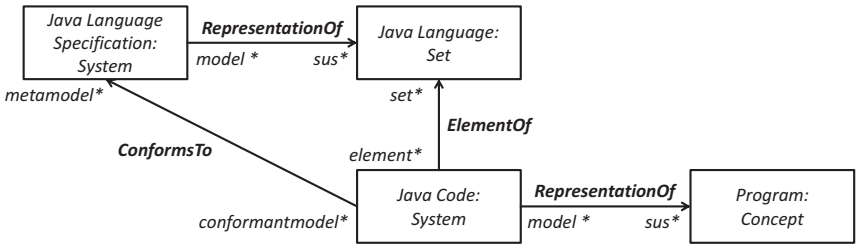

Figure 4.1. The notion of *metamodel* by the example of the Java programming language according to Favre and NGuyen [2005].

> system under study, and which (3) can give answers instead of the system under study that also hold for the system under study [Stachowiak, 1973; Seidewitz, 2003; Bézivin and Gerbé, 2001].

The idea of a metamodel is to formally represent the commonalities of a set of models in a system so that it can be checked if a model is an element of the set. If so, the model is said to be *conformant* to the metamodel. Figure 4.1 illustrates this relationship: A metamodel is relative [Völter and Stahl, 2006, p. 57]. The Java language specification plays the *role* of a metamodel to Java program code since it represents the Java language.

> ***Metamodel*** A metamodel is a model of a *set* of models [Favre and NGuyen, 2005].

4.1 The MDSD Metamodel

Figure 4.2 shows a metamodel that defines the MDSD domain. It is based on the work of Favre [2004a,b, 2005]; Favre and NGuyen [2005] on megamodels which model large-scale software evolution processes. It is fundamentally based on the concept of Systems linked with a well-defined set of relations such as Representation Of (μ), Conforms To (χ) and Transformation Instance (τ). Consequently, in their metamodel of megamodels, the concepts of models and metamodels are realised as *roles* that can be played by systems.

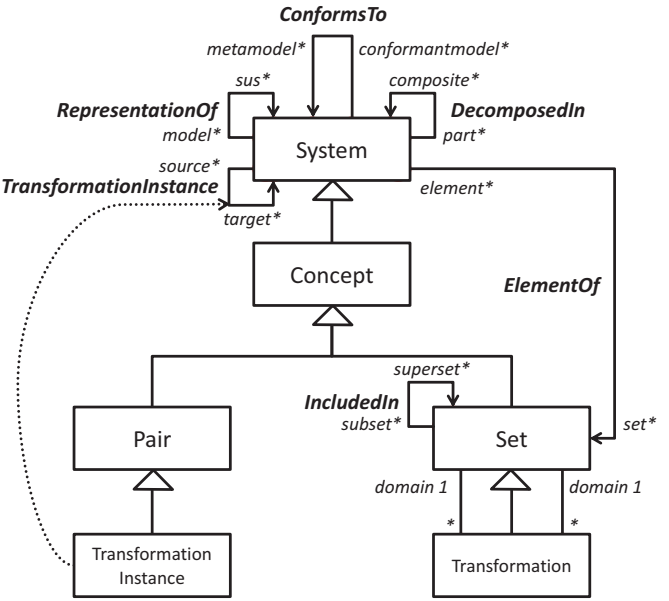

Figure 4.2. The MDSD metamodel (megamodel metamodel) based on Favre and NGuyen [2005].

We consider this conceptual megamodel metamodel as ideally suited to be used as a model for MDSD itself due to its flexibility. This flexibility originates from the use of fundamental entities from the system and set theory, and from the use of only the most basic relations (no aggregation, no composition). These can be considered as the axioms of MDSD.

- ▻ Representation Of (μ): A System can be a *model* which represents a certain system under study (*sus*).
- ▻ Element Of (ε): A System can be a single *element* of a Set.
- ▻ Conforms To (χ): A System can be an *element* of a Set that itself is a system under study to another system. This other System plays the role of a *metamodel* to the original System.

- ▻ Decomposed In (δ): A System can be a *composite* that can be decomposed in *parts*.
- ▻ Included In (ζ): A Set can be a *subset* to another Set denoted as *superset*.
- ▻ Transformation Instance (τ): A System that is a model of a system under study can be transformed into the system under study given an appropriate Transformation Instance.

The last relation, Transformation Instance, is of particular interest in the practical application of MDSD. We present model transformations and their transformation instances in more detail in the following section.

4.2 Model Transformations

The goal of MDSD is to create software in part or whole through transformations [Völter and Stahl, 2006, p. 61]. Therefore, model transformations play a key role in MDSD. Informally, a model transformation can be thought of as a computable mapping that translates source models into target models, cf. [Reussner and Hasselbring, 2009, p. 97]. Czarnecki and Helsen summarise the application of transformations in MDSD as follows [Czarnecki and Helsen, 2006]:

- ▻ Lower level model and program code generation from higher-level models
- ▻ Model synchronisation
- ▻ The creation of views of a system based on queries
- ▻ Evolutionary activities modelling (e. g., refactoring)
- ▻ Reverse engineering of lower level models and program code from higher level models

Although transformation is a recurring topic in computer science, there are several characteristics that distinguish model transformations from other transformation approaches. On the basis of the work of Czarnecki and Helsen [2006] on model transformations, we subsume them as follows:

- Object-Orientation (OO): Model transformations adopt an approach based on OO to represent and manipulate models.
- Traceability: Model transformations consider the traceability among models as a primary concern. Traceability refers to mechanisms to create and maintain trace links between source and target model elements that describe the runtime footprint of a transformation execution. These links can be used to analyse the impact of model changes to related models, and for stepwise transformation debugging.
- N-way transformations: Model transformations can be defined for multiple models as sources and targets.
- Multi-directionality: Transformations do not necessarily define a transformation direction for their execution.

More formally, one can distinguish between the notion of transformation as a function, the representation of a transformation in a transformation language, and the instance of a transformation at runtime. We provide definitions for these terms in the following. Figure 4.3 illustrates this operational context. We recommend the work of Favre and NGuyen [2005] for further information on this separation.

> ***Transformation (function)*** A Transformation, also denoted as *transformation function*, is a Set of Transformation Instances. The set of systems that can be transformed by the transformation function is denoted as the *domain* of the transformation, and the set of systems that can be obtained via a Transformation is denoted as its *range* [Favre and NGuyen, 2005].
>
> ***Transformation instance*** A Transformation Instance is a Pair of two Systems which represents the application of a Transformation (function) to a particular input (the *source*) to produce an output (the *target*) [Favre and NGuyen, 2005].

To increase comprehensibility, we additionally introduce the following definitions:

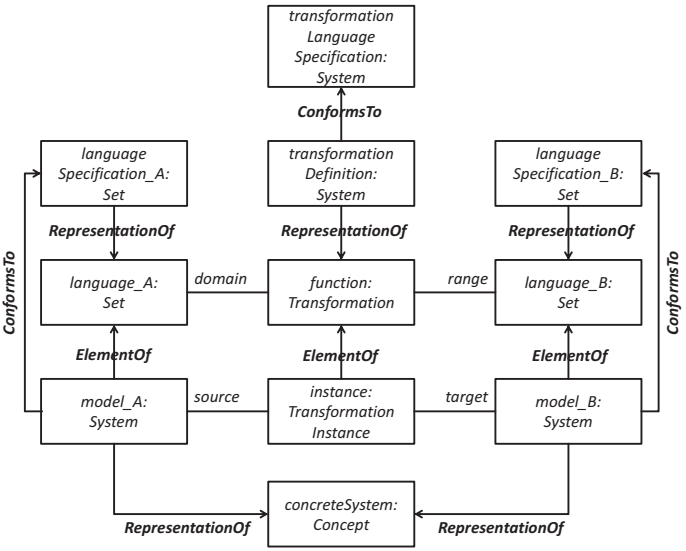

Figure 4.3. The operational context of a model transformation according to Favre and NGuyen [2005].

Transformation engine A transformation engine is a system that can instantiate transformations (functions) by applying them to the source systems to produce the target systems. The instantiation of a transformation (function) is referred to as transformation *execution*.

Transformation language A transformation language is the linguistic embodiment of a model of transformations.

Transformation definition A transformation definition is the description of a transformation (function) using a transformation language.

Approaches to model transformation are practically distinguished by what kind of target models are used. In generally, we differentiate between

Model-to-Model (M2M) transformations and Model-to-Text (M2T) transformations. While the former create or manipulate their targets as instances of explicit metamodels, the latter consider their target simply as text strings [Czarnecki and Helsen, 2006]. Essentially, M2T transformations are a special case of M2M transformations given a metamodel for the target programming language is available. The focus on string generation allows to reuse existing target programming language compiler technology [Czarnecki and Helsen, 2006; Reussner and Hasselbring, 2009]. Further classification is extensively discussed by Czarnecki and Helsen [2006].

In the following, we present examples for the two main classes of transformations: The M2M transformation language Query/View/Transformation (QVT), and the M2T transformation language Xpand.

4.2.1 Query/View/Transformation (QVT)

QVT is a standard that specifies a set of model transformation languages defined by the Object Management Group (OMG) [QVT, 2011]. As the name implies, QVT covers the issues of querying models, transforming models, and providing views on models that conform to the Meta Object Facility (MOF) metamodel standard [MOF, 2011]. The standard integrates the Object Constraint Language (OCL) for constraint definition and specifies the following parts, cf. [QVT, 2011, pp. 9-10]:

Relations is a declarative language that specifies transformations bidirectionally as a set of relations between two models which consist of object patterns. These patterns are applied model-wise either in checkonly mode or in enforce mode. While *checkonly* mode checks consistency by testing if the relation holds for one model with respect to the other, *enforce* mode enforces the consistency between the two models by manipulating one of them as specified. The language has both a textual and a graphical notation.

Core is a declarative language equally powerful to the Relations language but significantly simpler, trading language simplicity for verboseness. Trace models must be explicitly defined and cannot be derived as in

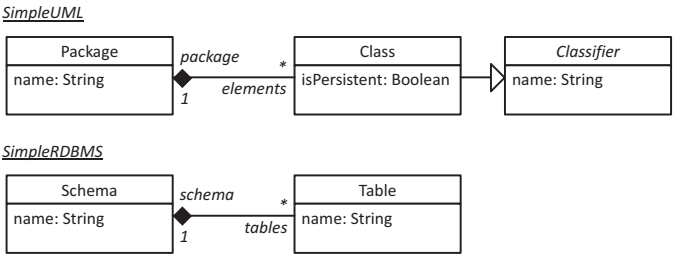

Figure 4.4. Metamodels for SimpleUML and SimpleRDBMS.

the Relations language. It is used to provide the semantic basis for the specification of the Relations language.

Operational Mappings is an imperative language for unidirectional transformations that extends the Relation language with side-effect-afflicted OCL expressions and imperative constructs. These constructs instantiate declarative object patterns from the Relations language.

Black Box operations allow to integrate external program code in the execution of a transformation, for example, to reuse existing domain-specific libraries. However, as their execution is not normally controlled by a QVT transformation engine, Black Box operations represent a potential risk for inconsistency.

Representative implementations of the standard are mediniQVT[1] for QVT Relations and SmartQVT[2] for QVT Operational Mappings.

The following listing illustrates a transformation written in QVT Relation to transform Unified Modeling Language (UML) class models into relational database tables based on the example that is delivered with mediniQVT. Note that we omitted class attributes and table columns for simplicity. The simplified self-describing metamodels are shown in Figure 4.4.

```
transformation uml2rdbms(uml:SimpleUML, rdb:SimpleRDBMS)
```

[1] http://projects.ikv.de/qvt/
[2] http://sourceforge.net/projects/smartqvt/

```
{
  -- map each package to a schema
  top relation Package2Schema {
    pname : String;
    checkonly domain uml p : SimpleUML::Package {
      name = pname
    };
    enforce domain rdb s : SimpleRDBMS::Schema {
      name = pname
    };
  }
  -- map each persistent class to a table
  top relation Class2Table
  {
    cname: String;
    checkonly domain uml c:Class {
      name = cname,
      ownerPackage = p:Package {},
      isPersistent = true
    };
    enforce domain rdb t:Table {
      name = cname,
      schema = s:Schema {}
    };
    when {
      Package2Schema(p, s);
    }
  }
}
```

The QVT Relations code declares a transformation with the two typed parameters *uml* and *rdb* for the involved models. The parameters are typed over the two metamodels shown in Figure 4.4. The first relation within the transformation enforces that an *rdb* schema's name equals that of the related *uml* package. The second relation enforces a table for each persistent class in the given *uml* model. The *when* clause defines a precondition under which the relation must hold. In this case, a table can only be enforced if its schema has been enforced from a corresponding package beforehand. Both relations are denoted as *top*, meaning that they are always executed when the transformation is executed. Regarding the application of the transformation, the user specifies the execution direction manually in the transformation engine.

4.2.2 Xpand and Xtend

Xpand is a template-based M2T transformation language that generates text from Eclipse Modeling Framework (EMF) Ecore models [Efftinge et al., 2004-2010]. Its purpose is effective template development with an easy-to-learn language and good tool support [Klatt, 2007; Efftinge et al., 2004-2010]. Xpand was originally developed as a part of the openArchitectureWare MDSD platform and is now available as a sub-project of the Eclipse Modeling Project [Efftinge et al., 2008; Xte, 2011]. The language features comprise template polymorphism, import and namespace concepts, nesting, loops, expressions to apply a template on a collection of elements, decisions, means for file generation, protected regions that are not overridden in subsequent text generations, variables, errors, a mechanism to control the generation of whitespaces, support for aspect-oriented programming, and comments [Klatt, 2007]. French quotation marks are used to escape template expressions from text fragments. Text generation is started via a workflow script that defines the entry point for execution. Additionally, Xpand can access functions implemented in Xtend, a functional language to extend metamodel types with additional logic. Xtend also originated from the openArchitectureWare MDSD platform [Efftinge et al., 2008]. Xpand and Xtend represent the foundation for the recent development of Xtend2, a statically-typed template language with a Java-like syntax in the context of the Xtext language workbench [Xte, 2011].

In the following listing, we illustrate an Xpand template that generates Java code from models that conform to the SimpleUML metamodel shown in Figure 4.4. Firstly, the SimpleUML metamodel is imported. Secondly, an Xtend extension is loaded. It is later used to calculate the fully qualified name of classes. Thirdly, the first `DEFINE` directive defines a template for the *Package* type that expands another template for each *Class* in the package. It represents the entry point for the text generation engine and must be referenced as such in the execution engines workflow script. The second `DEFINE` directive defines the template for a class to be generated. Thereby, a class is only generated when it is marked as being persistent. The outlet for text generation is specified via the `FILE` directive. We use the function `qualifiedName` from the imported Xtend extension to calculate the fully

qualified name of the class, and replace the point characters with the slash character to generate the corresponding folder structure. The minus character is used to remove supernumerary whitespaces and line breaks.

```
<<IMPORT SimpleUML>>
<<EXTENSION template::Extensions>>

<<DEFINE main FOR Package>>
  <<EXPAND class FOREACH elements>>
<<ENDDEFINE>>

<<DEFINE class FOR Class>>
  <<IF isPersistent == true>>
    <<FILE (qualifiedName().replaceAll("\\.","/") + ".java")->>
      package <<ownerPackage.name->>;
      public class <<name->>
      {
        public <<name->>(){}
      }
    <<ENDFILE>>
  <<ENDIF>>
<<ENDDEFINE>>
```

4.3 Model-Driven Systems

Model-driven software projects lead to very complex structures that are based on models, metamodels and transformations as their cornerstones, see Figure 4.5. Typically, such structures are inherently difficult to understand and consist of a variety of artefacts, for example, domain-specific models, technical models that abstract from the underlying programming environment, model transformations, model transformation engines, and additional software tools. We denote such structures as Model-Driven System (MDS). An important aspect of MDS is that they consider the tools that are used in a software project explicitly, since a metamodel may impose restrictions to the version of a tool to be used.

> ***Model-Driven System (MDS)*** An MDS is a system whose artefacts refer to models, metamodels, and transformations, and the tools that are used in a model-driven software project.

The purpose of an MDS is to handle and maintain the individual artefacts and their interrelations to automate the generation of the target system as

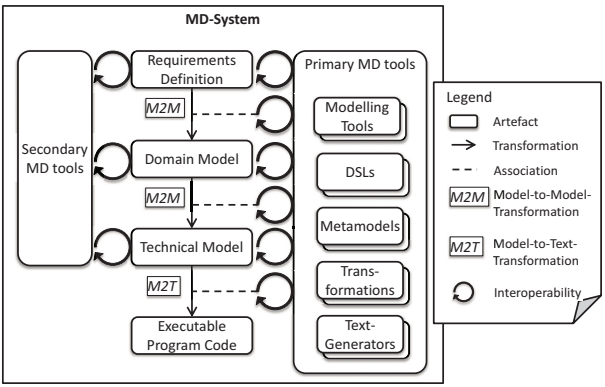

Figure 4.5. An exemplary MDS.

far as possible. Its application provides three advantages: reuse for software product families, reuse for software product lines, and the separation of concerns to handle complexity.

Reusing the MDS for the generation of technical artefacts such as program code and configuration files can decrease the time to market of *software product families*. The focus of the MDS is therefore shifted towards domain aspects such as the domain models.

> ***Software product family*** A (software) product family is a group of products that share at least a common generic (software) architecture. They are scoped primarily based on technical commonalities (cf. [Withey, 1996, p. 16], [Czarnecki and Eisenecker, 2005]).

Reusing domain models, on the other hand, can decrease the time to market of *software product lines*. The focus of the MD system is therefore shifted towards technical aspects such as the model transformations.

> ***Software product line*** "A [software] product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market" [Withey, 1996, p. 15]

and is therefore scoped primarily based on a marketing strategy (rather than technical commonalities).

The *separation* of the individual artefacts from each other, for example, the domain models from the technical transformations, allows to develop software for domains whose requirements become more and more complex.

Figure 4.6 illustrates a possible metamodel for an MDS family. It is a refinement of the megamodel metamodel presented in Figure 4.2 and separated into a platform-independent and a platform-specific MDS metamodel partition. The platform-independent partition comprises the concepts Model, Metamodel, M2M and M2T Transformation Definitions, and Text as first class entities instead of relations, and introduces the Tool entity as a base class for tools to be considered in MDS. In the figure, we consider Transformation Engines, Editors and Repositories as tools.

M2M and M2T Transformation Definitions represent Transformations (functions). We omitted the relation RepresentationOf to the element Transformation from the megamodel metamodel for visual clarity (see Figure 4.2). We introduce the relations MappedFrom and MappedTo as a shortcut to indicate that Transformation Definitions map Models as representations of Sets in the same manner as Transformations map Sets to one another as their domains and ranges. Consequently, we name the respective roles domainModel and rangeModel.

The platform-specific partition contains platform-specific generalisations. For example, Figure 4.6 illustrates that only Ecore Models are considered, which conform to the Ecore Metamodel from the EMF [Steinberg et al., 2009]. The metamodel recognises that Ecore Models can reference each other. MDS that conform to the MDS metamodel support only Xpand Templates and Java Source Code explicitly.

4.4 Architecture-Centric Model-Driven Software Development

Schmidt [2006] describes the use of MDSD to tackle the problem of platform complexity and the inability of General-Purpose Programming Languages

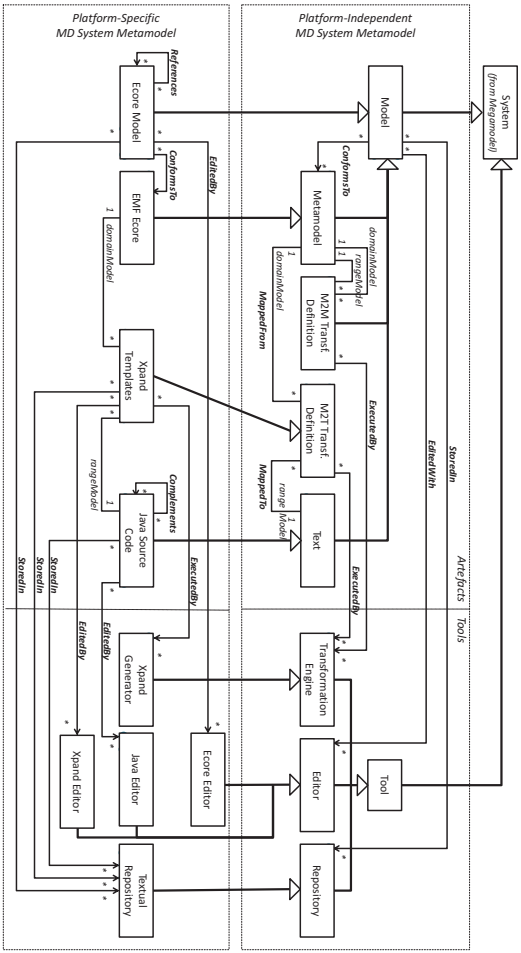

Figure 4.6. An exemplary MDS metamodel.

(GPLs) and libraries to alleviate this complexity.[3] Völter and Stahl [2006,

[3]We consider the term *platform* as synonymous to programming environment, as defined in Chapter 2.

pp. 21-22] conclude that the more a platform is used, the more code relates to platform usage. This leads to schematic and repetitive chunks of boilerplate code that is not specific to the application domain. Stahl and Völter refer to such code as *infrastructure code*. Architecture-Centric Model-Driven Software Development (AC-MDSD) is a pragmatic approach that is aimed at increasing the efficiency, quality, and reusability of/in software development by generating infrastructure code from the architecture of the software system [Völter and Stahl, 2006, pp. 21-22].

> ***Software architecture*** A software architecture is "[t]he fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [IEEE Architecture Working Group, 2000]

To employ AC-MDSD in the software development processes, three aspects must be considered: generative software architectures, role models, and reference implementations, cf. [Völter and Stahl, 2006, pp. 21-27].

A *generative software architecture* is an MDS that comprises a platform-independent model of the software architecture of a system to be developed, along with means to generate infrastructure code from the model. For example, the software architecture model can be modelled using a UML profile, and the code generator can be a template-based M2T transformation engine such as Xpand. Since a software architecture abstracts from the concrete platforms, generative software architectures can be used to develop entire software product families.

AC-MDSD favours the separation of the software architecture from domain-specific implementation and requires a tool infrastructure for generative software architectures. This leads to a natural separation of roles embodied into a *role model*:

- ▻ *MDS architects* develop and maintain the generative software architecture.
- ▻ *Application architects* develop the platform-independent model of the software architecture of the software system to be developed, along with transformations of these models to the platform.

- *Application developers* implement the domain-specific application logic. They are also responsible for integrating the application logic with the generated infrastructure code.

A *reference implementation* is an executable example application that is used as a blueprint for separating platform code from domain-specific application code. It uses the same platform as the generative software architecture and realises its concepts in its source code. Also, it represents a meaningful example from the application domain to be targeted. Its domain-specific application code can be used as a basis for the definition of M2T transformations.

AC-MDSD approaches can be identified by the following characteristics [Völter and Stahl, 2006, pp. 27-28]:

- Software system families: AC-MDSD enables the reuse of generative software architectures for the development of architecturally similar applications.
- Architecture-centric design: AC-MDSD focusses on the architecture of an application. It abstracts from those aspects of an application that do not represent domain-specific logic but recurring error-prone platform code.
- Forward engineering: AC-MDSD avoids roundtrip engineering. Changes in the generated code are not considered to be recognised by the architecture model.
- M2M for modularisation: The architecture model should be as abstract as possible without loosing the ability to generate platform code. M2M transformations and intermediate models can be used to modularize the overall code generation, but should be concealed from application developers.
- No final target metamodel: There is no direct target metamodel as a transformation target for code generation.
- Partial generation: AC-MDSD generates platform code. Domain-specific application logic must be supplemented in the target programming language.

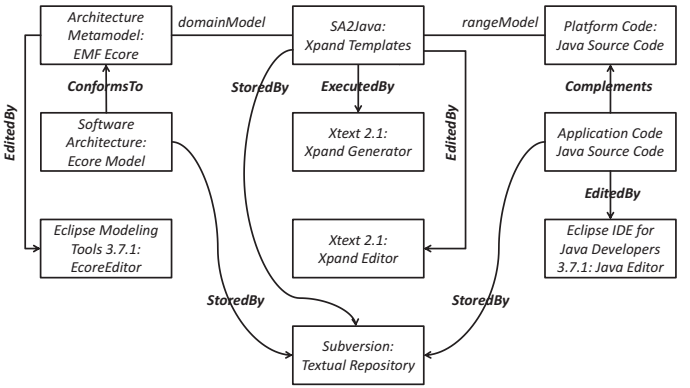

Figure 4.7. Example of AC-MDSE illustrated as an MDS.

▻ Separation of concerns: AC-MDSD separates the software architecture from domain-specific implementation. Programmers cannot break out of the boundaries of the generated platform code. The software architecture can not deteriorate on the program code level.

Figure 4.7 shows a simple MDS that conforms to the MDS metamodel shown in Figure 4.6. The MDS represents a generative software architecture and illustrates AC-MDSD. The main model is the Software Architecture of a software system to be developed. Its Architecture Metamodel is used as a domainModel for a software architecture-to-Java (SA2JAVA) transformation that generates Java Platform Code. Since this code is completely generated, it is not stored in the Subversion repository nor edited. However, it is complemented with domain-specific Application Code that is stored by the Subversion repository and provided by the programmers. The scenario assumes that the Architecture Metamodel is given and must not be edited during the software development process. The Software Architecture model is assumed to be edited by application architects with the Eclipse Modeling Tools 3.7.1 Ecore editor, and SA2JAVA Xpand Templates are developed by using the Xtext 2.1 Xpand editor. Application Code is provided by the programmers using the Eclipse IDE for Java Developers 3.7.1.

4.5 Domain-Specific Languages

DSLs play an important role in software engineering. They improve productivity by abstracting from the details of the underlying programming environments. While this can also be achieved with conventional mechanisms such as frameworks and libraries, DSLs often put abstractions of the solution space (i. e., the domain of computing technologies) in the centre of attention. Frameworks and libraries, on the other side, often only provide abstractions of the problem space (i. e., the respective application domain) [Schmidt, 2006; Visser, 2008].

There are numerous approaches in which DSLs are employed, including generative programming, software factories, domain-specific modelling, intentional software, language-oriented programming, and MDSD [Czarnecki and Eisenecker, 2005; Greenfield et al., 2004; Kelly and Tolvanen, 2008; Simonyi et al., 2006; Ward, 1994; Dmitriev, 2004; Völter and Stahl, 2006]. In the context of MDSD, the potential of domain-specific language has broadened. Firstly, DSLs are no longer focused on programming. Instead, they also consider comprehensive domain models for domain experts. Secondly, for the first time in computer science there are mature model-driven frameworks to specify and generate the infrastructure of a DSL given its language definition [Xte, 2011; Dmitriev, 2004]. As a result of this development, DSLs can now be regarded as a standard tool in the repertoire of software development [Visser, 2007].

> ***Domain-specific language (DSL)*** A DSL is a "high-level software implementation language that supports concepts [...] that are related to a particular application domain." [Visser, 2008]

In accordance with Visser [2008] we interpret the definition as follows: A DSL is a *language*, meaning it has a formally defined syntax and semantics. It is *high-level* in so far that it abstracts from the low level details of the programming environment. The notion of what is the respective programming environment is relative. *Software implementation* means that it contributes artefacts to the software development or maintenance process. Its concepts are *related* to a particular application domain since it dismisses generality for expressiveness in the domain in question.

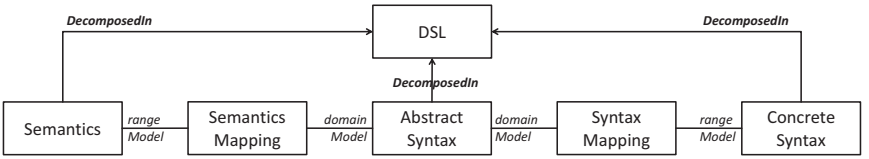

Figure 4.8. Overview of the parts of a DSL as a megamodel.

4.5.1 Components of a DSL

Figure 4.8 presents an overview of the term DSL in terms of MDSD. As shown, a DSL consists of the following parts [Völter and Stahl, 2006]:

- The *abstract syntax* defines the structure of a DSL and specifies the creation of conformant instances. The term abstract syntax is synonymous to *metamodel*.
- A *concrete syntax* of a DSL, or simply, its *notation*, provides a realisation of the abstract syntax. It describes the correct rendering of models that are created using the DSL and can be textual, graphical, or hybrid. A DSL can have more than one notation.
- The *semantics* of a DSL specifies its behaviour, or simply, the meaning of the metamodel of the language.

Between these main parts, a DSL can make use of the following mappings [Rivera et al., 2009]:

- The *mapping of the semantics* of a DSL with its abstract syntax is useful to reason about DSL models and to transform DSL models to semantic models for further analysis.[4] The definition of the semantic mapping represents a transformation, since it regards the language's semantics as its range, and the abstract syntax as its domain.

[4] In contrast to Rivera et al. [2009], we do not consider the semantic mapping as the definition of the semantics of a language itself but as a separate, desirable transformation that relates the semantics and the abstract syntax.

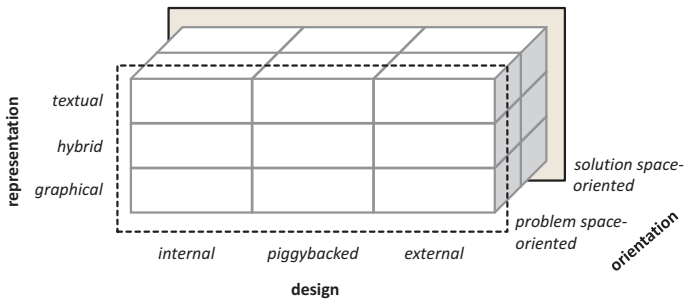

Figure 4.9. DSL categorisation.

▷ The *mapping of a concrete syntax* of a DSL with its abstract syntax is necessary to render DSL models. The definition of the semantic mapping represents a transformation, since it regards the language's concrete syntax as its range, and the abstract syntax as its domain.

4.5.2 Categorisation

The most important characteristics of a DSL are related to the questions of how the language represents itself to the user, how it is designed, and towards what kind of domain it is oriented. We discuss these three topics – representation, design, and orientation – in the following. Figure 4.9 presents an overview over the dimensions.

Firstly, the *representation* of the language can be distinguished as whether the languages notation is purely textual, purely graphical, or hybrid. Although there is no difference in expressiveness, graphical and textual notations have distinctive properties:

▷ A *textual notation* (serialisation) supports collaborative work naturally since modelling activities can be distributed across files and across blocks of text within separate files. Textual notations benefit from a broad tool support and the extensive experience of software engineers with textual programming languages. A typical shortcoming of textual notations is

the need for references to represent relations between individual blocks of text because of their serial structure.

- A *graphical notation* (rendering) can express multi-dimensionality by nature [Kleppe, 2008]. As a downside, collaborative modelling is often not supported by the language in terms of composability, and by tools in terms of versioning and storage. Also, layout can easily take more effort than modelling.

- A *hybrid notation* uses both textual and graphical elements. A common pattern is an overall graphical graph-based notation whose nodes and edges are annotated textually.

Secondly, DSLs can be distinguished by whether they are invented or developed on basis of a host language. A more detailed view on the *design* of a DSL is presented by Mernik et al. [2005].

- An *external* DSL is realised as an independent language from the programming environment it works with [Fowler, 2010]. They typically have a custom abstract syntax, but the re-use of existing languages as concrete syntaxes is not unusual, for example, by the use of the Extensible Markup Language (XML) as a notation.

- An *internal* DSL is realised by exploiting an existing host language and its corresponding language infrastructure. It produces valid code in terms of the host language but only uses a subset of the language's features [Fowler, 2010].

- A *piggybacked* DSL is realised by extending a host language with domain-specific elements. The challenge is to integrate these extensions with the host language (e. g., with a native extension mechanism) without producing proprietary dialects. A recent example of piggybacking is the Xbase language, an extensible DSL for expressions that supports static typing, type inference, closures, and operator overloading.[5] The language is implemented with the Xtext language workbench (see Section 4.5.5) on top

[5] http://www.eclipse.org/Xtext/#xbase

of the Java programming language platform, and can itself be embedded in DSLs developed with Xtext.

Finally, a DSL can be categorised by its *orientation*. This is a relative notion to the targeted domain.

- ▻ We consider a DSL as *solution space-oriented* when the DSL abstracts of the solution space, meaning that it provides language constructs over recurring idioms of the respective computing technologies and programming environments to be used, cf. [Schmidt, 2006].
- ▻ We consider a DSL as *problem space-oriented* when it abstracts of the problem space (the application domain) to be challenged, cf. [Schmidt, 2006]. This can be, for example, business processes, enterprise architectures, or telecommunications.

4.5.3 DSL Development

The provision of adequate DSLs plays an important role in the development of an MDS. If there are no adequate modelling languages abstract enough for a specific task or targeted domain, new ones can be developed. To do so, some general requirements must be considered:

1. DSL development requires knowledge of the application domain, knowledge of programming language development, and knowledge of MDS design.
2. The outcome of DSL development depends on the involved stakeholders, their collaboration, and the degree of sustainability provided by the tool infrastructure, maintenance processes, and standard utilisation [Mernik et al., 2005].
3. DSL development is not a sequential process [Mernik et al., 2005]. Decisions that are made in a certain phase of development can require revisions of earlier phases retroactively. These revisions can cascade over several phases of the whole development process, and over several layers of abstraction.

There are two fundamental approaches to DSL development frequently encountered in practice, *DSL-by-Examples* and *DSL-by-Domain-Engineering*.

If the intention is to abstract from a certain platform, a DSL can be developed by detecting patterns in the program code of one or more applications that are programmed against the platform. To do so, program code fragments are examined if one can differentiate between variable configuration parameters and redundant platform code. The latter is then used as a basis for M2T transformation templates, in which the configuration parameters must be inserted to generate code. The individual variabilities of the configuration parameters then form the basis for the DSL. Typically, the effort needed to model the configuration parameters with the resulting DSL is much lesser than implementing them in the identified program code patterns directly [Visser, 2007]. However, the applicability of the DSL is likely restricted to the technical platform, so that the conceptual portability across the boundaries of the platform is not guaranteed. Since the approach depends on the examination of concrete code fragments, it can be denoted as *DSL-by-Examples*. It is quite possible to apply the approach to the development of DSLs from model fragments, but less likely seen in practice than the development from code fragments.

If the intention is to primarily represent an application domain that may subsequently be mapped to one or more technical platforms, a DSL can be developed by decomposing the domain in question. To do so, various approaches can be used, for example, Domain Analysis and Reuse Environment (DARE), Domain Specific Software Architectures (DSSA), Family-oriented Abstractions, Specification and Translation (FAST), Feature-oriented Domain-Analysis (FODA), Megamodellierung, Ontology-based Domain Engineering (ODE), or Organization Domain Modeling (ODM) [Frakes et al., 1998; Taylor et al., 1995; Weiss and Lay, 1999; Kang et al., 1990; Favre, 2004b; Favre and NGuyen, 2005; Falbo et al., 2002; Simos and Anthony, 1998]. Regardless of the approach to be chosen, sufficient formality and formal semantics is emphasised to use domain models as a communication medium for domain experts on the one side, and as as means for domain-specific validation on the other. M2M and M2T transformations can usually be applied in subsequent processes to generate models of lower abstraction levels or platform code from the domain models developed with the DSL. Since in this case,

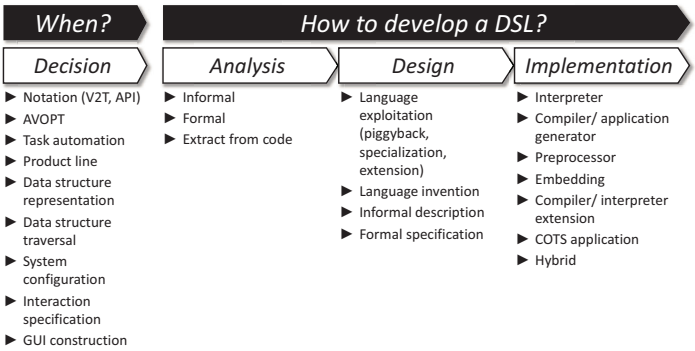

Figure 4.10. Patterns of DSL development according to Mernik et al. [2005]. The patterns are assigned to the development phases in which they can be applied.

DSL development starts with the decomposition of the application domain to be modelled, the approach can be denoted as *DSL-by-Domain-Engineering*.

Mernik et al. [2005] address the questions of when and how a DSL should be developed with a collection of design patterns. According to the presented categorisation of the patterns, the overall development process of a DSL consists of the phases decision, analysis, design and implementation. These phases can be used as a skeleton to construct a concrete method for DSL development.

In the following, we present the different phases and patterns of DSL development. The overview is compiled from Mernik et al. [2005], and extended by additional references as stated. Figure 4.10 shows an overview of the phases and patterns.

When to develop a DSL?

DSL development starts with the decision if a DSL should be developed at all. For example, it can be more effective to develop an internal DSL in the form of a program library instead of an external DSL when the domain in question is immature or when there is only few knowledge about the domain. The focus can then be placed on concept exploration and prototyping, instead of

domain-specific validation and productivity. Patterns that can be identified to justify DSL development comprise the following:

- ▻ Notation: The DSL supports a domain with a new or existing concrete syntax. Notable sub-patterns are the provision of a textual notation for an otherwise graphical language (e. g., to support collaborative work and composition of large programs), and the conversion of an Application Programming Interface (API) to a full-fledged language whose correct use can be validated.
- ▻ Analysis, Verification, Optimization, Parallelization, and Transformation (AVOPT): Domain-specific analysis, verification, optimisation, parallelisation, and transformation of domain models are facilitated.
- ▻ Task automation: Repetitive and redundant activities are eliminated.
- ▻ Product line: Members of a software product line are specified.
- ▻ Data structure representation: Common data structures are defined.
- ▻ Data structure traversal: Repetitive and complicated data structure traversals are specified.
- ▻ System configuration: System configuration is simplified.
- ▻ Interaction specification: The interactions of independent entities are formalised.
- ▻ Graphical User Interface (GUI) construction: User interface construction is simplified.

How to develop a DSL?

After the decision phase, the problem domain is identified and knowledge of the problem domain is gathered to develop a thorough domain understanding. To do so, various sources of knowledge are examined, for example, technical documents, existing source code, expert interviews, and surveys. These activities form the analysis phase.

- ▻ Informal: The problem domain is analysed informally.
- ▻ Formal: A method is used for domain analysis that produces a *domain model* consisting of a domain definition, a domain terminology, a description of individual domain concepts, and feature models that describe the commonalities and variabilities of domain concepts along with their interdependencies. Examples for formal analysis methods are DARE, DSSA, FAST, FODA, Megamodellierung, ODE, or ODM [Frakes et al., 1998; Taylor et al., 1995; Weiss and Lay, 1999; Kang et al., 1990; Favre, 2004b; Favre and NGuyen, 2005; Falbo et al., 2002; Simos and Anthony, 1998].
- ▻ Extract from code: Domain knowledge is gathered by examining legacy code manually, by software tool-based inspection, or both.

The following design phase can be regarded by the question of whether the DSL to be developed is related to already existing languages or not, and the question if the language description is accomplished formally or informally.

- ▻ Language exploitation: The DSL to be developed uses an existing GPL or DSL. The existing language can be restricted (specialisation), extended, or parts of it are used as a foundation for domain-specific features (piggyback).
- ▻ Language invention: A new DSL is developed from scratch without using an existing language and the corresponding language infrastructure.
- ▻ Informal description: The DSL in question is described informally, typically using natural language and DSL examples for illustration.
- ▻ Formal description: The DSL in question is described formally using existing formalisms for both syntax and semantics definition. Slonneger and Kurtz [1995] present a broad spectrum of semantics definition techniques.

The last phase is the implementation of the DSL. There are several patterns that can be applied for DSL implementation.

- Interpreter: DSL constructs are interpreted. The pattern is recommended for DSLs with dynamic behaviour and when execution speed is not relevant. The pattern is simpler to implement than compilation, can be easily extended, and facilitates control over the execution environment.

- Compiler/application generator: DSL constructs are translated into the constructs of a target language and corresponding library calls. The pattern enables the static analysis of DSL models.

- Preprocessor: DSL constructs are translated to constructs in a target language by a preprocessor. Static analysis is not performed at the domain level. Possible errors in generated code can only be detected in the target language or at run-time [Deursen et al., 2000]. An important subpattern is the expansion of macros (specifications that are independent of the target language whose syntactical correctness is not guaranteed at design-time).

- Embedding: DSL constructs are embedded in a host language by defining domain-specific concepts such as data structures with the features offered by the host language. The language infrastructure of the host language can be reused *as is*, but the expressiveness of the DSL is limited to the host language, and possibly compromised to a certain degree with respect to the targeted problem domain [Deursen et al., 2000]. The most common form of embedding is application libraries.

- Compiler/interpreter extension: A compiler or interpreter of the target language is extended with domain-specific optimisations or code generation facilities.

- Commercial Off-The-Shelf (COTS) application: Existing tools or notations are applied to the domain in question.

In addition to the implementation patterns, Mernik et al. provide guidelines when to choose a certain pattern for the implementation of a DSL, see Figure 4.11. The figure takes reference to the described patterns and is otherwise self-explaining. Mernik et al. provide a detailed discussion of the patterns to be chosen.

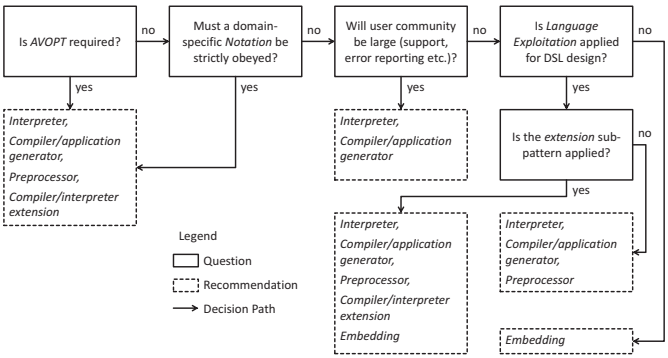

Figure 4.11. DSL implementation gudelines according to Mernik et al. [2005]. The figure illustrates when to choose which pattern for DSL implementation. Referenced patterns are shown in italics.

4.5.4 DSLs in Practice

Several surveys on the use and development of DSLs identify them as an established practice in software engineering, including [Mernik et al., 2005; Spinellis, 2001; Deursen et al., 2000]. Examples comprise the following:

FORTRAN is a programming language for scientific computing developed by Backus [1958] in the late 1950s. It was originally created as an abstraction of low-level machine code that resembles the mathematical notation. The example of FORTRAN illustrates the relativity of the high-level abstraction characteristic of DSLs.

Structured Query Language (SQL) is a DSL for queries over the relational database model [Chamberlin and Boyce, 1974; Codd, 1970]. Its use in general purpose programming language libraries often take the form of string literals, illustrating the shortcoming of libraries to provide means for syntactical and domain-specific validation.

dot (for graph drawing) and make (for incremental software building) are examples of DSLs used in UNIX environments [Bentley, 1986].

XML is a popular DSL for the textual serialisation of structured data

[Bray et al., 2008]. It is commonly used for implementation- and platform-independent data exchange between computer systems and can be regarded as a least common denominator in human-computer information exchange.

Scala, LISP, and Ruby are examples for GPLs that provide sufficient semantic and syntactic flexibility to construct internal DSLs. Scala integrates functional and object-oriented features in a single GPL. As Ghosh [2011] summarises, the language provides several facilities that renders it an excellent basis for internal DSL development: type inferencing, an extensible object system, modularity, lexically scoped open classes, implicit parameters, and structural types. LISP allows a kind of bottom-up program development that favours the modification of the language towards the problem domain, for example by developing more abstract operators, that can be regarded as an internal DSL [Graham, 1993]. Ruby has also a strong orientation towards internal DSLs. For example, the popular Ruby framework Rails can be regarded as a collection of DSLs [Fowler, 2010, p. 28].

Business Process Model and Notation (BPMN) is a DSL for specifying business processes [Axway et al., 2010]. It can be regarded as a de facto-standard in the business domain to describe the behaviour of socio-technical systems on different levels of granularity. The specification defines a formal metamodel and a graphical notation, but lacks a formal semantics.

4.5.5 Language Workbenches

A recent trend is that of language workbenches. A language workbench is a programming environment to develop DSLs, together with tools for using them efficiently [Fowler, 2010, p. 22]. These tools can include mechanisms for compilation/ interpretation/code generation, model transformation, validation, error handling, and syntax highlighting, for example. In the context of MDSD, they can be regarded as mature model-driven frameworks to specify and generate the infrastructure of an MDS given a set of DSL language definitions. Language workbenches can be used for two goals: Firstly, they can be used for the effective development of external DSLs with practical tool integration. Since this tool infrastructure is generated as far as possible, they are especially suited for the rapid prototyping of DSLs. Secondly, in the context of MD systems they can be used to develop and

maintain the tool infrastructure of an MD system. Each of the DSLs used in an MDS represents the definition of a *viewpoint* for the MDS, cf. [IEEE Architecture Working Group, 2000].

The Xtext language framework is a popular example of language workbenches. It is a mature parser-based language workbench for textual DSLs realised as a sub project in the Eclipse Modeling Project [Xte, 2011]. The overall approach is based on the definition of the abstract and concrete syntax of a DSL as a whole in a language similar to the Extended Backus-Naur Form (EBNF), and subsequent automatic generation of the DSL infrastructure, including an EMF Ecore model and stubs for validation, scoping, DSL tooling, and M2T transformation. Xtext's main benefits are the integration in the Eclipse ecosystem and the thorough use of text files as a serialisation mechanism for all kinds of artefacts. The first aspect, Eclipse ecosystem integration, enables to use complementary technologies (e. g., M2M transformations) and is crucial for setting up a consistent MDS tool chain. The second, thorough use of text files, is of major practical importance since text files can be collaboratively edited (file-wide and block-wise), stored with mature version management systems such as Subversion (SVN),[6] and easily exchanged between tools.

4.5.6 Benefits and Limitations

Domain-specific languages have several benefits and drawbacks. These must be carefully considered in addition to the decision patterns described by Mernik et al. [2005]. The following lists of benefits and limitations summarises the work of Deursen et al. [2000] and Fowler [2010]. Firstly, we subsume the benefits of DSLs:

- Knowledge preservation: DSLs embody domain knowledge and are to a large extent self-documenting. If properly serialised, they can be easily stored and put under version control.
- Reuse: Domain models created with DSLs can be reused to establish software product lines.

[6] http://subversion.tigris.org/

- ▻ Portability: Platform-independent DSLs can be mapped to various target platforms and programming environments.

- ▻ Abstraction: DSLs allow solutions to be expressed in terms of the problem domain instead of technical frameworks.

- ▻ Participation: Domain experts can understand, validate, manipulate, and create domain models using DSLs. When DSLs are aimed at a non-technical problem domain, even non-programmers can participate in domain modelling.

- ▻ Validation: DSLs enable to validate solutions on the domain-level instead of the implementation level.

Secondly, the drawbacks of DSLs are additional (up-front) costs, a potential loss of efficiency, and *ghettoisation*.

- ▻ Costs: There are various costs related with DSLs that must be considered. For example, DSLs must be designed, implemented, and maintained separately in addition to the software system(s) to be developed. Beforehand, the knowledge to do so must be acquired. Also, learning and teaching DSL usage requires additional effort.

- ▻ Potential loss of efficiency: Hand-crafted code can be optimised locally while DSLs favour general implementation patterns and code generation. A possible solution can be provided by fine-tuning the code generation, and by combining code generation with domain-specific auto-tuners.

- ▻ Ghettoisation: An overly strong commitment to a DSL that is developed and used only in-house makes it difficult to benefit from advances in related standards and best practices. It can also mislead to reimplement existing frameworks and libraries for the sake of conformance with in-house tooling. This ghettoisation can be regarded as a variant of the ominous *not-invented-here* anti-pattern.[7]

[7] We understand the *not-invented-here* anti-pattern as the disbelief that in-house implementation is always a more adequate solution than using existing comparable software components.

4.6 Summary of Model-Driven Software Engineering

MDSD emphasises to develop software by generating code from formally specified models as the main development artefacts.

A model is a representation of a system under study, where the model is itself a system, simpler than the system under study, and which can give answers instead of the system under study that also hold for the system under study.

A metamodel is a model of a set of conformant models.

Model transformations are computable mappings that translate source models into target models. In practice, it is distinguished by what kind of target models are used. M2M transformations create or manipulate their targets as instances of explicit metamodels, and M2T transformations consider their target simply as strings. M2T transformations are a special case of M2M transformations if a metamodel for the target programming language is available.

MDSs are systems whose artefacts refer to models, metamodels, and transformations, and the tools that are used in model-driven software projects. Their purpose is to handle and maintain the individual artefacts and the artefacts' interrelations to automate the generation of the target systems as far as possible. MDSs are a key factor to set up software product families and product lines.

AC-MDSD is a pragmatic approach that is aimed at increasing the efficiency, quality, and reusability of/in software development by generating infrastructure code from the architecture model of the software system.

DSLs are high-level software implementation languages that support the concepts of particular problem domains. They improve productivity by abstracting from the details of the underlying programming environments.

A DSL consists of an abstract syntax, a concrete syntax, and a description of the semantics of the language. The abstract syntax (metamodel) defines the structure of a DSL and specifies the creation of conformant instances. The concrete syntax (notation) provides a realisation of the abstract syntax that describes the correct rendering of conformant models. The semantics of a

DSL specifies its behaviour/the meaning of the metamodel of the language.

A DSL can be categorised by the realisation of its notation (textual, graphical, or hybrid), the utilisation of a host language (external, internal, piggybacked), and by its domain orientation (solution space-oriented or problem space-oriented).

The development of a DSL can be addressed by considering the phase model of Mernik et al. [2005]. It consists of the phases decision, analysis, design and implementation and comprises several design patterns to be employed within these phases. The phase model can be used as a skeleton to construct a concrete method for DSL development.

A language workbench is a programming environment to develop DSLs, together with tools for using them efficiently. In the context of MDSD, it can be regarded as a mature model-driven framework to specify and generate the infrastructure of an MDS given a set of DSL language definitions. Language workbenches can be used for the rapid prototyping of DSLs as well as to develop and maintain the tool infrastructure of an MDS. Each of the DSLs used in anMDS represents a viewpoint on the MDS.

The benefits of DSLs are the preservation of knowledge, the reuse of domain models, the portability of platform-independent models, abstraction (expressiveness in terms of the problem domain), and the increased participation of domain experts. The drawbacks of DSLs are design, implementation, maintenance, and learning costs, the potential loss of efficiency, and ghettoisation (i. e., an overly strong in-house commitment to one or more DSLs).

Part II

Coordination Engineering with SCOPE

Chapter 5

Coordination Engineering

Coordination design considers a parallel program as a coordinated system. Its goal is to produce the specification of the system as a whole consisting of coordinated components [Ortega-Arjona, 2010, pp. 315-327]. In the context of Model-Driven Software Development (MDSD), a coordination model is an artefact that represents the software architecture of a program to be developed from a behavioural viewpoint. As such, coordination models can be regarded as a special kind of the specification of a system. Combined with model-driven techniques they do not only serve the purpose of documentation but also that of realisation: models can be analysed, validated, communicated, used as a source for code generation, and possibly even be executed.

Coordination Engineering is an approach that combines coordination design with model-driven software development. Its goal is to consider the coordination model of a parallel program as the first and most important artefact in the development process and to use model-driven techniques to enforce the coordination model in program code. The target domain of Coordination Engineering is therefore *parallel software system engineering*.

Figure 5.1 presents the metamodel of Coordination Engineering. As illustrated, it is a specialisation of the Model-Driven System (MDS) metamodel from Section 4.3. The central artefact of Coordination Engineering is the development of a Coordination Model of the Program to be developed. This is done by so-called Application Engineers using an appropriate Coordination Workbench. The Coordination Model conforms to a Coordination Metamodel that is part of the Coordination Workbench, as a part of a coordination DSL (not shown). The Coordination Workbench is maintained by so-called Toolsmiths. Valid Coordination Models can then be transformed into Coordination Code for

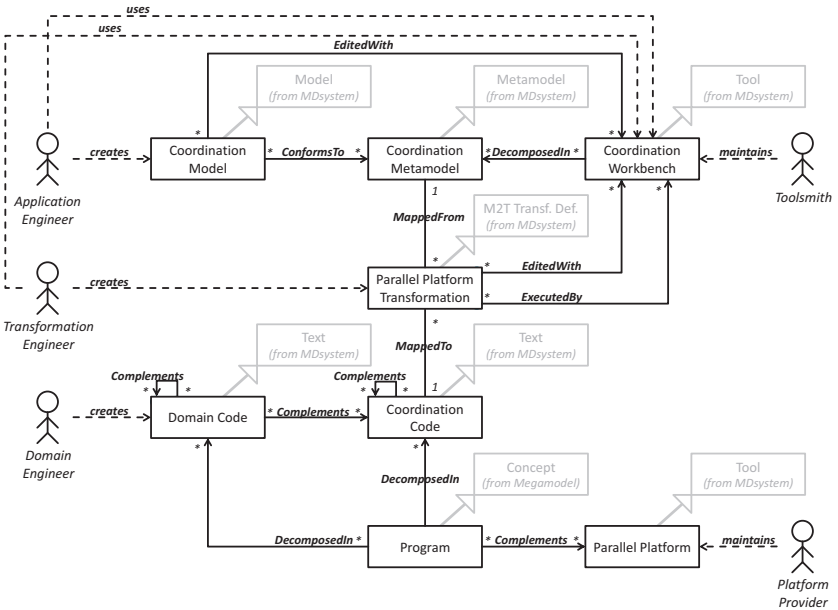

Figure 5.1. The Coordination Engineering metamodel as a specialisation of the MD system metamodel from Section 4.3.

a specific Parallel Platform. The necessary Parallel Platform Transformations are provided by so-called Transformation Engineers. They use the Coordination Workbench to both edit and execute the transformations. The generated Coordination Code represents the concurrency infrastructure of the Program, and does not necessarily comprise only program language code but can also include accompanying coordination-related artefacts such as run scripts and configuration files, if desired. Ideally, the generated Coordination Code is executable, but does not perform any domain-specific calculations. These are provided in the form of Domain Code that complements the Coordination Code and is conceptually regarded as sequential. Domain Code is provided by Domain Engineers. Note that the figure does not show the tools that are used to provide Domain Code, since the emphasis of Coordination Engineering

is on the Coordination Model and not on programming. Finally, the resulting Program can be executed by the Parallel Platform that is maintained by so-called Platform Providers.

Coordination Engineering shows the following characteristics:

1. Like coordination design, Coordination Engineering emphasises a top-down problem decomposition and is applicable to different problem domains and target platforms.
2. The artefacts of Coordination Engineering form a special kind of MDS. This MDS has to be maintained as a separate system in its own right, apart from the parallel program to be developed.
3. Established roles in the software development process – software architect and programmer – are revised and extended to support a clear separation of activities that relate to the establishment of coordination from those that establish domain-specific computation.

In the following, we discuss the artefacts and tools that are considered in Coordination Engineering, the role model, the individual stages of Coordination Engineering, the placement of Coordination Engineering, and its benefits and limitations.

5.1 Artefacts

Coordination Engineering considers the following artefacts:

- ▻ Coordination model
- ▻ Coordination metamodel
- ▻ Coordination workbench
- ▻ Parallel platform transformations
- ▻ Coordination code
- ▻ Domain code

▻ Program

▻ Parallel platform

The *coordination model* is a platform-independent representation of the program that considers only how the individual components of the program are coordinated.[1] The focus of the coordination model lies on the behavioural viewpoint on the software architecture of the program. The coordination model is explicit, meaning that it is an artefact that can be edited with the coordination workbench.

The *coordination metamodel* is the representation of the set of coordination models. It allows to check if a coordination model is valid, meaning that the coordination model conforms to the metamodel. The coordination metamodel is also an explicit artefact that can be manipulated by appropriate modelling environments. Since it represents a high-level abstraction over concurrent program code, it introduces non-determinism when it is needed, instead of cutting it away when it is not needed.

The *coordination workbench* is an Integrated Development Environment (IDE) for coordination models and parallel program transformations. Its goal is to foster the establishment of product families or product lines for parallel programs, rather than individual program development. The focus on product families or lines depends on the kind of artefact reuse: Reusing coordination models results in product lines, and reusing parallel program transformations results in product families. As a consequence, its lifecycle is longer than an individual program to be developed. The coordination workbench is the constitutive artefact of the Coordination Engineering MDS. It is a separate system in its own right that must itself be maintained. Also, its evolution can cascade over all other artefacts of Coordination Engineering, raising questions of co-evolution between the different artefacts of the MDS (e. g., between the coordination metamodel and transformations).

Figure 5.2 shows the components of a coordination workbench. It essentially consists of a coordination-specific Domain-Specific Language (DSL)

[1] For the sake of clarity, Figure 5.1 does not show the Representation Of relations that exist between Coordination Model, Domain Code, Coordination Code, and the Program. The Representation Of relation is explained in the paragraphs of this section.

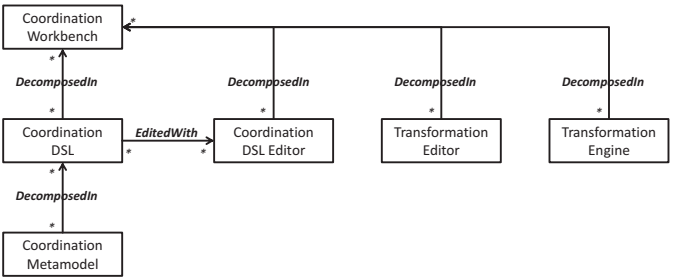

Figure 5.2. The coordination workbench metamodel.

based on the coordination metamodel to foster the definition of coordination models, a complementing editor for the DSL that provides coordination-specific mechanisms for validation and error handling, an editor for parallel program transformations, and a transformation engine to execute these transformations. Additional components of the coordination workbench are means for further analysis of coordination models, model simulators, and Model-to-Model (M2M) transformation support to facilitate interoperability.

Parallel platform transformations specify how coordination models are transformed into coordination code. They are edited with a transformation editor and executed by a transformation engine as parts of a coordination workbench. Typically, parallel platform transformations are Model-to-Text (M2T) transformations that specify how a coordination model is transformed into coordination code. Parallel platform transformations are the main means to enforce the coordination model in program code.

Coordination code uses the programming platform's specific concurrency constructs to express coordination. Since it is fully generated by the coordination workbench it is guaranteed that it is represented by the respective coordination model. Domain-specific computation is only declared so that the implementation of domain-specific logic must be complemented by appropriate domain code.

Domain code complements coordination code with domain-specific logic. From a conceptual point of view, it is considered as sequential, although domain engineers may use concurrency constructs within it. Domain

code is not further decomposed into coordination and computation. The responsibility for it lies exclusively in the hands of the domain engineer – guaranteeing its correctness and conformance to the interfaces provided by the coordination code. Both coordination code and domain code form the program.

The *parallel platform* executes the resulting program. It is synonymous to the term programming environment since it represents the entirety of hardware and software, including operating system, programming languages, compilers, runtime libraries etc., that is available to the roles in Coordination Engineering.

5.2 Role Model

There are several roles involved in Coordination Engineering. These are:

- ▻ Application Engineer
- ▻ Transformation Engineer
- ▻ Domain Engineer
- ▻ Toolsmith
- ▻ Platform Provider

The *application engineer* designs the coordination models and considers architectural coordination patterns to be applied. He is also responsible for specifying the target platform. The role focusses on the behavioural aspects of the software system. Therefore, an application engineer can be regarded as a software architect that gained experience in coordination modelling. The role is tightly related to the transformation developer since transformations may have to be adapted for reuse or developed from scratch.

The *transformation engineer* provides M2T transformations as a concrete embodiment of coordination. These transformations map coordination models that conform to the metamodel of a coordination DSL to the parallel

platform. The role is tightly related to the application engineer such that the application engineer specifies the target platform and technologies to be used for coordination implementation. Transformation engineers are experts in concurrent programming on the respective target platform technologies who gained experience in model transformation development.

The *domain engineer* develops sequential code as a concrete embodiment of application-specific computation. The role is concerned with designing and implementing application-specific functionality that is required by the generated coordination code in order to represent a complete software product. Domain engineers are programmers that gained experience in implementing thread-safe sequential code.

The *toolsmith* provides and maintains the coordination workbench. The role is responsible for the development and maintenance of the individual components and artefacts of the coordination workbench (e. g., DSL metamodels, mechanisms for validation and error handling, interoperability mechanisms), as well as for the amalgamation of these components and artefacts into a consistent whole. As a consequence, the toolsmith's interest is in the co-evolution of the components and artefacts of the coordination workbench. Toolsmiths are MDSD experts that gained experience in coordination modelling.

The *platform provider* provides and maintains the parallel platform. The role is responsible for the entirety of hardware and software that is available to the other roles, including operating system, programming languages, compilers, runtime libraries etc. Platform providers can be regarded as administrators from the other roles' point of view.

Note that other roles that are not directly associated with Coordination Engineering are not considered. For example, we omitted roles that are concerned with complementary issues such as continuous build management and test infrastructure management.

5.3 Application Processes

The application of Coordination Engineering can be considered as a set of processes executed by the different roles. The roles of interest are the

application engineer, the transformation engineer, and the toolsmith. The domain engineer and the platform provider are left out since their activities are more related to programming in general, or platform maintenance/administration respectively, than specific to Coordination Engineering. In the following, we present processes for the relevant roles. The processes should not be misunderstood as waterfall processes, as they only illustrate the ideal and most relevant dependencies between the individual activities.

5.3.1 Application Engineering

From the viewpoint of the application engineer, the application process consists of the development of a coordination model in an early stage of the overall software development process (*coordination first*). The approach is a variant of the *coordination design* stage in the Parallel Software Design method described by Ortega-Arjona [2010, pp. 299-357], and can be viewed as a coordination-specific elaboration of the "Formal Modeling/Design" activity within the Application Development Thread of the generic MDSD Process Building Blocks [Völter and Stahl, 2006, p. 261].

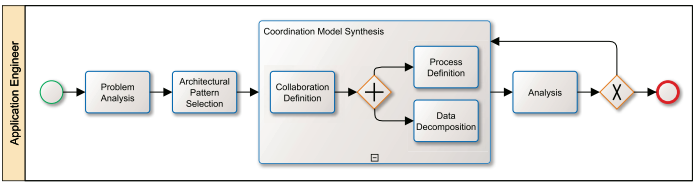

Figure 5.3. The application engineer's process for Coordination Engineering.

Figure 5.3 shows an overview of the different activities of the process using a top-level Business Process Model and Notation (BPMN) diagram.

First, the *problem analysis* establishes an understanding of the problem – usually in the form of the data to operate on and the general (often mathematical) algorithm to use the data. The outcome of the problem analysis is the *Specification of the Problem*, which ideally consists of an overview of the development project in form of an *executive summary*, a *problem statement* that describes the problem in user terms and lists the requirements,

a description of the *data* to operate on and the *algorithm* to be applied, information about the *platform/programming environment*, as it can strongly affect the performance of a concurrent program, and finally, quantified *performance and cost requirements* to be expected from the concurrent program to be developed [Ortega-Arjona, 2010, p. 309].

Second, one or more *architectural patterns* are selected for the coordination model. An architectural pattern encapsulates "design experience of coordination in parallel software design" [Ortega-Arjona, 2010, p. 316]. To execute the activity, the following selection process can be used.[2] Concrete architectural pattern systems are presented by Ortega-Arjona [2010, pp. 27-93], and Mattson et al. [2004, pp. 57-120].

1. Determine the *category of parallelism* based on the Specification of the Problem from the problem analysis activity. Archetypical categories comprise task parallelism, data parallelism, and activity parallelism [Ortega-Arjona, 2006].

   - ▻ *Task parallelism* focusses the decomposition of the algorithm into disjoint tasks that are executed simultaneously. Conceptually, all tasks start simultaneously, but the task activity can depend on the availability of required data. Data decomposition is secondary.
   - ▻ *Data parallelism* focusses the decomposition of the data into smaller parts, ideally of equal size. Algorithm decomposition is secondary and typically concerned with associating a set of operations to the individual data chunks. Conceptually, operations start simultaneously at the same time, but their activity is suspended when a certain piece of data cannot be processed until another activity makes the piece available.
   - ▻ *Activity parallelism* focusses the decomposition of the data *and* the algorithm into tasks that are not committed to certain chunks of data and can therefore operate in any order. Conceptually, tasks start

[2] The process is a variant of the architectural pattern selection process described by Ortega-Arjona [2010, pp. 321-322]. It omits the analysis and specification of the problem as the first process activity since it essentially is a summary of the problem analysis stage of the parallel software design method

simultaneously, each taking an arbitrary chunk of data, operate on it, and produce a result until all data has been processed. The operation on a chunk of data can require coordination among tasks.

2. Determine the *category of processing*. Categories comprise homogeneous processing and heterogeneous processing [Ortega-Arjona, 2010, pp. 320-321].

   - ▻ *Homogeneous processing* refers to multiple instances of components with the same behaviour. Homogeneous systems are typically composed of a large number of simple, identical components that communicate through data exchange operations.
   - ▻ *Heterogeneous processing* refers to different components with defined relations. Heterogeneous systems are typically composed of a small number of different components that communicate via function calls.

3. Identify appropriate architecture pattern candidates by comparing the description of the patterns with the results of step one and two.

4. Select an architectural pattern from the candidates for application. The classification of architectural patterns provided by Ortega-Arjona [2010, p. 321] can support the decision, see Table 5.1.

Third, the coordination model of the program is defined in terms of sufficiently detailed software components: *Collaboration* declares the process components that collaborate in the program to be developed. Then, *process definition* defines the processes that implement the behaviour of each participating component. Finally, *data decomposition* defines the data objects considered in the program.

Subsequently, *analysis* determines the performance and cost requirements from the problem specification and decides whether the coordination model serves its purpose or whether previous activities must be re-iterated. Ideally, this activity is guided by runtime analysis, which is possible when the coordination code generated from the coordination model is complete enough to be executed. For example, wait statements that were generated

Table 5.1. Architectural pattern classification according to Ortega-Arjona [2010, p. 321].

Architectural pattern	*Parallelism*			*Processing*	
	task	data	activity	homogeneous	heterogeneous
Parallel Pipes and Filters	X				X
Parallel Layers	X			X	
Commun. Seq. Elements		X		X	
Manager-Workers			X	X	
Shared Resource			X		X

from cost estimates in the coordination model can serve as place-holders for domain code and can be used for early bottleneck analysis.

We omitted *documentation* as a separate phase since we consider it as a cross-cutting activity that builds up across all phases in the process. Section 10.3 illustrates the application process from the viewpoint of the application engineer at the example of Mandelbrot set visualisation and Point-Feature Label Placement (PFLP).

5.3.2 Transformation Engineering

From the viewpoint of the transformation engineer, the application process consists of the development of M2T transformations. The process is a variant of the *DSL-by-Examples* approach that focuses on the parallel platform transformations (see Section 4.5.3), and can be viewed as a coordination-specific elaboration of the "Transformations" partition of the Domain Architecture Development Thread of the generic MDSD Process Building Blocks [Völter and Stahl, 2006, p. 255].

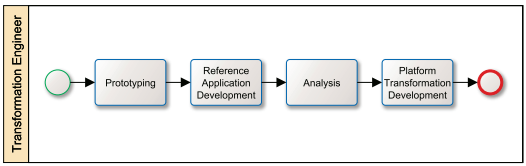

Figure 5.4. The transformation engineer's process for Coordination Engineering.

Figure 5.4 shows an overview of the different activities of the process using a top-level BPMN diagram.

Prototyping refers to provide one or more prototype applications based on the considered parallel platform. The intention is to gain experience in the platform and its concurrency features. The concrete application functionality of the prototype(s) is irrelevant. The outcome of the activity, concurrent prototype application(s), can be used as an input for the subsequent activity.

Reference application development provides an application that represents a relevant example for implementation on the parallel platform. Thereby, the reference application serves two purposes. First, it demonstrates how coordination code and domain code blend together to form a complete, executable program. Second, it provides the code basis from which the parallel platform transformations are derived. While the concrete application functionality is of minor interest, particular emphasis should lie on a sufficient coverage of the parallel platform's concurrency features. Thereby, the transformation engineer has to put considerable effort in adhering to the coordination model of the reference application. It must be strictly followed to provide a clear separation of domain code from coordination code. Incremental refinement and restructuring is likely to become necessary, especially, when the reference application is based on a previous prototype or on existing legacy code. This highlights the incremental nature of the overall process. However, the use of more fine-grained coordination mechanisms such as parallel design patterns and programming idioms are completely in the responsibility of the transformation engineer.

The *analysis* activity provides an overview of the relevant coordination code fragments in the program code. To do so, the program code of the

reference application is examined for repeating code patterns that represent fragments of the (implicit) coordination model of the application. While the configuration parameters of the code patterns represent a possible input for the domain analysis performed by the toolsmith, the code patterns itself represent the basis for platform transformation development.

Finally, in the *platform transformation development* activity, the transformation engineer produces the transformation to the parallel platform. This is usually an M2T transformation in the form of a set of templates. These describe which artefacts are generated from models that conform to the coordination metamodel. Therefore, the activity relies on the availability of an explicit coordination metamodel provided by the toolsmith. To produce the templates from the previously identified coordination code fragments, the transformation engineer replaces smaller code fragments that were identified as configuration parameters by expressions of the template language that refer to the coordination metamodel. Additional partitioning of templates can become necessary to untangle intricate dependencies, for example, type reference resolution with recursive invocations.

5.3.3 Tool Engineering

From the viewpoint of the toolsmith, the application process consists of the development of the coordination workbench. The process is a variant of the *DSL-by-Domain-Engineering* approach that focuses on coordination modelling (see Section 4.5.3). It can be viewed as a coordination-specific elaboration of the "Domain" partition of the Domain Architecture Development Thread of the generic MDSD Process Building Blocks [Völter and Stahl, 2006, p. 255].

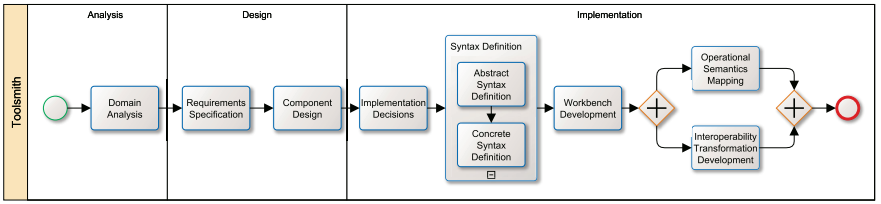

Figure 5.5. The toolsmith's process for Coordination Engineering.

Figure 5.5 shows an overview of the different activities of the process using a top-level BPMN diagram. We describe the process as a set of activities aligned to the "how to develop" phases of the phase model of Mernik et al. [2005] for DSL development. The reason is that the coordination workbench is essentially an embodiment of a DSL for coordination modelling, along with complementary tooling for transformation support. Consequently, the individual phases (analysis, design, and implementation) encourage the use of appropriate DSL development patterns as discussed by Mernik et al. [2005]

The *domain analysis* delivers an overview of the concepts of the considered domain. The outcome of the domain analysis is a conceptual coordination model that comprises the following artefacts: A definition of the domain, an informal description of the domain concepts, the definition of the operational semantics of the concepts, and a domain terminology in the form of a vocabulary, an ontology, or a domain metamodel. For Coordination Engineering with Space-Coordinated Processes (SCOPE), the domain analysis is described in Chapter 6 and in this chapter. The metamodel of Coordination Engineering is presented at the beginning of this chapter, see Figure 5.1.

Next, the *requirements specification* specifies the coordination workbench requirements that were implicitly exposed in the domain analysis. For Coordination Engineering with SCOPE, we describe the requirements that must be met by a coordination workbench for SCOPE in Section 7.2.

In the subsequent *component design* activity, the individual components of the coordination workbench, along with their interdependencies, are described. For the SCOPE coordination workbench, we describe the component

design in Section 8.2.

The first activity of the implementation phase is the *implementation decision* activity. It documents the decisions that were made to implement the component design. This includes to document how the individual requirements are addressed and which tools and third-party components are used to reduce implementation costs. Differences to the component design must be discussed and justified. The implementation decisions for the SCOPE coordination workbench are presented in Section 9.2.

The next activity is the *syntax definition*. It consists of a definition of an abstract and at least one concrete syntax (metamodel and notation) of the coordination DSL.

First, the *abstract syntax definition* activity defines the abstract syntax/metamodel of the coordination DSL embodied in the coordination workbench. This metamodel is the formal representation of the commonalities of all coordination models that conform to the coordination DSL. It allows to check if a given model is a valid element of the DSL or not. There are several languages and technologies that can be used to define the metamodel of the coordination DSL. Examples are the Object Management Group (OMG) standards Meta Object Facility (MOF) and Unified Modeling Language (UML), the Eclipse Modeling Framework (EMF) with its Ecore metamodel, Extensible Markup Language (XML) and XML Schema Definition (XSD), the Eclipse Xtext Language Development Framework, and the JetBrains Meta Programming System (MPS).[3] An additional part of the abstract syntax/metamodel is the definition of constraints that restrict the use of syntax elements. They can be defined with the Object Constraint Language (OCL) or the Check language of the Eclipse Xtext Language Development Framework, for example. For the SCOPE coordination workbench, we describe the abstract syntax specification in Section 9.2.

Second, the *concrete syntax definition* specifies the concrete syntax/notation of the coordination DSL. It defines the textual and/or graphical appearance of the DSL. Thereby, it is possible to define several concrete syntaxes for a single abstract syntax. This allows to address different

[3] See http://www.omg.org/mof/, http://www.uml.org/, http://www.eclipse.org/modeling/emf/, http://www.eclipse.org/Xtext/, and http://www.jetbrains.com/mps/

stakeholders on the coordination architecture of a parallel program with different notations. Also, it is possible to define a concrete syntax of a DSL together with its abstract syntax, for example, with the Extended Backus-Naur Form (EBNF) or an EBNF-like definition language for the purpose of rapid prototyping. For the SCOPE coordination workbench, we describe the concrete syntax specification in Section 9.2.

Workbench development denotes the development of the coordination workbench. It comprises the implementation of an editor for the coordination DSL, including mechanisms for validation, and the implementation or integration of complementary components, most prominently, transformation engines and editors. For the SCOPE coordination workbench, we describe the workbench development in Section 9.2.

Operational semantics mapping definition specifies a transformation between the abstract syntax of the coordination DSL and the abstract syntax of the formalism that was used to define the operational semantics of the conceptual coordination model. This can be realised in the form of an M2M transformation, requiring the metamodels of both, the coordination DSL and the semantics formalism, to be available. The activity enables the static analysis of concrete coordination architectures constructed with the coordination DSL. For the SCOPE coordination workbench, we describe the operational semantics specification in Section 9.2.

Additionally, *interoperability transformations definition* specifies transformations between the coordination DSL and third-party tools with which the coordination workbench must interoperate. These can be M2M transformations, in the case that the target tools provides an appropriate model interface, or M2T transformations, when interoperability is defined over file exchange interfaces.

5.4 Benefits and Limitations

The benefits of Coordination Engineering are as follows:

- Knowledge capture: Models provide a basis for communication between application engineers, transformation engineers, domain engineers, and other stakeholders.

- Reuse and portability: Reference models and transformations can be reused, providing a basis for software system families [Withey, 1996]. Different target platform transformation sets can also be applied to the same coordination model, forming a basis for software system lines.
- Quality: Model bugs, as well as the respective responsibilities, are separated from implementation bugs – the former having to be corrected only once in the transformation descriptions instead of multiple times in the source code.
- Information hiding: Transformations encapsulate platform-specific coordination implementation, thus relieving application engineers and domain engineers.
- Development time reduction: Reusing models and transformations can save development time.

However, each approach comes with additional costs and limitations:

- Learning costs: Each role can be regarded as a role in the conventional software development process that gained additional skills to participate in Coordination Engineering, see Section 5.2. Additionally, the generation of behavioural code is not well explored in practice in contrast to structural code generation [Reussner and Hasselbring, 2009]. Although the availability of an explicit coordination metamodel such as SCOPE eases the task of expressing coordination, transformation engineers will supposedly find it challenging to express coordination code as a set of transformation templates. We address this problem in Section 9.2 by presenting a strategy to organise the structure of coordination templates.
- Setup and maintenance costs: Coordination Engineering establishes a complex MDS. This represents a system in its own right that must be set up and maintained during/across parallel system development projects. As an analogy, the MDS can be regarded as a separate project infrastructure similar to test and continuous build infrastructures.
- Co-Evolution: In MDSD, issues of co-evolution can arise easily, for example when the metamodel of the coordination language evolves and

conforming models as well as transformations must be adapted to guarantee compliance [Kruse, 2011; Herrmannsdoerfer et al., 2011, 2009; Eysholdt et al., 2009]. While MDSs represent a promising step towards managing co-evolution, research on the issue is still underdeveloped and there is a lack of tools to support co-evolution by managing MDSs explicitly.

5.5 Related Work

Coordination Engineering can be described as a model-driven approach that can be architecture- and platform centric, depending on which artefacts are reused. When coordination and computation are regarded as cross-cutting concerns, the approach can also be considered as a form of aspect-oriented modelling.

5.5.1 Architecture- and Platform-Centric MDSD

Coordination Engineering can be regarded as Architecture-Centric Model-Driven Software Development (AC-MDSD), see Section 4.4. It favours the separation of the software architecture from a behavioural viewpoint from the domain-specific implementation of coordinated components. It also requires the coordination workbench as a tool infrastructure for coordination code generation, is primarily oriented towards forward engineering, and provides an elaborate role model. However, it differentiates itself from AC-MDSD since it is not restricted to establish software system *families*, which map different architecture models onto the same technical platform by reusing the platform transformation. Instead, it can also be used to establish software system *lines*, which map the same architecture onto different platforms by reusing the architecture model.

5.5.2 Aspect Oriented Modelling

Aspect orientation regards the modularisation of Cross-Cutting Concerns (CCCs) in software systems [Völter, 2005]. CCCs are concerns that can not

normally be associated with a single module or component in the respective systems because of the limited capabilities of the underlying programming languages and platforms. While aspect orientation in general aims at the modularisation and localisation of such CCCs, Aspect-Oriented Programming (AOP) specifically aims at introducing programming language constructs to handle CCC modularisation.

Although there are many commonalities between AOP and MDSD, we regard our approach clearly as an MDSD approach (cf. [Völter, 2005]). The reasons are two-fold:

- Abstraction level: A fundamental concept of MDSD is to express a problem solution on a higher level of abstraction that is more closely aligned with the problem domain. To do so, MDSD employs domain-specific languages. AOP, on the contrary, is bound to the level of abstraction of the system for which it handles CCCs.

- Pre-runtime: MDSD transformations are typically executed before the runtime of the system to be developed. Therefore, MDSD approaches can generate non-programming language artefacts such as configuration files, build scripts and documentation. On the contrary, AOP is dynamic in nature and contributes behaviour to specific points during the runtime of a system, but can not affect any artefacts that are relevant before runtime.

However, as Favre [2004a] notices, separation of concerns is an intrinsic property of model engineering (i. e., "the disciplined and rationalized production of models") that naturally leads to aspect-oriented modelling. Since Coordination Engineering eases parallel program development by the separation of concerns as the underlying principle, it can also be denoted as a form of Aspect-Oriented Modeling (AOM).

Chapter 6

Space-Coordinated Processes

This chapter introduces the Space-Coordinated Processes (SCOPE) coordination model. We discuss the requirements that apply to SCOPE, describe SCOPE as a beneficial combination of Space-Based Systems (SBS) with the Business Process Model and Notation (BPMN) specification for coordination engineering, and present the associated domain concepts. We also discuss the use of BPMN as a host language for SBS, identify a BPMN subset that represents the SCOPE coordination model, and define the operational semantics of the subset. The chapter concludes with a placement of the SCOPE coordination model in several contexts.

6.1 Requirements

The challenges of concurrent programming – a generalised abstraction over concurrent system behaviour, continuity with prevalent technologies, and the dissemination of patterns and practice – impose several requirements to any possible solution. These requirements are presented in the following. Also, they are subsumed in table form at the end of the section.

Programming languages mostly support concurrency by the highly non-deterministic and error-prone thread- or lock-based programming model [Lee, 2006], requiring an extensive use of synchronisation to prevent unwanted non-determinism. The solution must therefore provide a generalised abstraction over concurrent system behaviour that abstracts from lock-based programming, leading to the requirement of concurrency abstraction (requirement **CM00**).

With the focus on system behaviour, the *concurrency abstraction* requirement clearly identifies each solution as a kind of coordination model. In

general, such models are defined as the interaction of active components, whereas the interaction is subject to certain coordination laws that specify how components coordinate themselves through the given coordination media [Ciancarini, 1996]. This leads to the requirement that any solution must express coordination explicitly (requirement **CM01**).

The next requirement is that of the separation of concerns (requirement **CM02**): Inter-component activity and intra-component activity have to be separated in order to distinguish between the concerns of coordination and computation, cf. [Kielmann, 1996].

Stemming from the separation of concerns, the coordination model must be orthogonal to existing General-Purpose Programming Languages (GPLs) (requirement **CM03**). In particular, the coordination model must not force the considered programming languages to modify their functionality, cf. [Gelernter and Carriero, 1992; Freisleben and Kielmann, 1997].

Portability (requirement **CM04**) is the primary concern for development cost reduction and represents the degree to which applications can be moved between different machines and environments without yielding different results [Marowka, 2010, p. 78]. It must be possible to implement the coordination model on top of various types of parallel systems [Freisleben and Kielmann, 1997]. Overall generality in the sense of a general purpose coordination model is desirable [Gelernter and Carriero, 1992].

Finally, the coordination model must provide means for modularisation as a basic principle to support collaborative work and to reduce complexity by separating model elements from each other. This requires hierarchical abstractions that allow the composition of model elements (requirement **CM05**). Table 6.1 summarises these requirements.

6.2 Combining SBS and BPMN

Space-Based Systems (SBS) represent an abstraction of complex system behaviour that emphasises choreography for coordination. They focus on the coordination media (spaces) and the laws (operations) that are used to let client components communicate with the spaces. The coordination components are underspecified and an issue of *realisation*. SBS require a

Table 6.1. Domain requirements.

CM00	*Concurrency abstraction*
The solution must provide a generalised abstraction over concurrent system behaviour that abstracts from lock-based programming. It must not require to prevent unwanted non-determinism by means of synchronisation.	
CM01	*Coordination explicity*
Concurrency must be expressed explicitly as the interaction of active components that coordinate themselves according to some coordination laws through the given coordination media.	
CM02	*Separation of inter-component and intra-component activity*
Inter-component activity and intra-component activity must be separated.	
CM03	*Orthogonality*
The coordination model must be orthogonal to existing GPLs.	
CM04	*Portability*
It must be possible to implement the coordination model on top of various types of parallel systems [Freisleben and Kielmann, 1997]. Overall generality in the sense of a general purpose coordination model is desirable [Gelernter and Carriero, 1992].	
CM05	*Composition*
The coordination model must provide means for hierarchical abstractions that allow the composition of model elements.	

"host" language that defines the components. Typically, this host language is a general-purpose (sequential) computation language like C, C++, or Java in order to provide a complete concurrent programming language.

The Business Process Model and Notation (BPMN), on the other hand, emphasises both orchestration and choreography. It focuses on all three dimensions of a coordination model, the coordination components (Processes and Activities), the coordination laws (Sequence Flows and Message Flows), and the coordination media (Properties and Data Objects for intra-process coordination, Messages and Message Flows as channels for inter-process coordination). This means that it represents a complete coordination language in its own right. Several viewpoints on coordination exist, especially for high-level views on choreography. However, these are based on *message passing*, a low-level communication mechanism that tightly couples the coordination components and produces very detailed choreographies (thus motivating the high-level viewpoint on process choreographies).

To provide a complete high-level coordination model, we suggest SBS and BPMN to be combined. We denote this combination as Space-Coordinated Processes (SCOPE). This combination has fundamental advantages:

First, *space-based choreography* shows several desirable features that ease concurrency management. It abstracts from lock-based programming by introducing a shared virtual associative data structure called *space* that decouples concurrent components and guarantees the immutability of the data objects it manages (requirement **CM00**). The components indirectly communicate with each other by using a set of well-defined special operations to exchange data over these spaces (requirement **CM01**). Also, spaces are a natural mechanism to represent the *context* on which components can operate, and can evolve separately from the considered components (requirement **CM02**). Space-based choreography is orthogonal to widespread general purpose programming languages (requirement **CM03**), and can be implemented on top of various types of parallel systems (requirement **CM04**).

Second, *control-driven orchestration* clearly separates fine-grained activities of domain-specific computation from the space-based coordination primitives to publish, read, or consume data from spaces (requirement **CM02**). This resolves the drawback of SBS as a *data-driven* coordination

model regarding inter-component coordination: The arbitrary intermixture of coordination primitives and domain-specific computational code makes is hard to realise data-driven coordination models as separate languages. In contrast, control-driven coordination models are commonly realised as separate Domain-Specific Languages (DSLs). An existing control-driven coordination language can be used as a "host" language to separate the data-driven choreography primitives and declarations of domain-specific computation without the need of a computation language. The resulting model remains a complete coordination model in its own right. In addition, control-driven orchestration typically provides some mechanisms for activity composition, thus supporting the modularity of the resulting model and making it amenable to collaborative work (requirement **CM05**).

Third, the overall model is significantly higher-level than the concurrency mechanisms of general-purpose (sequential) programming languages (requirement **CM00**). It is no longer in the responsibility of the programmer to provide a clear separation between the coordination-related and the computational aspects among and within the components of a program. Instead, the combined model provides a viewpoint on the *architecture* of a concurrent program that is even understandable by non-programmers. It also provides an abstraction that is higher-level than the message passing inter-component coordination mechanism of BPMN. As such, it can be regarded as a representation of the *problem space* (the representation of the problem) instead of the *solution space* (the representation of the solution to the problem).

6.3 Domain Concepts

We define our domain as *Coordination Engineering*, see Chapter 5. Also, we regard concurrent programming as the primary domain from which we abstract. We further focus on the behavioural view on software architecture, regarding the behaviour of a software system as processes that represent active components. Within a process, application engineers are encouraged to decompose their problems in terms of sub-processes, activities to encapsulate (declare) domain logic or space access operations, and the control

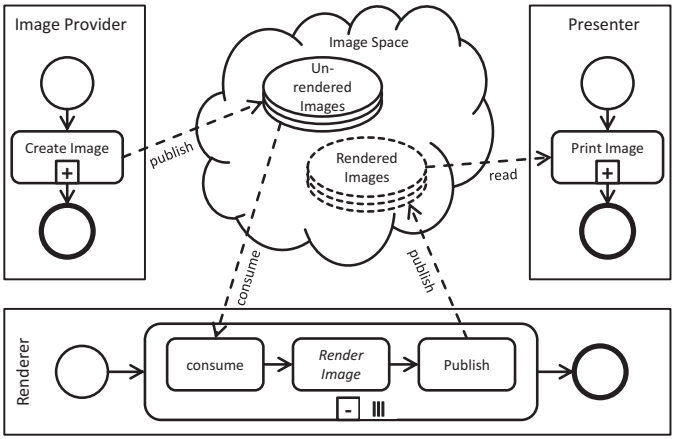

Figure 6.1. Domain concepts illustrated by the example of Mandelbrot set visualisation.

flows between them. Data flows and passive data objects are used to realise inter-process coordination. Processes only communicate with each other indirectly over spaces. The two kinds of concurrency that are considered are:

- Concurrent execution of *different* components
- Concurrent execution of multiple instances of *the same* component

Table 6.2 presents an overview of these concepts. They form a small and manageable set categorised in terms of the components to be coordinated, the media used for coordination, and the primitives that specify how component coordinate themselves through the given coordination media.

For the SCOPE coordination model, we adopt the space operations of the LighTS tuple space framework [Balzarotti et al., 2007], a small extensible open source Java implementation of an SBS for closed, non-distributed concurrent programs (as discussed in Section 6.7). Table 6.3, Table 6.4, and Table 6.5 describe the space operations used in SCOPE.

Table 6.2. Domain concepts.

	Coordination Components
Process	An active entity that executes activities and sub-processes in a predefined order, whereas the number of processes does not have to equal the number of actual processing units (processors) and is determined by the problem decomposition.
Sub-Process	A process that is a partition of a superordinate process.
Activity	An atomic active component of computation that can be executed sequentially or in parallel with other activities and sub-processes.
Domain Activity	An activity that represents (declares) domain-specific logic.
Space Activity	An activity that can publish data objects to spaces, or read or consume data objects from spaces.
	Coordination Media
Data Object	The representation of data that can be published, read, and consumed by space activities.
Space	An associative data structure that manages data objects. It can be accessed by processes through the use of space activities.
	Coordination Laws
Control Flows	The explicit specification of the order of execution of activities and sub-processes including decisions and loops.
Space Operations	The primitives performed by space activities to publish data objects to spaces, or read or consume data objects from spaces.

Table 6.3. Publishing space operations; *min* denotes the minimal number of data objects sent to a space; *max* the maximum number of data objects for an operation.

Operat.	*Data Obj.* min	max	*Behaviour*
out	1	1	Inserts a single data object into the space; guarantees the availability of the data object in the space after successful execution (synchronous)
outg	1	n	Inserts *n* data objects into the space; guarantees the availability of all data objects in the space after successful execution (synchronous); guarantees no data objects to be available before successful execution (atomic)

Figure 6.1 provides an overview of the domain concepts by the example of a parallel program that visualises the Mandelbrot set. It serves as an illustration of the concepts and does not cover the exact semantics of the individual concepts. The components of the program are Image Provider, Renderer, and Presenter. All three execute concurrently and have access to a single space, the Image Space. The order of execution is determined by the availability of data objects in the space. Firstly, the Image Provider publishes a set of unrendered slices (data objects) of the overall image. The orchestration of activities within the Image Provider is encapsulated with a sub-process (not shown in the diagram). The unrendered image slices are consumed by the Renderer. To do so, it starts a multiple-instance sub-process whose instances each consume a single unrendered image slice with a space activity, render it with a domain activity, and then publish the rendered image slice to the space with a space activity. Note that we assume that there are as many concurrent sub-process instances as there are image slices. The Presenter reads the rendered image slices, combines them to a single image, and prints the resulting image to a file. The orchestration of activities within the Presenter is encapsulated with a sub-process.

Table 6.4. Consuming space operations; *min* denotes the minimal number of data objects retrieved from a space; *max* the maximum number of data objects for an operation.

Operat.	*Data Obj.* min	max	*Behaviour*
in	1	1	Withdraws one data object from the space for the given query; if no matching data object is found, the component performing the operation is suspended until a matching object becomes available (synchronous); if multiple matches are found, only one data object is returned, as determined by the concrete implementation[a]
inp	0	1	Withdraws one data object from the space for the given query; if no matching data object is found, the component performing the operation is not suspended (asynchronous); if multiple matches are found, only one data object is returned, as determined by the concrete implementation
ing	0	n	Withdraws all data objects from the space for the given query; if no matching data object is found, the component performing the operation is not suspended (asynchronous); if multiple matches are found, all matching data objects are returned as a single unit (atomic)[b]

[a] In contrast to the LighTS tuple space framework, we do not stipulate how the space selects the data object to be returned from the matching candidates (e. g., picking the first one, as the LighTS framework does, or selecting one non-deterministically). Instead, we consider matching deliberately as a matter of implementation to enable the interoperability with a wide variety of space implementations.

[b] The *ing* operation can be considered as a symmetric sibling to the *rdg* primitive, which solves the *Linda multiple rd problem* [Rowstron, 1996]. It must be distinguished from the *collect* primitive described by Butcher et al. [1994], since the latter dismisses atomicity for increased performance and competition amongst space clients.

Table 6.5. Reading space operations; *min* denotes the minimal number of data objects retrieved from a space; *max* the maximum number of data objects for an operation.

Operat.	*Data Obj.* min	max	*Behaviour*
rd	1	1	Reads one data object from the space for the given query; if no matching data object is found, the component performing the operation is suspended until a matching object becomes available (synchronous); if multiple matches are found, only one data object is returned, as determined by the concrete implementation[a]
rdp	0	1	Reads one data object from the space for the given query; if no matching data object is found, the component performing the operation is not suspended (asynchronous); if multiple matches are found, only one data object is returned, as determined by the concrete implementation
rdg	0	n	Reads all data objects from the space for the given query; if no matching data object is found, the component performing the operation is not suspended (asynchronous); if multiple matches are found, all matching data objects are returned as a single unit (atomic)[b]

[a] In contrast to the LighTS tuple space framework, we do not stipulate how the space selects the data object to be returned from the matching candidates (e. g., picking the first one, as the LighTS framework does, or selecting one non-deterministically). Instead, we consider matching deliberately as a matter of implementation to enable the interoperability with a wide variety of space implementations.

[b] As Freisleben and Kielmann [1997] notice, the atomicity of the operation solves the *Linda multiple rd problem* [Rowstron, 1996], and potentially contributes to the performance of the operation by aggregating data objects to be returned into a single return message.

6.4 BPMN as a Host Language

The BPMN is a widely accepted modelling language for the definition of business processes. It emphasises graphical models over textual serialisations as its abstract syntax. In its current version (2.0), it represents a fully fledged coordination language. Due to its similarity, the language is often compared to Unified Modeling Language (UML) Activity Diagrams [White, 2004]. Although primarily targeting the business domain, it can be viewed as a generic language for the description of processes. We chose BPMN as a host language for SCOPE for the following reasons:

- BPMN is well-known by a variety of roles in IT industry, even non-technical roles. It is also widely supported by tools. As such, it provides a promising vehicle for the rapid dissemination and acceptance of SCOPE.
- It provides a comprehensive graphical outline view on behavioural architecture models, even across non-technical stakeholders.
- The BPMN specification already provides an operational semantics. Using BPMN as a host language allows to aggregate the semantics of SCOPE models from the operational semantics of the realising BPMN models.

BPMN also offers some challenges that must be overcome:

- Semantic correctness: The BPMN metamodel defines over 150 elements. These elements and their relations must not be used in a way that contradicts their semantics.
- Semantic clearness: BPMN defines its execution semantics in terms of token flow. Although it is denoted that "[t]he purpose of this execution semantics is to describe a clear and precise understanding of the operation of the elements." It is only defined informally [Axway et al., 2010, p. 425]. Additionally, it does not consider data or message flows. A more formal specification of SCOPE that conforms to the BPMN specification is desirable.
- Lack of validation: BPMN tools can only validate BPMN models, not SCOPE models. It is up to the modeller to provide valid coordination-specific SCOPE models when using BPMN tools for modelling.

- Lack of extensibility: SBS-specific aspects can make it necessary to extend the BPMN. The specification provides an explicit extension mechanism [Axway et al., 2010, pp. 57-61], but it is limited in several ways: custom graphical notation elements mapped to extensions are not guaranteed to be handled uniformly in tool chains, and the extensions themselves are not guaranteed to carry over in BPMN tool chains by definition [Axway et al., 2010, p. 57].

Regarding semantic correctness, industry practice recommends to identify and use only a meaningful subset of BPMN elements to cope with the complexity of the specification [Suarez et al., 2011]. Semantic clearness can be assured by providing a mapping of the informal execution semantics to a formal model that also considers the data and message flows. The lack of validation can be circumvented by providing a SCOPE modelling environment that supports the BPMN subset directly. Finally, the lack of extensibility can be addressed by a separate domain-specific language for SCOPE that can be directly mapped to BPMN.

In the remainder of this chapter, we address semantic correctness and semantic clearness by identifying a subset of the BPMN specification to represent SCOPE models, and by providing a semi-formal mapping of this subset to Petri Nets to clarify their behaviour.

6.5 BPMN Subset

In this section we identify a subset of the BPMN that can represent SBS and, more specifically, SCOPE models. We discuss the alternatives and select only elements that do not break the semantic correctness of both BPMN and SBS. If needed, we use the BPMN extension mechanism to introduce custom BPMN elements or graphical markers [Axway et al., 2010, pp. 44,57-61]. In particular, we address how to express the following elements:

- Spaces
- Client processes
- Intra-process activities

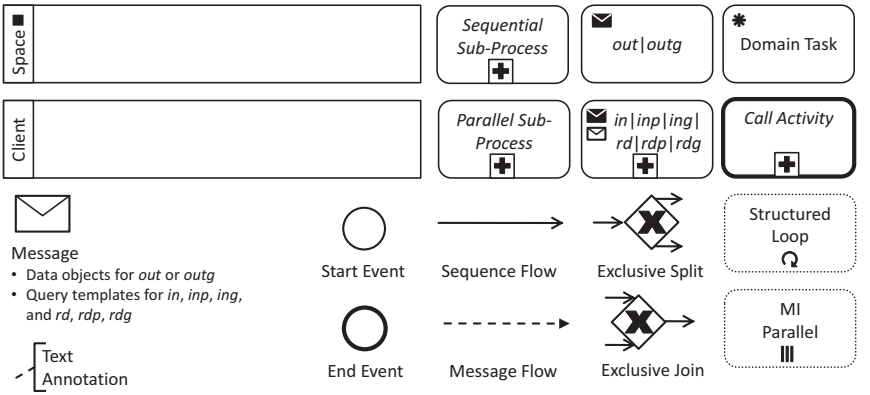

Figure 6.2. BPMN 2.0 subset for SCOPE.

- ▻ Space operations
- ▻ Data objects
- ▻ Intra-process control flows

Figure 6.2 presents an overview of the BPMN subset. In the following, we give a rationale for the elements we selected for the subset.

6.5.1 Spaces

In SBS, the operational semantics of a space is determined by its implementation. Spaces are considered as black-box components in SCOPE models. We identify BPMN Participants and Pools as the most appealing elements to model spaces although they represent a restriction of the composition of spaces. The space Pools are indicated by a black-box marker in its upper left corner to indicate their black-box execution semantics (see Figure 6.2). The rationale for the selection is discussed in the following.

SCOPE assumes the lifecycle of spaces to be bound to the lifecycle of the overall program. Therefore, BPMN Data Stores cannot be used to represent spaces since they persist beyond the lifecycle of any BPMN Process [Axway

et al., 2010, p. 208]. They would also complicate data object modelling since they are Item Aware Elements that require the sourceRef and the targetRef of their Data Associations to be also Item Aware Elements [Axway et al., 2010, p. 223]. A BPMN Activity that interacts with Data Stores would therefore require to define appropriate properties (Data Input Associations and Data Output Associations) [Axway et al., 2010, pp. 151-152].

Spaces can be represented as BPMN Participants. They represent concrete entities or roles that interact with each other and can execute individual Processes [Axway et al., 2010, p. 114]. The use of Participants for spaces has several benefits:

- There is no constraint if Participants are local or distributed. The BPMN subset would therefore abstract over the distribution of SBS. The definition of space location would be in the responsibility of subsequent mapping, modelling, or transformation stages.

- Participants do not necessarily have to define their processRef property [Axway et al., 2010, p. 116]. Also, they can be represented graphically by (black-box) Pools that may be defined without a Process [Axway et al., 2010, p. 114]. As a consequence, the behaviour of space Participants does not have to be modelled explicitly in BPMN diagrams.

- Data objects can be represented as Messages that are exchanged between client processes and spaces. This enables to consider the message exchange protocol between clients and spaces as BPMN Choreographies [Axway et al., 2010, pp. 345-366].

Another possibility would be to use BPMN Lanes to represent spaces. Lanes are sub-partitions of a Process that are often used within Pools [Axway et al., 2010, p. 305]. The outer Pool then would represent either the top-level space or the overall application. Although Lanes support hierarchical composition by nesting (requirement **CM05**) [Axway et al., 2010, p. 306], and would provide more compact diagrams than using Participants and Pools for spaces, they should not be used. Firstly, Lanes are intended as white-box elements that are "used to organize and categorize Activities within a Pool" [Axway et al., 2010, p. 306]. If the outer Pool would represent the top-level

space, they would require the outer Pool to be also white-box, forcing it and all inner spaces to reveal parts of their implementation. Secondly, they prevent to examine the interaction between client components and (inner) spaces on a higher level of abstraction as Choreographies. They would also require data objects to be represented as Data Inputs or Data Outputs since activities do not use Messages to communicate with each other within a single Process.

When spaces are modelled as Participants and Pools, there are two subsequent questions that are of particular interest:

1. Should we use multiple-instance Participant spaces to reduce the footprint of large BPMN models with many spaces?

2. Should we use Participant Associations to facilitate collaborative modelling?

Regarding the first question, the answer is "no". The BPMN specification states that "ParticipantMultiplicity is used to define the multiplicity of a Participant. For example, a manufacturer can request a quote from multiple suppliers in a Collaboration." [Axway et al., 2010, p. 117]. Participants can only have at most one Participant Multiplicity. To determine the current number of Participants in a concrete model, the standard introduces the concept of Participant Multiplicity *Instance* attributes that are *not* included in the BPMN metamodel: "numParticipants: integer [0..1] The current number of the multiplicity of the Participant for this Choreography or Collaboration Instance." [Axway et al., 2010, p. 118]. This means that the individual instances of a multiple-instance Participant remain anonymous since Participants can only be distinguished by their name and id attributes [Axway et al., 2010, p. 115]. Furthermore, it is also unclear how the actual message exchange pattern between two multiple-instances Participants is intended (many-to-many message exchange). Therefore, we refrain from using multiple-instances Participant spaces.[1]

[1] In [Gringel et al., 2012], we identified a related problem concerning the multiplicity of Participants, namely the inability of the BPMN to distinguish between the principle attendance of a Participant in a Collaboration, and the concrete participation of a certain number of instances of a Participant in a Choreography Task (a message exchange activity). We address this problem in [Gringel et al., 2012] by the introduction of a Participation as an indirection between Collaboration and Participant, instead of relying on Participant Multiplicity *Instance* attributes.

Regarding the second question, the answer is again "no". The BPMN specification states that "[t]here are situations where the Participants in different diagrams can be defined differently because they were developed independently, but represent the same thing." [Axway et al., 2010, pp. 118-119]. The Participant Association element can be used to associate different Participants with each other to denote them as the *same* Participant. The BPMN specification defines four application scenarios for the Participant Association where mapping can become necessary [Axway et al., 2010, pp. 118-119]:

1. A Collaboration references a Choreography for inclusion.
2. A Call Conversation references a Collaboration or Global Conversation.
3. A Call Choreography references a Choreography or Global Choreography Task.
4. A Call Activity within a Process has a definitional Collaboration and references another Process that also has a definitional Collaboration.

Cases two and three are not relevant since we do not use the BPMN elements Global Conversation, Global Choreography Task, and Call Conversation. Cases one (Choreographies to focus on the data exchange protocol) and four (Call Choreographies as a way of composition) are relevant, but can be circumvented by strictly using unique names for spaces. For the BPMN subset, we refrain from using Participant Associations.

Finally, using Participants to represent spaces imposes that client processes must also be modelled as Participants. The consequences are examined in the following.

6.5.2 Client Processes and Intra-Process Activities

As a consequence of modelling spaces as Participants, client components are also modelled as Participants. Regarding the further decomposition of client components in terms of BPMN elements, we consider to model the following aspects:

- ▻ Components that embody domain-specific logic are modelled as Domain Tasks.

- ▻ The parallel instantiation of the *same* component is modelled with Multi Instance Loop Characteristics.
- ▻ The parallel execution of *different* components is modelled with (Parallel) Sub-Processes.
- ▻ Component composition is modelled with Call Activities.
- ▻ Component structurisation is modelled with (Sequential) Sub-Processes.

A natural candidate for the representation of domain-specific logic is the BPMN (Abstract) Task element: It represents an atomic activity that cannot be broken down to a finer level of detail [Axway et al., 2010, p. 156]. Tasks are general (and extensible) enough to represent any kind of domain-specific logic [p. 158, BPMN]. The semantics of the element resembles its black-box nature: "Upon activation, the Abstract Task completes. This is a conceptual model only; an Abstract Task is never actually executed by an IT system." [Axway et al., 2010, p. 430]. To conform to the standard, we introduce the Domain Task as a concrete Task type that represents atomic domain-specific logic. We advice to indicate the Domain Task by an asterisk marker in its upper left corner.

The parallel execution of *different* components can be modelled with BPMN Sub-Processes: "Expanded Sub-Processes can be used as a mechanism for showing a group of parallel Activities in a less-cluttered, more compact way. [...]" [Axway et al., 2010, p. 174]. The use of expanded Sub-Processes as so-called "parallel boxes" does not require a Start Event or an End Event associated to the parallel Activities within the Sub-Process, thus reducing the footprint of BPMN models and diagrams [Axway et al., 2010, p. 174].

The BPMN multi-instance Activity fits nicely to model the parallel instantiation of the *same* component: "The multi-instance (MI) Activity is a type of Activity that acts as a wrapper for an Activity which has multiple instances spawned in parallel or sequentially" [Axway et al., 2010, p. 432]. It is defined as a conventional Activity whose behaviour is determined by the attributes of its Multi Instance Loop Characteristics [Axway et al., 2010, pp. 191-194]: First, isSequential specifies whether instances are instantiated sequentially (*true*) or in parallel (*false*). Second, the number of instances is

defined by the attribute loopCardinality or by the cardinality of a collection data item of the *Data Input* of the Activity.

Component composition can be modelled as Call Activities. Sub-Processes can not be used for composition since the contained elements are exclusive to the parent Process. This means that Sub-Processes represent means for structuring, but not for composition since reuse is not supported. In fact, the reusable Sub-Process element of the previous BPMN 1.2 specification corresponds to the Call Activity of BPMN 2.0 [Axway et al., 2010, p. 176].

However, we retain Sub-Processes as a means for structurisation and use meaningful subclasses of the generic Sub-Process. We already decided to use a dedicated Sub-Process for the parallel execution of different Activities. We also introduce a dedicated Sub-Process for the execution of a sequence of activities within a Process. This enables to model loops not only over single activities but also over complete sequences of activities. To distinguish the two uses, we refer to them as Parallel Sub-Process and Sequential Sub-Process.

A Call Activity is a task that can call a pre-defined (global) Process. It "acts as a wrapper" for the invocation of the Process [Axway et al., 2010, p. 183]. Several Call Activities can call the same pre-defined Process, identifying the element clearly as a means for composition. Graphically, a Call Activity can both be displayed as a collapsed element (similar to a collapsed Sub-Process, but with a thick line), or expanded (showing its contents similar to an expanded Sub-Process, but with a thick line). Regarding the SCOPE BPMN subset, using Call Activities for composition also distinguishes composition nicely from parallel execution with Sub-Processes.

6.5.3 Space Operations

Regarding the operations needed to access spaces, we model the following aspects:

- Publishing a data object to a space is modelled as a Send Task.
- Receiving a data object from a space is modelled as a predefined BPMN Sub-Process idiom that contains a Send Task to model assembling and sending a query template to the space, and a subsequent Receive Task to model receiving a response message from the space.

Publishing a data object to a space can be modelled using a Send Task [Axway et al., 2010, pp. 159-161]. A Send Task is designed to send a Message to an external Participant. This relates to the non-blocking space coordination primitives *out* (publish a data object) and *outg* (publish a group of objects). An alternative would be to use Intermediate Events [Axway et al., 2010, pp. 249-260]. We refrain from these because they only denote the event of receiving or sending a message, not the construction of the message itself. Send Tasks, on the other hand, can also symbolise that messages are constructed in a standardised way before sending occurs. A Send Task must be named either *out* or *outg*, depending on which space coordination primitive is used.

Receiving data objects from a space requires assembling and sending a query template to the space before matching data objects can be received. Also, it must be possible to differentiate between the blocking space coordination primitives *in* and *rd*, and their non-blocking siblings *inp*, *ing*, *rdp*, and *rdg*. Therefore, the conventional Receive Task cannot be used: "Once the Message has been received, the [Receive] Task is completed" [Axway et al., 2010, p. 161].

As a solution, we use a predefined BPMN Sub-Process idiom that contains a Send Task to model assembling and sending the query template to the space, and a subsequent Receive Task to model receiving a response message from the space (see Figure 6.3). Blocking space coordination primitives are distinguished from their non-blocking siblings only by the name of the Sub-Process and the content of the response message from the space: Non-blocking operations simply can contain messages with no data objects enclosed, or *null* data objects, respectively. The predefined Sub-Process must be named either *in*, *inp*, *ing*, *rd*, *rdp*, or *rdg*, depending on which space coordination primitive it should represent. We advise to indicate the predefined Sub-Process by two markers in its upper left corner: the black Message marker of a Send Task and the white Message marker of a Receive Task.

Another solution would be to extend the list of Task types with a custom Query Task. This is in accordance with the BPMN standard since "[t]he list of Task types MAY be extended along with any corresponding indicators." [Axway et al., 2010, p. 158]. We refrain from this since it conceals the

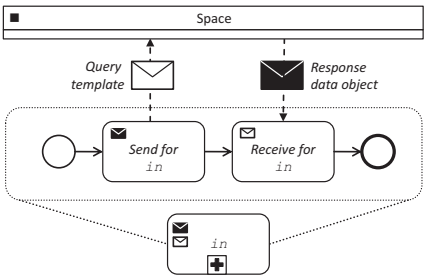

Figure 6.3. BPMN space query idiom.

message exchange pattern that is necessary to represent space coordination primitives that consume or read data objects.

Note that we refrain from using BPMN Standard Loop Characteristics to model space coordination primitives that handle a group of data objects (*outg* and *ing*). This enables to distinguish between looping space coordination primitives and space coordination primitives that atomically publish a group of data objects. For example, a looping *in* primitive can guarantee that all parts of a compound data object are available to a Process.[2] As a consequence, Messages can represent (contain) a collection of data objects.

6.5.4 Data Objects and Inter-Process Data Exchange

We represent inter-process data exchange as Message Flows as a consequence of modelling spaces as Pools, data objects as Messages. The rationale for this selection is as follows:

Much of the benefits of the SCOPE model rely on the fact that in SBS, the lifecycle of data objects is not necessarily bound to the lifecycle of the processes that produced them. This means that data objects and query templates cannot be represented as BPMN Data Objects since the lifecycle

[2]The use of looping for the other space primitives is questionable from a semantic viewpoint: A looping *rd* or *rdp* could possibly return always the same data object for a given query (see [Rowstron, 1996]), and looping non-blocking primitives represent multiple tries without checking the result of an individual trial.

of the latter is bound to the lifecycle of their parent Process or Sub-Process [Axway et al., 2010, p. 207].

In accordance with the previous selections, Messages are a natural choice to represent SBS data objects [Axway et al., 2010, p. 93]. The data object to be sent or received does not have to be modelled explicitly since the Item Definition is an optional part of the Message [Axway et al., 2010, p. 95]. Also, Messages can be used in a BPMN Choreography [Axway et al., 2010, p. 93]. A single Message Flow illustrates the flow of a Message between two Participants by connecting two separate Pools [Axway et al., 2010, p. 120]. Message Flows can not connect two elements that are located within the same Pool.

The BPMN standard differentiates between initiating and non-initiating Messages. Since we cannot guess the responses from space Participants, we consider only initiating Messages in SCOPE:

- Messages that represent query templates for the *in*, *inp*, *ing*, *rd*, *rdp*, and *rdg* coordination primitives
- Messages that represent data objects for the *out* and *outg* space coordination primitives

6.5.5 Intra-Process Control Flows

Process instantiation and termination We use the BPMN Start Event to model the instantiation of a client process [Axway et al., 2010, p. 238]. Analogously, we use the BPMN End Event to model the termination of a client process.

The BPMN standard employs the notion of *process levels* to discuss the use of events: "A Process MAY have more than one Process level (i. e., it can include Expanded Sub-Processes [...])" [Axway et al., 2010, p. 238]. We restrict Processes to only allow one single Start Event and one single End Event within a single process level [Axway et al., 2010, p. 426]. Additionally, we consider the Start Event and End Event as optional in Sub-Processes to support the "parallel box" concept.

Control flows Control flows are represented in BPMN in terms of Sequence Flows [Axway et al., 2010, p. 97]. In the BPMN subset, we use Sequence Flows

to model the flow of control between Start Events and End Events, Activities and Gateways. To enforce block-structured control flows as far as possible we adhere to the following conventions in Sequence Flows:

- Do not use arbitrary control flow loops.
- Do not use Gateways to control concurrency. Instead use Sub-Processes and Loop Activities.

Structured loops Structured loops in the control flow of a Process are modelled with the conventional Standard Loop Characteristics element [Axway et al., 2010, p. 432]. Its behaviour is determined by its attributes. As long as the loopCondition evaluates to *true*, the inner Activity is executed again. Thereby, testBefore determines if the loopCondition is evaluated before (*true*) or after (*false*) the Activity is executed. loopMaximum determines the maximum number of executions, including unbounded, if the attribute is not set. We differentiate between the following kinds of loops:

- Pre-testing iterating loop (for): testBefore is *true* and loopMaximum is evaluated to determine if the inner Activity shall be executed or not.
- Pre-testing conditional loop (while): testBefore is *true* and loopCondition is evaluated to determine if the inner Activity shall be executed or not.
- Post-testing conditional loop (do-while): testBefore is *false* and loopCondition is evaluated to determine if the inner Activity shall be executed or not.

Structured parallel control flows We differentiate between two kinds of parallel control flows:

- Parallel execution of *different* activities is modelled as a BPMN Sub-Processes.
- Parallel instantiation of the *same* activity is modelled as a BPMN multi-instance Activity.

Although the Parallel Gateway would be a natural choice to model parallel control flows, we refrain from using it in order to confine to block structured modelling. Else, it would be necessary to require each diverging Parallel Gateway to be merged by a subsequent converging Parallel Gateway, and to restrict a diverging Parallel Gateway to have only one single incoming Sequence Flow, and a converging Parallel Gateway to only have one single outgoing Sequence Flow.

Data-based decisions Exclusive Gateways are used to model data-based decisions. The reason is that their semantics are uncomplicated [Axway et al., 2010, p. 435]. We require each Exclusive Gateway to have its default Sequence Flow specified to prevent an exception is thrown when all conditions of the gateway evaluate to false.

We refrain from using Inclusive Gateways since they require that a backwards search for tokens in the upstream Sequence Flows is executed [Christiansen et al., 2010]. We also refrain from using Event-based Gateways since the subset currently does not consider Events except for Start Events and End Events. Finally, we do not use the Complex Gateway since it shares the complicated upstream search behaviour of the Inclusive Gateway and imposes the possibility of race conditions [Axway et al., 2010, p. 437].

Annotations We introduce Text Annotations in order to support unstructured information within SCOPE BPMN models [Axway et al., 2010, p. 71].

6.6 Operational Semantics

In this section, we establish a semi-formal mapping of the SCOPE BPMN subset to Petri net modules [Reisig, 2010]. This serves the purpose of further refining the operational semantics of SCOPE and providing a basis for the statical analysis of models that conform to the SCOPE coordination model. The mapping is based on the work of Dijkman et al. [2007] on the formal semantics of BPMN process models and adapted to conform to the SCOPE BPMN subset. Further formalisation of the operational semantics are left for future work.

Although the BPMN 2.0 specification already discusses its operational semantics, a more rigorous semantics definition for SCOPE is desirable. The reason is that the BPMN does not satisfy its claim to provide a precise understanding of the behaviour of its elements [Axway et al., 2010, p. 425]: The execution semantics of BPMN is specified only informally and does not take Message Flows into consideration [Axway et al., 2010, pp. 238,425].

Petri nets are chosen as the target formalism because their execution semantics is exact (mathematically defined) and there is a wide range of static analysis techniques. Additionally, the use of Petri nets is an obvious choice since the BPMN 2.0 specification discusses its operational semantics in terms of token flow [Axway et al., 2010, p. 238].

6.6.1 Basic Constructs for Client Components

Figure 6.4 presents the mapping of SCOPE Tasks, Events and Gateways to Petri net modules. A Start Event or End Event is mapped to a transition with an input place and an output place. A Domain Task is mapped to a similar module. The forking and merging Exclusive Gateways are mapped to Petri net modules that represent the routing behaviour of the gateways using a set of two transitions that follow a single space (fork) or precede a single space (join), respectively. The Call Activity is mapped to a module that uses two transitions to represent entering and returning from the referenced Process of the Activity. The Petri net module of the referenced Activity then needs to be copied into the Petri net module of the Call Activity to complete the construct. In general, Sequence Flows are mapped to places.

6.6.2 Sub-Processes

In SCOPE, we use Sub-Processes in three different scenarios: as a container to enable structured looping over a sequence of activities, as a container for parallel execution (parallel box), and as an idiom for space access. The corresponding Petri net modules for Sub-Processes are presented in Figure 6.5 and Figure 6.6.

The Petri net module for the Sequential Sub-Process is almost identical to the Petri net module for the Call Activity: It uses two transitions to represent

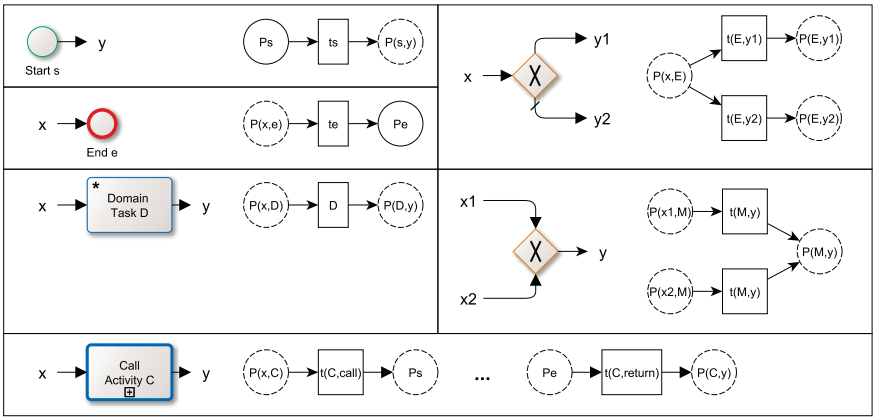

Figure 6.4. BPMN SCOPE subset to Petri net mapping for basic elements; the individual SCOPE BPMN model fragments are displayed on the left hand side and the corresponding Petri net modules to the right; dashed Petri net module elements represent interface connection points for Petri net composition.

entering and returning from the module contained in the Sub-Process. However, as it does not reference a (reusable) Activity but represents a partition of the Process, the Petri net modules corresponding to the partition must be inserted (instead of copied) into the module to complete the construct.

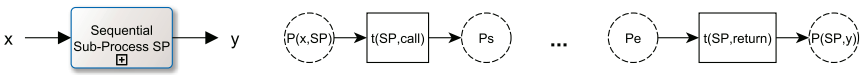

Figure 6.5. BPMN SCOPE subset to Petri net mapping for the Sequential Sub-Process; the SCOPE BPMN model fragment is displayed on the left hand side and the corresponding Petri net module on the right; dashed Petri net module elements represent interface connection points for Petri net composition.

The Parallel Sub-Process can be described as a combination of the Petri net module of the Sequential Sub-Process with Petri net constructs for parallel fork and join. The latter are modelled with a single transition that precedes a set of places (tfork in case of the fork), and a set of places places that

precede a single transition (tjoin in case of the join).

The Sub-Process idiom for space access is discussed in Section 6.6.5.

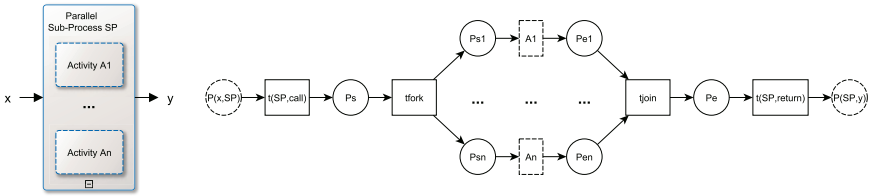

Figure 6.6. BPMN SCOPE subset to Petri net mapping for the Parallel Sub-Process; the SCOPE BPMN model fragment is displayed on the left hand side and the corresponding Petri net module on the right; dashed Petri net module elements represent interface connection points for Petri net composition.

6.6.3 Loop Characteristics

Loop behaviour must be considered in more detail, depending on the actual Loop Characteristics of a BPMN Activity. First, we discuss Standard Loop Characteristics and second, the Multi Instance Loop Characteristics. The corresponding Petri net modules are presented in Figure 6.7

When the Standard Loop Characteristics is *pre-test conditional* (a "while"-loop), the loopCondition must be checked before a loop iteration can be executed. This is modelled with the place Pcheck and the two subsequent transitions tcheck_true and tcheck_false. If the loopCondition evaluates to *true*, the transition tcheck_true fires and initiates the construct to invoke the Activity A. After A has executed, the transition trepeat fires and produces a token to Pcheck, meaning that an iteration has executed and the condition must again be evaluated. If the loopCondition evaluates to *false* (tcheck_false fires), no (further) iteration can execute and the flow of control continues with subsequent Petri net modules.

The *pre-test iteration* loop behaviour (a "for"-loop) can be regarded as a special case of *pre-test conditional* loop behaviour: loopCondition evaluates to *true* when the loopCounter has reached (equals) loopMaximum. Else, it evaluates to *false* (the value of loopCounter is lesser than the value of loopMaximum).

Conceptually, the firing of trepeat also means that the value of loopCounter is increased to represent a successful execution of A.

When the loop behaviour is *post-test conditional* (a "do while"-loop), the loopCondition must be checked after a loop iteration has executed. This means that always, at least one iteration executes. This is modelled again with the transitions tcheck_true and tcheck_false, but this time, the two transitions are located after the invocation of Activity A. After A is executed, the transition tcheck_true fires only when loopCondition evaluates to *true*. It produces a token to Ps, meaning that another iteration can execute. When loopCondition evaluates to *false*, tcheck_false fires and the flow of control continues with subsequent Petri net modules.

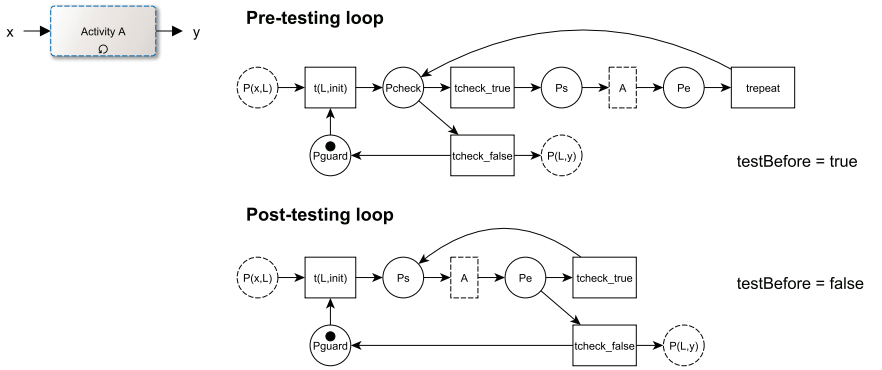

Figure 6.7. BPMN SCOPE subset to Petri net mapping for loop behaviour; the SCOPE BPMN model fragment is displayed on the left hand side and the corresponding Petri net module to the right; dashed Petri net module elements represent interface connection points for Petri net composition.

As Dijkman et al. [2007] notice, *parallel* Multi Instance Loop Characteristics represents a "for-each" programming construct (isSequential is *false*). We consider the case that the number of instances n is known *a priori* at design time.[3] In this case, the Petri net module for Parallel Sub-Process (Figure 6.6)

[3] As Dijkman et al. [2007] notice, *a priori* knowledge at runtime would require high-level net constructs such as arc expressions in Coloured Petri nets, which is beyond the scope of this

can be reused. The only difference is that the transitions $A1$ to An represent instances of the actual Activity A to be executed.

6.6.4 Spaces

Figure 6.8 shows the mapping of a space to a Petri net module. It represents a conceptual abstraction over the behaviour of a space that preserves its black-box behaviour regarding the actual matching algorithm. The module is necessary to construct complete Petri nets.

In principle, messages and data objects are mapped to tokens. The places marked with dashed lines represent interfaces for the space primitives. They are annotated with groups that identify the corresponding space primitive. Each of these places consider tokens on them as messages to be send or received. Beside these interface places, the individual parts of the module decompose incoming messages into individual data object tokens, and produce outgoing messages from data object tokens. Throughout the module, we use n-weighted edges to represent a number of data objects. This enables to consider each data object as a single token.[4]

At the core of the module, the place Pstore models the data object store maintained by the space. In addition, the place Pnull models a special *null* data object that can be returned to indicate that no match was found for a given query. For the sake of clearness, Figure 6.8 breaks the module down into separate parts for each space primitive. These module parts are connected with each other by sharing the two places Pstore and Pnull. Within each part, we provide places that represent the access interface for client

thesis.

[4] Avoiding this kind of *a priory*-knowledge would require to consider a single token as a *set* of data objects. We refrain from this alternative since it would complicate the semantics of the module: The production of a token that represents the result for a given query would require to split and reassemble these sets. Consider, for example, a space that manages the set A and B, where A consists of the data objects a and b, and B consists of the data objects c and d. As well, consider a space operation *ing* that leads to a matching that identifies the data objects b and c as matches. A Petri net construct that models the consumption of these data objects from a data store place would then require to consume the two sets A and B, extract b and c from them, produce a new set *Result* containing b and c, and produce sets for the remaining data objects a and d to be returned to the data store place – for example, as A' and B', or as a single set *Remain*.

components (the places with the dashed lines and the corresponding space primitives annotated), and two transitions that represent the interfaces for a synchronisation policy that can be employed. In the following, we discuss the Petri net module for the space component. After that, we discuss the role of the synchronisation policy.

The publishing space primitives *out* and *outg* are modelled with the two transitions taquire_write_<out/outg> and trelease_write_<out/outg>. These transitions also represent an interface for a possible synchronisation policy (therefore the dashed lines). The first transition conceptually can acquire a lock, while the second releases it to indicate the successful publication of one (or more) data object(s) to Pstore. The place Ptemp_<out/outg> is an intermediate place to connect the two transitions. In the case of the *outg* primitive, the atomic publication of multiple data objects is modelled with *n*-weighted edges.

Similarly, the consuming primitives *in*, *inp* and *ing* also use two transitions (taquire_write_<in/inp/ing> and trelease_write_<in/inp/ing>) as an interface for a synchronisation policy. The transition taquire_write_<in/inp/ing> fires when the space received a query for one or more data objects. It produces a token that represents a query to the place P<in/inp/ing>_query. The place models a query store that is used as a precondition to let the transition that models the matching strategy fire (tmatch_withdraw for *in*, tmatch_withdraw_probing for *inp*, and tmatch_multi_withdraw_probing for *ing*). These matching transitions, in turn, consume the query token from P<in/inp/ing>_query, and one (or more) data object token(s) from Pstore in order to provide the necessary data object(s) to be returned. Finally, firing of trelease_write_<in/inp/ing> represents the return message from the space to a client component.

The actual abstraction from the matching algorithm is located in the transitions tmatch_withdraw, tmatch_withdraw_probing, and tmatch_multi_withdraw_probing. When the transitions tmatch_withdraw and tmatch_withdraw_probing fire, they each consume a single matching data object token from Pstore. In the case of the *ing* primitive, tmatch_multi_withdraw_probing consumes a number of matching data object tokens from Pstore.

In case of the non-blocking space access operations *inp* and *ing*, the space must be able to inform client components when there was no matching data object available. This is modelled with the transition tmatch_<inp/ing>_null. It

fires when no matching data object could be found in Pstore, thus consuming the query token from P<inp/ing>_query. In the case of the *inp* primitive, tmatch_inp_null simply generates a token to Pinp_result that represents a *null* object to be returned by trelease_write_inp.

For *ing*, it is necessary to model the preparation of the return message explicitly. The reason is that we must distinguish between weighted and non-weighted edges: A successful match produces n matching data objects to Ping_result, and a mismatch produces a single token. As a solution, we introduce the place Ping_result_msg that represents a temporary store for the return message, and the transition tprepare_msg_ing that produces the result message from the matching data objects to Ping_result_msg. Thus, a token produced to Ping_result_msg represents a message that either contains a set of data objects, or a single *null* data object.

The read primitives (*rd*, *rdp*, and *rdg*) are modelled similarly to the consuming primitives. The only difference is that the matching transitions tmatch_copy, tmatch_copy_probing, and tmatch_multi_copy_probing do not only remove a token from Pstore but also generate one back to Pstore in order to model a copy behaviour. The interface transitions are taquire_read_<rd/rdp/rdg> and trelease_read_<rd/rdp/rdg>.

As mentioned before, the Petri net mapping shown in Figure 6.8 provides interface transitions for each space primitive that allow to formulate a synchronisation policy for space access. The transitions can be identified by their dashed lines. Figure 6.9 presents two examples of synchronisation policies. Both have in common that tokens that are initially placed on Pready_read represent read locks, and tokens that are initially placed on Pready_write represent write locks. The transitions tacquire_read, trelease_read, tacquire_write, and trelease_write represent placeholders that illustrate two aspects: First, they show how the interface transitions of the space module (Figure 6.8) can access the places Pready_read, Pbusy_read, Pready_write, and Pbusy_write. Second, they show how the places Pbusy_read and Pbusy_write inhibit the firing of the transitions (modelled with inhibitor edges that prevent a transition from being fired).

The module on the left side of Figure 6.9 represents a simple locking policy that does not profit from the distinction of read and write locks. The module on the right side represents a read-write lock that shows the

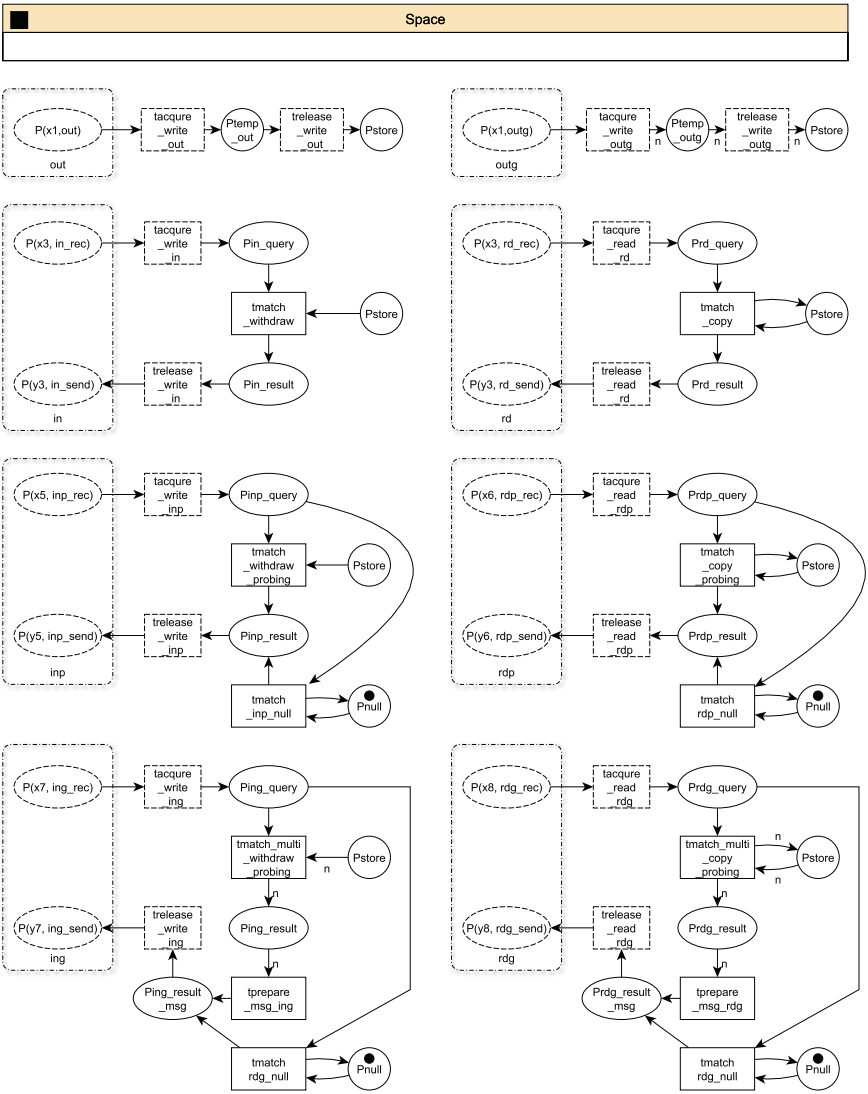

Figure 6.8. BPMN SCOPE subset to Petri net mapping for spaces; the SCOPE BPMN model fragment is displayed top and the corresponding Petri net module at the bottom; dashed Petri net module elements represent interface connection points for Petri net composition; the places Pstore and Pnull connect (are shared by) the individual module parts.

following characteristics:

- ▻ Any number of reading space accesses can acquire and hold a read lock.
- ▻ No read lock can be acquired when a write lock is held.
- ▻ No write lock can be acquired when a read lock is held.
- ▻ Only one writing space access can acquire and hold a write lock at a time.

When such a policy is formulated as a Petri net module, it can be regarded as a contract for the operation of a concrete space implementation. For example, the first policy in Figure 6.9 could be implemented in Java with the `synchonized` keyword, while the second policy could be implemented in Java using `java.util.concurrent.locks.ReentrantReadWriteLock`.[5]

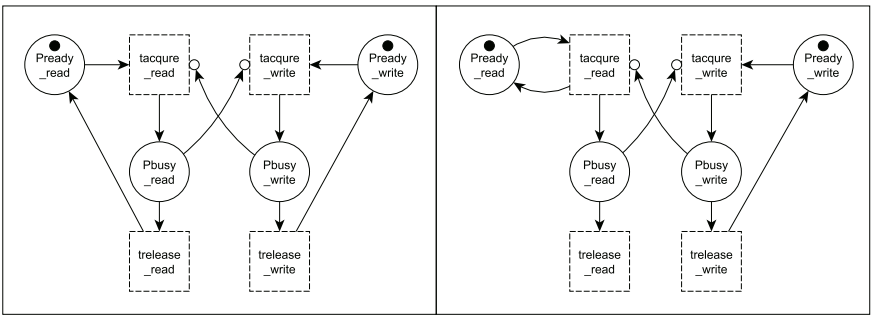

Figure 6.9. Exemplary Petri net modules for space access synchronisation policies; the left module illustrates a conventional locking policy while the right module illustrates a multiple read locking policy; dashed Petri net module elements represent interface connection points for Petri net composition.

[5] http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/locks/ReadWriteLock.html states that "[a] ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive."

6.6.5 Space Access

The mapping of space access operations to Petri net modules is presented in Figure 6.10 by the example of the space primitives *out* and *in*. As the interaction between spaces and client components is expressed in terms of message exchange, BPMN Message Flows have to be modelled in terms of token flow. We use places to model Message Flows.

First, publishing a data object to a space (BPMN Send Task) requires to provide a data object token for the place that is required by the space's tacquire_write_out transition to fire. This is realised with the transition tsend_out on the side of the client. For the *out* primitive, it produces one data object token to the place P(x1,out), and for *outg*, it produces one or more data object tokens to the place P(x1,outg) (using a weighted edge).

Second, we demonstrate the idiom at the example of the *in* operation. The Petri net modules for the BPMN idioms to consume or read data objects require to consider the interface places of the space Petri net module (Figure 6.8): P(x2,in_rec) expects a query, and P(x2,in_send) provides the space's return message. The transition tsend_in produces a query token for the place P(x2,in_rec). In contrast, the client's trec_in transition can only fire when the space returned a response message, represented by a data object token produced by trelease_write_in to the place P(x2,in_send). In the case of a *inp*, *rdp*, *ing*, or *rdg* primitive, the token can also represent a *null* value, as discussed above.

6.7 Related Work

6.7.1 DSL for AVOPT, Product Lining, and Interaction

The development of the SCOPE coordination model can be justified in terms of DSL development patterns as discussed by Mernik et al. [2005]. The main purpose of SCOPE is to provide a domain-specific language for Analysis, Verification, Optimization, Parallelization, and Transformation (AVOPT). To increase the accessibility of coordination architectures among stakeholders, SCOPE was defined as a subset of the BPMN, providing a well-known notation for coordination specification. In the context of Model-Driven Systems

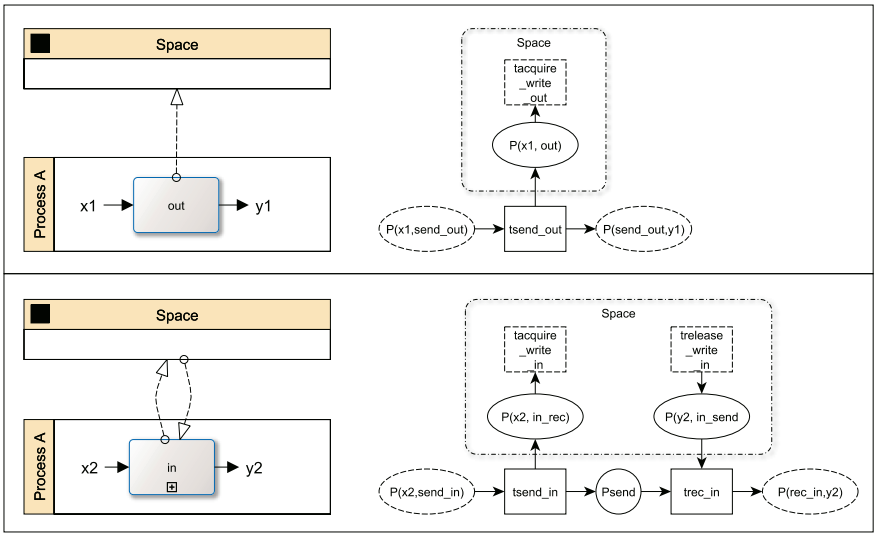

Figure 6.10. BPMN SCOPE subset to Petri net mapping for space access; the individual SCOPE BPMN model fragments are displayed on the left hand side and the corresponding Petri net modules to the right; dashed Petri net module elements represent interface connection points for Petri net composition.

(MDSs) (see Chapter 5) SCOPE can be regarded as a means for product lining and product family establishment, depending on which artefacts are reused (the coordination architecture models or the platform transformation sets). Finally, since SCOPE is clearly focused on coordination, it specifies the interaction of the individual components of a software architecture.

6.7.2 Closed Local Parallel Systems

There are several variations of the space operations in SBS. Kielmann [1996] presents a set of operations that comprise both the operations from the original Linda SBS for closed local systems (*Linda-set*), as well as the minimal set of operations for open and distributed systems. Open systems are systems in which new components (often denoted as agents) may dynamically join

and leave [Agha, 1985; Kielmann, 1996], adding the requirement of *dynamics* to coordination models. Distributed systems are systems that have to cope with the remoteness of their components and the absence of an overall compile time [Kielmann, 1996], adding the requirement of *decentralisation*. The resulting operation set can be regarded as an archetypical SBS operation set (*Kielmann-set*).

For the SCOPE coordination model, we adopt the space operations of the LighTS tuple space framework [Balzarotti et al., 2007], a small extensible open source Java implementation of an SBS for closed, non-distributed concurrent programs. Its orientation towards closed local systems can be explained by two prerequisites:

- ▻ There exist a single compile-time for the whole concurrent program.
- ▻ All components the concurrent program consists of are identified, modelled, and implemented before the runtime of the application.

We leave an extension of SCOPE's coordination primitives, for example, for open and distributed systems and to increase the performance and competition amongst space clients by a *collect* primitive [Butcher et al., 1994], as future work.

6.7.3 Software Architecture Description

Software architecture can be defined as "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" [Garlan and Perry, 1995; IEEE Architecture Working Group, 2000]. It can be specified by Architectural Description Languages (ADLs) – formal languages that deal with the specification of a system's global structure by defining its basic elements denoted as *components*, and the interactions between them in form of *interfaces* as *connectors*. Thereby, ADLs can both describe single architectures denoted as *configurations*, or whole families of architectures denoted as *architectural styles*. Coordination, on the other hand, can be defined as "the management of dependencies between activities" [Malone and Crowston, 1994]. This

is achieved by using coordination languages, whereas a coordination language is "the linguistic embodiment of a coordination model" [Gelernter and Carriero, 1992].

Although ADLs and coordination languages have evolved separately and with different goals, they share common abstractions. This is especially the case when the architectures of complex parallel and distributed systems are considered. ADLs have at least the potential to coordinate interactions between the components of a software architecture and can therefore be considered as coordination languages or *dynamic* ADLs, and a certain coordination model can be described either as an architectural style or as the foundation of an ADL [Quintero et al., 2001]. Conversely, coordination languages can describe the structures of programs (i. e., their architectures) that change over time, and can therefore be considered as dynamic ADLs. Although we emphasise coordination before architectural description, thus denoting SCOPE as the foundation for a coordination language, we regard the two terms *dynamic ADL* and *coordination language* as interchangeable. We recommend the work of Medvidovic and Taylor [1997]; Papadopoulos and Arbab [1998] as a starting point for a further study of existing ADLs and coordination languages for the interested reader. In their work on dynamic coordination architecture through the use of reflection, Quintero et al. [2001] provide an overview of the interrelationship between Coordination Theory (CT) and software architecture, and provide *PiLar* as an example that is both considered as a dynamic ADL and a coordination language.

Chapter 7

Requirements for a Coordination Engineering Library and Workbench

This chapter presents the requirements for a coordination library and a coordination workbench for Space-Coordinated Processes (SCOPE). In the following, we distinguish between the *conceptual* SCOPE coordination model (see Chapter 6), the SCOPE metamodel (an abstract syntax that realises the SCOPE coordination model as the foundation of a domain-specific language), and SCOPE models (models that are conformant to the SCOPE metamodel).

7.1 Coordination Library

The purpose of the coordination library is to provide an experimental environment for the justification of the SCOPE coordination model, and to provide a prototype platform from the viewpoint of the platform provider to support parallel systems designed with SCOPE.

Several requirements can be derived from these goals, see Table 7.1. In the following, we describe the requirements, separated into functional requirements (*LF*, describe what the system should do) and non-functional requirements (*LNF*, describe how the system should be).

Table 7.1. Coordination library requirements.

LF-00	*Performance Measurement*
The library must provide mechanisms to measure the performance of parallel programs that were developed using it. It should be possible to measure the overall time behaviour of parallel programs, as well as the time behaviour of the individual processes of the program. When the implementation is based on a managed run-time environment, measures must be taken to reduce the effects of bytecode interpretation and Just-In-Time (JIT) compilation [Goetz, 2009].	
LF-01	*Performance Analysis*
The library must provide mechanisms to analyse the measured performance of parallel programs that were developed using it. In particular, it must be possible to compute the 95% confidence interval for a series of measurements because of the importance of confidence intervals for statistically rigorous performance evaluation [Georges et al., 2007].	
LF-02	*Performance Visualisation*
The library must provide mechanisms to visualise the analysed performance properties of parallel programs that were developed using it. In particular, it must be possible to plot arithmetic means and confidence intervals computed for a series of measurements.	
LF-03	*Benchmark Execution*
The library must support the execution of benchmarks in order to analyse its intrinsic time behaviour. The benchmark execution must support micro-benchmarks that execute individual space operations, macro-benchmarks that execute compound space operations and application-specific benchmarks that execute complete SCOPE programs. In conformance to Fiedler et al. [2005] on Space-Based Systems (SBS) performance measurement, scalability should be measured in terms of throughput and response time.	
LNF-00	*Domain Conformance*
The library must be conformant to domain concepts, ideally providing distinct constructs for each distinct concept.	

LNF-01	*Time Behaviour*
The library must be scalable on multi-core machines. Additionally, the library must perform well in contrast to lock-based programming.	
LNF-02	*Portability*
The library must be portable across different parallel platforms. The parallel platforms targeted are described as multi-core machines that provide a shared memory abstraction.	
LNF-03	*Genericity*
The library must be applicable to coordination in arbitrary application domains. It must not be restricted to be used in a certain application domain (e. g., business, health, or energy).	
LNF-04	*Extensibility*
The library must be extensible and adaptable to future requirements and platforms.	
LNF-05	*Usability*
The library must be as comfortable as possible from the programmer's viewpoint. This includes module test support, continuous build integration, and dependency management.	

7.2 Coordination Workbench

The purpose of the coordination workbench is to provide an implementation of the conceptual SCOPE coordination model as a concrete Domain-Specific Language (DSL). It is supported by appropriate tooling that can be used by application engineers and transformation engineers for Coordination Engineering. The implementation of the workbench illustrates the application of the Tool Engineering process (Section 5.3.3) within Coordination Engineering. In the following, we describe the requirements we derived from this goal, separated into functional requirements (*WF*, describe what the system should do) and non-functional requirements (*WNF*, describe

how the system should be), see Table 7.2.

Table 7.2. Coordination workbench requirements.

WF-00	*Textual Representation*
In contrast to a graphical representation, a textual representation supports the iterative collaboration among stakeholders due to its block-based structure. It allows to separate models across different files and to separate work within a single file across different model blocks. Additionally, textual representations profit from broad tool support. This is in particular the case with version management systems. The SCOPE DSL must therefore provide a textual concrete syntax.	
WF-01	*Visual Representation*
Developing parallel applications using traditional programming languages can be tedious and error-prone due to the linearity of textual source code. Visual representations are multi-dimensional, thus able to present multiple concurrent components naturally, while fine-grained concurrency control can be encapsulated in appropriate model feature semantics [Browne et al., 1994; Kleppe, 2008]. The SCOPE DSL must therefore provide a graphical concrete syntax.	
WF-02	*Model Editing*
The workbench must provide an editor for SCOPE models. In particular, models must be editable by their textual representation in order to support collaborative work.	
WF-03	*Model Validation*
The workbench must support the structural and domain-specific validation of SCOPE models. Errors and warnings should be indicated instantaneously and corrected automatically, if possible.	
WF-04	*Model Visualisation*
The workbench must provide a mechanism to render the visual representation of SCOPE models in order to support the dissemination and understanding of models.	

WF-05	*Iterative Development Process*
The workbench must support Coordination Engineering as an iterative development process to reduce cascading change effects when existing SCOPE models are manipulated.	
WF-06	*Transformation Editing*
The workbench must provide a mechanism to edit Parallel Platform Transformation sets.	
WF-07	*Transformation Execution*
The workbench must provide a mechanism to execute Parallel Platform Transformation sets.	
WNF-00	*Domain Conformance*
The SCOPE DSL's metamodel must be conformant to domain concepts, ideally providing distinct constructs for each distinct concept.	
WNF-01	*Business Process Model and Notation (BPMN) 2.0 Conformance*
The DSL must be conformant to the BPMN specification in order to benefit from its wide use and tool support. In particular, conformance to the SCOPE subset is required.	
WNF-02	*BPMN 2.0 Interoperability*
The workbench must be interoperable with BPMN tools in order to establish seamless development tool chains.	
WNF-03	*Genericity*
The workbench must be applicable to coordination in arbitrary application domains. It must not be restricted to be used in a certain application domain (e. g., business, health, or energy).	
WNF-04	*Extensibility*
The workbench must be extensible and adaptable to future requirements and uses.	
WNF-05	*Usability*
The workbench must be as comfortable as possible and meet current expectations to contemporary tooling.	

Chapter 8

Design of a Coordination Engineering Library and Workbench

In this chapter, we define the fundamental software architecture of a coordination library and a workbench to support Coordination Engineering with Space-Coordinated Processes (SCOPE). The chapter can be used as the basis to implement a SCOPE coordination library and a SCOPE coordination workbench.

For each of the two systems, the library and the workbench, we describe the platform- and implementation-independent components.

8.1 Coordination Library

The coordination library consists of the following components that are interconnected via interfaces, see Figure 8.1. The specification of the interfaces is not part of the component design.

- Scope: The component realises the SCOPE coordination model. It internally consists of the components Space Based System and Process Orchestration. It provides interfaces to access the two inner components, and requires an interface to attach the Logging component.
- Space Based System: The component realises an Space-Based Systems (SBS). It provides the functionalities to create and use spaces that provide the space operations considered in the SCOPE model, see Section 6.3. The component provides an interface to access its implementation.

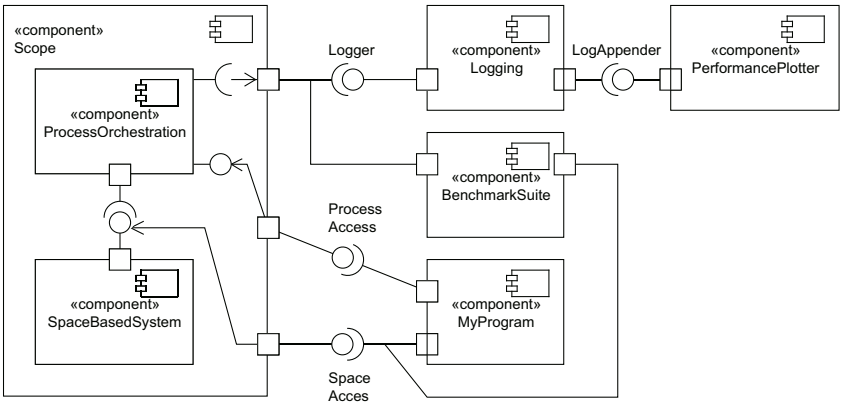

Figure 8.1. Component diagram of the coordination library.

- ▻ Process Orchestration: The component provides coordination-specific process components and abstract components to realise application-specific processes. These use the SBS to coordinate themselves, and control flows to orchestrate their internal activities. The component provides an access interface, and requires an interface to the Space Based System and an interface to attach the Logging component.
- ▻ Logging: The component realises a logging component. The component can be regarded as a generic logging service, but should be adaptable/able to produce custom log files. It provides an interface to access the Logging service, and requires an interface to attach custom log appenders to process log messages.
- ▻ Performance Plotter: The component realises a custom log appender that examines log messages to analyse and report the time behaviour of SCOPE programs in a statistically rigorous way. It provides an interface to append its functionality to the Logging component.
- ▻ Benchmark Suite: The component realises a benchmark suite to conduct several benchmarks on the SBS component. It provides the functionality

to conduct micro-benchmarks (individual space operations) and macro-benchmarks (compound space operations). The component requires an interface to access the Space Based System component, and an interface to attach the Logging component.

- My Program: The component is a placeholder for parallel programs realised with SCOPE. This includes application-specific benchmarks. It requires an interface to access the Space Based System component and one to access the Process Orchestration component. An interface to the Logging component is not required since application logging occurs through the Process Orchestration component.

8.2 Coordination Workbench

The coordination workbench consists of the following components that are interconnected via interfaces, see Figure 8.2. The specification of the interfaces is not part of the component design.

The component design assumes the coordination workbench to realise an *external* Domain-Specific Language (DSL) *piggybacked* on the Business Process Model and Notation (BPMN) specification, instead of an internal DSL embedded in a programming language, see Section 4.5.3. The reasons are a clear separation of higher-level coordination-related DSL code from lower-level computation-related programming code, sophisticated coordination-specific validation, and accessibility by non-programmers.

- Coordination Workbench: The component realises the coordination workbench. It consists of the components Model Registry, Scope Model, Scope Ui, Code Generator, and Bpmn Generator. It can be regarded as a generic platform for language workbenches that was extended by coordination-specific components. Additional generic platform details, such as generic workspace, user interface, and plug-in toolkits, are omitted.

- Model Registry: The component realises a registry for models that can be accessed by the components of the platform. This is, most importantly, the metamodel of the SCOPE coordination model. The purpose of the

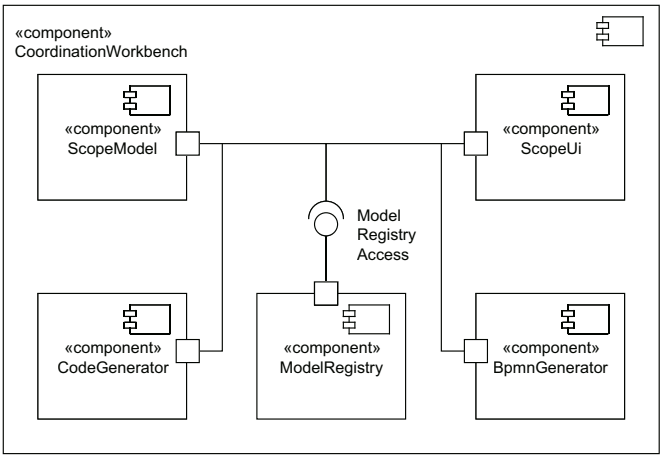

Figure 8.2. Component diagram of the coordination workbench.

central registry is to decouple components from each other. This, for example, makes it easier to extend the workbench by components that provide additional notations and interoperability transformations for the SCOPE metamodel. The Model Registry provides an interface to its service to make metamodels available and to access available metamodels.

- Scope Model: The component realises the SCOPE coordination model as a domain-specific language (DSL). This comprises a metamodel for SCOPE as well as a concrete textual syntax as the main notation of the language. The textual notation is preferred over a graphical syntax as the main notation of the DSL because it better supports editing, collaborative work, and version management. The component requires an interface to the Model Registry in order to make the SCOPE metamodel available to the other components.

- Scope Ui: The component realises an editor for the SCOPE DSL, including mechanisms for coordination validation and error fixing. To do so, it requires an interface to the Model Registry to access the SCOPE metamodel.

- Code Generator: The component realises a generator for coordination code. It comprises a transformation engine to execute Model-to-Text (M2T) transformations to the parallel platform, as well as an editor for the transformation engineer to produce sets of transformation, each set for a different parallel platform. It requires an interface to the Model Registry to access the SCOPE metamodel.

- Bpmn Generator: The BPMN generator represents a special generator that executes a predefined model-to-model transformation that maps the SCOPE metamodel to the metamodel of the BPMN specification. This enables to generate BPMN models and diagrams from SCOPE models that were created with the SCOPE DSL. The component requires an interface to the Model Registry to access the SCOPE metamodel and to provide the BPMN metamodel.

Part III

Evaluation

Chapter 9

Implementation

This chapter presents the implementation of a coordination library and a coordination workbench for Space-Coordinated Processes (SCOPE). For each of the two, we first describe the implementation decisions and then discuss the individual sub-projects.

9.1 Coordination Library

In this section we present the implementation of the Process Coordination Library (PROCOL), a library-based embodiment of the SCOPE coordination model.[1] Its purpose is two-fold: It provides a proof of concept for SCOPE, and eases the development of concurrent space-based programs in Java. PROCOL can be considered as an internal Domain-Specific Language (DSL) for SCOPE. All PROCOL projects are integrated with Maven for build support.[2]

9.1.1 Implementation Decisions

We made several decisions for the implementation of the coordination library. We document these decisions in the following. For convenience and traceability, we associated the decisions to the requirements described in Section 7.1. We also make references to the design of the coordination library, see Section 8.1.

[1] http://procol.sourceforge.net/
[2] http://maven.apache.org/

LF-00	*Performance Measurement*

Performance measurement is realised in the component Benchmark Suite. Since we adopt the space operations of the LighTS tuple space framework [Balzarotti et al., 2007] for the SCOPE coordination model (see Section 3.3.4 and Section 6.3), it uses the interface class `ITupleSpace` as its Space Access interface. The implementation makes use of the libraries *apache-commons-cli* to provide a robust command line interface, *log4j* as a generic logging component (i. e., the Logging component in Section 8.1), and *junit* for testing.

LF-01	*Performance Analysis*

Performance analysis is realised in the component Performance Plotter. The implementation distinguishes between start-up and steady state execution. Confidence intervals are used to discuss independent measurement series. Different techniques are used to compute confidence intervals based on whether the number of individual measurements for a series is large or not. The implementation makes use of the libraries *apache-commons-math* as a statistics component, *JFreeChart* to plot the average means and the confidence intervals of measurements series, *STAX* for processing Extensible Markup Language (XML)-based log files, *log4j* as a generic logging component (i. e., the Logging component in Section 8.1), and *junit* for testing.

LF-02	*Performance Visualisation*

The implementation of the Performance Plotter makes use of the library *JFreeChart* to plot the average means and the 95% confidence intervals of measurements series in high quality charts.

LF-03	*Benchmark Execution*

The experiments executed by the Benchmark Suite are directly concerning the space operations, thus supporting micro-benchmarks and macro-benchmarks natively. Application-specific benchmarks are not supported by the Benchmark Suite since PROCOL applications make use of the generic logging component *log4j* to measure their overall and per-component response time. For convenience, the logging functionality is encapsulated in the abstract classes of the implementation of the Process Orchestration component. In conformance to Fiedler et al. [2005] on Space-Based Systems (SBS) performance measurement, scalability is measured in terms of throughput and response time.

LNF-00 *Domain Conformance*

The domain concepts described in Section 6.3 are realised in the implementation of the components Space Based System and Process Orchestration. We selected the LighTS framework as a basis for implementing the component Space Based System since it provides a minimal, non-distributed (local) SBS implementation and an adaptation layer for own extensions. *junit* is used for testing. The implementation of the component Process Orchestration realises the coordination components of the SCOPE coordination model, see Section 6.3. Its implementation uses *apache-commons-cli* for a robust command line interface, *log4j* as a generic logging component, and *junit* for testing.

LNF-01 *Time Behaviour*

In SBSs, scalability is to a great extend determined by the ability of coordination components to exchange data objects via spaces. To provide a scalable coordination library, we provide several alternative space data structure implementations in the component Space Based System. The actual scalability and performance are examined in several experiments discussed in Section 10.2.

LNF-02 *Portability*

The coordination library is implemented in Java, a widely used General-Purpose Programming Language (GPL) based on a managed-runtime system, the so-called Java Virtual Machine (JVM). Java guarantees portability by providing JVM implementations for several platforms. As a consequence, the coordination library can be used on a variety of platforms, as long as there exists a JVM implementation for the platform.

LNF-03 *Genericity*

The support of genericity is two-fold: First, the use of the Java GPL does not restrict the implementation to a certain application domain. Second, the implementations of the components Space Based System and Process Orchestration do not make any assumptions about the application domain.

LNF-04	*Extensibility*
Extensibility is guaranteed by the adaptation layer provided by the LighTS framework in the Space Based System component, and the interface classes of the Process Orchestration component. Additionally, the ability of SBS to deal with incomplete knowledge and the LighTS-specific extensions for context-based systems makes it easy to provide future extensions for open systems. The coordination library itself is purely written in Java, which reduces the need to use additional technologies for own extensions.	
LNF-05	*Usability*
To increase the usability of the coordination library from the programmer's viewpoint, all component implementations of the coordination library are integrated with Maven for continuous build integration and dependency management.[a] Module tests are supported by *junit*.	

[a] http://maven.apache.org/

9.1.2 Project *procol-tuplespace*

The project *procol-tuplespace* realises an SBS. Regarding the coordination library software architecture described in Section 8.1, it represents the implementation of the component Space Based System.

We already constituted the use of the LighTS tuple space framework as a basis for implementation. We extended the framework as follows but retained its original interface (see Section 3.3.4). We refer to the original authors for an extensive overview of the LighTS framework [Balzarotti et al., 2007]. Table 9.1 summarises the synchronisation mechanisms we used in our extensions.

1. We exchanged the original synchronised `Vector`-based tuple space by a version based on the unsynchronised `java.util.LinkedList` to allow for more coarse-grained synchronisation and constant-time insert and removal operations.[3] Synchronising the tuple space operations is achieved

[3] http://docs.oracle.com/javase/tutorial/collections/implementations/list.html

using the `synchronized` keyword. The resulting tuple space (**TS**) is regarded as a straight-forward reference implementation.

2. We added a *CopyOnWriteTupleSpace* (**CoWTS**) based on `java.util.concurrent.CopyOnWriteArrayList`. Synchronising the blocking tuple space operations is achieved using the `synchronized` keyword. The non-blocking read operation is not synchronised since it is allowed to return a null value and operates on a local copy of the underlying array of the `CopyOnWriteArrayList`.[4] The operation `inp` must be synchronised. Otherwise, possibly interleaving `inp` operations can consume a tuple that conceptionally has already been consumed by another component.[5] The CoWTS is expected to perform well for read operations, but writes should add a performance penalty since mutative operations on the `CopyOnWriteArrayList` internally make a fresh copy of the underlying array.

3. We added a *ReadWriteLockedTupleSpace* (**RWLTS**) based on `LinkedList`, and `java.util.concurrent.locks.ReentrantReadWriteLock` and `java.util.concurrent.locks.Condition` for synchronisation. The RWLTS should show the effect of differentiating read locks from write locks in contrast to the TS, since `ReentrantReadWriteLock` allows multiple read locks to be held on the `LinkedList`.

4. We added a *ReadWriteLockedCopyOnWriteTupleSpace* (**RWLCoWTS**) based on `CopyOnWriteArrayList`. Synchronisation is achieved using `ReentrantReadWriteLock` and `Condition`. The non-blocking read operations are not synchronised.

5. We added a *ConcurrentHashMapTupleSpace* (**CHMTS**) based on `java.util.concurrent.ConcurrentHashMap`. Synchronising the tuple space operations is achieved using the `synchronized` keyword. Tuples are stored as values and keys are generated upon insertion as simple unique identifiers using `java.util.UUID`. The `count` operation does not require synchronisation.

[4] *"The iterator provides a snapshot of the state of the list when the iterator was constructed. No synchronization is needed while traversing the iterator. The iterator does NOT support the remove method."* (Java SE 1.5 API)

[5] We denote this behaviour as *defective reduplication*.

6. We added a *ReadWriteLockedConcurrentHashMapTupleSpace* (**RWLCHMTS**) based on `ConcurrentHashMap`. Synchronisation is achieved using `ReentrantReadWriteLock` and `Condition`. Tuples are stored as values and keys are generated upon insertion as simple unique identifiers using `java.util.UUID`.

7. We added a *PreselectingReadWriteLockedConcurrentHashMapTupleSpace* (**PRWLCHMTS**) based on `ConcurrentHashMap`. Synchronisation is achieved using `ReentrantReadWriteLock` and `Condition`. Tuples are stored as values and keys are generated upon insertion. A generated key consists of tuple arity (i. e., the number of fields), an encoded representation of field types, and a generated unique id using `java.util.UUID`. This enables to preselect tuple *candidates* before the actual matching by filtering with the first two matching rules *arity* and *type equivalence*. Filtering relies on splitting the keys to compute the candidates. Therefore, we suppose that this implementation can show a decreased performance in contrast to the other data structures. This is especially likely when there are many similar tuples in the space, since both the key set as well as the candidate set must be iterated and examined.

8. For convenience, we added a simple factory interface (and exemplary factories for CoWTS, RWLCoWTS, RWLTS, and TS) for setting up consistent tuple space infrastructures, and added Maven build integration, additional tests, and documentation. The overall project structure resembles that of the original LighTS framework with only small differences.

9.1.3 Project *procol-tuplespace-benchmark*

The project *procol-tuplespace-benchmark* realises an experimentation suite for the project *procol-tuplespace*. Regarding the coordination library software architecture described in Section 8.1, it represents the implementation of the component `Benchmark Suite`. Table 9.2 provides an overview of the package structure of the project.

Figure 9.1 and Figure 9.2 show Unified Modeling Language (UML) class diagrams that illustrate the main classes of the experimentation suite. The

Table 9.1. Overview of space operation synchronisation used in the different space implementations; • denotes synchronisation with the `synchronized` keyword; ⊗ denotes the use of `ReentrantReadWriteLock` and `Condition` (*r* for `readLock`, *w* for `writeLock`); ◯ denotes that no synchronisation is used for the respective space operation.

	TS	CoW-TS	RWL-TS	RWL-CoW-TS	CHM-TS	RWL-CHM-TS	PRWL-CHM-TS
out	•	•	⊗ *w*	⊗ *w*	•	⊗ *w*	⊗ *w*
outg	•	•	⊗ *w*	⊗ *w*	•	⊗ *w*	⊗ *w*
in	•	•	⊗ *w*	⊗ *w*	•	⊗ *w*	⊗ *w*
inp	•	•	⊗ *w*	⊗ *w*	•	⊗ *w*	⊗ *w*
ing	•	•	⊗ *w*	⊗ *w*	•	⊗ *w*	⊗ *w*
rd	•	•	⊗ *r*	⊗ *r*	•	⊗ *r*	⊗ *r*
rdp	•	◯	⊗ *r*	◯	•	⊗ *r*	⊗ *r*
rdg	•	◯	⊗ *r*	◯	•	⊗ *r*	⊗ *r*
count	•	◯	⊗ *r*	◯	◯	◯	◯

Table 9.2. Overview of the package structure of the project procol-tuplespace-benchmark.

`scalability.benchmark` Contains the main class of the project, `BenchmarkDriver`, and auxiliary enumeration types
`scalability.benchmark.*` Contains the individual experiments supported by the project
`scalability.exception` Defines a `BenchmarkException`
`scalability.interfaces` Defines the interface `IBenchmark` and appropriate abstract classes

implementation is based on a `BenchmarkDriver` that executes one or more experiments via `runExperiments`. These, in turn, execute a set of `IBenchmark` implementations via `runBenchmarks`. Between each benchmark, `runBenchmarks` makes use of the method `pause` to minimise possible interferences of the individual benchmarks: it pauses the suite for a certain time, invokes Garbage Collection (GC), and again pauses the application for a certain time. The other methods of the `BenchmarkDriver` are either auxiliary methods for logging or for command line interface construction. Table 9.3 shows an overview of the command line interface parameters and their effects on the experimentation suite. An exemplary command line invocation is shown in the following:

```
java jar procol-tuplespace-benchmark-<version>.jar -cl 1 -cu 128 -b CONSUMERPRODUCER
-bi 8 -li 10000 -rt 10000 -pt 2000 -a 0 -ts ALL
```

Implementations of `IBenchmark` instantiate a set of concurrent workload simulators that implement the `Callable` interface, see Figure 9.2. The simulators either simulate work given a fixed run-time for throughput measurement (i. e., implement `AbstractFixedRuntimeWorkloadSimulator`) or given a fixed number of workload iterations to measure the response time (i. e., implement `AbstractFixedIterationCountWorkloadSimulator`). The operations to be simulated in each workload iteration are implemented in the method `simulateWorkloadIteration`, which is required by the abstract superclass of the respective simulator. Conceptually, each benchmark consists of an implementation of `IBenchmark` (or `AbstractBenchmark`, to be more specific), and an implementation of one of the two abstract workload simulators. The benchmark suite considers the following benchmarks:

- `OutBenchmark`: Each workload iteration of a workload simulator publishes a simple tuple in the tuple space via the `out` operation.
- `OutInBenchmark`: Each workload iteration checks the availability of a certain tuple with a non-blocking `rdp`, and if there is no result, it produces one with `out`. Else, it consumes the tuple with the non-blocking `inp` operation.
- `OutReadBenchmark`: Each workload iteration checks the availability of a certain tuple with a non-blocking `rdp` and if there is no result, it produces

Table 9.3. Command line parameters for the project *procol-tuplespace-benchmark*.

`cl: int` The lower border of the examined concurrency interval; beginning with this value, the benchmark driver starts experiments with a number of workload simulators equal to the current concurrency level, doubling the level until the upper border is reached or excessed;[a] the value of *cl* is stored in the field `CONCURRENCY_LEVEL_LOWER`
`cu: int` The upper border of the examined concurrency interval; the value of *cu* is stored in the field `CONCURRENCY_LEVEL_UPPER`
`b: String` The benchmarks to be executed; allowed values are *OUT*, *OUTIN*, *OUTREAD*, *CONSUMERPRODUCER*, and *ALL*
`bi: int` The number of benchmark executions (iterations); the value of the parameter is stored in the field `BENCHMARK_ITERATIONS`
`li: long` The number of workload executions (iterations) used in each benchmark execution; the value of the parameter is stored in the field `WORKLOAD_ITERATIONS`
`rt: long` The benchmark runtime (if runtime-dependent) in milliseconds; the value of the parameter is stored in the field `RUN_TIME`
`pt: long` The pause-time with GC between individual benchmark executions in milliseconds; the value of the parameter is stored in the field `PAUSE_TIME`
`a: int` The tuple prepopulation count to simulate aged tuple spaces; the value of the parameter is stored in the field `AGEING_POPULATION_COUNT`
`ts: String` The tuple spaces to be benchmarked; allowed values of the parameter are *TS*, *RWLTS*, *COWTS*, *RWLCOWTS*, CHMTS, *RWLCHMTS*, and *ALL*
`h: boolean` A flag parameter that prints a help text on the command line, if used

[a] e. g., experiments with 1, 2, 4, 8, and 16 workload simulators for the interval $[1, 16]$

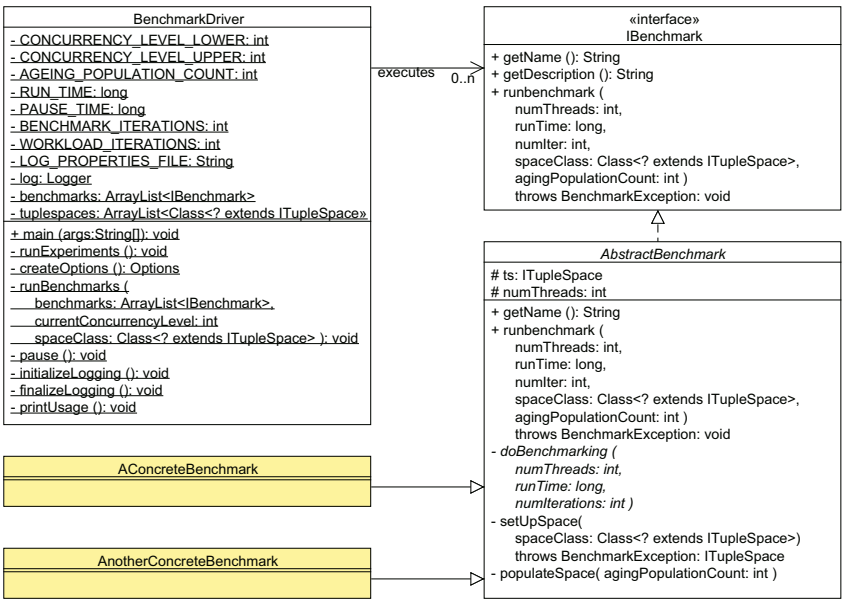

Figure 9.1. The UML class diagram for the benchmark driver and the benchmark interface classes.

one with `out`. Since the first workload iteration produces a tuple, this is a non-blocking read test.

▷ `ConsumerProducerBenchmark`: The Benchmark starts a number of n consumer (blocking `in` per workload iteration) and n producer simulators (`out` per workload iteration), where n is the number of workload simulators specified by the respective command line parameter.

The output of the benchmark suite is in XML format. The schema of the XML format is presented in Appendix A. Essentially, it consists of experiments that each consist of a number of benchmarks, whereas each benchmark comprises a set of measurements. A measurement, in turn, contains a data sample and defines its unit of measurement.

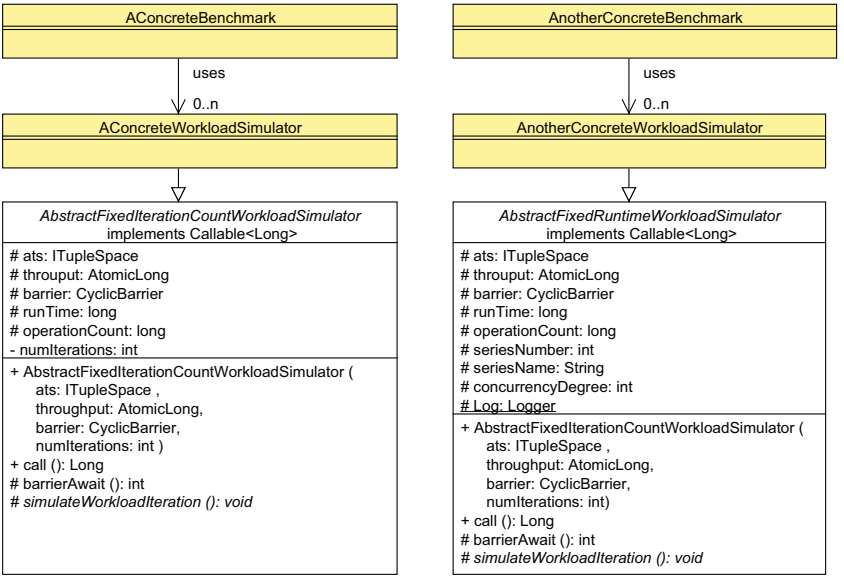

Figure 9.2. The UML class diagram for the workload simulators.

9.1.4 Project *procol-benchmark-logging*

The project *procol-benchmark-logging* realises a log analysis and plotting tool for the project *procol-tuplespace*. Regarding the coordination library software architecture described in Section 8.1, it represents the implementation of the component Performance Plotter.[6]

The project analyses the log files of the benchmark suite. It computes and plots the arithmetic mean and the 95% confidence interval for a series of measurements that describes the behaviour of a tuple space. Example plots can be found in Chapter 10. The confidence intervals are computed using the Central Limit Theorem and the sample standard deviation if the number of measurements is large ($n \geqslant 30$), and the Student's t-distribution

[6] As described in Section 9.1.1, the Logging component of the coordination library software architecture (Section 8.1) is realised by the generic *log4j* logging service.

when the number of measurements is small ($n < 30$). We refer to the work of Georges et al. [2007] as an excellent reading on the importance of confidence intervals for statistically rigorous Java performance evaluation.

Table 9.4. Overview of the package structure of the project procol-benchmark-logging.

`org.procol.framework.logging` Contains the main classes of the project, `PerformanceEvaluator`, `IntervalValuePlotter`, and `IntervalValuePlotterAppender`
`org.procol.framework.logging.util` Contains the auxiliary classes `SimpleStatistics` and `ConfidenceLevel`

Table 9.4 provides an overview of the package structure of the project. The implementation is based on the classes `PerformanceEvaluator`, `IntervalValuePlotter`, and `IntervalValuePlotterAppender`.

- `PerformanceEvaluator` parses XML log files from the benchmark suite and invokes `IntervalValuePlotter` for plotting. The XML files must conform to the schema presented in Appendix A. `PerformanceEvaluator` is either used as a standalone tool or invoked by `IntervalValuePlotterAppender`.

- `IntervalValuePlotterAppender` is a log appender for the *log4j* logging service. By implementing `org.apache.log4j.AppenderSkeleton`, it can be used as a log appender for the benchmark suite at runtime.

- `IntervalValuePlotter` represents the file plotter. It uses the `DeviationRenderer` from the *jFreeChart* library to plot the average mean and the confidence intervals for the given measurement series. The implementation supports up to eight different series with an individual layout color scheme each.

- `SimpleStatistics` is an auxiliary class that implements the formula for calculating confidence intervals, and `ConfidenceLevel` is an auxiliary class that represents a confidence level and the corresponding properties.

9.1.5 Project *procol-components*

The project *procol-components* provides coordination-specific process components for the realisation of application-specific processes. Regarding the coordination library software architecture described in Section 8.1, it represents the implementation of the component Process Orchestration. Table 9.5 provides an overview of the package structure of the project.

Table 9.5. Overview of the package structure of the project procol-components.

`org.procol.framework` Defines the generic abstract main class of the project, `AbstractAppRunner`
`org.procol.framework.components` Defines interfaces, abstract classes, and generic concrete implementations for coordination-specific process components
`org.procol.framework.exception` Defines exceptions for the project

The implementation fosters orchestration by introducing the interfaces `IComponent`, `ICompositeComponent`, and `IMultipleInstanceComponent`. While the former extends `java.util.concurrent.Callable`, the latter two extend `IComponent`. By convention, `IComponents` are only allowed to (a) communicate with each other indirectly using SBS, (b) concurrency is implemented by `java.util.concurrent.ExecutorService`, and (c) domain-specific computational *activities* are implemented using parameterless methods that use the private fields of the component.

Figure 9.3 presents an overview of the project as a UML class diagram. The implementation consists of the following interfaces and abstract classes:

- `IComponent` is a marker interface that specifies a `call` method that returns a *Boolean* value upon successful termination.
- `ICompositeComponent` specifies the methods `add`, `remove` and `getComponent` to handle *composition* (internal `IComponents`).
- `IMultipleInstanceComponent` describes components that instantiate and

execute *multiple instances* of a *single component*. The method `setInvokee` specifies the component to be invoked.

- `AbstractComponent` is an abstract implementation of `IComponent` that provides a constructor to initialise components with a map of known tuple spaces with which component instances can communicate. The `call` method is implemented to initialise runtime measurement, if `isReported` is *true*. The method `call` invokes `doExecute`, an abstract method that represents execution logic. It must be implemented by concrete implementations of `AbstractComponent`.

- `AbstractCompositeComponent` is an abstract implementation of `ICompositeComponent` that provides implementations for the methods `add`, `remove`, `getComponent`, and `call`. The class also extends `AbstractComponent`, meaning that the method `call` invokes `doExecute`, which, in turn, must be implemented by concrete implementations.

- `AbstractMultipleInstanceComponent` is an abstract implementation of `IMultipleInstanceComponent` that provides implementations for the methods `setInvokee` and `call`. The class also extends `AbstractComponent`, meaning that the method `call` invokes `doExecute`, which, in turn, must be implemented by concrete implementations.

- `AbstractAppRunner` is an abstract class that provides methods to enable performance monitoring for PROCOL programs. It is intended to be implemented by the main class of a PROCOL program. The `execute` method uses the methods `initializeLogging` and `finalizeLogging` to set-up and shut-down logging, and `runApp`. `runApp` is an abstract method that executes the coordination, including the set-up of the used SBS, and the initialisation of the participating process components.

To address concurrency and composition, the project provides the following concrete process component implementations:

- `MultipleInstanceParallelComponent` is an extension of `AbstractMultipleInstanceComponent` that *concurrently* instantiates and executes multiple instances of a single component as specified by the method `setInvokee`.

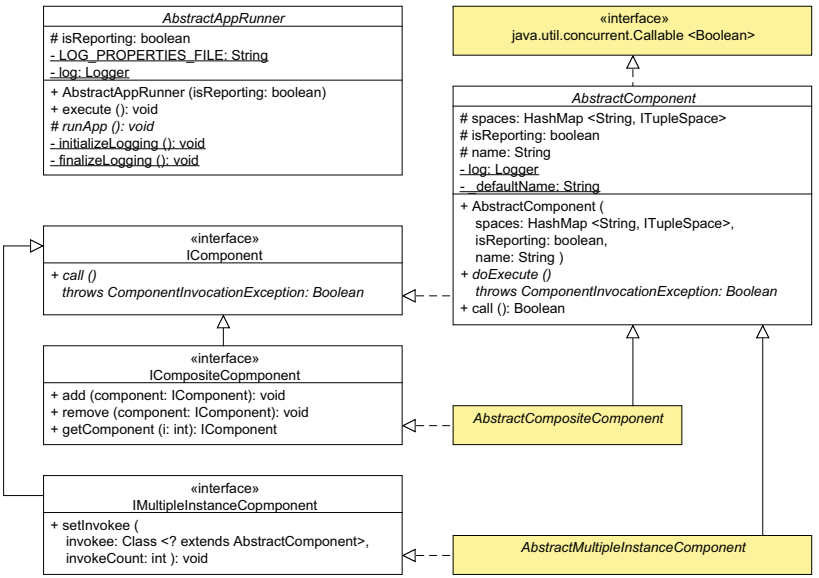

Figure 9.3. The UML class diagram for process orchestration.

To do so, it makes use of `java.util.concurrent.ExecutorService` and its `newFixedThreadPool` method to instantiate and execute implementations of `AbstractComponent` as `java.util.concurrent.Futures`.

▻ `MultipleInstanceSequentialComponent` is an extension of `AbstractMultipleInstanceComponent` that *sequentially* instantiates and executes multiple instances of a single component as specified by the component's constructor.

▻ `ParallelInvocationCompositeComponent` is an extension of `AbstractCompositeComponent` that *concurrently* instantiates and executes a number of different components as specified in the internal `List` populated by the method `add`. To do so, it makes use of `java.util.concurrent.ExecutorService` and its `newFixedThreadPool` method to instantiate and execute im-

plementations of `AbstractComponent` as `java.util.concurrent.Futures`.

▷ `SequentialInvocationCompositeComponent` is an extension of `AbstractCompositeComponent` that *sequentially* instantiates and executes a number of different components as specified in the internal `List` populated by the method `add`.

9.2 Coordination Workbench

In this section we present the implementation of a coordination workbench for SCOPE, including an external DSL.[7] The SCOPE DSL workbench uses the PROCOL coordination library as a platform for code generation.

9.2.1 Implementation Decisions

We made several decisions for the implementation of the coordination workbench that we document in the following. For convenience and traceability, we associated the decisions to the requirements described in Section 7.2. We also make references to the design of the coordination workbench, see Section 8.2.

We used the Xtext framework as a basis for DSL and workbench implementation [Xte, 2011]. Xtext is a mature language workbench framework that is well-integrated with the Eclipse Modeling Project, see Section 4.5.5. Its functionalities for syntax definition, model validation, and editor definition makes it an reasonable choice for rapid prototyping. Regarding the design of the coordination workbench described in Section 8.2, Xtext provides project structures for the components `Scope Model` and `Scope Ui` in the form of the projects *scope* and *scope.ui*, and acts as an interface framework to use the Eclipse Modeling Framework (EMF) as both a development platform and as a `Model Registry` component.

[7] http://scope-dsl.sourceforge.net/

WF-00 *Textual Representation*

The Xtext framework supports concrete syntaxes by nature. Syntax definition and validation are realised in the project *scope*, an implementation of the component Scope Model from Section 8.2.

WF-01 *Visual Representation*

The graphical concrete syntax of the SCOPE DSL is defined as the Business Process Model and Notation (BPMN) notation subset presented in Section 6.5. BPMN model output is guaranteed by the project *scope.generator.bpmn*, an implementation of the component Bpmn Generator from Section 8.2. The project uses *medini QVT* for Model-to-Model (M2M) transformation since it is integrated in the Eclipse Integrated Development Environment (IDE) environment and since it can support both the BPMN 2.0 metamodel as well as the SCOPE metamodel using EMF Ecore.[a]

WF-04 *Model Visualisation*

Model visualisation is considered as a matter of third party tooling in the responsibility of the application engineer. As an example, in-editor model visualisation can be realised by using the *BPMN2* component of the Eclipse *Model Development Tools (MDT)* project.[b].

WF-02 *Model Editing*
WF-03 *Model Validation*
WNF-05 *Usability*

The project *scope.ui* realises a textual editor for SCOPE models, including quick-fixes for common model errors, model completion, and an appropriate searchable outline view. The outline view uses BPMN notation elements to decrease the conceptual distance between the textual representation of the SCOPE DSL and the SCOPE BPMN subset defined in Section 6.5.

WF-05 *Iterative Development Process*

The transformation engine of *medini QVT* makes use of tracing information over multiple executions. Only the modified parts of a SCOPE model are transformed to BPMN when transformations are repeated. This facilitates a robust update behaviour of the SCOPE tooling. Changes to SCOPE models can be propagated to BPMN models even after the BPMN models were modified.

[a] http://projects.ikv.de/qvt
[b] http://www.eclipse.org/modeling/mdt/?project=bpmn2

WF-06	*Transformation Editing*
WF-07	*Transformation Execution*

Transformation editing and execution is supported by the integration of *medini QVT* in the coordination workbench.

WNF-00	*Domain Conformance*

Domain conformance is guaranteed by the abstract and concrete syntax of the SCOPE DSL, see Section 9.2.2.

WNF-01	*BPMN 2.0 Conformance*
WNF-02	*BPMN 2.0 Interoperability*

BPMN 2.0 conformance is based on the SCOPE BPMN subset from Section 6.5 and the identification of the related metamodel fragments in the BPMN 2.0 metamodel. The interoperability between the SCOPE workbench and third party tools that conform to the BPMN 2.0 specification is guaranteed by a M2M transformation defined with Query/View/Transformation (QVT) QVT [2011]. The QVT transformation maps all SCOPE elements and their relations to the corresponding BPMN elements and establishes the operational conformance of the SCOPE coordination model to the BPMN 2.0 specification.

WNF-03	*Genericity*

Neither the *Scope Model* nor the *Scope Ui* component implementation makes any restrictions to the targeted application domain. The SCOPE DSL provides a simple type system that allows to define domain specific data types by aggregation as part of the language.

WNF-04	*Extensibility*

Future extensions can be provided in different ways. First, toolsmiths can use the Eclipse plug-in mechanism for workbench extensions such as additional model editors, version management interfaces, and analysis tools. Second, transformation engineers can provide additional generator projects to map the SCOPE DSL to additional target platforms. Third, toolsmiths can extend the SCOPE DSL by modifying the Xtext syntax definition.[a]

[a] Language extension is relatively easy concerning the SCOPE DSL since both the textual notation of the language as well as its metamodel are inferred almost completely from the Xtext syntax definition. However, such modification can require cascading modifications of the validation, quickfix, and outline mechanisms of the workbench.

9.2.2 Project *scope*

The project *scope* realises the SCOPE coordination model as a DSL. It represents the implementation of the component `Scope Model` in the coordination workbench software architecture described in Section 8.2. Its development refers to the activity `Syntax Definition` in the Tool Engineering process, see Section 5.3.3.

The basic project structure is provided by the Xtext framework, see [Xte, 2011], and was extended as needed. It includes the source code folders `src`, `src-gen` and `xtend-gen`.

- `src` contains the implementation of the SCOPE DSL provided by the toolsmith. Table 9.6 provides an overview of the package structure of the `src`-folder of the project.

- `src-gen` contains DSL infrastructure code that is generated from the contents of the `src`-folder.

- `xtend-gen` contains the generated class `ScopeGenerator` that can be used for code generation.

Main Artefacts The package `org.xtext.scope` contains the main artefacts of the SCOPE DSL and its infrastructure:

- The `ScopeRuntimeModule` represents the main runtime configuration of the DSL infrastructure. It can be used to register custom components to be used at runtime.

- The `ScopeStandaloneSetup` allows the SCOPE DSL to run in standalone mode and initialises the Guice dependency injection framework used by Xtext.[8][9]

[8] http://code.google.com/p/google-guice/

[9] The term *standalone* refers to the use of an Xtext language without using the Eclipse IDE, or Eclipse Equinox extension points, respectively (http://www.eclipse.org/equinox/).

Table 9.6. Overview of the package structure of the `src`-folder of the *scope* project.

`org.xtext.scope` Contains the syntax definition of the SCOPE DSL and the main artefacts of the DSL infrastructure.
`org.xtext.scope.formatting` Contains means for language formatting. The package content is unmodified since we do not consider custom formatting.
`org.xtext.scope.generator` Contains means for code generation. The package is intended for Model-to-Text (M2T) transformation templates. We refrain from the Xtext convention of considering code generation as a part of the language definition since it impedes a proper separation of concerns and the establishment of software product lines by different generator sets, see Section 9.2.4.
`org.xtext.scope.scoping` Contains means for scoping.[a] The package content is unmodified since we leave custom scoping for future work.
`org.xtext.scope.serializer` Contains means for serialisation.[b] The package content is unmodified since we do not consider custom EMF serialisation.
`org.xtext.scope.typesystem` Contains the class `TypeSystemHelper`, a helper class that we introduced to provide several convenience methods for type system navigation.
`org.xtext.scope.validation` Contains means for validation.

[a] The term *scoping* denotes the calculation of which model elements can be referenced by a certain reference.

[b] The term *serialisation* denotes the transformation of EMF models to its textual representation to support DSL parsing and lexing.

- `GenerateScope` is a Modeling Workflow Engine (MWE) 2 workflow that triggers the Xtext framework to generate several DSL infrastructure components in the `src-gen` folders of the projects *scope* and *scope.ui*. The workflow allows to configure the generation with various components, for example, for grammar access and validation.

- `Scope` is the definition of the abstract and concrete syntax of the SCOPE DSL in the Extended Backus-Naur Form (EBNF)-like Xtext language.

- `ScopePostProcessor` is an Xtend file that modifies the Ecore metamodel of the SCOPE DSL after it was generated from the `Scope` Xtext syntax definition. We use it to create bidirectional relations between model elements, which are not directly supported by Xtext but exceptionally useful for M2T transformation template definition. The implementation creates a `EReference` back-reference from the contained element to its container, and sets each other as their respective `EOpposite`. The following Xtend code illustrates this pattern by the example of `Definitions` and their `Imports`.

```
process(ecore::EPackage this) :
  let d = this.eClassifiers.typeSelect(ecore::EClass) :

  d.select(e|e.name == "Import").first()
    .eStructuralFeatures
    .add(this.createBackRefFromImportToDefinitions())
  ->

  d.select(e|e.name == "Definitions").first()
    .eStructuralFeatures.typeSelect(ecore::EReference)
    .select(e|e.name == "imports").first()
    .setEOpposite(
      this.eClassifiers.typeSelect(ecore::EClass)
        .select(e|e.name == "Import")
        .eStructuralFeatures.typeSelect(ecore::EReference)
        .select(e|e.name=="definitionsRef").first())
  ->
  <...>
;

create ecore::EReference createBackRefFromImportToDefinitions(ecore::EPackage p) :
  this.setName("definitionsRef") ->
  this.setEType(p.eClassifiers.typeSelect(ecore::EClass)
  .select(e|e.name == "Definitions").first()) ->
  this.setEOpposite(p.eClassifiers.typeSelect(ecore::EClass)
```

```
    .select(e|e.name == "Definitions")
      .eStructuralFeatures.typeSelect(ecore::EReference)
      .select(e|e.name=="imports").first()) ->
    this.setLowerBound(0) ->
    this.setUpperBound(1) ->
    this
;
```

Abstract Syntax The concrete syntax of the SCOPE DSL is defined by the EBNF-like Xtext file `Scope` in the package `org.xtext.scope`. The abstract syntax of the language is defined in the Ecore metamodel that was inferred from the `Scope` Xtext file and modified by the `ScopePostProcessor`. It is located in the package `org.xtext.scope` in the folder `src-gen`. We discuss the abstract syntax in terms of the metamodel. The Xtext language definition can be found in Appendix B.

Figure 9.4 presents an overview of the SCOPE metamodel. The actual metamodel development can be considered as *language piggybacking* [Mernik et al., 2005]: Domain-specific elements were added to parts of a host language (the previously identified BPMN 2.0 subset).

First, we discuss which metamodel elements we retained from the BPMN 2.0 specification: The main element of a SCOPE model is Definitions. In accordance with the specification, we use it to define the scope of visibility and the namespace for all contained elements [Axway et al., 2010, p. 51]. The Import class is used to reference elements contained in other Definitions [Axway et al., 2010, p. 53]. The interaction between spaces and clients are defined by Collaboration, which defines them as Participants. Client Participants can reference a Process in order to be executable. Processes have Properties (renamed to Process Properties) and a Sequence Flows elements. We refactored the graph-based Sequence Flow modelling of the BPMN 2.0 standard into a block-based structure: Instead of defining the sourceRef and the targetRef of a single Sequence Flow, we define the order of execution of Flow Nodes within a Process by the order of references to them with the nodes property of the Sequence Flows or Conditional Sequence Flows elements. Start Events and End Events can be omitted since the order of the references indicate the start and end of the Sequence Flows. Flow Node is an abstract superclass for elements in Sequence Flows. As Flow Nodes, we consider the Exclusive Gateway and Activities. Activities can have Loop Characteristics to model looping or multiple

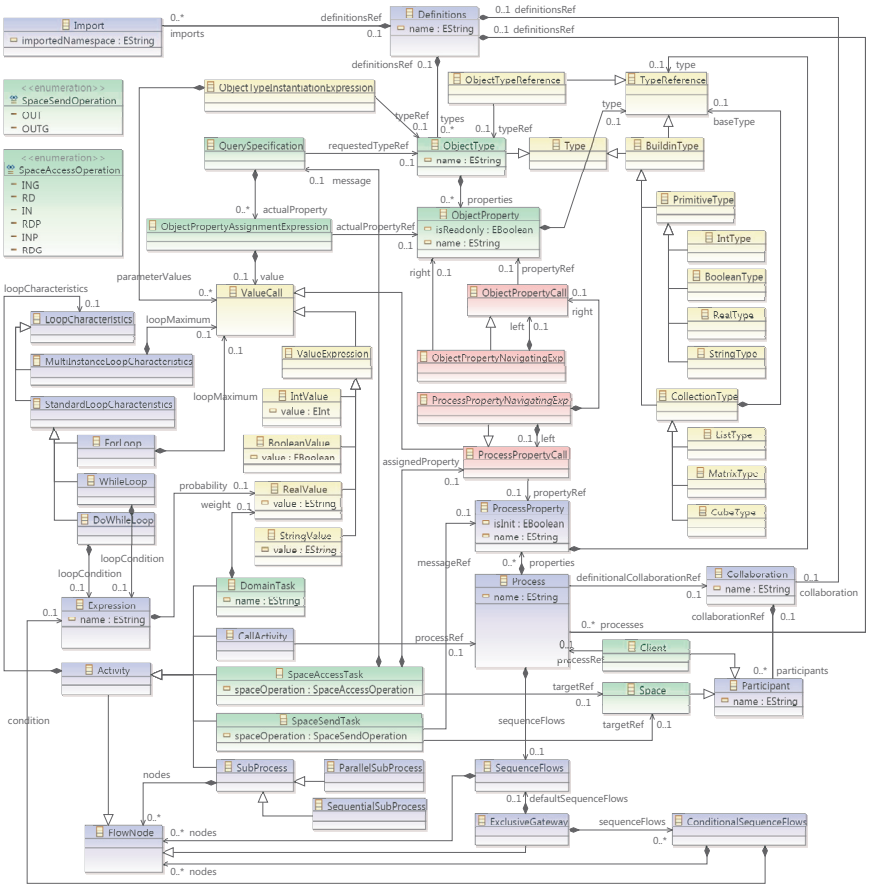

Figure 9.4. The metamodel of the SCOPE DSL.
Color print version: *blue*: elements from BPMN; *green*: SBS-specific elements; *yellow*: primitive type system; *red*: support elements from grammar definition with Xtext.

instantiation (using Standard Loop Characteristics or Multiple Instance Loop Characteristics). Conventional looping behaviour is distinguished in For Loop (pre-testing iterating loop), While Loop (pre-testing conditional loop), and Do While Loop (post-testing conditional loop). Structurisation is supported by Sub Processes: Sequential Sub Process allows to structure sequential activities (e. g., to loop over sequences), and Parallel Sub Process allows to concurrently execute different activities. The Call Activity can be used to invoke other Processes from the current one. Finally we model the conditions of Conditional Sequence Flows, While Loop, and Do While Loop as Expression declarations instead of providing a complete expression system.

We added the following elements to model space-based inter-process communication: We differentiate between Space and Client Participants. Spaces do not have a reference to a Process since their behaviour is assumed to be black-box. The Domain Task is an Activity (more specifically a custom BPMN 2.0 Task) that represents domain-specific logic. It has a weight attribute that allows software architects to specify a relative processing time estimate that can be used for simulation before domain-specific logic is actually implemented in subsequent *analysis* phases (see Section 5.3.1). The Space Send Task is a BPMN 2.0 Send Task that publishes data objects to a Space. The data objects to be published are specified by a reference to a Process Property. Complex data objects are defined as Object Types that have Object Properties. The Space Access Task represents the BPMN 2.0 idiom for consuming or reading data objects from Spaces. The data objects to be consumed or read are specified via a Query Specification that requests an Object Type. Actual values to be matched are identified with an Object Property Assignment Expressions that allows to assign the expected values to the Object Properties of the requested data object. Finally, Expressions have probabilities using Real Values. They can be used for simulation before domain-specific logic is implemented.

There are some additional classes in the metamodel: Process Property Call, Process Property Navigation Expression, Object Property Call, Object Property Navigation Expression, and Object Property Assignment Expression. They support the grammar definition in the Xtext framework. The differentiation between Process Properties and Object Properties also simplifies grammar definition. The elements Type and Value Call and their subclasses represent a simple

type system that can be extended in future versions of the SCOPE language.

Concrete Syntax The concrete syntax of the SCOPE DSL was developed with the Xtext language, see Appendix B. Appendix D and Appendix E shows examples of the concrete syntax. The example are discussed in the experiments described in Chapter 10.

In principle, sequences of activities are represented by a semicolon mark (";") after the individual statements, while concurrency is represented as either multiple-instantiated Call Activities ("`multi-instance(...) call ...`") or Parallel Sub-Processes ("`parallel {...}`").

We also use meaningful names for the coordination primitives: "`publish`" denotes the operation `out` and "`publish-all`" denotes `outg`. The notation for the consuming operations are "`take`" for `in`, "`take-available`" for `inp`, and "`collect`" for `ing`. Analogously, the notation elements for the reading operations are "`read`" for `rd`, "`read-available`" for `rdp`, and "`scan`" for `rdg`.

Finally, the concrete syntax employs an Structured Query Language (SQL)-like rendering for Object Property Assignment Expressions: "`where`"-clauses greatly simplify the specification of templates to be sent as queries to spaces. The ordering of the clause (the "`where`"-part before the "`from`"-part) is restricted by the language definition in Xtext. The following listing shows an example:

```
take Image where (Image.isRendered = false) from Mandelbrot.ImageSpace
```

Validation The package `org.xtext.scope.validation` implements DSL validation by the class `ScopeJavaValidator`. Table 9.7, Table 9.8, and Table 9.9 summarise the implemented model validations.

Table 9.7. Overview of the *information* validation for the SCOPE metamodel.

Informations
`DOMAIN_TASK_NO_WEIGHT` Informs that a Domain Task should be weighted.

Table 9.8. Overview of the *warning* validations for the SCOPE metamodel.

Warnings
`COLLABORATION_NAME_NOT_CAPITALIZED` Warns that the name of a Collaboration should start with a capital letter.
`PARTICIPANT_NAME_NOT_CAPITALIZED` Warns that the name of a Participant should start with a capital letter.
`PROCESS_NAME_NOT_CAPITALIZED` Warns that the name of a Process should start with a capital letter.
`OBJECT_TYPE_NAME_NOT_CAPITALIZED` Warns that the name of a Object Type should start with a capital letter.
`CLIENT_NO_PROCESS_REFERENCE` Warns that Client should reference a Process to be executable.
`EXPRESSION_EQUALS_ONE` Warns that the probability of a condition Expression should be lesser than `1.0` for simulation.

9.2.3 Project *scope.ui*

The project *scope.ui* realises an editor for the SCOPE DSL. Regarding the coordination workbench software architecture described in Section 8.2, it represents the implementation of the component Scope UI. Its development contributes to the activity Workbench Development in the Tool Engineering process, see Section 5.3.3.

Figure 9.5 shows a screenshot of the editor. The main editor window is located on the upper left side of the workbench. It shows an example of a SCOPE model with validation errors. Validation errors are also presented in the *Problems view* at the bottom of the workbench. In the main editor window, a pop-up dialogue provides so-called *quick fixes* for the error. To the right side the *outline view* presents an overview of the SCOPE model. It is synchronised with the main editor so that model elements selected in the outline view are highlighted in the textual notation. Additionally, the outline view provides a shortcut-triggered pop-up search dialogue

Table 9.9. Overview of the *error* validations for the SCOPE metamodel.

Errors
SPACE_ACCESS_GROUP_COLLECTION_TYPE_REQUIRED Checks that group Space Access Operations must be assigned to List Type properties.
SPACE_ACCESS_NONGROUP_NO_COLLECTION_TYPE_ALLOWED Checks that non-looping non-group Space Access Operations must not be assigned to a List Type properties.
SPACE_SEND_GROUP_COLLECTION_TYPE_REQUIRED Checks that group Space Send Operations must publish List Type properties.
SPACE_SEND_NONGROUP_NO_COLLECTION_TYPE_ALLOWED Checks that non-group Space Send Operations must not publish List Type properties.
PROCESS_MISSING_ATTENDANCE_IN_COLLABORATION Checks that Processes must attend a Collaboration.
EXPRESSION_GREATER_THAN_ONE Checks that the probabilities of condition Expressions must not be greater than 1.0.
EXPRESSION_BELOW_ZERO Checks that the probabilities of condition Expressions must not be smaller than 0.0.

$(STRG + o)$.

The basic project structure is provided by the Xtext framework, see [Xte, 2011]. It includes the source code folders `src`, `src-gen`.

- `src` contains the configuration and implementation of the SCOPE DSL editor provided by the toolsmith. Table 9.10 provides an overview of the package structure of the `src`-folder of the project.
- `src-gen` contains DSL editor code that is generated from the project *scope* and complemented by the contents of the `src`-folder.

Table 9.10 provides an overview of the package structure of the `src`-

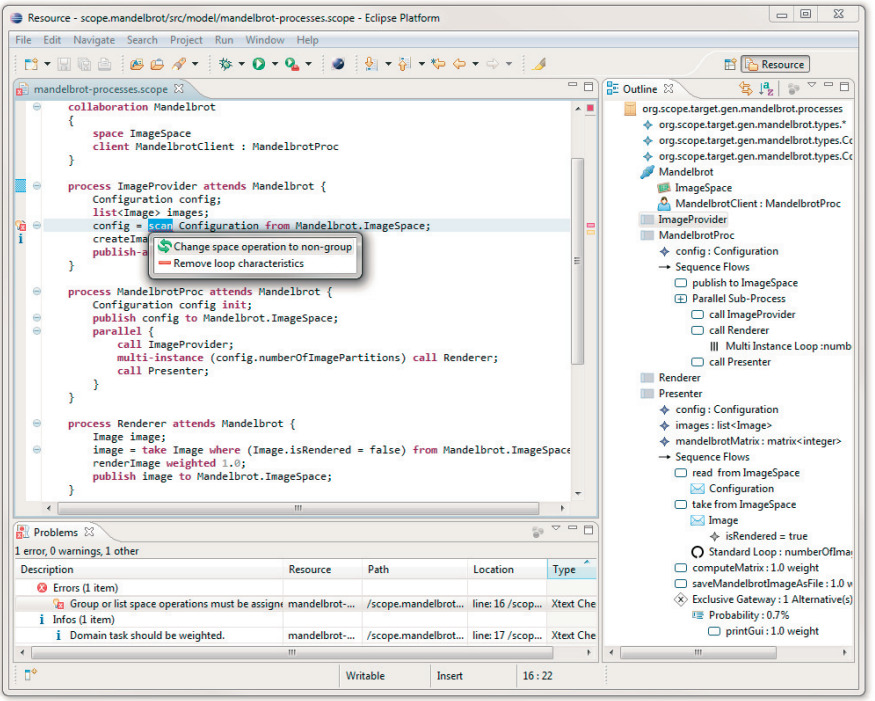

Figure 9.5. The editor for the SCOPE DSL.

folder of the project.

Outline View Xtext provides an outline view for model navigation that provides a hierarchical view on domain models [Xte, 2011]. By default, it shows the complete containment hierarchy without referenced elements. To improve the outline view we modified and extended the default implementation. First, we added several label implementations in the class `ScopeLabelProvider` located in the package `org.xtext.scope.ui.labeling`.

- Labels for metamodel elements that have references to other elements do not only contain the name of the elements, but also the referenced

Table 9.10. Overview of the package structure of the of the src-folder of the project scope.ui.

`org.xtext.scope.ui` Contains the generated `ScopeUiModule` that can be used to register additional custom components.
`org.xtext.scope.ui.contentassist` Contains means for content assistance such as proposals for element naming.
`org.xtext.scope.ui.labeling` Contains means for labelling. Text and image labels are used by the outline view, for example.
`org.xtext.scope.ui.outline` Contains the configuration of the outline view of the SCOPE DSL editor.
`org.xtext.scope.ui.quickfix` Contains means to provide quick fixes for validation.
`org.xtext.scope.wizard` Contains means to define a project wizard dialogue to set up new SCOPE projects.

element (e. g., the type of a Process Property).

- Labels of Exclusive Gateways also name the number of the alternative Sequence Flows.
- Conditional Sequence Flows labels show the probability of their execution in percent.
- Space Access Task and Space Send Task labels also name their target Ref. Additionally, Object Property Assignment Expression labels show the complete assignment.
- Domain Task labels show the weight attribute.
- Labels for Loop Characteristics show the loop Condition or the loop Maximum, respectively.

Second, we customised the default structure of the outline view by implementing the `ScopeOutlineTreeProvider` located in the package `org.xtext.scope.ui.outline`.

- ▻ We disabled the generation of child nodes in the outline hierarchy for the elements Object Property, Process Property, Object Property Assignment Expression, and Loop Characteristics.
- ▻ For Conditional Sequence Flows, we enabled child node creation for referenced nodes.
- ▻ For Call Activities and Domain Tasks, we enabled child node creation for referenced Loop Characteristics, if available.
- ▻ For Space Access Tasks and Space Send Tasks, we enabled child node creation for message Ref and referenced Loop Characteristics, if available.

Finally, we defined several image icons for the relevant metamodel elements in the class `ScopeLabelProvider` located in the package `org.xtext.scope.ui.labeling`. The image icons illustrate BPMN elements to minimise the conceptual distance between the SCOPE BPMN subset and the textual syntax of the SCOPE DSL.

Quick Fixes The Xtext framework allows to define so-called *quick fixes* to resolve model validation issues defined by implementations of `AbstractDeclarativeValidator`. Table 9.11 shows the implemented quick fixes for errors defined in the `ScopeJavaValidator` from the *scope* project[10]

9.2.4 Project *scope.generator*

The project *scope.generator* realises a generator for Java coordination code that uses the PROCOL library. Regarding the coordination workbench software architecture described in Section 8.2, it represents the implementation of the component Code Generator.

[10]The `ScopeJavaValidator` extends the `AbstractScopeJavaValidator`, which, in turn, is an extension of `AbstractDeclarativeValidator`. The `AbstractScopeJavaValidator` is generated by the Xtext framework.

Table 9.11. Quick fixes for model validation errors.

`fixSpaceAccessRemoveGroup` fixes `ScopeJavaValidator.SPACE_ACCESS_GROUP_COLLECTION_TYPE_REQUIRED` Changes the space operation to request non-group data objects.
`fixSpaceAccessRemoveLoop` fixes `ScopeJavaValidator.SPACE_ACCESS_GROUP_COLLECTION_TYPE_REQUIRED` Removes the loop characteristics of a Space Access Task to request non-group data objects.
`fixSpaceAccessNongroupDoesNotUseCollectionType` fixes `ScopeJavaValidator.SPACE_ACCESS_NONGROUP_NO_COLLECTION_TYPE_ALLOWED` Changes the Space Access Operation to query List Type data objects.
`fixSpaceSendGroupUsesCollectionType` fixes `ScopeJavaValidator.SPACE_SEND_GROUP_COLLECTION_TYPE_REQUIRED` Changes the Space Send Operation to publish non-List Type data objects.
`fixSpaceSendNongroupDoesNotUseCollectionType` fixes `ScopeJavaValidator.SPACE_SEND_NONGROUP_NO_COLLECTION_TYPE_ALLOWED` Changes the Space Send Task to publish List Type data objects.
`fixProcessAttendsCollaboration` fixes `ScopeJavaValidator.PROCESS_MISSING_ATTENDANCE_IN_COLLABORATION` Makes the Process attend an existing Collaboration; also creates a new Collaboration if none exists yet.

In contrast to the Xtext convention to consider code generation as a part of the language definition, the *scope.generator* is held separate from the language definition in the *scope* project. The reason lies in the separation of concerns: Coordination Engineering considers workbench and transformation development as two different activities that are executed by different roles – the toolsmith and the transformation engineer (Chapter 5). Holding the generator projects separate from the language definition allows to provide additional generators without modifying the language definition, thus fostering the establishment of software product lines.

Table 9.12 provides an overview of the package structure of the project.

Table 9.12. Overview of the package structure of the project scope.generator.

`org.scope.generator.java` Contains the `ScopeGenerator` MWE 2 workflow that performs the actual code generation.
`org.scope.generator.java.helper` Contains convenience classes. This is currently only the class `GeneratorHelper`. It provides internal Universally Unique Identifiers (UUIDs) reference names for the anonymous `Parallel` Sub-Processes and `Sequential` Sub-Processes.
`org.scope.generator.java.templates` Contains Xpand templates used by the `ScopeGenerator` workflow for M2T transformation.

Template Dependencies The *scope.generator* generates PROCOL Java coordination code by executing the `ScopeGenerator` MWE 2 workflow located in the package `org.scope.generator.java`. The `ScopeGenerator` uses the Xtext component `org.eclipse.xpand2.Generator` to expand the Xpand M2T transformation templates located in the package `org.scope.generator.java.templates`.

Figure 9.6 shows the call dependencies of the template files. For the sake of clarity, the figure shows the template files as black boxes, the individual template definitions are omitted. Instead, Table 9.13 presents a classification of the template definitions on a per-file basis:

- Top-level templates are directly expanded (i. e., executed) by the `ScopeGenerator`. They initiate code generation by calling file generator templates.
- File generators generate code files. The granularity of the individual template definitions is very coarse-grained. They call contextual templates for repetitive chunks of configurable code.
- Contextual templates are used to generate repetitive chunks of code that is configured by its context. The granularity of the individual template definitions is medium to coarse-grained. Contextual templates call type-system related templates.

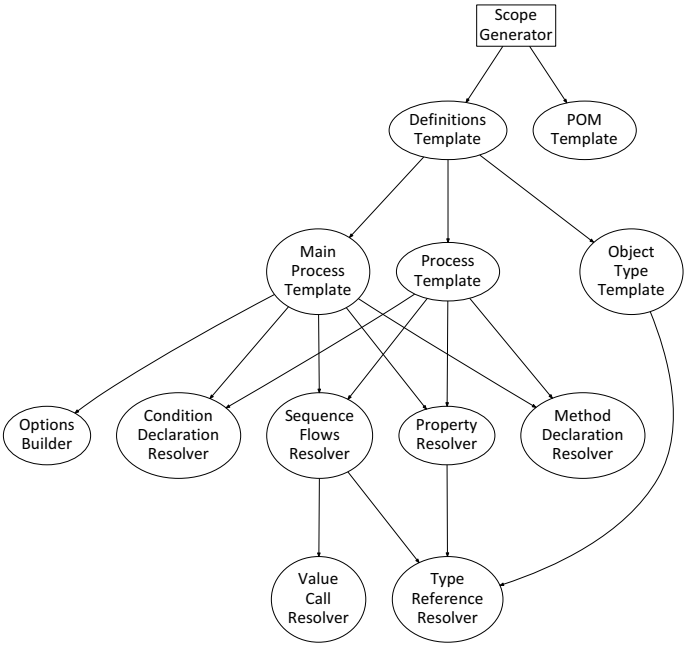

Figure 9.6. The call dependencies of the generator templates.

▻ Type-system related templates are used to generate type-system relevant code snippets. They map the primitive type system of the SCOPE DSL to the type and data object system of Java and the PROCOL library. Type-system related templates are fine-grained and recursive.

Code Generation The code generation for SCOPE programs is as follows. First, the `POMTemplate` generates a *pom.xml* file for dependency management and build integration with Maven. The *pom.xml* file represents the Project Object Model (POM) – the fundamental project configuration required by Maven to build target SCOPE projects. Appendix C discusses the project structure of target SCOPE projects.

Table 9.13. Generator template classification.

Template	*Classification*	*Description*
`Definitions-Template`	Top-level template	Triggers code generation for *src/main/java* and *src-gen/main/java*.
`POMTemplate`	Top-level template, file generator	Generates the fundamental POM configuration required by Maven to build SCOPE projects.
`MainProcess-Template`	File generator	Generates Java classes for SCOPE Processes that are directly referenced by a Client. Also generates the class `ExpressionHelper` for condition evaluation simulation to make the entire SCOPE program executable before domain code is supplemented.
`ProcessTem-plate`	File generator	Generates classes for all other SCOPE Processes that attend the Collaboration.
`ObjectType-Template`	File generator	Generates Java classes for SCOPE Object Types.
`Options-Builder`	Contextual template	Generates the parameter options for the command line interface of target SCOPE projects.
`MethodDecla-rationResol-ver`	Contextual template	Generates method declarations for the abstract and concrete classes that represent SCOPE Processes.
`Sequence-FlowsResolver`	Contextual template	Generates Java code for the Sequence Flows and the Flow Nodes of SCOPE models.
`ConditionDe-clarationRe-solver`	Contextual template	Generates abstract and concrete Java methods for condition Expressions.
`PropertyRe-solver`	Type-system template	Generates Java code for the Process Properties of a Process.
`Type-Reference-Resolver`	Type-system template	Generates Java code for Type References. Uses the fall-through semantics of Xpand templates to deal with complex metamodel element hierarchies.
`ValueCallRe-solver`	Type-system template	Generates Java code for Value Calls.

Second, the MainProcessTemplate, the ProcessTemplate, and the ObjectTypeTemplate generate Java code files. The Java files are generated into the source folders *src/main/java* and *src-gen/main/java*. The ScopeGenerator employs a conventional Generation Gap pattern [Vlissides, 1996]. Class inheritance is used to encapsulate the generated coordination code in abstract classes and domain code in concrete implementations:

The MainProcessTemplate generates classes for SCOPE Processes that are directly referenced by a Client. These are regarded as the main classes (entry points for execution). For each Process, the template generates three files:

- An abstract class Abstract<process-name> that extends org.procol.framework.AbstractAppRunner. It contains the coordination code that uses the PROCOL library. The file is generated to the folder *src-gen/main/java*.

- An abstract class Initialized<process-name> that extends Abstract<process-name>. It uses the library *apache-commons-cli* to provide a robust command line interface for the SCOPE project. The command line parameters are constructed from those Process Properties of the SCOPE Process whose isInit attribute is set. The names of a parameter is initially generated using the first character of the corresponding Process Properties supplemented with a deterministically generated integer id to prevent ambiguities. Object Type Reference properties are recursively explored until their primitive properties can be mapped to command line parameters. The file is generated to the folder *src/main/java*.

- A concrete class <process-name> that extends Initialized<process-name>. It contains empty method stubs for domain-specific logic to be provided by domain engineers. The file is generated to the folder *src/main/java*.

The ProcessTemplate generates classes for all other SCOPE Processes that attend the Collaboration. For each Process, the template generates two files:

- An abstract class Abstract<process-name> that extends org.procol.framework.components.AbstractComponent. It contains the coordination code that uses the PROCOL library. The file is generated to the folder *src-gen/main/java*.

- A concrete class `<process-name>` that extends `Abstract<process-name>`. It contains empty method stubs for domain code to be provided by domain engineers. The file is generated to the folder *src/main/java*.

`ObjectTypeTemplate` generates Java classes for SCOPE Object Types. The generated Java classes implement the interface `lights.utils.ITuplable` to flatten the Object Types into tuples with the method `toTuple`, and to recreate the objects from a retrieved tuple with `setFromTuple`, see Section 3.3.4. Both methods are completely generated, along with getter and setter methods for the corresponding `Object Properties`. The classes also implement the interface `java.io.Serializable` to enable the SBS component of the PROCOL library to handle deep object hierarchies. Java classes for Object Types are generated to the folder *src/main/java*.

Code generation was designed to allow the prototyping of SCOPE programs. In particular, the following aspects were considered to facilitate prototyping. First, if the weight attribute of a Domain Task is set, the `MethodDeclarationResolver` generates a `sleep` statement for the process. The statement simulates that the process takes time to perform some computation. The value of the weight attribute is multiplied with the factor 500, considered as milliseconds by the `sleep` statement. Second, the `MainProcessTemplate` generates the class `ExpressionHelper`. The class provides the method `boolExpr(double probability)` that returns *true* with a given probability. The class is used by Conditional Sequence Flows, While Loop, and Do While Loop to simulate the evaluation of boolean expressions to manipulate control flow behaviour. Both, computation simulation and expression evaluation simulation, should be replaced by domain code for further implementation.

9.2.5 Project *scope.generator.bpmn*

The project *scope.generator.bpmn* represents a M2M generator that generates BPMN models from SCOPE models. Regarding the coordination workbench software architecture described in Section 8.2, it represents the implementation of the component Bpmn Generator. Table 9.14 provides an overview of the folder structure of the project.

The package `org.scope.generator.bpmn` contains the Java implementation for command line usage. The class `ScopeToBpmn2QVTHeadlessGenerator`

Table 9.14. Overview of the folder structure of the project scope.generator.bpmn

src Contains the Java sources to execute the SCOPE to BPMN model transformation using the *medini QVT* transformation engine as a standalone command line tool.
model Can be used to contain example SCOPE models to test the QVT transformation.
qvt Contains the *scope2bpmn2.qvt* QVT transformation file.
traces Target folder for trace models generated by the *medini QVT* transformation engine upon transformation execution.

implements a QVT generator that uses the *scope2bpmn2.qvt* transformation file located in the project folder *qvt*. The implementation of the class is based on a modified version of the *medini QVT* Java integration example from the *medini QVT* web site.[11] `ScopeBpmnGenerator` extends `ScopeToBpmn2QVTHeadlessGenerator` and adds a command line interface using the Apache library *apache-commons-cli*. The command line parameters are described in Table 9.15. An exemplary command line invocation is shown in the following:

```
java jar scope.generator.bpmn-<version>.jar -i model/my.scope -o my.bpmn2 -d model/result"
```

At its core, the module executes a QVT Relations M2M transformation *scope2bpmn2.qvt*. The QVT transformation was developed with *medini QVT*. It is located in the folder *qvt*.

Figure 9.7 presents an overview on the transformation approach. The transformation rules map the SCOPE metamodel to the BPMN 2.0 metamodel. Thereby, a single BPMN model is generated from the SCOPE model by mapping the main Definitions and all imported Definitions namespaces of the SCOPE model to a single Definitions elements in the BPMN model. The project

[11] http://projects.ikv.de/qvt/wiki/integration

Table 9.15. Command line parameters of the project *scope.generator.bpmn*.

`input (-i): String` The input file path
`output (-o): String` The output file name
`outputDir (-d): String` The output directory path

uses the following plug-in dependencies as the source and target metamodels for the QVT transformation:

- *scope* defines the source metamodel for the transformation (see Section 9.2.2)
- *org.eclipse.bpmn2* defines the BPMN 2.0 target metamodel for the transformation; the component is a part of the MDT.[12]

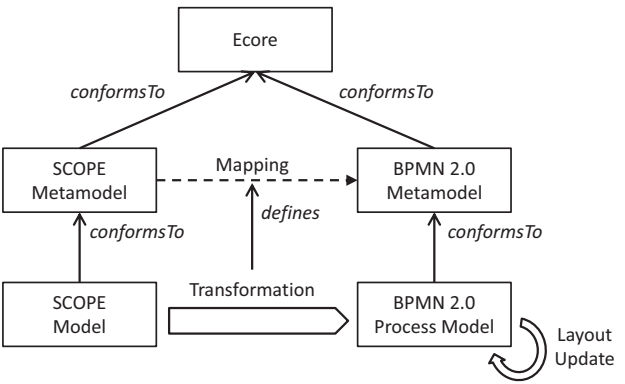

Figure 9.7. SCOPE-to-BPMN transformation approach.

[12]http://www.eclipse.org/modeling/mdt/?project=bpmn2#bpmn2

Graphical BPMN models require additional layout information. These can be generated after the transformation. We considered simple default layout information since the development of layout algorithms for complex graphical languages such as the BPMN 2.0 are out of the scope of this work. If necessary, an appealing layout can be created by subsequent stakeholders in any BPMN 2.0 conformant modeling tool.

The QVT transformation engine makes use of tracing information over multiple executions. In subsequent transformations, only *changes* to a SCOPE model are transformed to BPMN. This facilitates a robust update behaviour of the tooling. Changes to SCOPE models can be propagated to BPMN models even after the BPMN models were modified. Thus, the QVT transformation establishes the conformance of the SCOPE coordination model to the BPMN 2.0 specification.

9.3 Categorisation of the Implementation

We can categorise our SCOPE implementation considering the taxonomy for high-level concurrent software engineering presented in Section 2.5. Table 9.16 shows an overview of the categorisation of the SCOPE implementation as a high-level concurrent software engineering approach.

The SCOPE workbench can clearly be identified as an IDE for Model-Driven Software Development (MDSD). It applies Parallel Algorithmic Skeletons in the form of M2T transformation templates. The SCOPE DSL can be used to illustrate and transport coordination architecture patterns. Since the generated coordination code is almost directly executable by itself, the SCOPE workbench can also be used for prototyping (i. e., the exploration of system features in an early state of development). The workbench can be considered as the embodiment of a coordination-specific DSL for concurrent programs that is based on a graph-based abstraction (i. e., the BPMN 2.0 specification) and intended to be mapped to a SBS-based coordination library as its execution platform. Data parallelism can be modelled with Multi Instance Loop Characteristics.

The PROCOL coordination library can be considered as a conventional object-oriented library that can also be regarded as an internal DSL. Data

Table 9.16. Categorisation of the SCOPE implementation; • denotes direct category membership; ◯ denotes amenability.

	SCOPE DSL	PROCOL
Techniques		
Autotuning		◯
Pattern languages	◯	◯
Skeletons	•	◯
Prototyping	◯	◯
IDEs	•	◯
MDSD	•	◯
Approaches		
Library	◯	•
Data parallelism	◯	◯
Functional programming		
Logic programming		
Functional logic programming		
COOP		•
Coordination	•	•
Graph	•	
DSL	•	•

parallelism can be expressed using `MultipleInstanceParallelComponents`. In principle, PROCOL can be combined with other high-level techniques such as autotuning to find the best degree of concurrency for multi-instance process components. As a conventional programming library, it is also amenable for design patterns, skeleton/template definition, IDE supported programming, and code generation with MDSD techniques.

Chapter 10

Experiments

This chapter documents several experiments that we conducted to illustrate the feasibility of our approach. First, we discuss the relevant performance metrics. Second, we justify the SCOPE coordination model by protobenchmarks we conducted with the PROCOL coordination library. Third, we illustrate the Coordination Engineering approach by two experiments using the Space-Coordinated Processes (SCOPE) coordination workbench: an experiment on the visualisation of the Mandelbrot set on the complex plane, and one on Point-Feature Label Placement (PFLP) in Enterprise Architecture Visualisation (EAV). For each of the experiments, we also document the experimental configuration. The chapter is concluded with a discussion of the threats to validity.

10.1 Performance Metrics

The performance metrics used in this chapter focus on throughput and response time. Following the work of Fiedler et al. [2005], we measured the scalability of a Space-Based Systems (SBS) S_i

- in terms of the throughput $X_{bench_q}(S_i, O_{bench})$ in relation to the number of workload simulators q given a fixed response time T_{bench_q}, or
- as the response time T_{bench_q} given a fixed number of completed space operations $L_{bench_q}(S_i, O_{bench})$, see equation (10.1).

Thereby, *bench* denotes the respective experiment and O_{bench} denotes the (primitive or compound) operations to be completed. Subsequently,

we define the scalability of a space-based data structure as presented in Definition 1.

$$X_{bench_q}(S_i, O_{bench}) = \frac{L_{bench_q}(S_i, O_{bench})}{T_{bench_q}} \tag{10.1}$$

Definition 1 (Scalability) *A space-based system S_i is scalable, if*

$$\left(\begin{array}{ll} R_{throughput_{bench}}(S_i, O_{bench}, n, m) = \frac{Xbench_n(S_i, O_{bench})}{Xbench_m(S_i, O_{bench})} \leqslant 1 & \wedge \\ T_{bench_n} = T_{bench_m} \quad \wedge \quad T_{bench_n}, T_{bench_m} > 0 & \wedge \\ n < m \quad \wedge \quad n, m \in \mathbb{N} & \end{array} \right) \quad \vee$$

$$\left(\begin{array}{ll} R_{resptime_{bench}}(S_i, O_{bench}, n, m) = \frac{T_{bench_n}}{T_{bench_m}} \geqslant 1 & \wedge \\ L_{bench_n}(S_i, O_{bench}) = L_{bench_m}(S_i, O_{bench}) & \wedge \\ L_{bench_n}(S_i, O_{bench}), L_{bench_m}(S_i, O_{bench}) > 0 & \wedge \\ n < m \quad \wedge \quad n, m \in \mathbb{N} & \end{array} \right)$$

10.2 Justification of the Coordination Model

To justify the SCOPE coordination model presented in Chapter 6 as a reasonable solution for concurrent programming in Java, we investigate the behaviour of the Process Coordination Library (PROCOL) library (an internal Domain-Specific Language (DSL) for SCOPE). We consider two quantitative questions regarding the realisation and application of PROCOL, as shown in Figure 10.1:

1. Which space-based data structures *scale well* on multi-core machines (*realisation*)?

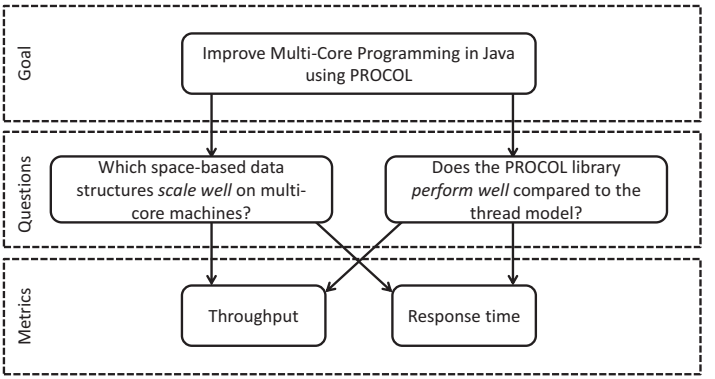

Figure 10.1. The quantitative questions and metrics considered according to the Goal Question Metric approach by Basili et al. [1994].

2. Does the PROCOL model show a *reasonable performance* compared to the lock-based thread model (*application*)?

To answer these questions, we conducted a comparative evaluation of two representative scenarios: the scalability of different PROCOL tuple space data structures, and the scalability of a Mandelbrot application that is implemented with the PROCOL library compared to an equivalent application implemented with the standard Java thread model. The conclusions drawn from these experiments are that in principle, SBS implementations can be provided that scale reasonably well on multi-core architectures, and that the performance overhead imposed by the PROCOL library is at least for the considered application a reasonable trade-off for the provided benefits.

In the following, we first present the experimental configuration. Second, we evaluate the scalability of several space-based data structures that we have implemented (question 1). Third, we evaluate two equivalent programs that compute the Mandelbrot set concurrently and that show it on the complex plane. The first is implemented using the PROCOL library, and the second using the Java standard thread model (question 2).

10.2.1 Experimental Configuration

To minimise interferences between the individual benchmark runs of an experiment the `BenchmarkDriver` pauses for 5,000 ms, then invokes the Java Garbage Collection (GC), and pauses again for 5,000 ms between benchmark runs. We executed 12 runs for each experiment, where the first two are excluded as warm-up runs to reduce the effects of bytecode interpretation and Just-In-Time (JIT) compilation [Goetz, 2009]. We use the arithmetic mean and the 95% confidence interval for the remaining ten results. The confidence intervals allow us to address two questions: Do different tuple space implementations show significantly different behaviour or not?[1] How large is the performance variation of the individual measurements for a single tuple space implementation? Experiments that implement `Abstract-FixedRuntimeWorkloadSimulator` executed workload simulation iterations for 10,000 ms. Unless otherwise stated, the benchmarks used primitive type tuples as data objects (floats and integers). The benchmarks were conducted on four machines M0 to M3. M1 and M2 are identical machines with different operating systems and Java environments. M0, M2 and M3 use the same operating system. M0 and M2 also use the same Java Version. This means that differences in the execution behaviour of M0 and M2 must be a matter of hardware, and differences between M1 and M2 must be a matter of different software stacks:

- **(M0)** T6340, 2xUltraSparcT2+ 8core 1.4Ghz, 16x4GB FB DIMM, Solaris 10, Java Version 1.6.0_21, Java(TM) SE Runtime Environment (build 1.6.0_21-b06), Java HotSpot(TM) Server VM (build 17.0-b16, mixed mode)
- **(M1)**: X6270, 2xIntel Xeon E5540 4core 2.53GHz, 12x2GB DDR3-1066MHz ECC, Debian Lenny, Java Version 1.6.0_0, OpenJDK Runtime Environment (build 1.6.0_0-b11), OpenJDK 64-Bit Server VM (build 1.6.0_0-b11, mixed mode)
- **(M2)** X6270, 2xIntel Xeon E5540 4core 2.53GHz, 12x2GB DDR3-1066MHz ECC, Solaris 10, Java Version 1.6.0_16, Java(TM) SE Runtime Environment

[1] If the confidence intervals for two implementations do not overlap, the performance difference between the two is significant. Else, it is most likely caused by random performance variations in the system under measurement [Georges et al., 2007].

(build 1.6.0_16-b01), Java HotSpot(TM) Server VM (build 14.2-b01, mixed mode)

▻ **(M3)** X6240, 2xAMD Opteron 2384 4core 2.7GHz, 8x2GB DDR2-667, Solaris 10, Java Version 1.5.0_20, Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_20-b02), Java HotSpot(TM) Server VM (build 1.5.0_20-b02, mixed mode)

10.2.2 Data Structure Benchmark Results

Figures 10.2, 10.3, 10.4, 10.5, and 10.6 show the results of different experiments regarding the appropriateness of different Java Data structures as a SBS component. The experiments can be regarded as microscopic (individual operations) and macroscopic (compound operations) proto-benchmarks.[2] In the following, we discuss the results of each experiment separately, and summarise the general observations at the end of the section.

The figures do not contain the measurements for the PRWLCHMTS since it showed catastrophic response time behaviour. For example, in previous test runs on M0 the PRWLCHMTS took an average of 2,013,632.33 milliseconds (33.56 minutes) to complete the `ConsumerProducerBenchmark` using 2 workload simulators. A possible reason is that there are many similar tuples in the space so that both the key set as well as the candidate set contain all entries of the tuples so far produced. Both must be iterated and examined for matching. However, the response time of the PRWLCHMTS is about magnitudes worse than that of the RWLCHMTS so that we must consider the possibility that the implementation of the PRWLCHMTS contains defects that we have not yet identified. Deemed as impractical in its current implementation, we excluded the PRWLCHMTS from the discussion of benchmarks.

[2] Sim et al. [2003] state that a benchmark has three components, a *motivating comparison* that states its purpose for comparison and its need for research, a representative *task sample*, and *performance measures* that illustrate its fitness for purpose by showing how the benchmarked technology is used. In addition, they denote benchmarks that lack one of these components as *proto-benchmarks*. Consequently, we identify ours as proto-benchmarks since there is no standardisation upon the task samples nor the measures.

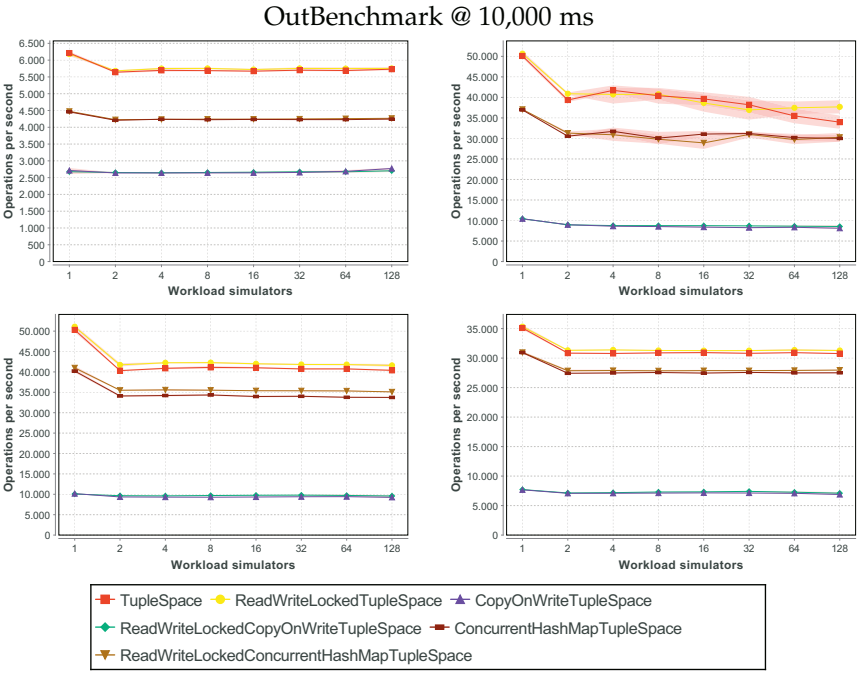

Figure 10.2. Results of the OutBenchmark on machines M0 to M3 from left to right, top to bottom.

OutBenchmark Each workload iteration of the workload simulator publishes a simple tuple in the tuple space via the `out` operation. Regarding the *OutBenchmark*, we see a drop of throughput between the use of 1 and 2 workload simulators on all machines. For example, $R_{throughput_{Out}}$ $(S_{TS}, O_{Out}, 1, 2)$ = 1.25 on M2. Except for that, all six tuple space implementations scale well on M0, M2, and M3. On M1, the throughput of the TS increases slightly from 2 to 4 workload simulators ($R_{throughput_{Out}}$ $(S_{TS}, O_{Out}, 2, 4)$ = 0.94) and then decreases again ($R_{throughput_{Out}}$ $(S_{TS}, O_{Out}, 4, 128)$ = 1.23). We can see that the 95% confidence intervals of the TS and the RWLTS, and of the CHMTS and the RWLCHMTS mostly overlap. This means that the

behaviour of the traditionally synchronised implementations is not significantly different from the read-write-locked versions. In general, the CoWTS and the RWLCoWTS show a much lower performance than the TS and the RWLTS due to the copy-on-write behaviour of the underlying `CopyOnWriteArrayList`. Also, the CHMTS and the RWLCHMTS show a lower performance than the TS and the RWLTS.

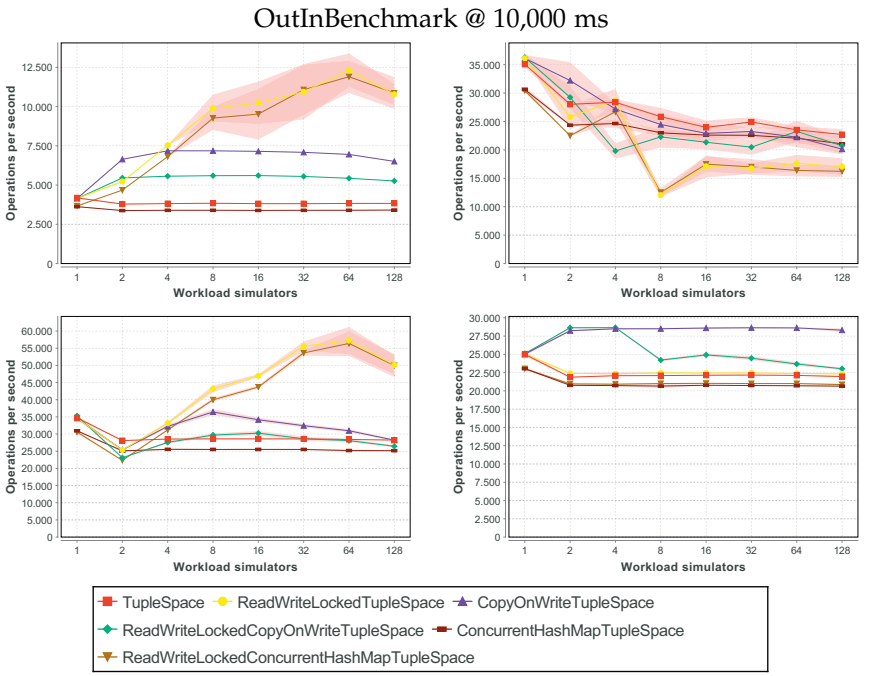

Figure 10.3. Results of the OutInBenchmark on machines M0 to M3 from left to right, top to bottom.

OutInBenchmark Each workload iteration checks the availability of a certain tuple with a non-blocking `rdp`, and if there is no result, it produces one with `out`. Else, it consumes the tuple with the non-blocking `inp` operation.

The RWLTS and the RWLCHMTS perform surprisingly well on M0 and M2 with increasing performance until reaching an observable peak at 64 workload simulators, then levelling off. A possible reason is the differentiation of read and write locks that let both excel in reading tuples. The decreasing performance after the peak can be explained by effects of write locking when the number of concurrent simulators increases. We can also see a large performance variation for both implementations on M0, which starts when 4 workload simulators are used, as indicated by their confidence intervals. When 8 workload simulators or more are used both implementations show similar behaviour since their confidence intervals begin to overlap. On M1 we see a strong decrease in throughput and a fluctuating behaviour of the RWLTS and the RWLCHMTS. This is especially the case between 4 and 8 workload simulators. For example, $R_{throughput_{OutIn}}(S_{RWLTS}, O_{OutIn}, 4, 8)$ = 2.38 and $R_{throughput_{OutIn}}(S_{RWLTS}, O_{OutIn}, 8, 16)$ = 0.70. Using 16 simulators or more stabilises throughput. A possible explanation is contention with OS and VM threads. As their overlapping confidence intervals indicate, the RWLTS and the RWLCHMTS show similar behaviour. In contrast, the RWLCoWTS shows significantly different behaviour. This is likely due to the fact that the RWLCoWTS does not have to synchronise the `rdp` operation. All in all, throughput decreases from 1 to 128 workload simulators for all implementations on M1. On machine M2, the overall throughput of the RWLTS and the RWLCHMTS is much higher than on M1, and the other implementations scale better than on M1. Since M2 is hardware-wise similar to M1, we can identify the software stack of M1 as inferior compared to that of M2. On M3, the CoWTS and the RWLCoWTS scale well, with increasing throughput between 1 and 2 workload simulators. While the CoWTS scales constantly when using more than 2 simulators, the throughput of the RWLCoWTS decreases between 4 and 8 simulators before it stabilises. Again, a possible explanation is OS and VM thread contention. The other implementations, on the contrary, show a small drop of throughput between the use of 1 and 2 workload simulators ($R_{throughput_{OutIn}}(S_{RWLTS}, O_{OutIn}, 1, 2)$ = 1.12) and a constant scalability from 2 to 128 simulators. Except for that, the TS and the CHMTS scale well on M0, M2, and M3. They show a worse performance than the CoWTS and the RWLCoWTS since the former employ conventional synchronisation for the space operations, and the latter do not

employ any synchronisation for non-blocking read operations at all. Finally, the CHMTS and the RWLCHMTS show a slightly worse performance than the TS and the RWLTS. A possible reason is the additional synchronisation in the `ConcurrentHashMap` of the CHMTS and the RWLCHMTS in contrast to the TS and the RWLTS, which internally use the unsynchronised `LinkedList`.

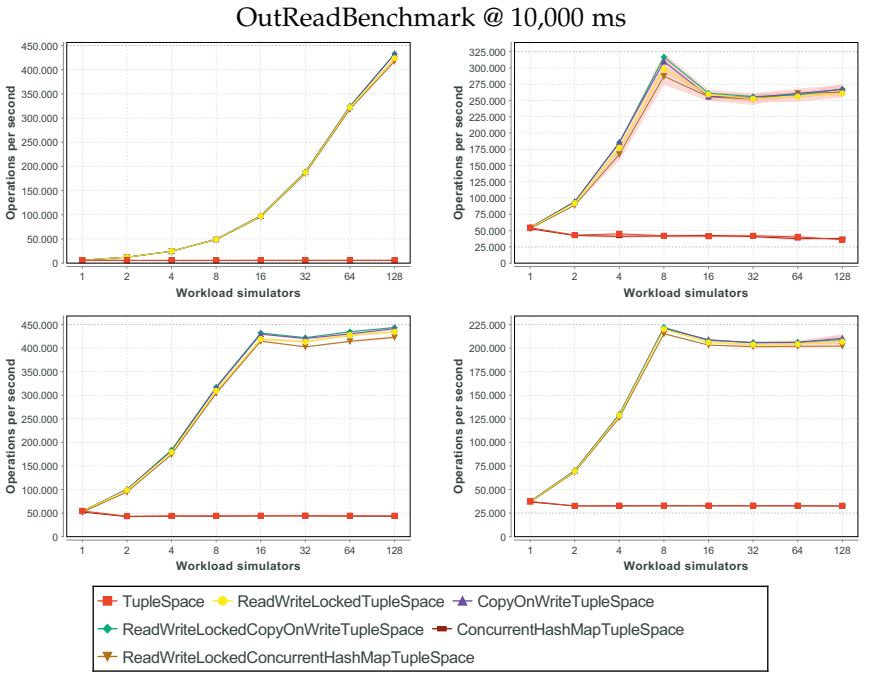

Figure 10.4. Results of the OutReadBenchmark on machines M0 to M3 from left to right, top to bottom.

OutReadBenchmark Each workload iteration checks the availability of a certain tuple with a non-blocking `rdp` and if there is no result, it produces one with `out`. Since the first workload iteration produces a tuple, this is a non-blocking read test. The TS and the CHMTS scale constantly since

their operations are synchronised with the `synchronized` keyword. The CoWTS, the RWLTS, the RWLCoWTS, and the RWLCHMTS perform exceptionally well, with increasing performance until 8 workload simulators on M1 and M3, and until 16 workload simulators on M2 ($R_{throughput_{OutRead}}(S_{RWLCHMTS}, O_{OutRead}, 1, 16) = 0.13$). Then their performance turns into a near-constant throughput on a high level of performance ($R_{throughput_{OutRead}}(S_{RWLCHMTS}, O_{OutRead}, 16, 128) = 0.98$). An obvious explanation is that concurrency can not further be exploited for real concurrency. On M1, the decease of throughput of the CoWTS, the RWLTS, the RWLCoWTS, and the RWLCHMTS between 8 and 16 workload simulators is much greater than on M2. Again, we can identify the software stack of M1 as inferior compared to that of M2 since both are hardware-wise similar to each other. On M0, throughput continues to increase up to 128 workload simulators, the exact number of logical processing elements of M0 (2 processors with 8 cores each, which each support 8 logical processing elements). The reason for the overall behaviour of the implementations is that the OutReadBenchmark is essentially a non-blocking read benchmark. The CoWTS's and the RWLCoWTS's `CopyOnWriteArrayList` can excel since their `rdp` operation is not synchronised. The differentiation of read and write locks suggest that the RWLTS and the RWLCHMTS also excel at read benchmarks. The TS and the CHMTS can not speed up since their `rdp` operation is conventionally synchronised.

ConsumerProducerBenchmark The Benchmark starts a number of n consumer (blocking `in` per workload iteration) and n producer simulators (`out` per workload iteration), where n is the number of workload simulators specified by the respective command line parameter. We observe constant and almost identical response time behaviour for the TS and the RWLTS, and the CHMTS and the RWLCHMTS given a fixed number of operations to be performed. Thereby, the CHMTS and the RWLCHMTS show a lesser and less scaling performance than the TS and the RWLTS. Again, an explanation is the additional synchronisation in the `ConcurrentHashMap` of the CHMTS and the RWLCHMTS. The CoWTS and the RWLCoWTS show less scaling response time behaviour and noticeable performance variations in the measurements as indicated by their confidence intervals. Most striking,

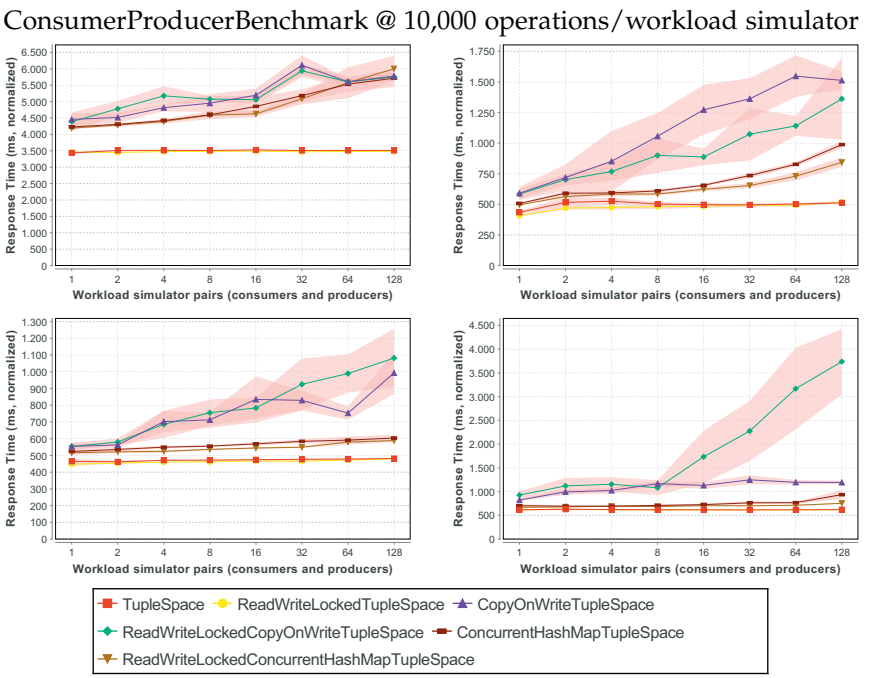

Figure 10.5. Results of the ConsumerProducerBenchmark on machines M0 to M3 from left to right, top to bottom.

on M3 the response time of the RWLCoWTS starts to increase continuously from 8 to 128 workload simulators in contrast to the other implementations. Surprisingly, the conventionally synchronised CoWTS scales well on M3 although we would have expected a similar performance penalty for both implementations caused by the copy-on-write behaviour of the underlying `CopyOnWriteArrayList`. The reason for the observed behaviour remains to be revealed by further investigation.

AgeingBenchmark We exemplary examined the effects of tuple space ageing on the results of the ConsumerProducerBenchmark, see Figure 10.6.

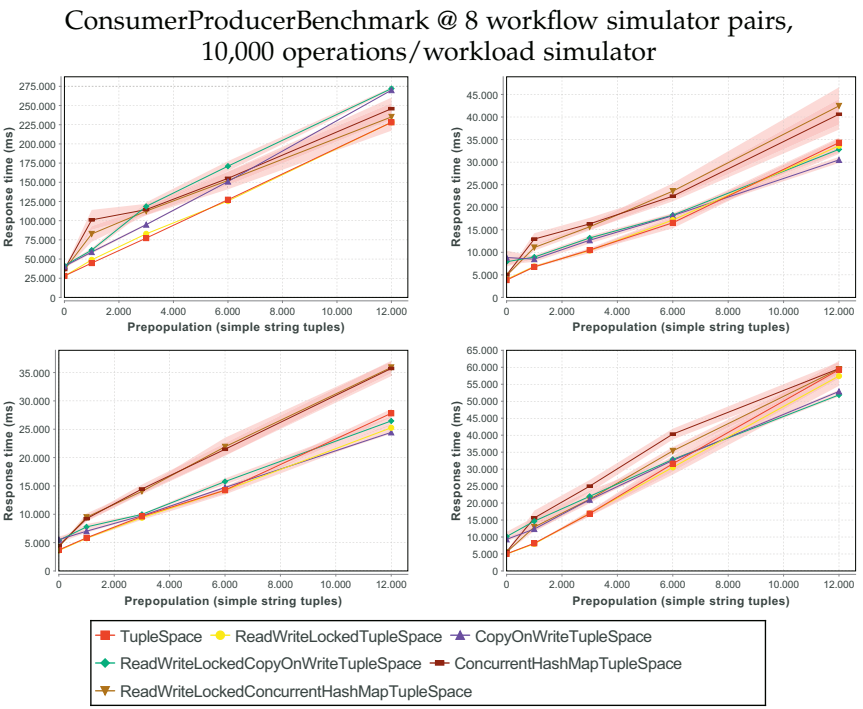

Figure 10.6. Effects of tuple space ageing on the ConsumerProducerBenchmark on machines M0 to M3 from left to right, top to bottom.

The benchmarks used 8 producer and 8 consumer workload simulators as the degree of concurrency. We pre-populated the tuple spaces with different numbers of tuples that contained a single string value each. We used different values for each string tuple. The benchmark indicates worst-case performance since the matching algorithm for tuple lookup iterates the tuple space in a straight-forward manner. Response time increased by about 741.97% from a zero pre-population to a pre-population of 12,000 tuples regarding the RWLTS on M1, for instance. The reason is the primitive matching algorithm. It has to iterate over all the string tuples in the aged

space each time a relevant tuple should be retrieved. A more sophisticated matching algorithm can certainly mitigate the effects of ageing up to a certain degree.

Table 10.1. Percentaged results of benchmark runs on M0 compared to M1, M2, and M3 using the TS with one (1) workflow simulator.

	Performance of TS on M0 compared ...			
	$X_{bench_1}(S_{TS}, O_{bench})$			T_{bench_1}
	Out	OutIn	OutRead	ConsumerProd.
to M1	12.40%	11.93%	11.39%	12.66%
to M2	12.35%	12.04%	11.39%	13.54%
to M3	17.69%	16.74%	16.63%	17.99%

Table 10.2. Percentaged results of benchmark runs on M0 compared to M1, M2, and M3 using the TS with eight (8) concurrent workflow simulators.

	Performance of TS on M0 compared ...			
	$X_{bench_8}(S_{TS}, O_{bench})$			T_{bench_8}
	Out	OutIn	OutRead	ConsumerProd.
to M1 :	14.08%	14.86%	13.69%	14.32%
to M2 :	13.84%	13.43%	12.96%	13.44%
to M3 :	18.42%	17.39%	17.39%	17.55%

General observations We observed a lower overall performance of M0 compared to the other machines. For example, the number of completed space operations per second $X_{Out_1}(S_{TS}, O_{Out})$ is only about 12.4% of that on M1, as shown in Table 10.1 and Table 10.2. Except for the AgeingBenchmark, the benchmarks were executed as single VM invocations. Benchmarking with multiple parallel VMs can show different results. Second, the reason for the different behaviour of the space implementations on M1 in contrast to M2 must be a matter of using different operating systems and Java VMs

since M1 and M2 are hardware-wise similar. This is especially the case for the OutInBenchmark, in which the performance of the RWLTS and the RWLCHMTS on M1 differs significantly to that on M2. It seems that for non-block `inp` in operations, the software stack of M2 provides better and more reliable results than that of M1. We suppose that the OpenJDK VM is not as optimised as its Java HotSpot counterpart. For most of the tuple spaces we observed a noticeable overhead for data structure management. As a consequence, further optimisations of the PROCOL framework should not only focus on improving the matching algorithm but on reducing space management overhead in general to improve scalability. However, in each of the benchmarks we found tuple space implementations for which our definition of scalability (Definition 1) holds or is only slightly violated. The conclusion drawn from the results is that scalable SBS implementations can be provided given that the effects of ageing are not yet considered. This is at least the case for multi-core machines with a number of cores that is in the single-digit range, since the machines on which the benchmarks were conducted only have 4 and 8 core processors.

We give the following recommendations which tuple spaces should be used:

- The CoWTS and the RWLCoWTS represent a reasonable choice when non-blocking read operations are used heavily since they do not need to synchronise these. On the other hand, they do not show a significantly better performance than the RWLTS, and are clearly not advisable for applications that favour blocking `in` and `out` operations over `rd`.

- As a general recommendation, we advise to use the RWLTS over the other implementations since its throughput and response time behaviour scale reliably in most benchmarks, and since it performs very well for applications that heavily depend on read operations. The RWLCHMTS can be regarded as an alternative to the RWLTS. It shows similar reliable behaviour as the RWLTS, but its performance is worse.

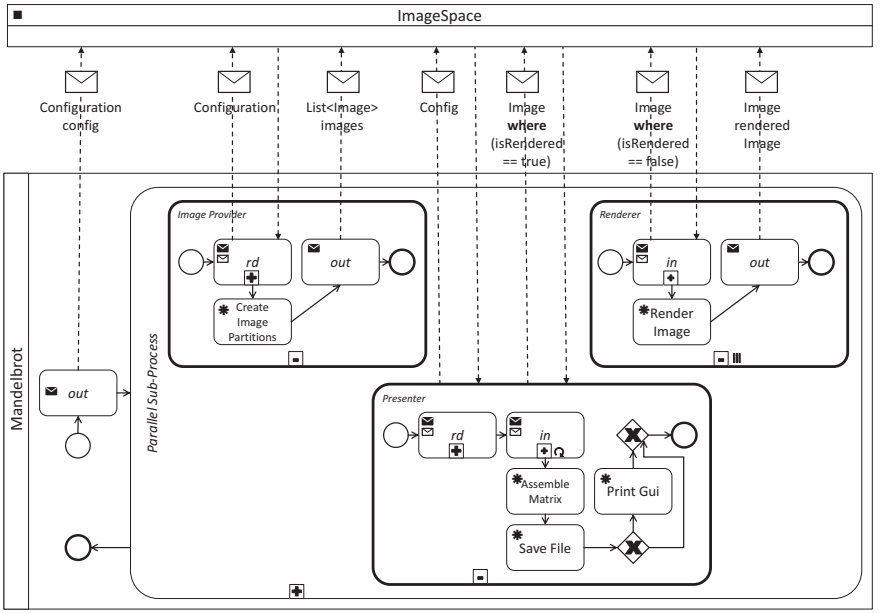

Figure 10.7. Coordination model of the Mandelbrot application (illustrated using the SCOPE BPMN subset).

10.2.3 Mandelbrot Set Visualisation Results

Using PROCOL, we created a program that computes the Mandelbrot set concurrently and shows it on the complex plane. The program represents an application benchmark for massive concurrency since each point of the complex plane can be computed independently. Figure 10.7 shows an overview on the coordination model of the application using the SCOPE Business Process Model and Notation (BPMN) subset.

The two participants of the program are ImageSpace and Mandelbrot. The former represents the SBS, and the latter represents the main client process. The Mandelbrot process mainly consists of a Parallel Sub-Process that invokes the processes Image Provider, Renderer, and Presenter. While the Mandelbrot process represents an implementation of `AbstractAppRunner`, the called pro-

cesses are IComponents that extend the abstract class AbstractComponent. The Parallel Sub-Process is realised as a ParallelInvocationCompositeComponent. The multiple-instance behaviour of the Renderer is realised using the concrete class MultipleInstanceParallelComponent.

The behaviour of the program is as follows: Firstly, the Mandelbrot's main method outs a configuration data object which is constructed from the program's command line parameters. Second, all domain-specific components are started concurrently with a Parallel Sub-Process. The components coordinate themselves along the data flows of the space operations. The ImageProvider provides a set of unrendered image slices that are consumed by as many concurrent Renderer instances as there are image slices in order to produce rendered images. While Renderer encapsulates the domain knowledge to compute the Mandelbrot set on the plane, Presenter consumes rendered image slices, assembles them into a single image, and saves it into a file.

As Figure 10.8 shows, the application scales very well on the given machines using the RWLTS until 8 concurrent *Renderer* instances. The figure shows the mean of ten measurements after two warm-up runs. Regarding M1, we see that the response time decreased from 2,781.2 ms to 1,806.8 ms ($R_{resptime_{Mandel}}(S_{RWLTS}, O_{Mandel}, 1, 8) = 1.54$) until 8 concurrent *Renderer* instances were used, then begun to increase back to 2,561.6 ms using 265 concurrent *Renderers* ($R_{resptime_{Mandel}}(S_{RWLTS}, O_{Mandel}, 16, 256) = 0.71$). Also, we can see that the application scales better on machines M0 and M3 than on M1 and M2 when more than 8 Renderer instances are used. A possible reason is that incipient effects of hyperthreading impose a performance penalty while up to using 8 renderers all renderers can be mapped directly to the cores of the two Xeon processors of M1 and M2. We also measured the execution times of each component individually to provide a hint for the increase in response time.

Measuring and logging the execution times of the individual components of the program has virtually no effect on the overall response time (Mandelbrot-Procol vs. Mandelbrot-Procol-Silent). The performance variation of Mandelbrot-Procol-Silent at 128 Renderers on M3 originates from a noticeable outlier in the measurements. Figure 10.9 illustrates that the Presenter component represents a constant serial fraction of the overall response time

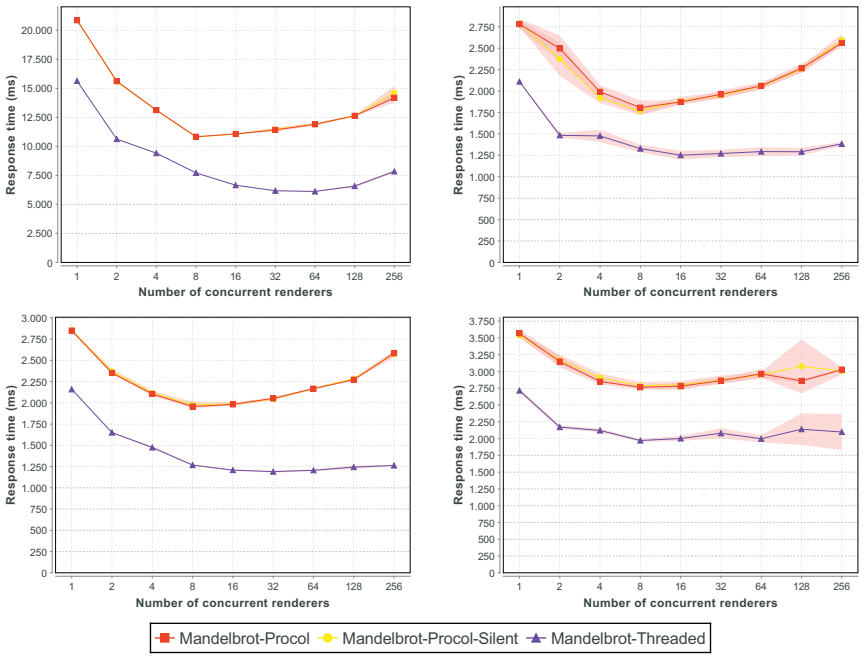

Figure 10.8. Results of the Mandelbrot program using the RWLTS on machines M0 to M3 from left to right, top to bottom; the command line parameters were: image width (*-w*): `1200` pixels, image height (*-ht*): `1024` pixels, number of image partitions and Renderer instances (*-r*), top left complex number on plane (*-tl*): $(-1) + 1i$, bottom right complex number on plane (*-br*): $1 + (-1)i$.

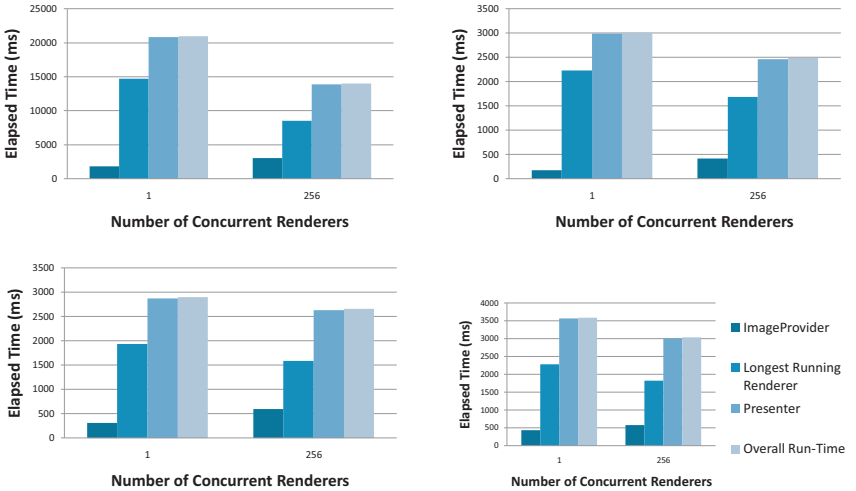

Figure 10.9. Results of the individual Mandelbrot program components at 1 and 256 concurrent Renderer instances on machines M0 to M3 from left to right, top to bottom.

and that the execution time of the Renderers decrease as more Renderer instances are used. The response time of the ImageProvider increases slightly with an increasing number of Renderer instances. A possible reason is that it must prepare and publish more image partition tuples the more Renderer instances are used. Another explanation is that the `ExecutorService`'s thread management can show a noticeable effect on response time with an increasing number of Renderer instances although there is no data to be shared among threads. In each case, however, the response time using 256 Renderer instances never exceeded the time elapsed when only one Renderer instance was used.

As a reference application, we implemented a version of the Mandelbrot application that is solely based on the standard Java thread model. Its implementation follows the instructions of the project course "Mandelbrot

Set Visualization" held by Daniel Tang at the Purdue University in 2009[3]. Since there is no data to be shared among threads, there is no synchronisation needed and the application measurements can be regarded as a best-case measurements. We see that in all cases the difference between the measurements of the space-based solution and those of the reference application was less than an entire order of magnitude (which we regarded as a KO criterion). For example, when using 8 (256) concurrent Renderer instances, the response time of the PROCOL implementation takes 1.33 (1.81) times longer to execute on M0, 1.32 (1.85) times longer on M1, 1.32 (2.05) times longer on M2, and 1.31 (1.44) times longer on M3. We conclude that the performance overhead that the PROCOL programming model imposes is at least for the Mandelbrot application a reasonable trade-off for the benefits that the PROCOL programming model provides.

10.3 Application of Coordination Engineering

In this section, we demonstrate the application of Coordination Engineering with SCOPE by two examples, a variant of the Mandelbrot visualisation program from Section 10.2.3, and a program component to address the PFLP problem in an existing framework for EAV. The two examples serve the following purposes:

The Mandelbrot visualisation program is used to answer the question if programs development with Coordination Engineering and SCOPE scale well for a best-case example. The motivation is to collect evidence about if the approach can exploit a provided opportunity for massive parallelism or not. It is also used to make a comparison to the behaviour of the Mandelbrot visualisation program version that was realised with the conventional Java thread model (see Section 10.2.3).

The PFLP-solving program component is used to illustrate the Application Engineering process within Coordination Engineering (see Section 5.3). It can be considered as a non-best-case example since PFLP is an Non-Deterministic Polynomial-Time (NP)-hard global (i. e., combinatorial) optimisation problem, see [Christensen et al., 1995]. Finally, the example

[3] http://web.ics.purdue.edu/~cs180/Fall2009Web/projects/p3/

provides insights on applying Coordination Engineering with SCOPE within an existing application.

10.3.1 Experimental Configuration

We conducted performance experiments with SCOPE programs for concurrency degrees ranging from 1 to 256. The mapping of the degree to concurrent components is dependent on the respective program. To minimise interferences between individual program runs we pause between each run for 5,000 ms. Furthermore, we executed 12 runs for each experiment, where the first two are excluded as warm-up runs to reduce the effects of bytecode interpretation and JIT compilation [Goetz, 2009]. We use the arithmetic mean and the 95% confidence interval for the remaining ten results. The confidence intervals allow us to address the questions of how large the performance variation of the individual measurements for a single programs is, and if different program versions show significantly different behaviour or not (if confidence intervals overlap or not).

The benchmarks were conducted on four machines M0 to M3. For the label positioning example (Section 10.3.3), we also used a test machine that represents the environment in which the program is most likely used. Note that the machines M0 to M3 show a slightly different configuration than the configuration described in Section 10.3.1 regarding the Java versions used:

- **(M0)** T6340, 2xUltraSparcT2+ 8core 1.4Ghz, 16x4GB FB DIMM, Solaris 10, Java Version 1.5.0_32, Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_32-b05), Java HotSpot(TM) Server VM (build 1.5.0_-32-b05, mixed mode)
- **(M1)**: X6270, 2xIntel Xeon E5540 4core 2.53GHz, 12x2GB DDR3-1066MHz ECC, Debian Lenny, Java Version 1.6.0_18, OpenJDK Runtime Environment (IceTea6 1.8.13) (6b18-1.8.13-0+squeeze1), OpenJDK 64-Bit Server VM (build 14.0-b16, mixed mode)
- **(M2)** X6270, 2xIntel Xeon E5540 4core 2.53GHz, 12x2GB DDR3-1066MHz ECC, Solaris 10, Java Version 1.5.0_32, Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_32-b05), Java HotSpot(TM) Server VM (build 1.5.0_32-b05, mixed mode)

- **(M3)** X6240, 2xAMD Opteron 2384 4core 2.7GHz, 8x2GB DDR2-667, Solaris 10, Java Version 1.5.0_32, Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_32-b05), Java HotSpot(TM) Server VM (build 1.5.0_32-b05, mixed mode)
- **(Test Machine)** Intel(R) Core(TM) i5-2410M CPU @ 2.30 GHz 2.30 GHz, 4GB DDR3-666.7MHz, Windows 7, Java Version 1.7.0_03, Java(TM) SE Runtime Environment (build 1.7.0_03-b05), Java HotSpot(TM) 64-Bit Server VM (build 22.1-b02, mixed mode)

10.3.2 Mandelbrot Set Visualisation

Using the SCOPE coordination workbench, we created an alternative version of the Mandelbrot set program from Section 10.2.3. The program is used to gather evidence on the question if SCOPE programs can exploit the opportunity of massive parallelism. Appendix D shows the coordination architecture of the program developed with the SCOPE DSL. An overview of the coordination model of the application is presented in Figure 10.7 using the SCOPE BPMN subset.

Target Project Structure Figure 10.10 shows an overview of the project structure of the Mandelbrot program after code generation. The Java classes relate to the SCOPE model as follows:

- The namespaces of the project directly conform to the namespaces of the Definitions of the SCOPE model.
- `AbstractMandelbrotProc`, `InitializedMandelbrotProc` and `MandelbrotProc` represent the Mandelbrot Proc process that implements the Mandelbrot Client.
- `AbstractImageProvider` and `ImageProvider` represent the Image Provider process of the SCOPE model.
- `AbstractRenderer` and `Renderer` represent the Renderer process of the SCOPE model.

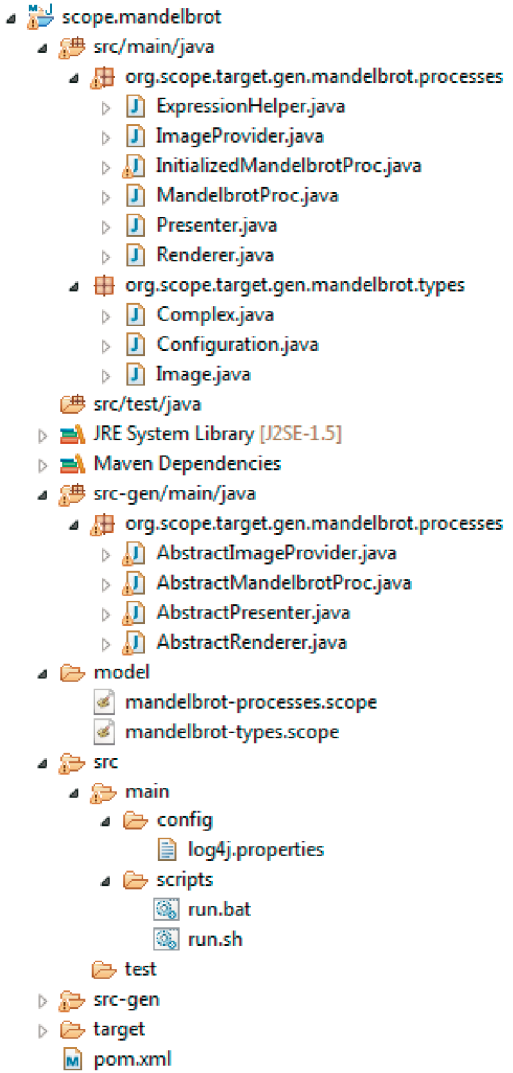

Figure 10.10. Project structure of the SCOPE Mandelbrot program.

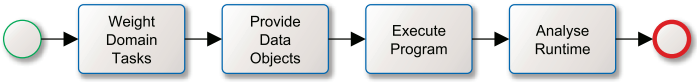

Figure 10.11. Development process for SCOPE prototype program.

- `AbstractPresenter` and `Presenter` represent the `Presenter` process of the SCOPE model.
- The `ExpressionHelper` is generated by the *scope.generator*. By default, it is used to simulate expression evaluation for prototyping (execution without supplemented domain code). The class is not used in the final program.

Prototyping After we modelled the coordination architecture of the program, we used the generated coordination code for *prototyping* – the analysis of the behaviour of the program before domain code is supplemented. Figure 10.11 illustrates the conducted activities to produce a program prototype.

First, we weighted the `Domain Tasks` of the SCOPE model of the program. We weighted each `Domain Task` equally with a value of 1.0. Second, we initialised the data objects in the coordination code that are published to the `Image Space` with mock-up data. Else, client processes would possibly wait indefinitely for data objects when the non-blocking space operations `take` and `read` are used. As a rule of thumb, the mock up data should resemble the data that is expected in practical usage scenarios as closely as possible. Third, we executed the prototype and analysed its runtime behaviour on the Test Machine. To do so, we used the log files of the PROCOL library and the NetBeans 7.1 Integrated Development Environment (IDE).[4] The latter was chosen for its extensive support for out-of-the-box profiling including CPU bottleneck identification, memory usage tracking, and thread status monitoring.[5] More sophisticated analysis could be achieved with the application performance monitoring and dynamic software analysis

[4] http://netbeans.org/
[5] http://netbeans.org/features/java/profiler.html

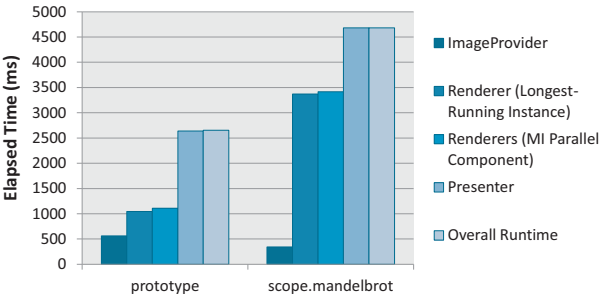

Figure 10.12. Sample run of the SCOPE Mandelbrot prototype using 16 concurrent `Renderer` instances on Test Machine; *Renderers (MI)* denotes the anonymous `MultipleInstanceParallelComponent` that instantiated the `Renderers`.

tool suite Kieker [van Hoorn et al., 2009].[6] The performance analysis of prototype sample runs revealed that the `Presenter` is most likely the performance bottleneck in the implemented program as it acts completely sequentially and must assemble the individual image slices rendered by the `Renderer` instances into a single image file. Figure 10.12 illustrates the findings by the example of a sample run using 16 concurrent Renderer instances.

Results After analysis, we supplemented the Mandelbrot program with domain code similar to the implementation of the PROCOL version. Test samples on the Test Machine using the NetBeans profiler were used to confirm that all concurrent `Renderers` are actually busy with rendering, see Figure 10.13. They also indicate that the performance bottlenecks imposed by the `Image Provider` and the `Presenter` are not as severe as the prototype suggested, see Figure 10.12.

Figure 10.14 and Figure 10.15 show the results of experiments conducted on machines M0 to M3. The figures show the average mean of ten measurements after two warm-up runs were excluded. The absence of confidence intervals in the figures show that there were hardly any variations in the

[6] http://se.informatik.uni-kiel.de/kieker/

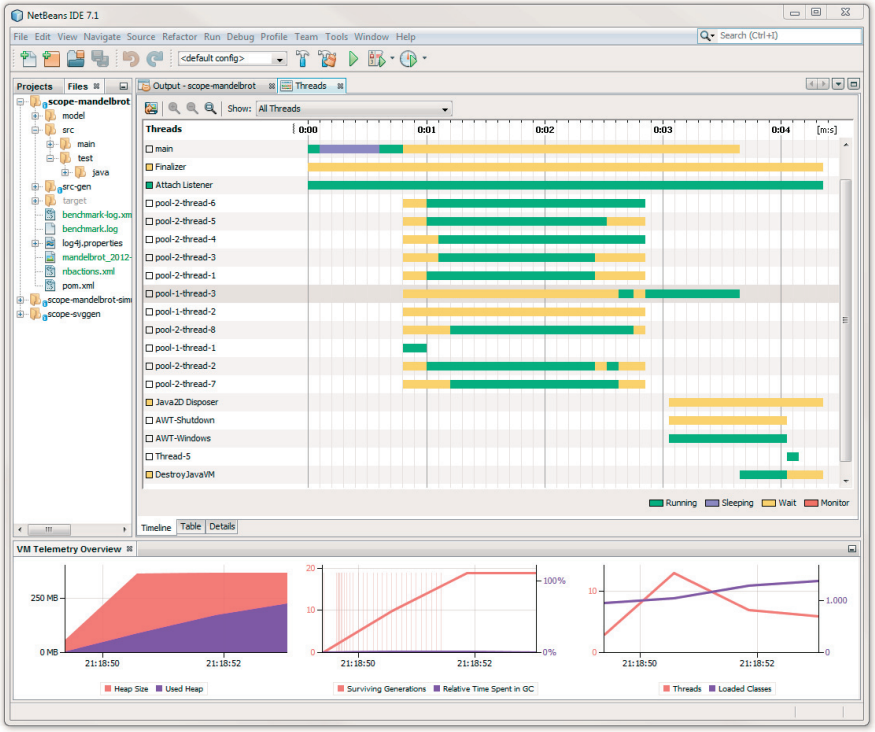

Figure 10.13. Test samples on the Test Machine using the NetBeans profiler.

individual measurements.

As Figure 10.14 shows, the program scales best on M0, until 32 concurrent Renderer instances. The response time decreased from 39,497.8 ms to 13,947.1 ms ($R_{resptime_{ScopeMandel}}(S_{RWLTS}, O_{ScopeMandel}, 1, 32) = 2.83$). However, the overall runtime behaviour of M0 is significantly worse than that of the other machines, emphasising its eligibility as a server before its usefulness for high-performance computing.

On M1, the program scales very well until 16 concurrent Renderer instances are used (2,179.1 ms), and then again starts to increase slightly, see

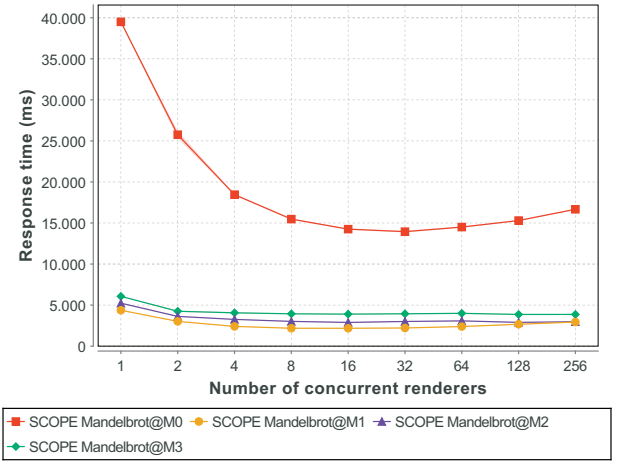

Figure 10.14. Results of the SCOPE Mandelbrot program using the RWLTS on machines M0 to M3.
The command line parameters were: image width (*-w*): `1200` pixels, image height (*-ht*): `1024` pixels, number of image partitions and Renderer instances (*-r*), top left complex number on plane (*-tl*): $0.402589755868683 + 0.1921543496459i$, bottom right complex number on plane (*-br*): $0.402679825967559 + 0.192064279547024i$, reporting (*-rp*): `true`.

Figure 10.15. However, its overall behaviour is similar to the behaviour of the program on M2 and M3: The most significant performance increase can be observed between 1 and 2 Renderer instances. The response time decreased from 4,371.6 ms to 3,017.9 ms ($R_{resptime_{ScopeMandel}}(S_{RWLTS}, O_{ScopeMandel}, 1, 2) = 1.45$). After that, the runtime does not change as dramatically between each concurrency step in the scale.

Given the slightly different experimental configuration of the machines in contrast to the experiments described in Section 10.2, we could not observe a performance penalty on M1 and M2 between 8 and 16 concurrent Renderers that may stem from incipient effects of hyperthreading.[7]

[7] Up to using 8 Renderers all Renderers can be mapped directly to the cores of the two Xeon processors of M1 and M2.

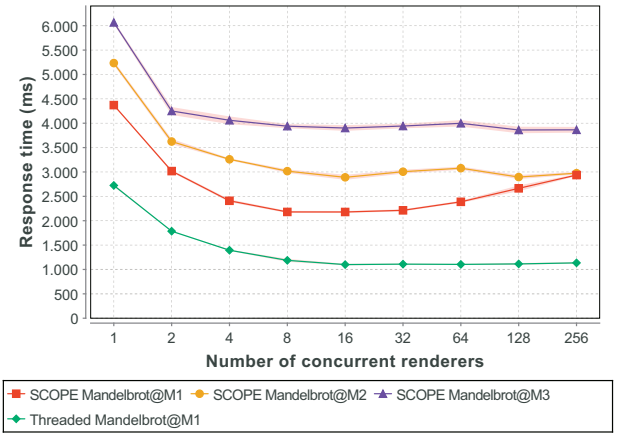

Figure 10.15. Results of the SCOPE Mandelbrot program using the RWLTS on machines M1 to M3 compared to the results of the conventionally threaded program version from Section 10.2.3.
The command line parameters were: image width (*-w*): `1200` pixels, image height (*-ht*): `1024` pixels, number of image partitions and Renderer instances (*-r*), top left complex number on plane (*-tl*): $0.402589755868683 + 0.19215434964 59i$, bottom right complex number on plane (*-br*): $0.402679825967559 + 0.192064279547024i$, reporting (*-rp*): `true`.

When compared to the conventionally threaded version of the Mandelbrot application (see Section 10.2.3), we see that the SCOPE implementation of the program does not scale as well as the threaded version on M1, but that the performance gain between 1 and 16 concurrent Renderers is high. For the SCOPE program, the response time decreased from 4,371.6 ms to 2,179.1 ms ($R_{resptime_{ScopeMandel}}(S_{RWLTS}, O_{ScopeMandel}, 1, 16) = 2.01$), and for the threaded version, response time decreased from 2,723.9 ms to 1,100.1 ms ($R_{resptime_{ThreadMandel}}(S_{RWLTS}, O_{ThreadMandel}, 1, 16) = 2.48$). Overall, the SCOPE implementation does not exploit the opportunity for massive parallelism as well as the conventionally threaded version, but is not even more than 2 times worse than the threaded version using 16 Renderers (the exact number of logical Processing Elements (PEs) of the machine).

10.3.3 Label Positioning

In EAV [Kruse et al., 2009], the Application Interface Overview (AIO) map is an important diagram type to visualise the data transfer interfaces between the individual applications of an organisational unit. In its basic form, applications are represented as boxes that are connected by edges that represent the transfer paths, see Figure 10.16. To render the data transfers, their paths are routed around the applications considered as obstacles, and around or across each other. As these paths have meaningful names, their names are also rendered as labels. The degree to which labels obscure features of a map significantly affects its clarity. Therefore, it must be clear which label belongs to which path, and labels must not overlap with each other, see Figure 10.17. The problem can be identified as an instance of PFLP, described by Christensen et al. [1995] as "the problem of placing text labels adjacent to point features on a map or diagram so as to maximize legibility". It represents a form of global (combinatorial) optimisation problem and is NP-hard, leading to solutions that are either incomplete or show exponential time complexity [Christensen et al., 1995].

Executive Summary The subject of interest is an existing application for EAV, which produces several map types (including AIO maps) for different file formats such as Scalable Vector Graphics (SVG) and Microsoft Visio diagram files. The main use of the application is that of a demonstrator for EAV map types for arbitrary enterprise architecture models provided as a Microsoft Excel file or in a proprietary Extensible Markup Language (XML) format. The application currently employs a sequential algorithm for label positioning that shows a low performance even with small domain models, see Table 10.3. The used example domain model is an anonymised real-world model in Excel format retrieved from industry. It contains 29 obstacles and 39 transfer paths.

Problem Statement To illustrate Coordination Engineering with SCOPE, the label positioning functionality of the program should be replaced with a concurrent component that either yields a better runtime given the same result quality, or better results (i. e., less label collisions) given the same

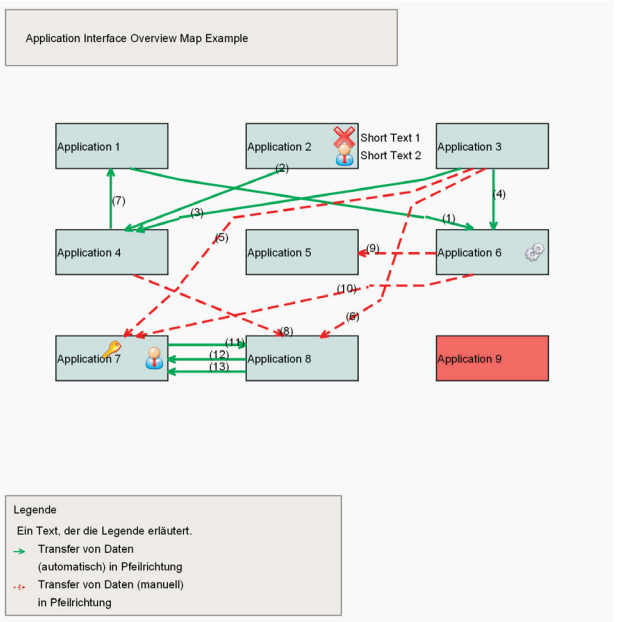

Figure 10.16. Generic example of an Application Interface Overview map.

Table 10.3. Average response time and number of overlapping labels of the EAV application on Test Machine.
The configuration was: Number of obstacles: 29, Number of transfer paths: 39

No label positioning	*Label positioning*	*Overlapping*
569.4 ms	16,504.5 ms	3.0 per file

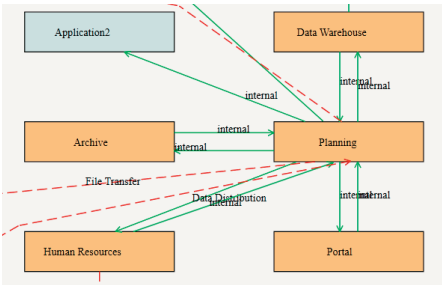

Figure 10.17. Example of overlapping labels in an Application Interface Overview map.

runtime.

Algorithm and Data Structures The application distinguishes the conceptual domain model from its visualisation. The most important data objects for the positioning of labels to data transfer interfaces are `Path`, the representation of a path that has a start `Point` and an end `Point` and may have bendpoints, and `PathWalker`, a movable content container that can travel along a given `Path` and can hold an `Object` (i. e., the text label). These data objects are, amongst others, mapped to SVG model representation elements to draw the AIO map. Labels are positioned after the transfer paths were routed from their start point to their end point applying a shortest path routing algorithm.

There are several alternative algorithms to solve PFLP. An overview is provided by Christensen et al. [1995]. The current implementation uses a sequential local search algorithm that can be trapped in local minima. Simulated annealing, developed by Kirkpatrick et al. [1983], is a heuristic technique for solving combinatorial problems that avoids local minima. The name of the technique stems from the analogy of metallurgic annealing, where controlled cooling is carried out to prevent defects in the structure of a heated solid, producing a more durable result [Heaton, 2007]. Simulated annealing emulates this process by the notion of a temperature in the optimisation process: As the temperature is high, label relocations may be

accepted at random, even if they are not better than the original position. As the temperature decreases, the likelihood for such acceptance decreases also, down to the point where only better positions are accepted. Following Christensen et al. [1995], the algorithm can be described by the following outline:

1. For each `Path`, place the corresponding `PathWalker` in any of the `Points` on the `Path`.
2. Repeat until a given threshold is reached; this can be a maximum number of cycles or when the rate of improvement falls under a given threshold.
   (a) Decrease the `temperature` T, according to an annealing schedule
   (b) Pick a `PathWalker` and move it to another position
   (c) Compute ΔE, the change in the objective function which calculates the quality of the repositioning
   (d) If the repositioning is worse that the previous position, undo the repositioning with a temperature-dependent probability

We consider parallel simulated annealing as the basis for label positioning. An overview on parallel simulated annealing is provided by Onbaşoğlu and Özdamar [2001], examples are presented by Jelenković and Poljak [1998] and Kliewer and Tschöke [2000]. Onbaşoğlu and Özdamar [2001] observed that an approach that occasionally applies partial information exchange among processors yields particularly notable results.

Platform Description The target programming platform is the Java Virtual Machine (JVM). In addition to the machines M0 to M3, we also introduced a Test Machine that represents the environment in which the program is most likely used.

Performance and Cost Requirements The current implementation takes an overall run-time of 16,504.5 ms for the example model, compared to only 569.4 ms when label positioning is disabled. For the SCOPE implementation of a new label positioning component, we consider a response time below 5,000 ms as desirable.

Category of Parallelism The category of parallelism can be described as activity parallelism, focussing on the decomposition of the data *and* the algorithm into tasks that are not committed to certain chunks of data and can therefore operate in any order.

Category of Processing From the description of the algorithm, it is clear that the overall solution is obtained by applying the same computation on every `PathWalker`. The processing is therefore a homogeneous one.

Coordination Model Synthesis According by categorisation of parallelism and processing and the pattern recommendations provided by Ortega-Arjona [2010], the Manager-Workers pattern is best suited for parallel simulated annealing. Appendix E shows the synthesised coordination model for PFLP with parallel simulated annealing. The model can be described as follows:

The `LabelPositioning` collaboration contains the space `LabelSpace` and `ObstacleSpace`, and the client `AIOMapBuilder`. `LabelSpace` contains the `PathWalkerWrappers` that contain the `PathWalkers` to be repositioned, and `ObstacleSpace` contains the obstacles with which overlapping must not occur. These are, in fact, also the `PathWalkerWrappers`, but can be extended with other obstacles in future extensions. Holding the walkers (i. e., the `PathWalkerWrappers`) in the `LabelSpace` and the `ObstacleSpace` distributes the required space operations, but requires occasional data exchanges to deal with updated positions.

We introduced the data objects `PathWalker` and `PathWalkerWrapper`. The former is a stub for the existing `PathWalker`, which will not be overwritten by the code generation component of the SCOPE workbench. The latter is a wrapper for the `PathWalker` that has the properties `trials` and `id`. `trials` the number of how many times a `PathWalker` was processed by a worker, and `id` provides a unique identifier for `PathWalkers`. We also introduce a `Configuration` object. It consists of the properties `maxCycles`, the maximum number of cycles to be performed by a worker, `maxTrials`, the maximum number of times a `PathWalker` should be processed, `spacing`, the minimum distance between two `PathWalker` not considered as a collision, `temperature`,

the current temperature of the overall problem, `temperatureDecreaseFactor`, the factor by which the temperature is decreased, and `partitionSize`, the number of `PathWalkers` processed in each processing cycle.

The behaviour of the `AIOMapBuilder` is implemented by the process `ConcurrentAIOMapBuilder`. It simply publishes all walkers to the `LabelSpace` and the `ObstacleSpace`, calls the process `PositionSolver`, and finally collects all solved walkers from the `LabelSpace`. The `PositionSolver`, in turn, publishes the `Configuration` to the `LabelSpace`, and invokes multiple instances of the `SolverWorker`. `SolverWorker` contains the main logic of the coordination:

1. Each worker reads the `Configuration` and initialises itself.
2. Each worker repeats the following steps as long as `notMaxCyclesReached` is true.[8]
   (a) Read all obstacles from the `ObstacleSpace`
   (b) Do the following for a number of `PathWalkerWrappers` defined by `partitionSize`
      i. Take an available walker from the `LabelSpace`
      ii. If the walker should be processed (at least if it exists), solve it by simulated annealing, and update the corresponding copy in the `ObstacleSpace` if its position was changed
      iii. Put the walker back to the `LabelSpace`
   (c) Increase the cycle count and reset the obstacles for the next iteration

Obviously, `SolverWorkers` will likely resolve the position of `PathWalkerWrappers` with respect to a subset of obstacles, since other `SolverWorkers` may just update a `PathWalkerWrappers` as an obstacle, leading to missed overlapping. The drawback is taken, but quality control must assure the quality of the overall label positioning.

Note that after coordination code generation, the Java classes for the process `ConcurrentAIOMapBuilder`, including the abstract classes, were modified to integrate the generated code into the existing application. Otherwise, it

[8]The actual implementation of condition evaluations such as `notMaxCyclesReached` is provided by the domain engineer.

would be necessary to provide a modified template set for the *scope.generator* to re-establish the Generation Gap pattern.

Domain Engineering The configuration is represented by the Configuration provided by the `PositionSolver`. Table 10.4 presents the configuration parameters. The parameter `maxTrials` was not used and is seen as a possible refinement to control the maximum number of times a `PathWalker` should be processed. `PathWalkers` are initially placed at a third of the length of their associated path.

The objective function contains an integer value `candidateCollisions` that is increased once if the distance of a `PathWalker` is smaller that `spacing`, and once again if the distance is too near ($dist < spacing/3$).

The annealing schedule is realised as the method `anneal(int delta)`, borrowing its implementation from Heaton [2007]. The method takes a value `delta` as a parameter, defined as the difference between the number of collisions of the repositioning and the number of collisions of the former best result. Essentially, the method checks if the `temperature` has fallen below a certain threshold or not. If the `temperature` has fallen below a threshold of $1.0E-4$, the old position is retained if the `delta` is greater than zero, else accepted. If the `temperature` is above the threshold, the `delta` and the `temperature` are used to determine if the position should be retained or not. See [Heaton, 2007] for further details on the implementation.

To guarantee the copy behaviour of the SBS component of the PROCOL library, the relevant data objects that already existed in the application were modified to implement the interface `java.io.Serializable`.

Analysis Figure 10.18 shows the results of the SCOPE-based label positioning component. The figure shows only the response time of the label positioning component, as generated by the coordination code generator of the SCOPE workbench.

The implementation scales reasonably well on machine M0 until 32 concurrent workers, ranging from 16,974.1 ms to 13,719.8 ms ($R_{resptime_{ScopeEav}}(S_{RWLTS}, O_{ScopeEav}, 1, 32) = 0.81$). However, the overall runtime of the implementation is much worse on M0 than on the other machines, providing

Table 10.4. Configuration of the SCOPE EAV application.

Parameter	*Value*
`partitionSize`	$\begin{cases} \lfloor \frac{n}{m} + 0.5 \rfloor & \text{if } n > m \\ 1 & \text{if } n \leqslant m \end{cases}$ where n denotes the number of walkers and m denotes the number of workers
`maxCycles`	10
`maxTrials`	10
`spacing`	25
`temperature`	10.0
`temperatureDecreaseFactor`	0.97

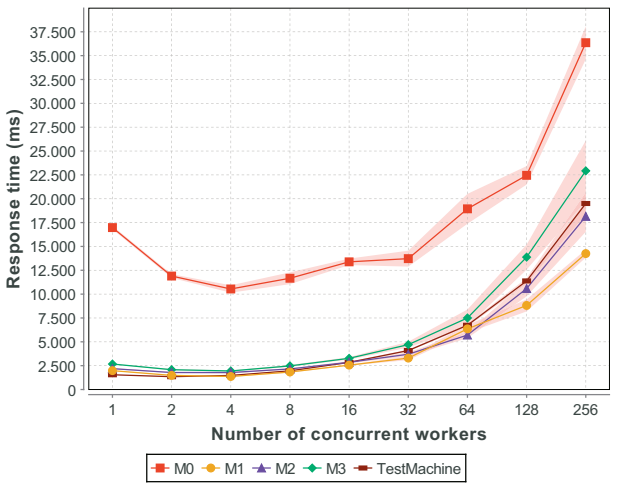

Figure 10.18. Response time results of the label positioning component.

further evidence that M0 is best used as a server for multiple requests instead of heavy computation. On M0, the required response time below 5,000 ms could not be met. On machines M1, M2, M3 and the Test Machine the response times do not differ as much from each other as they do to M0.

Table 10.5 illustrates the best result for each measurement series. For machines M0 to M3, the best results were achieved using 4 concurrent worker instances. For the Test Machine, a conventional end user system, the best results were achieved using only 2 concurrent renderers. All measurement series show an average mean of overlapping labels per file that is considerably lower than the original version.

Table 10.5. Best results of the SCOPE EAV application.

	M0	M1	M2	M3	Test Machine
Best result	10,539.5 ms	1,365.6 ms	1,778.4 ms	1,944.0 ms	1,344.6 ms
Number of workers (n)	4	4	4	4	2
$R_{resptime_{ScopeEav}}(S_{RWLTS}, O_{ScopeEav}, 1, n)$	0.62	0.69	0.81	0.73	0.87
Average mean of overlapping labels per file @ n	0.4	0.5	0.3	0.8	0.3

Increasing the number of workers further above the best measurement point yields a considerable performance decrease (see Figure 10.18). The response time shows an exponentially growth. For example, on the Test Machine, the application is rendered unacceptable when 64 or more concurrent workers are used (32 workers: 4,085.5 ms, 64 workers: 6,737.7 ms). A possible explanation for the exponential growth rate is that the number of cycles of the main loop is relative small. When too much workers are available, each can take only a small number of walkers to be solved. This,

in turn, will result in more frequent read accesses to the `ObstacleSpace` as a whole – a considerable expensive space operation (PROCOL's atomic `rdg`) that is necessary to avoid the multiple read problem [Rowstron, 1996]. Figure 10.19 illustrates the unfavourable ratio between computation and waiting when more than the ideal number of workers are used at the example of a test sample on the Test Machine using 8 concurrent worker instances (4 times as much workers as the optimum). In contrast, using the ideal number of workers results in a proper workload distribution, see Figure 10.20. Further refinements of the coordination architecture should therefore consider a parameter space exploration on the respective platform to determine the best degree of concurrency for the given coordination architecture and the current problem size.

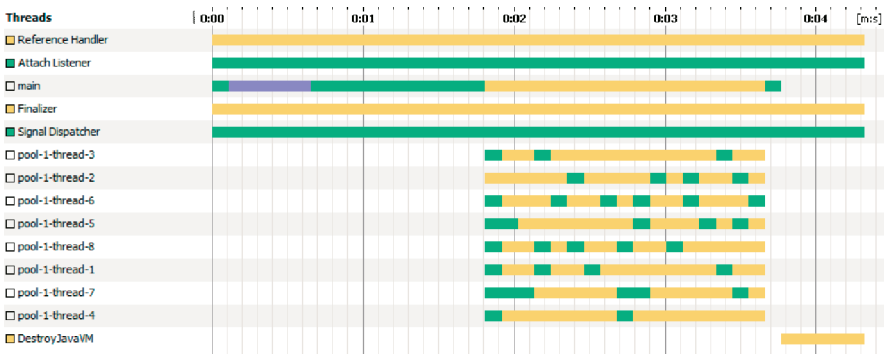

Figure 10.19. Test samples on the Test Machine using 8 concurrent worker instances.

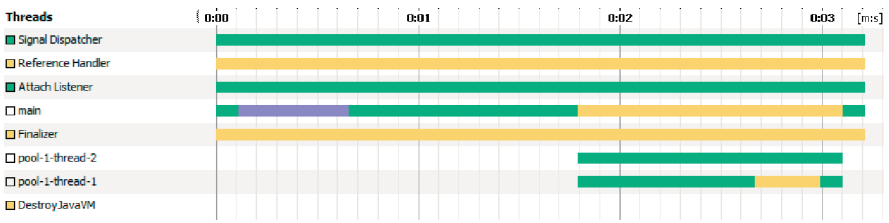

Figure 10.20. Test samples on the Test Machine using 2 concurrent worker instances.

10.4 Threats to Validity

Any empirical study like ours is vulnerable to certain threats to validity. We consider the threats discussed in this section as the most severe. The discussion focusses on construct validity and internal validity as they represent the pre-requisites for a high degree of external validity [Wright et al., 2010]. Wright et al. [2010] provide us the descriptions for the terms construct validity, internal validity, and external validity.

Construct validity denotes the degree to which the studied experimental configuration and its parameters are relevant to the research questions. Our experiments are amenable to the following construct validity threats:

1. The experiments focussed on the micro- and macro-benchmarks OutBenchmark, OutInBenchmark, OutReadBenchmark, ConsumerProducerBenchmark, and an AgeingBenchmark, and on the two application experiments on Mandelbrot set visualisation and label positioning. Further micro- and macro-benchmarks (e. g., for the `ing` and `rdg` operations) can simply be integrated by implementing the appropriate abstract benchmark classes in the project *procol-tuplespace-benchmark*, and further application experiments (e. g., for programs that apply other architectural patterns than Manager-Worker) can be integrated by providing additional SCOPE programs that conform to the project structure discussed in Appendix C.

2. The benchmarks and the SCOPE programs have only been executed on multi-core machines and the performance on many-cores is yet to prove. The threat can be addressed by providing additional data structure implementations for the extended LighTS framework and by using many-core machines, when available.

3. The ageing technique applied to the SBS component only considers single different strings as data objects. The observed ageing behaviour may therefore not adequately represent real-world ageing. Addressing ageing in more detail can be achieved by populating the SBS component with data objects of different size and type before benchmark execution [Fiedler et al., 2005].

4. Although stemming from practice, we only considered a single input model for the application experiment on label positioning. It is likely that the size of the input model as well as the actual positioning of the obstacles and the transfer paths could have a significant impact on the performance of the program and the quality of the outcomes. The issue can be addressed with further experiments with other input models. However, the example serves us mainly as a demonstration of the Application Engineering process of Coordination Engineering.

Internal validity denotes confounding factors in experiments. As Wright et al. [2010] observe, selection bias is a prevalent threat to the internal validity in software engineering research:

1. Our benchmark suite could contain defects that we have not identified. These could cause benchmarks to be executed improperly or metrics to be calculated incorrectly. We addressed this threat by using existing and established libraries for statistics calculation and plotting. Additionally, we tested the calculations itself using unit tests.

2. Third party effects such as uncontrolled user logins and program interferences on the benchmark machines could compromise the experiments. We prevented such interference by inspecting the running processes on the machines and coordinating user logins with an agreed usage schedule amongst users.

External validity denotes the applicability of experimental results to domains beyond those under study, or simply, the generalisability of the results. We perceive the following external threats:

1. The different Mandelbrot programs based on PROCOL and the SCOPE DSL) were only compared to an equivalent version that is based on the Java thread model. Comparative analyses to other programming models (e. g., light-weight publish-subscribe frameworks) are desirable to address the applicability of SCOPE beyond the domain of SBS as a general technology for concurrency.

Chapter 11

Related Work

We separate the discussion of related work into the categories high-level concurrent programming with Space-Based Systems (SBS), model-driven approaches for concurrent software development, rigorous performance evaluation and experimentation, Domain-Specific Language (DSL) development, and future target platforms. Finally, we categorise the Space-Coordinated Processes (SCOPE) implementation and compare the SCOPE DSL with several workflow languages.

11.1 Concurrent Programming with SBS

Balzarotti et al. [2007] present LighTS, a minimalistic SBS originally developed as the core tuple space layer for the LIME middleware.[1] The framework can be extended in various ways, for example, by exchanging the back-end implementation. Also, it finds a reasonable compromise between object encapsulation stemming from object-orientation and the requirement to publish the internals of a data object stemming from generative communication, see Section 3.3.4. Although the framework emphasises its use for context-aware applications, we use LighTS as a basis for the SBS component of our Process Coordination Library (PROCOL) library (see Section 9.1.2).

There are several other Java implementations of SBSs that are mostly oriented towards distributed computing. Examples are Blitz JavaSpaces, SemiSpace, and the LighTS framework.[2] Wells et al. [2004] present a survey of selected SBSs in Java. Their discussion focusses on the commer-

[1] http://lime.sourceforge.net/

[2] http://www.dancres.org/blitz/, http://www.semispace.org/semispace/

cial offerings of the implementations and on the capability to provide alternative mechanism for matching. They state that their performance measurements for a simple ray-tracing application indicate that none of the implementations considered were particularly suitable for fine-grained parallel processing. However, they also state that the employed hardware was barely adequate to benchmark Java applications. It is also unclear how representative the results are, as the number of individual measurements is not stated.

11.2 Model-Driven Concurrent Software Development

There are few comparable approaches regarding the use of Model-Driven Software Development (MDSD) for concurrent software engineering. Tan et al. [2003] present an infrastructure to use design patterns to generate parallel code for distributed and shared memory environments. Programmers are required to select appropriate parallel design patterns, and to adapt the selected patterns for the specific application by selecting appropriate code-configuration options. Our approach does not necessarily consider the modification of generated code for performance fine tuning.

Hsiung et al. [2009] present an approach of model-driven development of multi-core embedded software. The approach is based on SysML models as an input, and generates multi-core embedded software code in C++. The code architecture consists of an OS, the Intel Threading Building Blocks (TBB) library [Reinders, 2007], a framework for executing concurrent state machines, and the application code. The described approach is restricted to multi-core embedded software and abstracts from a specific target platform. Our Coordination Engineering-approach is instead applicable to different target platforms.

Pllana et al. [2009] propose an intelligent programming environment that targets multi-core systems. This environment is envisioned to combine model-driven development with software agents and high-level parallel building blocks to automate time-consuming tasks such as performance tuning. Unified Modeling Language (UML) extensions are proposed for

graphical program composition. Our approach and that of Pllana et al. share the focus on multi-core systems. While the latter exclusively considers those, we consider ours to be applicable to distributed system development by providing additional generators that include appropriate transformation sets.

In his Intel white paper, Lingam [2009] briefly presents the Microsoft Integration Services platform as a means for building parallel applications by the example of a video conversion and File Transfer Protocol (FTP) program example, emphasising that "Model Driven Development is a fantastic way to take advantage of multi-core computing infrastructure". While the article falls short on presenting details on the realisation of the example, we regard it as an evidence that model-driven software development is gaining momentum in the parallel programming domain.

11.3 Rigorous Performance Analysis

Our benchmark suite (Section 9.1.3) was inspired by the chapter on testing in the book "Java Concurrency in Practice" from Goetz [2009] and by the work of Fiedler et al. [2005]. The latter present an approach to measure the throughput and the response time of a tuple space when it handles concurrent local space interactions [Fiedler et al., 2005]. They also present an ageing technique to populate a tuple space before the execution of a benchmark. The approach considers the phases *Aging*, *Startup*, *Write* (equal to `out`), *Pause*, *Take*, *Shutdown*, *Aging Cleanup*. It is used to measure the performance of a JavaSpace implementation. The presented benchmarks were focused on blocking operations to characterise the worst-case performance of the JavaSpace `take` operation (equal to `in`).

Our tool for performance analysis (Section 9.1.4) and the data analysis itself (Chapter 10) are motivated by the work of Georges et al. [2007] on statistical rigorous Java performance evaluation. We followed their advice on distinguishing start-up from steady state execution, and on using the confidence intervals to discuss independent measurement series. We also followed their advice on using different techniques to compute the confidence interval based on whether the number of individual measurements

for a series is large or not.

11.4 DSL Development

Mernik et al. [2005] describe patterns for the analysis, design, and implementation of a DSL. We already justified the development of the SCOPE coordination model in terms of DSL decision patterns – *AVOPT*, *product lining*, and *interaction specification* – see Section 6.7. The domain analysis can be described as *formal* and *extract from model*: While Coordination Engineering is modelled using an explicit megamodel (see Chapter 5 and Favre and NGuyen [2005]), The SCOPE coordination model is extracted from the Business Process Model and Notation (BPMN) 2.0 specification. Because we used parts of the BPMN 2.0 specification and extended them by elements for SBS, its design can be categorised as *language piggybacking* (a special case of *language exploitation*). The implementation of our Process Coordination Library (PROCOL) applies the *embedding* pattern, and the implementation of the SCOPE coordination workbench applies the *compiler/generator* pattern.

In their work on building blocks for performance-oriented DSLs, Rompf et al. [2011] state that the higher-level of abstraction and willingness to sacrifice generality make it feasible for a DSLs compiler and runtime system to generate high-performance code that can even target parallel heterogeneous architectures that consists of a number of Central Processing Units (CPUs) and Graphics Processing Units (GPUs): "Reasoning across domain constructs [...] enables more powerful and aggressive optimizations that are infeasible otherwise". To alleviate the challenge of building appropriate DSLs, they developed the Delite Compiler Framework, a compiler and runtime infrastructure for heterogeneous target code generation and domain-specific optimisations that can be understood as a toolbox to create internal parallel DSLs.

At its core, the framework relies upon *language virtualisation* [Chafi et al., 2010], a hybrid approach that strives to obtain the flexibility and performance of external DSLs with internal DSLs, argumenting that the development of external DSLs requires extreme effort and thus does not scale [Lee et al., 2011]. As a case study, Rompf et al. [2011] present OptiML,

a DSL for machine learning, and demonstrate its performance on a system with multi-core CPUs and GPUs.

In contrast to the language virtualisation approach of the Delite Compiler Framework, the *language piggybacking* approach applied in this thesis produced SCOPE implementations as both an internal and an external DSL (PROCOL and SCOPE), including opportunities for sophisticated domain-specific validation and a clear separation of roles and concerns. It is this organisational separation of concerns that let us establish software product lines as well as factories in Coordination Engineering, while language virtualisation favours an amalgamation of roles and concerns to the level of the programmer.

The practicability of language piggybacking for external DSLs is guaranteed by the recent development and availability of language workbenches – technologies that greatly alleviate the development of external DSLs. As illustrates by the development of the SCOPE DSL at the example of the Xtext framework [Xte, 2011], such language workbenches can support scalability by a proper separation of concerns according to the roles involved.

There are many examples for the extension of the BPMN for domain-specific aspects. For instance, Awad et al. [2009] describe how BPMN 1.1 process diagrams can be extended by constraints for resource assignment. The authors extend a BPMN metamodel for additional classes such as roles that are assigned to task instances via resources, while constraints on the assignment of resources to tasks are modelled using the Object Constraint Language (OCL).

Rodríguez et al. [2007] developed their own metamodel based on BPMN 1.0 process diagrams, and extended it by security requirements aspects. These examples have in common that they rely on an own metamodel interpretation for their pre-BPMN-2.0 dialect. In contrast to our piggybacking approach, it remains to be proven that these approaches conform to the current BPMN 2.0 metamodel.

Schleicher et al. [2010] describe an extension of the BPMN 2.0 metamodel. They use the extension mechanism of the BPMN 2.0 specification to add so-called *compliance scopes* to process diagrams. In contrast to the language piggybacking approach, the approach is limited in interoperability. The BPMN 2.0 extension mechanism is not widely and uniformly supported

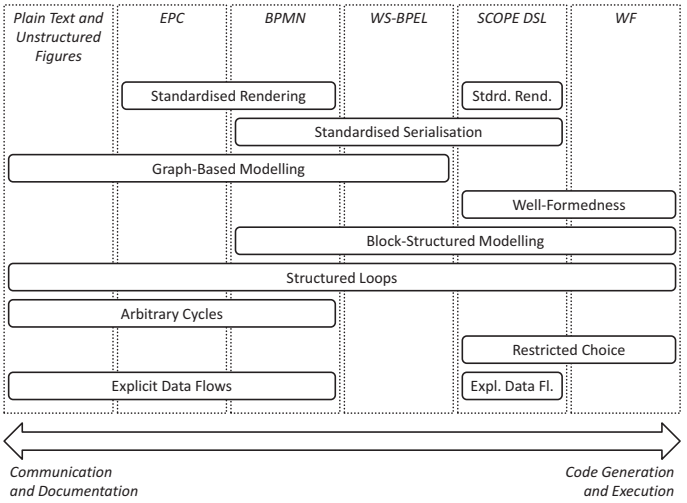

Figure 11.1. Comparison of the SCOPE DSL with the workflow languages BPMN, EPC, WS-BPEL, and WF.

across tool chains, and means for higher-level validation of BPMN models extended by domain-specific elements are often missing.

11.5 Comparison with Workflow Languages

Based on the work of Kopp et al. [2008], we characterise the SCOPE DSL in terms of workflow language features to compare it with four common workflow languages – BPMN, Event Driven Process Chains (EPCs), Web Services Business Process Execution Language (WS-BPEL), and Windows Workflow Foundation (WF). Figure 11.1 summarises the comparison. The individual characteristics of the SCOPE DSL are discussed in the following. We refer to Kopp et al. [2008] for the characterisation of the workflow languages BPMN, EPCs, WS-BPEL, and WF.

Intention The *intention* of a coordination language expresses whether the language has been designed primarily for human-to-human or human-to-machine communication. While human-to-human communication is mostly used for *documentation*, the use of human-to-machine communication can be differentiated in executable *code generation* and direct *automatic execution*. Also human-to-machine communication requires a language to provide clearly defined and unambiguous semantics for code generation or execution. The SCOPE DSL is a human-to-machine language that it is aimed on code generation.

Representation The *representation* refers to whether the language specification defines a *standardised rendering* (graphical representation) or a *standardised serialization* (textual representation), or both. Graphical representations are multi-dimensional, thus able to present multiple concurrent control flows naturally, while fine-grained concurrency control can be encapsulated in appropriate language feature semantics [Browne et al., 1994; Kleppe, 2008]. On the other side, they are often unambiguous and take much space. Developing concurrent programs using complex textual representations (such as provided by general programming languages) can be tedious and error-prone due to the linearity of the textual representations. On the other side reduced textual representations are more compact and formal than graphical representations. Since SCOPE DSL is oriented towards collaborative work a textual representation is preferred over a graphical one. However, mapping the SCOPE language to the graphical BPMN is an intrinsic part of the language definition.

Control Flows When regarding the *control flows* dimension, there are several sub-dimensions. These sub-dimensions concern the control style, well-formedness, join condition types, and loop structurisation. The *control style* denotes how control flows are expressed by a language. *Graph-oriented* control flow modelling is more applicable to graphical representations and usually allows to model arbitrary cycles without the notion of dedicated loop constructs. *Block-structured* control flow modelling, on the other hand, is more applicable to textual representations and restricts cycles to structured loops represented by dedicated loop constructs with well-defined

semantics. Being realised as a textual DSL, we selected block-structured control flow modelling as the primary control style of the SCOPE language.

Well-formedness refers to whether the language restrict consecutive split and join operations (also often denoted as choices or decisions) to the same type or not. For example, a well-formed language requires a control flow that is split using an XOR-split-choice to be joined through an XOR-join-choice and not otherwise. As a block-oriented language we require the SCOPE language to be well-formed.

The selection of *join condition types* is tightly related to the well-formedness of the language. There are two types of join conditions. Coordination languages with *restricted choice* only allow control flows to be joined by a restricted set of operators, for example AND, OR, and XOR. The negation operator is normally not included. On the other side, languages that allow *arbitrary expressions* allow to define arbitrary boolean expressions over either the status of incoming links, or over the status of processes by the notion of process variables. Well-formed languages can not support arbitrary boolean join conditions and so the SCOPE DSL cannot either.

Regarding *loop structurisation* there are typically two types of loops, block-structured loops and graph-based loops. While *block-structured loops* are characterised by explicit loop constructs with exit conditions (for example, *repeat until* and *while*-clauses), graph-based loops can be modelled simply by defining arbitrary control flow cycles between processes. The SCOPE DSL is restricted to block-structured loops to avoid semantic ambiguities that can arise with arbitrary graph-based cycles.

Data Flows *Data flows* define the availability of passive data structures to active processes. The most prominent and archetypical forms of data flow modelling are message passing and data sharing. While *message passing* explicitly defines to send and receive data structures over distributed memory or address spaces, *data sharing* regards data flows as accessing data structures that reside in memory that is shared among processes. For data flow modelling in the SCOPE language, we consider a data-sharing approach that is based on SBS.

Data Type System When regarding the types of passive data structures that shall be supported by a coordination language, there are two main types. A *well-typed* language provides a *defined type set* that is especially suited for executable languages (for example basic types such as *boolean*, *string*, and *integer*). However, for code generation, there must be a dedicated mapping of the type system to the type system of the target programming language. An *arbitrary type set*, on the other hand, does require an ad-hoc mapping of the types used in the respective coordination models of the language, or the adoption of the type system of the target programming language. For the SCOPE language, we consider a simple defined type set.

Part IV

Conclusion and Outlook

Chapter 12

Conclusion

The present thesis is motivated by three prominent challenges to software engineering that are imposed by the ongoing transition from sequential to parallel machines. Portability and human perception require the provision of standardised higher-level concurrency models, continuity with prevalent technologies must be guaranteed, and existing patterns and practice of parallel software design must be disseminated to software engineers of all kinds and domains.

The goal of the thesis is to address these challenges with a novel combination of coordination models and languages with techniques from Model-Driven Software Development (MDSD). We use the Business Process Model and Notation (BPMN) 2.0 specification as a vehicle for dissemination, model transformations to enforce compliance, a combination of BPMN and Space-Based Systems (SBS) as a coordination model, and a concrete method that is embedded in the Parallel Software Design method as application guidance.

The essential contribution of the thesis is as follows:

Space-Coordinated Processes (SCOPE) is a conceptual coordination model that introduces SBS-based choreography of independent processes, which internally orchestrate fine-grained workflow activities. The model combines BPMN and SBS.

Coordination Engineering is a method that regards the coordination model of a concurrent software system as the first development artefact in the software development process, and facilitates the separation of concerns according to the roles application engineer, domain engineer, and toolsmith. The method targets concurrent software development as a Model-Driven System (MDS) that emphasises the establishment of

concurrent software product lines and families.

The evaluation of our contribution comprises the implementation of two prototypes for SCOPE and several experiments on performance and applicability. The description of the implementation illustrates the Tool Engineering process of the Coordination Engineering method. The implemented prototypes can be regarded as two kinds of a Domain-Specific Language (DSL):

Process Coordination Library (PROCOL) is a library-based embodiment of the SCOPE coordination model as a proof of feasibility. It eases the development of concurrent SBS-based programs for the Java platform and can be considered as an *internal* DSL for SCOPE.

SCOPE DSL and Workbench is an embodiment of the SCOPE coordination model in an *external* textual DSL that conforms to the BPMN, and a workbench for the SCOPE DSL that is based on the Xtext language framework. M2M and Model-to-Text (M2T) transformations guarantee the interoperability with arbitrary BPMN-conformant third party tools and the compliance of PROCOL coordination code to the corresponding SCOPE coordination architecture models.

We conducted several experiments to justify the SCOPE coordination model and to show the application of Coordination Engineering with SCOPE. Statistical rigour is guaranteed by considering several measurement series from which measurements were excluded that were likely amenable for effects of JIT compilation and GC. For the rest, the average mean and the 95% confidence interval were considered.

Micro- and macro-benchmarks investigate the question if we can provide a scalable SBS implementation for the Java programming platform as the core of our PROCOL library. The findings show that in each of the benchmarks an implementation for which our definition of scalability holds or is only slightly violated was provided.

The conclusion drawn from the results is that scalable SBS implementations can be provided given that the effects of ageing are not yet

considered. This is at least the case for multi-core machines with a number of cores that is in the single-digit range.

Application-specific benchmarks on Mandelbrot set visualisation in the complex plane investigate if PROCOL and SCOPE programs exploit a given opportunity for massive parallelism. They also compare the PROCOL and SCOPE programs with an equivalent program realised with conventional Java threading mechanisms.

For the PROCOL program,[1] the findings show that the program scales very well on the given machines until 8 concurrent renderer instances. They also show the performance of the PROCOL program in comparison to the conventionally threaded program. For example, when using 8 concurrent renderer instances, the response time of the PROCOL implementation takes between 1.31 and 1.33 times longer to execute, and between 1.44 and 2.05 times longer to execute when 256 concurrent renderers are used, depending on the respective machine. In all cases the difference between measurements was less than an entire order of magnitude.

For the SCOPE program,[2] the findings show that the program scales very well on the given machines until 16, and in one case until 32 concurrent renderer instances. The response time of the SCOPE program is not even more than 2 times higher than the threaded version using 16 renderers on the considered machine.

We conclude that the performance overhead that PROCOL and SCOPE impose is at least for the considered applications a reasonable trade-off for the benefits that they provide.

An application-specific benchmark on parallel simulated annealing illustrates the Application Engineering process of the Coordination Engineering method at the example of Point-Feature Label Placement (PFLP)

[1] The program was directly programmed using the PROCOL library.

[2] The program was developed using the SCOPE DSL to develop its coordination architecture, and the SCOPE coordination workbench to generate PROCOL coordination code from the architecture model. Domain code was supplemented after prototyping to implement a complete program.

in Enterprise Architecture Visualisation (EAV). The results show that Coordination Engineering was successfully employed to develop a concurrent component for PFLP in an existing EAV application.

The benchmark also motivates further optimisation of the PROCOL library and the use of parameter space exploration to determine the best degree of concurrency for the size of a given problem on the considered parallel system: For the relatively small size of the input domain model to be labelled, the best response time results were achieved using only 4, and in one case only 2, concurrent worker instances. Increasing the number of workers further resulted in exponentially growing response times since SBS access prevails over computation, rendering the component as unacceptable at 32, or in one case at 64, concurrent workers.

The applicability of Coordination Engineering with SCOPE showed its eligibility as an engineering approach. The implementation of SCOPE guarantees interoperability and conformance with the widely known BPMN, both a means for high-level coordination modelling and for dissemination. Experimentation showed the eligibility of SCOPE as a high-level approach for concurrent engineering. We therefore consider Coordination Engineering with SCOPE as a promising candidate to address the three prevailing challenges of concurrent software engineering – standardised higher-level concurrency models, continuity with prevalent technologies, and the dissemination of existing patterns and practice of parallel software design.

However, despite our positive conclusion, further extensions and improvements are necessary to bring Coordination Engineering and the SCOPE coordination model forward from a mere prototype to a professional alternative for concurrent software engineering.

Chapter 13

Outlook

The implementation of the SCOPE coordination model and its application in Coordination Engineering provides several opportunities for extension and improvement. First, the Coordination Engineering can be supplemented with the following extensions:

- The analysis activity in the Application Engineering process determines the performance and cost requirements from the problem specification and decides whether the coordination model serves its purpose or whether previous activities must be re-iterated (Section 5.3.1). It can be worthwhile to further separate concerns by regarding analysis as a sub-process that can contain the following activities:
  - In Section 10.3.2 we employed a simple prototyping process. It can be considered as a separate prototyping activity to anticipate the dynamic behaviour of Space-Coordinated Processes (SCOPE) programs.
  - Section 6.6 describes the operational behaviour of SCOPE with Petri nets. A separate static analysis activity can be introduced that checks the corresponding Petri net model of a SCOPE program, for example, for deadlocks. Tool support can possibly be realised with the Model-Checking Kit – a toolkit that allows to apply a variety of analysis techniques to a single model representation that can be provided by a variety of modelling languages.[1]
- Autotuning could explore the parameter space of SCOPE prototypes and programs in order to determine the best degree of concurrency for the given target platform.

[1] http://www.fmi.uni-stuttgart.de/szs/tools/mckit/

Second, we consider the following extensions as promising to further refine the implementation of the SCOPE coordination model:

- The Process Coordination Library (PROCOL) framework represents a prototypical implementation of an internal Domain-Specific Language (DSL) for the SCOPE coordination model. The optimisation of the Space-Based Systems (SBS) component is appealing to increase the performance of inter-process communication. In particular, a more sophisticated matching strategy could mitigate the effects of ageing (see Section 10.2.2). Separating the matching algorithm from the space data structures and replacing it with a strategy design pattern could increase the configurability of the implementation using its factory interface. For highly concurrent systems, space implementations with multiple internal data structures and a concurrent matching algorithm could be worthwhile investigating. Finally, the introduction of a *collect* primitive that dismisses atomicity in favour to competition amongst client processes could further improve the overall performance.

- To show the feasibility of our approach, PROCOL maps the SCOPE coordination model to the Java programming language and the Java Virtual Machine (JVM), used as a closed local platform. As SBS are also used for open and distributed systems, additional generator projects for such platforms provide a basis for product lining and further experimentation.

- The SCOPE coordination workbench represents a prototypical implementation of an external DSL for the SCOPE coordination model. Its components are realised as a set of Eclipse plug-ins that can be combined with third-party plug-ins to form custom workbench configurations. When considered as an essential part of Model-Driven Systems (MDSs) for Coordination Engineering, different workbench configurations can be considered as viewpoints on the MDS that alleviate the activities of the different participating roles. Such viewpoints itself require configuration versioning and documentation, and application guidance. While the former can be realised with Yoxos workspace configuration profiles for the Eclipse Integrated Development Environment (IDE),[2] the latter can

[2] http://eclipsesource.com/en/yoxos/

be supported with intelligent project wizards.

- ▻ Interesting combinations of space data structures and matching strategies could be abstracted into DSL extensions: They could be represented by parameters to configure the Spaces that attend a Collaboration, for example, to specify that a read-optimized Space should be used.
- ▻ To foster the statical analysis of SCOPE models, a SCOPE-to-Petri net generator component could transform SCOPE models to Petri nets. Reasonable output formats are Petri Net Markup Language (PNML) [Hillah et al., 2009], and one of the many input formats of the Model-Checking Kit toolkit.
- ▻ Further optimisation of the code generator templates can reduce code redundancies by considering finer-grained context-dependent templates, possibly increasing the performance of SCOPE programs. Optimisations of the Query/View/Transformation (QVT) transformation in the Business Process Model and Notation (BPMN) generator can map the different Definitions namespaces of a SCOPE model to different Definitions elements in BPMN models, using its import mechanism. The result is the generation of compound output models that consists of separate BPMN models for each SCOPE Definitions element.
- ▻ Heiser [2009] makes a case for Virtual Shared Memory (VSM) as an attractive model for future many-core systems. VSM is a shared memory abstraction that is implemented over physically distributed memory by a hypervisor. It allows operating systems to directly access all memory of a many-core system. The model provides several benefits for hardware manufacturers since it integrates well with virtualisation for resource management, and simplifies to handle message loss in the interconnect, faulty cores, and core heterogeneity [Heiser, 2009]. Heiser also presents results from a cluster-based prototype that indicate that the VSM abstraction does not impose a significant overhead when shared memory is not required. We consider VSM-based many-cores as an attractive future target architecture for SCOPE since it provides a natural abstraction for the SBS model.[3]

[3] An overview of the architecture of future many-core systems in general is presented

Finally, the experiments can be extended by the following aspects:

- ▻ In our experiments, we investigated the operational behaviour of our PROCOL library. Further experiments can perform comparative analyses to other programming models, for example, light-weight publish-subscribe frameworks.
- ▻ The experiments on Mandelbrot Set visualisation and Point-Feature Label Placement (PFLP) employed the Manager-Worker pattern for their coordination architectures. Developing and analysing applications that employ different architectural patterns could provide hints for useful extensions of the SCOPE coordination model that further increase its expressiveness.
- ▻ To investigate the effects of ageing, we pre-populated the SBS component of the PROCOL library with different-string-values data objects. Addressing ageing in more detail can be achieved by populating the SBS component with data objects of different size and type before benchmark execution.
- ▻ For the analysis of SCOPE, we employed a mechanism that measures response time and throughput. More sophisticated analysis can be achieved with continuous performance monitoring and dynamic analysis frameworks, such as Kieker [van Hoorn et al., 2009].
- ▻ Further experiments could identify interesting combinations of space data structures and matching strategies, as discussed above.

in condensed form by Borkar [2007]. It discusses the topics power management, memory bandwidth, on-die-networks, and system resiliency.

Part V

Appendix

Appendix A

XML Schema Definition for SBS Benchmarks

This chapter describes a schema for benchmarks, see Section 9.1.3. It is formulated as an XML Schema Definition (XSD). The schema is used by the project *procol-tuplespace-benchmark*, a part of the Process Coordination Library (PROCOL) library for Space-Based Systems (SBS) programming in Java. Essentially, the schema defines experiments that each consist of a number of benchmarks, whereas each benchmark comprises a set of measurements. A measurement, in turn, contains a data sample and defines its unit of measurement.

```
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified" version="1.0"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="experiments" type="experimentsType" />
  <xsd:complexType name="experimentsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="experiment"
          type="experimentType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="experimentType">
    <xsd:sequence>
      <xsd:element name="timestamp" type="xsd:dateTime" />
      <xsd:element name="type" type="xsd:string" />
      <xsd:element name="concurrency-degree" type="xsd:int" />
      <xsd:element name="ageing-population" type="xsd:int" />
      <xsd:element name="run-time" type="xsd:double" />
      <xsd:element name="pause-time" type="xsd:double" />
      <xsd:element name="benchmark-iterations" type="xsd:int" />
      <xsd:element name="workload-iterations" type="xsd:int" />
      <xsd:element name="benchmarks" type="benchmarksType" />
```

```
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="benchmarksType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="benchmark" type="benchmarkType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="benchmarkType">
    <xsd:sequence>
      <xsd:element name="benchmark-name" type="xsd:string" />
      <xsd:element name="measurements" type="measurementsType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="measurementsType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="measurement"
          type="measurementType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="measurementType">
    <xsd:sequence>
      <xsd:element name="measurement-name" type="xsd:string" />
      <xsd:element name="index" type="xsd:int" />
      <xsd:element name="sample" type="xsd:double" />
      <xsd:element name="unit" type="xsd:string" />
      <xsd:element name="concurrency-kind" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Appendix B

SCOPE Xtext Syntax Definition

This chapter describes the Xtext language definition for the SCOPE DSL, see Section 9.2. It comprises both the concrete and the abstract syntax of the language. The language definition is formulated using the Extended Backus-Naur Form (EBNF)-like Xtext language of the Xtext language workbench [Xte, 2011].

File *Scope.xtext*

```
grammar org.xtext.scope.Scope with org.eclipse.xtext.common.Terminals
generate scope "http://www.xtext.org/scope/Scope"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

// ********************************************************************
// PART I: Definitions
// ********************************************************************

Definitions :
'definitions' name=QualifiedName
'{'
( imports+=Import )*
( types+=ObjectType )*
( collaboration=Collaboration )?
( processes+=Process )*
'}' ;

QualifiedName :
ID ('.' ID)* ;

QualifiedNameWithWildCard :
QualifiedName ( '.*' )? ;

Import :
'import' importedNamespace=QualifiedNameWithWildCard ;

// ********************************************************************
// Collaboration and Participants
```

```
// ****************************************************************

Collaboration :
'collaboration' name=ID
'{'
( participants+=Participant )*
'}' ;

Participant :
Space | Client ;


Space :
'space' name=ID ;

Client :
'client' name=ID ':' processRef=[Process] ;

// ****************************************************************
// Processes
// ****************************************************************

Process :
'process' name=ID ( 'attends' definitionalCollaborationRef=[Collaboration] )?
'{'
( properties+=ProcessProperty )*
sequenceFlows=SequenceFlows
'}' ;

ProcessProperty :
type=TypeReference name=ID
( isInit ?= 'init' )? ';' ;

// ****************************************************************
// Sequence Flows
// ****************************************************************

SequenceFlows :
{SequenceFlows} ( nodes+=FlowNode )* ;

ConditionalSequenceFlows :
'case' '(' condition=Expression ')'  ':' ( nodes+=FlowNode )* ;

Expression :
name=ID probability=RealValue ;

ExclusiveGateway :
{ExclusiveGateway}
'xor'
'{'
( sequenceFlows+=ConditionalSequenceFlows)*
('default' ':' defaultSequenceFlows=SequenceFlows )?
```

```
'}' ;

// ****************************************************************
// Activities
// ****************************************************************

FlowNode :
Activity | ExclusiveGateway ;

Activity :
DomainTask | SpaceSendTask | SpaceAccessTask | SubProcess | CallActivity
;

enum SpaceSendOperation :
OUT = 'publish'  |
OUTG = 'publish-all';

enum SpaceAccessOperation :
IN  = 'take'  |
INP = 'take-available' |
ING = 'collect' |
RD  = 'read '  |
RDP = 'read-available' |
RDG = 'scan' ;

DomainTask :
( loopCharacteristics=LoopCharacteristics )?
name=ID ( 'weighted' weight=RealValue )?
';' ;

SpaceSendTask : //
( loopCharacteristics=LoopCharacteristics )?
spaceOperation=SpaceSendOperation  messageRef=[ProcessProperty] 'to' targetRef=[Space|QualifiedName]
';' ;

SpaceAccessTask :
( assignedProperty=ProcessPropertyCall '=' )?
( loopCharacteristics=LoopCharacteristics )?
spaceOperation=SpaceAccessOperation message=QuerySpecification 'from' targetRef=[Space|QualifiedName]
';' ;

QuerySpecification :
requestedTypeRef=[ObjectType]
( 'where' '(' ( actualProperty+=ObjectPropertyAssignmentExpression  )*  ')' )? ;

ObjectPropertyAssignmentExpression : // differ between fromType or fromValue/Instance
actualPropertyRef=[ObjectProperty|QualifiedName] '=' value=ValueCall (',')? ;

ValueCall :
ValueExpression | ProcessPropertyCall // | ObjectTypeInstantiationExpression ;

ObjectTypeInstantiationExpression :
```

```
'new' typeRef=[ObjectType] '(' ( parameterValues+=ValueCall )* ')' ;

ProcessPropertyCall :
propertyRef=[ProcessProperty|ID]
( {ProcessPropertyNavigatingExp.left=current} '.' right=ObjectPropertyCall )? ;

ObjectPropertyCall :
propertyRef=[ObjectProperty]//|ID
( {ObjectPropertyNavigatingExp.left=current} '.' right=[ObjectProperty] )? ;

SubProcess :
ParallelSubProcess | SequentialSubProcess ;


ParallelSubProcess :
{ParallelSubProcess}
( loopCharacteristics=LoopCharacteristics )?
'parallel' // anonymous name=ID
'{'
( nodes+=FlowNode )*
'}' ;

SequentialSubProcess :
{SequentialSubProcess}
( loopCharacteristics=LoopCharacteristics )?
'{'
( nodes+=FlowNode )*
'}' ;

CallActivity :
( loopCharacteristics=LoopCharacteristics )?
'call' processRef=[Process] //calledElementRef
';' ;

// ********************************************************************
// Loops
// ********************************************************************

LoopCharacteristics :
StandardLoopCharacteristics | MultiInstanceLoopCharacteristics ;

StandardLoopCharacteristics :
ForLoop | WhileLoop | DoWhileLoop ;

ForLoop :
'for' '(' loopMaximum=ValueCall ')' ;

WhileLoop :
'while' '(' loopCondition=Expression ')' ;

DoWhileLoop :
'do-while' '(' loopCondition=Expression ')' ;
```

```
MultiInstanceLoopCharacteristics :
{MultiInstanceLoopCharacteristics}
'multi-instance' ( '(' loopMaximum=ValueCall ')' )? ;


// ********************************************************************
// PART II: Types and Object Definitions
// ********************************************************************

Type :
BuildinType | ObjectType ;

BuildinType :
PrimitiveType | CollectionType ;

CollectionType :
ListType | MatrixType | CubeType ;

PrimitiveType :
IntType | BooleanType | RealType | StringType ;

// ********************************************************************
// Collection Types
// ********************************************************************

ListType :
{ListType} 'list' '<'baseType=TypeReference'>'
;

MatrixType :
{MatrixType} 'matrix' '<'baseType=TypeReference'>'
;

CubeType :
{CubeType} 'cube' '<'baseType=TypeReference'>'
;

// ********************************************************************
// Object Types
// ********************************************************************

ObjectType :
'type' name=ID
'{'
( properties += ObjectProperty )*
'}' ;

ObjectProperty :
( isReadonly?='readonly' )? type=TypeReference name=ID ';' ;

TypeReference :
```

```
 BuildinType | ObjectTypeReference ;

ObjectTypeReference :
typeRef=[ObjectType] ;

// ****************************************************************
// Primitive Types
// ****************************************************************

IntType :
{IntType} 'integer' ;

BooleanType :
{BooleanType} 'boolean'
;

RealType :
{RealType} 'real' ;

StringType :
{StringType} 'string' ;

// ****************************************************************
// Literals and Terminals
// ****************************************************************

RealLiteral :
INT '.' INT ;

BoundLiteral returns ecore::EInt :
INT | '*' ;

terminal BOOLEAN returns ecore::EBoolean: 'true' | 'false' ;

// ****************************************************************
// Value Expressions
// ****************************************************************

ValueExpression :
IntValue | BooleanValue | RealValue | StringValue ;

IntValue :
value=INT ;

BooleanValue :
value=BOOLEAN ;

RealValue :
value=RealLiteral ;

StringValue :
value=STRING ;
```

Appendix C

Project Structure of PROCOL Projects developed with SCOPE

This chapter describes the project structure of concurrent programs that use the Process Coordination Library (PROCOL) coordination library and are developed with the Space-Coordinated Processes (SCOPE) coordination workbench.

Project Folders A PROCOL project developed with SCOPE is a conventional Maven project extended with an additional source folder for generated source files and a model folder for SCOPE Domain-Specific Language (DSL) models, see Figure C.1. The code generation strategy uses the conventional Generation Gap pattern, see Section 9.2.4. The individual project folders are:

- src/main/java contains the Java source code edited by a domain engineer. Contents of the folder typically include source files for SCOPE Object Types and Processes that are generated by SCOPE code generators once (i. e., that are not overwritten in subsequent generator runs).
- src/test/java contains test code provided by a domain engineer.
- src-gen/main/java contains generated Java source code that represents the coordination architecture of the project. Contents of the folder should not be modified nor put under the control of a version management system. SCOPE code generators can overwrite the contents.
- model contains the SCOPE DSL model files from which the coordination architecture of the program is generated.

- config contains additional configuration files such as the generic *log4j.properties* required by the PROCOL library for response time reporting.
- scripts is intended to contain run scripts provided by the domain engineer.
- *target* contains the compiled project sources after a Maven build is executed (e. g., via `mvn install`).

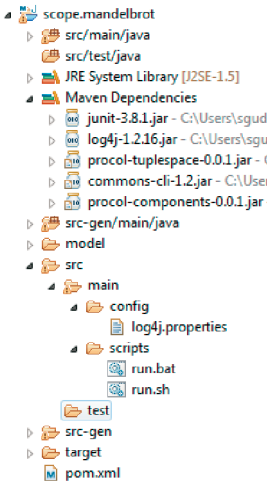

Figure C.1. Project Structure of PROCOL Projects developed with SCOPE.

Dependencies By default, a PROCOL project developed with SCOPE has the following project dependencies:

- *junit* is used for module tests provided by the domain engineer
- *log4j* is used by the PROCOL library for response time reporting.
- *procol-tuplespace* represents the Space Based System component of the PROCOL library.

- ▻ *commons-cli* provides a robust command line interface for SCOPE programs.
- ▻ *procol-components* represents the Process Orchestration component of the PROCOL library.

Maven Project Object Model The Project Object Model (POM) is the fundamental project configuration file required by Maven to build target projects. It is represented by the file *pom.xml*. For SCOPE projects, the *pom.xml* file is generated by the SCOPE coordination workbench. The following listing illustrates a *pom.xml* file for SCOPE projects. Project-specific configuration parameters are encapsulated in « and -»

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <!-- <parent>
    <artifactId>procol</artifactId>
    <groupId>org.procol</groupId>
    <version>0.0.1</version>
    <relativePath>./..</relativePath>
  </parent> -->
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.scope.target.gen</groupId>
  <artifactId><<projectName->></artifactId>
  <version>0.0.1</version>
  <packaging>jar</packaging>

  <name>scope-app-<<projectName->></name>
  <url>http://scope-dsl.sourceforge.net</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
```

```
      <version>1.2.16</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>org.procol.framework.tuplespace</groupId>
      <artifactId>procol-tuplespace</artifactId>
      <version>0.0.1</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.16</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>commons-cli</groupId>
      <artifactId>commons-cli</artifactId>
      <version>1.2</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>org.procol.framework.components</groupId>
      <artifactId>procol-components</artifactId>
      <version>0.0.1</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-source-plugin</artifactId>
        <executions>
          <execution>
            <id>attach-sources</id>
            <goals>
              <goal>jar</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <executions>
    <execution>
      <id>attach-javadocs</id>
      <goals>
        <goal>jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>

<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.5</version>
  <executions>
    <execution>
      <id>copy-resources</id>
      <!-- here the phase you need -->
      <phase>validate</phase>
      <goals>
        <goal>copy-resources</goal>
      </goals>
      <configuration>
        <outputDirectory>${basedir}/target</outputDirectory>
        <resources>
          <resource>
            <directory>src/main/config</directory>
            <filtering>true</filtering>
          </resource>
          <resource>
            <directory>src/main/scripts</directory>
            <filtering>true</filtering>
          </resource>
        </resources>
      </configuration>
    </execution>
  </executions>
</plugin>

<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <phase>install</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}</outputDirectory>
      </configuration>
```

```
          </execution>
        </executions>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.3.1</version>
        <configuration>
          <archive>
            <manifest>
              <mainClass><<qualifiedPackageName->>.<<this.name->></mainClass>
              <addClasspath>true</addClasspath>
            </manifest>
          </archive>
        </configuration>
      </plugin>

    </plugins>

    <pluginManagement><!-- Needed to tell the eclipse m2e plugin to execute
        maven-dependency-plugin (under Eclipse 3.7 Indigo)-->
      <plugins>
        <plugin>
          <groupId>org.eclipse.m2e</groupId>
          <artifactId>lifecycle-mapping</artifactId>
          <version>1.0.0</version>
          <configuration>
            <lifecycleMappingMetadata>
              <pluginExecutions>
                <pluginExecution>
                  <pluginExecutionFilter>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-dependency-plugin</artifactId>
                    <versionRange>[2.0,)</versionRange>
                    <goals>
                      <goal>copy-dependencies</goal>
                    </goals>
                  </pluginExecutionFilter>
                  <action>
                    <execute /><!-- else use 'ignore' -->
                  </action>
                </pluginExecution>
              </pluginExecutions>
            </lifecycleMappingMetadata>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>

  </build>
</project>
```

In addition to the project dependencies, the POM makes use of the following plugins for build support:

- *maven-source-plugin* creates a Java *jar* archive from the source files of the project into the project's *target* directory.
- *maven-javadoc-plugin* uses the Java *Javadoc* tool to generate the source code documentation for the project.
- *maven-resources-plugin* copies project resources from *src/main/config* to the project's *target* directory. The location of the *config* folder under *src/main* is a requirement of this plugin.
- *maven-dependency-plugin* provides the capability to copy and/or unpack artefacts from local or remote repositories to a specified location. It is used to copy all project dependencies (libraries) to the project's *target* directory.
- *maven-jar-plugin* defines the manifest for the Java *jar* archive of the project that specifies the project's main class.

Appendix D

SCOPE Model for Mandelbrot Set Visualisation

This chapter describes a Space-Coordinated Processes (SCOPE) model for the visualisation of the Mandelbrot set in the complex plane, see Section 10.3.2. It is formulated using the SCOPE coordination Domain-Specific Language (DSL), see Section 9.2. The model is used to generate the coordination code of the program.

File *mandelbrot-processes.scope*

```
definitions org.scope.target.gen.mandelbrot.processes
{
  import org.scope.target.gen.mandelbrot.types.*
  import org.scope.target.gen.mandelbrot.types.Complex.*
  import org.scope.target.gen.mandelbrot.types.Configuration.*

  collaboration Mandelbrot
  {
    space ImageSpace
    client MandelbrotClient : MandelbrotProc
  }

  process ImageProvider attends Mandelbrot
  {
    Configuration config;
    list<Image> images;

    config = read Configuration from Mandelbrot.ImageSpace;
    createImagePartitions weighted 1.0;
    publish-all images to Mandelbrot.ImageSpace;
  }

  process MandelbrotProc attends Mandelbrot
  {
    Configuration config init;
```

```
    publish config to Mandelbrot.ImageSpace;
    parallel
    {
      call ImageProvider;
      multi-instance (config.numberOfImagePartitions) call Renderer;
      call Presenter;
    }
  }

  process Renderer attends Mandelbrot
  {
    Image image;

    image = take Image where (Image.isRendered = false) from Mandelbrot.ImageSpace ;
    renderImage weighted 1.0;
    publish image to Mandelbrot.ImageSpace;
  }

  process Presenter attends Mandelbrot
  {
    Configuration config;
    list<Image> images;
    matrix<integer> mandelbrotMatrix;

    config = read Configuration from Mandelbrot.ImageSpace;
    images = for (config.numberOfImagePartitions) take Image
        where (Image.isRendered = true) from Mandelbrot.ImageSpace;
    computeMatrix weighted 1.0;
    saveMandelbrotImageAsFile weighted 1.0;
    xor
    {
      case (printCondition 0.7) : printGui weighted 1.0;
    }
  }
}
```

File *mandelbrot-types.scope*

```
definitions org.scope.target.gen.mandelbrot.types
{
  type Configuration {
    integer height;
    integer width;
    integer numberOfImagePartitions;
    Complex topleft;
    Complex bottomRight;
    boolean isGuiPresented;
  }

  type Complex {
    real a;
```

```
    real b;
  }

  type Image {
    integer id;
    Complex topLeft;
    Complex bottomRight;
    readonly matrix<integer> store;
    integer height;
    integer width;
    boolean isRendered;
  }
}
```

Appendix E

SCOPE Model for Label Positioning

This chapter describes a Space-Coordinated Processes (SCOPE) model for label positioning, see Section 10.3.3. It is formulated using the SCOPE coordination Domain-Specific Language (DSL), see Section 9.2. The model is used to generate the coordination code of a label positioning component for an exiting tool for Enterprise Architecture Visualisation (EAV).

File *AIOLabelPositioning.scope*

```
definitions de.offis.alis.svggen.concurrent.routing.labeling
{
  import de.offis.alis.svggen.concurrent.routing.labeling.types.*
  import de.offis.alis.svggen.concurrent.routing.labeling.types.PathWalkerWrapper.*
  import de.offis.alis.svggen.concurrent.routing.labeling.types.Configuration.*
  import de.offis.alis.svggen.routing.labeling.*

  collaboration LabelPositioning
  {
    space LabelSpace
    space ObstacleSpace
    client AIOMapBuilder : ConcurrentAIOMapBuilder
  }

  process ConcurrentAIOMapBuilder attends LabelPositioning
  {
    list<PathWalkerWrapper> walkers;

    publish-all walkers to LabelPositioning.LabelSpace;
    publish-all walkers to LabelPositioning.ObstacleSpace;
    call PositionSolver;
    walkers = collect PathWalkerWrapper from LabelPositioning.LabelSpace;
  }

  process PositionSolver attends LabelPositioning
```

```
  {
    Configuration  config;
    integer        numWorkers;

    createConfig;
    publish config to LabelPositioning.LabelSpace;
    multi-instance (numWorkers) call SolverWorker;
  }

  process SolverWorker attends LabelPositioning
  {
    Configuration       config;
    integer             cycles;
    integer             partitionSize;
    PathWalkerWrapper   walker;
    list<PathWalkerWrapper> obstacles;

    config = read Configuration from LabelPositioning.LabelSpace;
    initWorker;
    while (notMaxCyclesReached 0.9)
    {
      obstacles = scan PathWalkerWrapper from LabelPositioning.ObstacleSpace;
      for (partitionSize)
      {
        walker = take-available PathWalkerWrapper
            from LabelPositioning.LabelSpace;
        xor
        {
          case (ShouldProcessWalker 0.9) : // at least if walker exists
          solve; // by Simulated Annealing; includes to increase the trial count of the walker
          xor
          { // update obstacle space
            case (changeOccured 0.5) :
            take-available PathWalkerWrapper
                where (PathWalkerWrapper.id = walker.id)
                from LabelPositioning.ObstacleSpace;
            publish walker to LabelPositioning.ObstacleSpace;
          }
          publish walker to LabelPositioning.LabelSpace;
        }
      }
      increaseCycleCount;
      resetObstacles;
    }
  }
}
```

File *AIOLabelPositioningTypes.scope*

```
definitions de.offis.alis.svggen.concurrent.routing.labeling.types
{
  import  de.offis.alis.svggen.routing.labeling.*
```

```
  type Configuration {
    integer    maxCycles;
    integer    maxTrials;
    integer    spacing;
    real       temperature;
    real       temperatureDecreaseFactor;
    integer    partitionSize;
  }

  type PathWalkerWrapper {
    PathWalker walker;
    integer trials;
    integer id;
  }
}
```

File *AIOLabelPositioningTypesStubs.scope*

```
definitions de.offis.alis.svggen.routing.labeling
{
  type PathWalker { } // Stub for existing PathWalker
}
```

List of Acronyms

AC-MDSD	Architecture-Centric Model-Driven Software Development
ADL	Architectural Description Language
AIO	Application Interface Overview
API	Application Programming Interface
AGG	Attributed Graph Grammar System
AOM	Aspect-Oriented Modeling
AOP	Aspect-Oriented Programming
AVOPT	Analysis, Verification, Optimization, Parallelization, and Transformation
BPMI	Business Process Management Initiative
BPMN	Business Process Model and Notation
CCC	Cross-Cutting Concern
CL	Coordination Language
ccNUMA	Cache-Coherent Non-Uniform Memory Access
COOP	Concurrent Object-Oriented Programming
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
CT	Coordination Theory

DARE	Domain Analysis and Reuse Environment
DMM	Distributed Memory Machine
DSL	Domain-Specific Language
DSSA	Domain Specific Software Architectures
EAV	Enterprise Architecture Visualisation
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
EPC	Event Driven Process Chain
FAST	Family-oriented Abstractions, Specification and Translation
FODA	Feature-oriented Domain-Analysis
FTP	File Transfer Protocol
GC	Garbage Collection
GPL	General-Purpose Programming Language
GPU	Graphics Processing Unit
GReAT	Graph Rewriting and Transformation
GUI	Graphical User Interface
HPC	High-Performance Computing
IDE	Integrated Development Environment
ISAT	Intel Software Autotuning Tool
JIT	Just-In-Time
JVM	Java Virtual Machine
LHS	Left-Hand Side

M2M	Model-to-Model
M2T	Model-to-Text
MDS	Model-Driven System
MDSD	Model-Driven Software Development
MDT	Model Development Tools
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MOF	Meta Object Facility
MPI	Message Passing Interface
MPP	Massively Parallel Processors
MPS	JetBrains Meta Programming System
MWE	Modeling Workflow Engine
NP	Non-Deterministic Polynomial-Time
NUMA	Non-Uniform Memory Access
OCL	Object Constraint Language
ODE	Ontology-based Domain Engineering
ODM	Organization Domain Modeling
OMG	Object Management Group
OO	Object-Orientation
OS	Operating System
PE	Processing Element
PFLP	Point-Feature Label Placement

PNML	Petri Net Markup Language
POM	Project Object Model
PROCOL	Process Coordination Library
PSM	Platform-Specific Model
PTP	Eclipse Parallel Tools Platform
QVT	Query/View/Transformation
RAM	Random Access Machine
RHS	Right-Hand Side
SBS	Space-Based Systems
SCOPE	Space-Coordinated Processes
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMM	Shared Memory Machine
SMP	Symmetric Multiprocessor
SOA	Service-Oriented Architectures
SQL	Structured Query Language
SVG	Scalable Vector Graphics
SVN	Subversion
TBB	Intel Threading Building Blocks
TPL	Microsoft Task Parallel Library
UE	Unit of Execution
UML	Unified Modeling Language

UUID	Universally Unique Identifier
VIATRA2	VIsual Automated model TRAnsformations
VSM	Virtual Shared Memory
WF	Windows Workflow Foundation
WS-BPEL	Web Services Business Process Execution Language
WS-CDL	Web Services Choreography Description Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XPDL	XML Process Definition Language
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations

Bibliography

[XPD 2008] XML Process Definition Language, October 2008. URL http://www.wfmc.org. (cited on page 42)

[MPI 2009] MPI: A Message-Passing Interface Standard Version 2.2, September 2009. (cited on page 3)

[MOF 2011] OMG Meta Object Facility (MOF) Core Specification, August 2011. (cited on page 61)

[Ope 2011] OpenMP Application Program Interface Version 3.1, 2011. July. (cited on pages 3 and 30)

[QVT 2011] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, January 2011. (cited on pages 61 and 178)

[Xte 2011] Xtext 2.1 Documentation. Technical report, The Eclipse Foundation, 2011. (cited on pages 64, 72, 84, 176, 179, 187, 188, 245, and 267)

[Adiga et al. 2002] N. R. Adiga, G. Almási, G. S. Almasi, Y. Aridor, R. Barik, D. K. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. A. Blumrich, A. A. Bright, J. R. Brunheroto, C. Cascaval, J. G. Castaños, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. L.-T. Chiu, T. M. Cipolla, P. Crumley, K. M. Desai, A. Deutsch, T. Domany, M. B. Dombrowa, W. E. Donath, M. Eleftheriou, C. C. Erway, J. Esch, B. G. Fitch, J. Gagliano, A. Gara, R. Garg, R. S. Germain, M. Giampapa, B. Gopalsamy, J. A. Gunnels, M. Gupta, F. G. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, E. Krevat, M. P. Kurhekar, A. P. Lanzetta, D. Lieber, L. K. Liu, M. Lu, M. P. Mendell, A. Misra, Y. Moatti, L. S. Mok, J. E. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, A. J. Oliner, V. Pandit, R. B. Pudota, R. A. Rand, R. D. Regan, B. Rubin, A. E. Ruehli, S. Rus, R. K. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song,

V. Srinivasan, B. D. Steinmacher-Burow, K. Strauss, C. W. Surovic, R. A. Swetz, T. Takken, R. B. Tremaine, M. Tsao, A. R. Umamaheshwaran, P. Verma, P. Vranas, T. J. C. Ward, M. E. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. J. Krolak, C. T. Li, T. A. Liebsch, J. A. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. D. Wait, J. Wittrup, M. Bae, K. A. Dockser, L. Kissel, M. K. Seager, J. S. Vetter, and K. Yates. An overview of the BlueGene/L Supercomputer. Technical report, IBM and Lawrence Livermore National Laboratory, 2002. (cited on page 24)

[Agha 1990] G. Agha. Concurrent Object-Oriented Programming. *Commun. ACM*, 33(9):125–141, 1990. (cited on page 31)

[Agha 1985] G. A. Agha. *Actors: A Model Of Concurrent Computation In Distributed Systems*. PhD thesis, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, June 1985. (cited on pages 31 and 145)

[Allweyer 2009] T. Allweyer. *BPMN 2.0 - Business Process Model and Notation: Einführung in den Standard für die Geschäftsprozessmodellierung*. Books on Demand, 2nd edition, November 2009. (cited on page 51)

[Alves et al. 2007] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0, April 2007. URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`. (cited on page 42)

[AMD 2011] AMD. Multi-Core Processing with AMD, August 2011. URL `http://www.amd.com/us/products/technologies/multi-core-processing/Pages/multi-core-processing.aspx`. (cited on page 2)

[Antoy and Hanus 2010] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53:74–85, Apr. 2010. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1721654.1721675. URL `http://doi.acm.org/10.1145/1721654.1721675`. (cited on pages 30 and 31)

[Arenas Sánchez et al. 2011] P. Arenas Sánchez, S. Estévez Martín, A. J. Fernández Leiva, A. Gil Luezas, F. J. López Fraguas, M. Rodríguez Artalejo, and F. Sáenz Pérez. TOY: A Multiparadigm Declarative Language Version 2.3.2. Technical report, Universidad Complutense de Madrid, October 2011. (cited on page 31)

[Armstrong 2007] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Programmers, LLC., 2007. (cited on pages 30 and 31)

[Awad et al. 2009] A. Awad, A. Grosskopf, A. Meyer, and M. Weske. Enabling Resource Assignment Constraints in BPMN. BPT Technical Report 04-2009, Ahmed Awad, Alexander Grosskopf, Andreas Meyer, Mathias Weske, 2009. (cited on page 245)

[Axway et al. 2010] Axway, BizAgi, Bruce Silver Associates, IDS Scheer, IBM Corp., MEGA International, Model Driven Solutions, Object Management Group, Oracle, SAP AG, Software AG, TIBCO Software, and Unisys. Business Process Model and Notation (BPMN) Version 2.0. Technical report, Object Management Group, Inc. (OMG), 2010. (cited on pages 5, 32, 39, 42, 48, 49, 83, 121, 122, 123, 124, 125, 126, 127, 128, 129, 131, 132, 133, 134, and 182)

[Backus 1958] J. W. Backus. Automatic programming: properties and performance of FORTRAN systems I and II. In *Proceedings Symposium on the Mechanisation of Thought Processes*, pages 232–255, Teddington, Middlesex, England, The National Physical Laboratory, November 1958. Her Majesty's Stationary Office (HMSO). (cited on page 82)

[Balzarotti et al. 2007] D. Balzarotti, P. Costa, and G. P. Picco. The LighTS tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems*, 5(2):215–231, 2007. (cited on pages 7, 29, 39, 44, 46, 116, 145, 162, 164, and 241)

[Basili et al. 1994] V. R. Basili, G. Caldiera, and H. D. Rombach. The Goal Question Metric Approach. In *Encyclopedia of Software Engineering*. Wiley, 1994. (cited on page 203)

[Ben-Ari 2006] M. Ben-Ari. *Principles of Concurrent and Distributed Programming, Second Edition*. Addison-Wesley, 2006. ISBN 9780321312839. (cited on pages 18, 19, and 21)

[Benoit et al. 2005] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 613–613. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-28700-1. URL http://dx.doi.org/10.1007/11549468_83. (cited on page 28)

[Bentley 1986] J. Bentley. Programming pearls: little languages. *Commun. ACM*, 29:711–721, August 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/6424.315691. URL http://doi.acm.org/10.1145/6424.315691. (cited on page 82)

[Bézivin and Gerbé 2001] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 273, Washington, DC, USA, 2001. IEEE Computer Society. (cited on page 56)

[Borkar 2007] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: 10.1145/1278480.1278667. URL http://doi.acm.org/10.1145/1278480.1278667. (cited on page 260)

[Bray et al. 2008] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008. (cited on page 83)

[Browne et al. 1994] J. C. Browne, J. Dongarra, S. I. Hyder, K. Moore, and P. Newton. Visual Programming and Parallel Computing. Technical report, University of Tennessee, Knoxville, TN, USA, 1994. (cited on pages 150 and 247)

[Buschmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. John Wiley & Sons Ltd., Chichester, UK, 1996. (cited on page 27)

[Butcher et al. 1994] P. Butcher, A. C. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency - Practice and Experience*, 6(6):505–516, 1994. (cited on pages 119 and 145)

[Buyya 2000] R. Buyya. *The Design of PARAS Microkernel*, chapter Parallel Computing at a Glance. June 2000. URL http://www.buyya.com/microkernel/. e-book. (cited on page 2)

[Carriero and Gelernter 1989] N. Carriero and D. Gelernter. Linda in Context. *Commun. ACM*, 32(4):444–458, 1989. (cited on page 39)

[Chafi et al. 2010] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 835–847, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869527. URL http://doi.acm.org/10.1145/1869459.1869527. (cited on page 244)

[Chamberlin and Boyce 1974] D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM. doi: http://doi.acm.org/10.1145/800296.811515. URL http://doi.acm.org/10.1145/800296.811515. (cited on page 82)

[Christensen et al. 1995] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Trans. Graph.*, 14(3):203–232, July 1995. ISSN 0730-0301. doi: 10.1145/212332.212334. URL http://doi.acm.org/10.1145/212332.212334. (cited on pages 219, 228, 230, and 231)

[Christiansen et al. 2010] D. R. Christiansen, M. Carbone, and T. Hildebrandt. Formal Semantics and Implementation of BPMN 2.0 Inclusive

Gateways. In *Proc. of Web Services and Formal Methods (WS-FM'10)*, Berlin, Heidelberg, 2010. Springer-Verlag. URL http://www.itu.dk/people/maca/papers/CD10.pdf. (cited on page 133)

[Ciancarini 1996] P. Ciancarini. Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/234528.234732. (cited on pages 4, 35, 37, 38, 40, and 112)

[Ciechanowicz et al. 2009] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli: A Comprehensive Overview. In J. Becker, K. Backhaus, H. L. Grob, B. Hellingrath, T. Hoeren, S. Klein, H. Kuchen, U. Müller-Funk, U. W. Thonemann, and G. Vossen, editors, *Working Papers*. ERCIS European Research Center for Information Systems, 2009. (cited on page 28)

[Clocksin and Mellish 1994] W. F. Clocksin and C. S. Mellish. *Programming in Prolog (4. ed.)*. Springer, 1994. ISBN 978-3-540-58350-9. (cited on page 30)

[Codd 1970] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362384.362685. URL http://doi.acm.org/10.1145/362384.362685. (cited on page 82)

[Cole 1989] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989. (cited on page 28)

[Crowston et al. 2006] K. Crowston, J. Rubleske, and J. Howison. *Human-Computer Interaction in Management Information Systems*, chapter Coordination theory: A ten-year retrospective, pages 120–138. M. E. Sharpe, Inc., 2006. (cited on page 36)

[Czarnecki and Eisenecker 2005] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tool, and Applications*. Addison-Wesley, 6th edition, April 2005. (cited on pages 66 and 72)

[Czarnecki and Helsen 2006] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3): 621–646, 2006. (cited on pages 58 and 61)

[DeMarco 1979] T. DeMarco. *Structured Analysis and System Specification*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1979. ISBN 0138543801. URL http://portal.acm.org/citation.cfm?id=1102012. (cited on page 32)

[Deursen et al. 2000] A. Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35: 26–36, 2000. (cited on pages 81, 82, and 84)

[Dijkman et al. 2007] R. M. Dijkman, M. Dumas, and C. Ouyang. Formal Semantics and Analysis of BPMN Process Models. 2007. URL http://eprints.qut.edu.au/7115/. (cited on pages 133 and 137)

[Dijkstra 1968] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968. http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF. (cited on page 26)

[Dmitriev 2004] S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. Technical report, JetBrains Inc., November 2004. (cited on page 72)

[Edwards 2000] W. K. Edwards. *Core Jini*. Prentice Hall PTR, 2nd edition, December 2000. (cited on page 44)

[Efftinge et al. 2004-2010] S. Efftinge, P. Friese, A. Hase, D. Hübner, C. Kadura, B. Kolb, J. Köhnlein, D. Moroff, K. Thoms, M. Völter, P. Schönbach, M. Eysholdt, S. Reinisch, D. Jockel, and A. Arnold. Xpand Documentation. Technical report, 2004-2010. (cited on page 64)

[Efftinge et al. 2008] S. Efftinge, P. Friese, A. Haase, D. Hübner, C. Kadura, B. Kolb, J. Köhnlein, D. Moroff, K. Thoms, M. Völter, P. Schönbach, M. Eysholdt, D. Hübner, and S. Reinisch. openArchitectureWare User Guide Version 4.3.1. Technical report, 2008. (cited on page 64)

[Eysholdt et al. 2009] M. Eysholdt, S. Frey, and W. Hasselbring. EMF Ecore Based Meta Model Evolution and Model Co-Evolution. *Softwaretechnik-Trends*, 29(2):20–21, May 2009. (WSR 2009 Proceedings). (cited on page 108)

[Falbo et al. 2002] R. A. Falbo, G. Guizzardi, and K. C. Duarte. An Ontological Approach to Domain Engineering. In ACM, editor, *Inproceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 351–358, 2002. (cited on pages 77 and 80)

[Falcou 2009] J. Falcou. Parallel Programming with Skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009. (cited on page 28)

[Favre 2004a] J.-M. Favre. Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development*, 2004a. (cited on pages 56 and 109)

[Favre 2004b] J.-M. Favre. Towards a Basic Theory to Model Model Driven Engineering. In *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004b. (cited on pages 56, 77, and 80)

[Favre 2005] J.-M. Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2005/21. (cited on page 56)

[Favre and NGuyen 2005] J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):59–74, April 2005. ISSN 15710661. doi: 10.1016/j.entcs.2004.08.034. URL http://dx.doi.org/10.1016/j.entcs.2004.08.034. (cited on pages 56, 57, 59, 60, 77, 80, and 244)

[Fiedler et al. 2005] D. Fiedler, K. Walcott, T. Richardson, G. M. Kapfhammer, A. Amer, and P. K. Chrysanthis. Towards the measurement of

tuple space performance. *SIGMETRICS Perform. Eval. Rev.*, 33(3):51–62, Dec. 2005. ISSN 0163-5999. doi: 10.1145/1111572.1111574. URL http://doi.acm.org/10.1145/1111572.1111574. (cited on pages 148, 162, 201, 238, and 243)

[Floyd 1984] C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to prototyping*, pages 1–18, Berlin, 1984. Proceedings of the Working Conference on Prototyping, Springer. (cited on page 28)

[Flynn 1972] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21:948–960, September 1972. ISSN 0018-9340. URL http://dl.acm.org/citation.cfm?id=1952456.1952459. (cited on page 16)

[Fowler 2010] M. Fowler. *Domain-Specific Languages*. Publisher: Addison-Wesley Professional, October 2010. (cited on pages 75, 83, and 84)

[Frakes et al. 1998] W. Frakes, R. Prieto-Diaz, and C. Fox. DARE: Domain Analysis and Reuse Environment. In *Annals of Software Engineering*, number 5, pages 125–141, 1998. (cited on pages 77 and 80)

[Freeman et al. 1999] E. Freeman, S. Hupfer, and K. Arnold. *Javaspaces Principles, Patterns, and Practice*. Prentice Hall, June 1999. (cited on page 44)

[Freisleben and Kielmann 1997] B. Freisleben and T. Kielmann. Object-Oriented Parallel Programming with Objective Linda. *Journal of Systems Architecture*, 1997. (cited on pages 43, 112, 113, and 120)

[Freund and Rücker 2010] J. Freund and B. Rücker. *Praxishandbuch BPMN 2.0*. Carl Hanser Verlag GmbH & CO. KG, 2nd edition, September 2010. (cited on page 51)

[Garlan and Perry 1995] D. Garlan and D. E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Trans. Softw. Eng.*, 21(4): 269–274, 1995. ISSN 0098-5589. (cited on page 145)

[Gelernter 1985] D. Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/2363.2433. (cited on pages 31, 39, and 43)

[Gelernter and Carriero 1992] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35:97–107, 1992. (cited on pages 4, 31, 35, 39, 40, 112, 113, and 146)

[Georges et al. 2007] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. volume 42 of *OOPSLA '07*, pages 57–76, New York, NY, USA, October 2007. ACM. doi: http://doi.acm.org/10.1145/1297105.1297033. URL http://doi.acm.org/10.1145/1297105.1297033. (cited on pages 148, 172, 204, and 243)

[Ghosh 2011] D. Ghosh. DSL for the Uninitiated. *Queue*, 9:10:10–10:21, June 2011. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1989748.1989750. URL http://doi.acm.org/10.1145/1989748.1989750. (cited on page 83)

[Goetz 2009] B. Goetz. *Java Concurrency in Practice*. Addison-Wesley, Pearson Education, Inc., 7th Printing edition, January 2009. (cited on pages 26, 148, 204, 220, and 243)

[Graham 1993] P. Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall PTR, June 1993. (cited on page 83)

[Greenfield et al. 2004] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004. (cited on page 72)

[Gringel et al. 2012] P. Gringel, S. Gudenkauf, and S. Kruse. DeCREE: Eine Domänenspezifische Sprache zur Modellierung von Choreography-First Szenarien. In D. C. Mattfeld and S. Robra-Bissantz, editors, *Multikonferenz Wirtschaftsinformatik 2012: Tagungsband der MKWI 2012*, pages 1661–1674, Berlin, 2012. GITO mbH Verlag. ISBN 978-3-942183-63-5. (cited on page 125)

[Gudenkauf 2010] S. Gudenkauf. A Coordination-Based Model-Driven Method for Parallel Application Development. In S. Ghosh, editor, *Lecture Notes in Computer Science (LNCS): Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, pages 26–35, Heidelberg, 2010. Springer. (cited on page 7)

[Gudenkauf and Hasselbring 2011] S. Gudenkauf and W. Hasselbring. Space-Based Multi-Core Programming in Java. In C. Wimmer and C. W. Probst, editors, *PPPJ 2011: Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, pages 41–50, New York, 2011. The Associacion for Computing Machinery, Inc. ISBN 9781450309356. (cited on pages 7 and 45)

[Gudenkauf et al. 2012] S. Gudenkauf, S. Kruse, and W. Hasselbring. Domain-Specific Modelling for Coordination Engineering with SCOPE. In E. J. Sinz and A. Schürr, editors, *Modellierung 2012*, volume P-201 of *GI-Reihe Lecture Notes in Informatics (LNI)*, pages 203–218. GI, 2012. ISBN 978-3-88579-295-6. (cited on page 8)

[Halloway 2009] S. Halloway. *Programming Clojure*. Pragmatic Programmers. The Pragmatic Programmers, LLC., 1 edition, 2009. ISBN 1934356336. (cited on page 30)

[Hanus 2005] M. Hanus. Functional Logic Programming: From Theory to Curry. Technical report, Institut für Informatik, CAU Kiel, October 2005. (cited on page 31)

[Hanus 2007] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007. (cited on page 30)

[Harel 1987] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. (cited on page 32)

[Harris et al. 2005] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065952. URL http://doi.acm.org/10.1145/1065944.1065952. (cited on page 30)

[Hasselbring 1998] W. Hasselbring. The ProSet-Linda approach to prototyping parallel systems. *Journal of Systems and Software*, 43(3):187–196, 1998. (cited on pages 28 and 43)

[Hasselbring 2000] W. Hasselbring. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv.*, 32(1):43–79, 2000. (cited on pages 28 and 29)

[Heaton 2007] J. Heaton. *Introduction to Neural Networks for Java, 1st Edition*, chapter Understanding Simulated Annealing. Heaton Research, Inc., 1 edition, 2007. URL `http://www.heatonresearch.com/book/programming-neural-networks-java.html`. (cited on pages 230 and 234)

[Heiser 2009] G. Heiser. Many-Core Chips – A Case for Virtual Shared Memory. In *Proceedings of the 2nd Workshop on Managed Many-Core Systems*, Washington, DC, USA, March 2009. (cited on page 259)

[Henderson et al. 2012] F. Henderson, T. Conway, Z. Somogyi, D. Jeffery, P. Schachte, S. Taylor, C. Speirs, T. Dowd, R. Becket, M. Brown, and P. Wang. The Mercury Language Reference Manual Version 11.07. Technical report, The University of Melbourne, 2012. (cited on page 31)

[Herrmannsdoerfer et al. 2009] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - Automating Coupled Evolution of Metamodels and Models. In S. Drossopoulou, editor, *ECOOP 2009 Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 52–76. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-03012-3. (cited on page 108)

[Herrmannsdoerfer et al. 2011] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In B. Malloy, S. Staab, and M. van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 163–182. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-19439-9. (cited on page 108)

[Hillah et al. 2009] L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Trèves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter*, 76:9–28, October 2009. URL `http://www.pnml.org/papers/pnnl76.pdf`. (originally presented at the 10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools – CPN'09). (cited on page 259)

[Hillis and Steele 1986] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/7902.7903. URL http://doi.acm.org/10.1145/7902.7903. (cited on page 30)

[Hoare 1974] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/355620.361161. URL http://doi.acm.org/10.1145/355620.361161. (cited on page 26)

[Hoare 1978] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978. (cited on page 26)

[Hoare 1985] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153271-5. (cited on page 26)

[Horton and Kleinman 2010] N. J. Horton and K. Kleinman. *Using R for Data Management, Statistical Analysis, and Graphics*. CRC Press, 2010. (cited on page 30)

[Hsiung et al. 2009] P.-A. Hsiung, S.-W. Lin, Y.-R. Chen, N.-L. Hsueh, C.-H. Chang, C.-H. Shih, C.-S. Koong, C.-S. Lin, C.-H. Lu, S.-Y. Tong, W.-T. Su, and W. C. Chu. Model-Driven Development of Multi-Core Embedded Software. *ICSE Workshop on Multicore Software Engineering*, 0:9–16, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/IWMSE.2009.5071378. (cited on pages 29 and 242)

[Hutton 2007] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007. (cited on page 30)

[IEEE Architecture Working Group 2000] IEEE Architecture Working Group. IEEE Std 1471-2000, Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000. (cited on pages 15, 18, 39, 40, 69, 84, and 145)

[Intel Corporation 2011a] Intel Corporation. Intel Many Integrated Core Architecture, August 2011a. URL http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. (cited on page 2)

[Intel Corporation 2011b] Intel Corporation. Parallel Programming. Multicore processors TODAY, many-core co-processors READY. Technical report, Intel, 2011b. (cited on page 2)

[Intel Corporation 2011c] Intel Corporation. Intel(R) Threading Building Blocks Reference Manual. Technical report, Intel Corporation, 2011c. (cited on pages 29, 32, and 39)

[Jelenković and Poljak 1998] L. Jelenković and J. Poljak. Multithreaded Simulated Annealing. In *Proc. 20th Int. Conference ITI'98*, pages 525–530, Pula, June 14-17 1998. (cited on page 231)

[Kang et al. 1990] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990. (cited on pages 77 and 80)

[Karcher and Pankratius 2011] T. Karcher and V. Pankratius. Auto-Tuning Multicore Applications at Run-Time with a Cooperative Tuner. publikation, February 2011. (cited on page 27)

[Kavantzas et al.] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language. (cited on page 42)

[Kelly and Tolvanen 2008] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008. (cited on page 72)

[Kielmann 1996] T. Kielmann. Designing a Coordination Model for Open Systems. In *COORDINATION '96: Proceedings of the First International Conference on Coordination*, pages 267–284, London, UK, 1996. Springer-Verlag. (cited on pages 112, 144, and 145)

[Kirkpatrick et al. 1983] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983. (cited on page 230)

[Klatt 2007] B. Klatt. Xpand: A Closer Look at the model2text Transformation Language. Technical report, Chair for Software Design and Quality (SDQ), Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, Germany, July 2007. (cited on page 64)

[Kleppe 2008] A. Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008. ISBN 0321553454, 9780321553454. (cited on pages 75, 150, and 247)

[Kliewer and Tschöke 2000] G. Kliewer and S. Tschöke. A General Parallel Simulated Annealing Library and its Application in Airline Industry. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00)*, pages 55–62. IEEE Computer Society, 2000. (cited on page 231)

[Kopp et al. 2008] O. Kopp, D. Martin, D. Wutke, and F. Leymann. On the Choice Between Graph-Based and Block-Structured Business Process Modeling Languages. In P. Loos, M. Nüttgens, K. Turowski, and D. Werth, editors, *Lecture Notes in Informatics (LNI), Modellierung betrieblicher Informationssysteme (MobIS 2008)*, pages 59–72. Gesellschaft für Informatik e.V. (GI), 2008. (cited on page 246)

[Kruse 2011] S. Kruse. On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In *Preliminary Proceedings of the International Workshop on Models and Evolution*, ME 2011. ACM/IEEE, October 2011. (cited on page 108)

[Kruse et al. 2009] S. Kruse, J. S. Addicks, M. Postina, and U. Steffens. Decoupling Models and Visualisations for Practical EA Tooling. In A. Dan, F. Gittler, and F. Toumani, editors, *ICSOC/ServiceWave Workshops*, volume 6275 of *Lecture Notes in Computer Science*, pages 62–71, 2009. ISBN 978-3-642-16131-5. URL http://dx.doi.org/10.1007/978-3-642-16132-2. (cited on page 228)

[Kuhn 1996] T. S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1996. (cited on page 2)

[Lea 1999] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman, Amsterdam, 2 edition, 1999. (cited on page 26)

[Lee 2006] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5): 33–42, May 2006. URL http://www.truststc.org/pubs/96.html. (cited on pages 3, 22, and 111)

[Lee et al. 2011] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky. Implementing Domain-Specific Languages for Heterogeneous Parallel Computing. *IEEE Micro*, 31:42–53, 2011. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2011.68. (cited on page 244)

[Leijen and Hall 2007] D. Leijen and J. Hall. Optimize Managed Code For Multi-Core Machines, October 2007. URL http://msdn.microsoft.com/en-us/magazine/cc163340.aspx. (cited on pages 29 and 39)

[Lingam 2009] C. Lingam. Rapid Parallel Application Development. Technical report, Intel Software Network, 2009. URL http://software.intel.com/en-us/articles/rapid-parallel-application-development/. (cited on page 243)

[Liu and Gorton 1998] A. Liu and I. Gorton. PARSE-DAT: An Integrated Environment for the Design and Analysis of Dynamic Software Architectures. In *PDSE*, pages 146–, 1998. (cited on page 28)

[Luk 2010] C.-K. Luk. The Intel(R) Software Autotuning Tool (ISAT) User Manual (version 1.0.0). Technical report, Intel Corporation, 2010. (cited on page 27)

[Luk et al. 2011] C.-K. Luk, R. Newton, W. Hasenplaugh, M. Hampton, and G. Lowney. A Synergetic Approach to Throughput Computing on x86-Based Multicore Desktops. *IEEE Software*, 28(1):39–50, 2011. (cited on page 27)

[Malone and Crowston 1994] T. W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994. (cited on pages 31, 36, and 145)

[Marowka 2007] A. Marowka. Parallel Computing on any Desktop. *Commun. ACM*, 50(9):74–78, 2007. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1284621.1284622. (cited on page 2)

[Marowka 2010] A. Marowka. Pitfalls and Issues of Manycore Programming. *Advances in Computers*, 79:71–117, 2010. (cited on pages 2, 3, 27, and 112)

[Mattson et al. 2004] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley. Pearson Education, Inc., Boston, September 2004. ISBN 0321228111. (cited on pages 3, 17, 21, 22, 23, 24, 25, 27, 28, and 99)

[Medvidovic and Taylor 1997] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *ESEC / SIGSOFT FSE*, pages 60–76, 1997. (cited on page 146)

[Melzer 2007] I. Melzer. *Service-orientierte Architekturen mit Web Services*. Elsevier, München, 2nd edition, 2007. (cited on pages 38 and 42)

[Mernik et al. 2005] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37 (4):316–344, Dec. 2005. ISSN 03600300. doi: 10.1145/1118890.1118892. URL `http://portal.acm.org/citation.cfm?doid=1118890.1118892`. (cited on pages 75, 76, 78, 82, 84, 87, 104, 143, 182, and 244)

[Meyer 2000] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, March 2000. (cited on pages 31 and 43)

[Murphy et al. 2006] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15:279–328, July 2006. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1151695.1151698. URL `http://doi.acm.org/10.1145/1151695.1151698`. (cited on page 46)

[Nichols et al. 1996] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996. ISBN 1-56592-115-1. (cited on page 22)

[Oechsle 2007] R. Oechsle. *Parallele und verteilte Anwendungen in Java*. Hanser Fachbuch, 2 edition, September 2007. (cited on page 19)

[Omicini and Papadopoulos 2001] A. Omicini and G. A. Papadopoulos. Editorial: Why Coordination Models and languages in AI? *Applied Artificial Intelligence*, 15(1):1–10, 2001. (cited on pages 31 and 36)

[Omicini and Zambonelli 1998] A. Omicini and F. Zambonelli. TuCSoN: a Coordination Model for Mobile Information Agents. In D. G. Schwartz, M. Divitini, and T. Brasethvik, editors, *1st International Workshop on Innovative Internet Information Systems (IIIS'98)*, pages 177–187. IDI – NTNU, Trondheim (Norway), 1998. (cited on page 45)

[Omicini and Zambonelli 1999] A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190, San Antonio, TX, USA, 28 Feb. – 2 Mar. 1999. ACM. ISBN 1-58113-086-4. Special Track on Coordination Models, Languages and Applications. (cited on page 45)

[Onbaşoğlu and Özdamar 2001] E. Onbaşoğlu and L. Özdamar. Parallel Simulated Annealing Algorithms in Global Optimization. *J. of Global Optimization*, 19(1):27–50, Jan. 2001. ISSN 0925-5001. doi: 10.1023/A:1008350810199. URL http://dx.doi.org/10.1023/A:1008350810199. (cited on page 231)

[Ortega-Arjona 2006] J. L. Ortega-Arjona. *Architectural Patterns for Parallel Programming: Models for Performance Estimation*. PhD thesis, Department of Computer Science, University College London, 2006. (cited on page 99)

[Ortega-Arjona 2010] J. L. Ortega-Arjona. *Patterns for Parallel Software Design*. Wiley Series in Software Design Patterns. John Wiley and Sons, Ltd., Chichester, West Sussex, UK, 2010. (cited on pages 3, 5, 11, 23, 24, 25, 26, 27, 28, 39, 91, 98, 99, 100, 101, and 232)

[Pankratius et al. 2009] V. Pankratius, A. Jannesari, and W. F. Tichy. Parallelizing Bzip2: A Case Study in Multicore Software Engineering. *IEEE Software*, 26(6):70–77, Nov. 2009. Download: http://dx.doi.org/10.1109/MS.2009.183. (cited on page 3)

[Papadopoulos and Arbab 1998] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *Advances in Computers*, pages 329–400. Academic Press, 1998. (cited on pages 4, 35, 40, and 146)

[Pllana et al. 2009] S. Pllana, S. Benkner, E. Mehofer, L. Natvig, and F. Xhafa. Towards an Intelligent Environment for Programming Multicore Computing Systems. In C. Eduardo, M. Alexander, A. Streit, J. L. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and J. Shantenu, editors, *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415, pages 141–151, Berlin / Heidelberg, 2009. Springer. (cited on pages 29 and 242)

[Pountain 1987] D. Pountain. *A tutorial introduction to Occam programming - including a specification section defining the extended Occam language*. INMOS, 1987. (cited on page 26)

[Quintero et al. 2001] C. E. C. Quintero, P. de la Fuente, and M. Barrio-Solórzano. Dynamic Coordination Architecture through the use of Reflection. In *SAC*, pages 134–140, 2001. (cited on page 146)

[Rauber and Rünger 2007] T. Rauber and G. Rünger. *Parallele Programmierung*. Springer, Berlin, auflage: 2 edition, 2007. (cited on pages 16 and 21)

[Reinders 2007] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007. (cited on pages 29 and 242)

[Reisig 2010] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 15 July 2010. 248 pages; ISBN 978-3-8348-1290-2. (cited on pages 32 and 133)

[Reussner and Hasselbring 2009] R. Reussner and W. Hasselbring, editors. *Handbuch der Software-Architektur*. Dpunkt Verlag, 2nd edition, 2009. (cited on pages 58, 61, and 107)

[Rivera et al. 2009] J. E. Rivera, J. R. Romero, and A. Vallecillo. Models in Software Engineering. pages 60–65, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-01647-9. doi: http://dx.doi.org/10.1007/978-3-642-01648-6_7. URL http://dx.doi.org/10.1007/978-3-642-01648-6_7. (cited on page 73)

[Rodríguez et al. 2007] A. Rodríguez, E. Fernández-Medina, and M. Piattini. A BPMN Extension for the Modeling of Security Requirements in Business Processes. *IEICE - Transactions on Information and Systems*, E90-D(4):745–752, 2007. (cited on page 245)

[Roman and Cunningham 1990] G.-C. Roman and H. C. Cunningham. Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Trans. Softw. Eng.*, 16:1361–1373, December 1990. ISSN 0098-5589. doi: 10.1109/32.62445. URL http://dl.acm.org/citation.cfm?id=99197.99202. (cited on page 41)

[Rompf et al. 2011] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-Blocks for Performance Oriented DSLs. In O. Danvy and C. chieh Shan, editors, *DSL*, volume 66 of *EPTCS*, pages 93–117, 2011. URL http://dblp.uni-trier.de/db/series/eptcs/eptcs66.html#abs-1109-0778. (cited on page 244)

[Rowstron 1996] A. I. T. Rowstron. *Bulk Primitives in Linda Run-Time Systems*. PhD thesis, Department of Computer Science, University of York, 1996. (cited on pages 119, 120, 130, and 237)

[Scherp et al. 2009] G. Scherp, A. Höing, S. Gudenkauf, W. Hasselbring, and O. Kao. Using UNICORE and WS-BPEL for Scientific Workflow Execution in Grid Environments. In *Euro-Par 2009 Workshops - Parallel Processing*, volume LNCS 6043 of *LNCS*, pages 335–344, Heidelberg, 2009. Springer. (cited on page 38)

[Schleicher et al. 2010] D. Schleicher, F. Leymann, D. Schumm, and M. Weidmann. Compliance scopes: Extending the BPMN 2.0 meta model to specify compliance requirements. In *2010 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8, Perth, WA, 2010. IEEE. doi: http://dx.doi.org/10.1109/SOCA.2010.5707154. (cited on page 245)

[Schmidt 2006] D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006. (cited on pages 67, 72, and 76)

[Seidewitz 2003] E. Seidewitz. What Models Mean. *IEEE Softw.*, 20(5):26–32, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231147. (cited on page 56)

[Shalom 2006] N. Shalom. Space-Based Architecture and The End of Tier-based Computing. Technical report, GigaSpaces Technologies Ltd., 2006. (cited on page 45)

[Sim et al. 2003] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proc. of the 25th Int. Conf. on Software Engineering (ICSE 2003)*, pages 74–83. IEEE Computer Society, 2003. (cited on page 205)

[Simonyi et al. 2006] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 451–464. ACM, 2006. ISBN 1-59593-348-4. (cited on page 72)

[Simos and Anthony 1998] M. Simos and J. Anthony. Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domain Engineering. In IEEE Computer Society, editor, *Proceedings of the 5th International Conference on Software Reuse*, pages 94–102, 1998. (cited on pages 77 and 80)

[Slonneger and Kurtz 1995] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Publishing Company, Inc., 1995. (cited on page 80)

[Spinellis 2001] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001. (cited on page 82)

[Stachowiak 1973] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien - New York, 1973. (cited on page 56)

[Steinberg et al. 2009] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885. (cited on pages 40 and 67)

[Suarez et al. 2011] G. N. Suarez, J. Freund, and M. Schrepfer. Best Practice Guidelines for BPMN 2.0. In L. Fischer, editor, *BPMN 2.0 Handbook*, chapter 2. Future Strategies Inc., Lighthouse Point FL, USA, 2011. ISBN 978-0-9819870-7-1. (cited on page 122)

[Sutter 2005] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. (cited on page 1)

[Syme 2010] D. Syme. *Expert F# 2.0*. Apress, 2010. (cited on page 30)

[Tan et al. 2003] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. *ACM SIGPLAN Notices*, 38(10): 203, October 2003. ISSN 03621340. doi: 10.1145/966049.781532. URL http://portal.acm.org/citation.cfm?doid=966049.781532. (cited on page 242)

[Taylor et al. 1995] N. R. Taylor, W. Tracz, and L. Coglianese. Software Development using Domain-Specific Software Architectures. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 27–37, 1995. (cited on pages 77 and 80)

[van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Bericht 0921, Christian-Albrechts-Universität zu Kiel, November 2009. (cited on pages 224 and 260)

[Viry 2010] P. Viry. Ateji PX for Java - Parallel programming made simple. Technical report, Ateji, June 2010. (cited on page 30)

[Visser 2007] E. Visser. Domain-Specific Language Engineering: A Case Study in Agile DSL Development (Mark I). Technical report, Delft University of Technology, Software Engineering Research Group, 2007. (cited on pages 72 and 77)

[Visser 2008] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008. (cited on page 72)

[Vlissides 1996] J. Vlissides. Generation Gap [software design pattern]. 8 (10):12, 14–18, Nov.-Dec. 1996. ISSN 1040-6042. (cited on page 195)

[Völter 2005] M. Völter. Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development. Technical report, EuroPLoP 2005, 2005. copyright 2005 Markus Völter. Permission is hereby granted to copy and distribute this paper for the purposes of the EuroPLoP 2005 conference. (cited on pages 108 and 109)

[Völter and Stahl 2006] M. Völter and T. Stahl. *Model-Driven Software Development*. Wiley & Sons, 1 edition, May 2006. (cited on pages 29, 55, 56, 58, 68, 69, 70, 72, 73, 98, 101, and 103)

[Voss 2011] M. J. Voss. The Intel Threading Building Blocks Flow Graph, October 2011. URL `http://drdobbs.com/tools/231900177#`. [last visited 11/2011]. (cited on page 32)

[Wang 2006] P. Wang. *Parallel Mercury*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia, October 2006. 37 pages. (cited on page 31)

[Ward 1994] M. P. Ward. Language-Oriented Programming. *Software - Concepts and Tools*, 15(4):147–161, 1994. (cited on page 72)

[Wegner 1996] P. Wegner. Coordination as Comstrainted Interaction (Extended Abstract). In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, First International Conference, COORDINATION '96, Cesena, Italy, April 15-17, 1996, Proceedings*, volume 1061 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 1996. (cited on pages 31 and 40)

[Wegner 1997] P. Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5):80–91, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/253769.253801. (cited on pages 3, 27, 36, and 38)

[Weiss and Lay 1999] D. Weiss and C. T. R. Lay. *Software Product Line Engineering*. Addison-Wesley, 1999. (cited on pages 77 and 80)

[Wells et al. 2004] G. C. Wells, A. G. Chalmers, and P. G. Clayton. Linda implementations in Java for concurrent systems: Research Articles. *Concurr. Comput. : Pract. Exper.*, 16(10):1005–1022, Aug. 2004. ISSN 1532-0626. doi: 10.1002/cpe.v16:10. URL http://dx.doi.org/10.1002/cpe.v16:10. (cited on page 241)

[White 2004] S. A. White. Process Modeling Notations and Workflow Patterns. Technical report, IBM Corp., United States, May 2004. URL http://www.bpmn.org. (cited on page 121)

[Withey 1996] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1996. www.sei.cmu.edu. (cited on pages 66 and 107)

[Wright et al. 2010] H. K. Wright, M. Kim, and D. E. Perry. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 411–414, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0427-6. doi: 10.1145/1882362.1882446. URL http://doi.acm.org/10.1145/1882362.1882446. (cited on pages 238 and 239)