Domain-Specific Modelling for Coordination Engineering with SCOPE

Stefan Gudenkauf, Steffen Kruse
Software Engineering for Business Information Systems,
OFFIS - Institute for Computer Science
stefan.gudenkauf@offis.de, steffen.kruse@offis.de
Wilhelm Hasselbring
Software Engineering Group, Department of Computer Science,
Christian-Albrechts-Universität zu Kiel
wha@informatik.uni-kiel.de

Abstract:
The demand for increasing performance is a continuous trend in computing. Today's multi-core processors and future many-core processors require software developers to exploit concurrency in software as far as possible. To ease the task of developing concurrent software we present our *Coordination-First* approach and the coordination modelling language *SCOPE* that introduces the space-based choreography of processes, which internally orchestrate fine-grained workflow activities. The main contributions are (1) the *Coordination-First* approach that addresses the conformance to higher-level concurrency models in a standardised way by regarding the coordination model of a concurrent program as the first artefact in the software development process using model-driven software engineering techniques, (2) the coordination language *SCOPE* which conforms to the well-known BPMN 2.0 and differentiates between the space-based choreography of multiple concurrent process components and the orchestration of fine-grained activities within a single process component, and (3) the *SCOPE workbench* - an implementation of *SCOPE* based on the Xtext language framework to show the feasibility of our approach.

1 Motivation

Today, multi-core processors are a commodity [Mar07, AMD11, Int11b] and many-cores are on the verge of consumer market introduction [Int11a]. This shift imposes two consequences for software development: Programs need to be concurrent to further exploit performance gains in hardware development [Sut05], and parallel software design must become far easier, more portable, and more standardised to decrease investment risks and to be accepted by software developers in various domains. Current programming languages mostly 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 [Lee06]. Concurrency libraries can be regarded as a loophole, but can turn out to be non-portable and platform-specific. On the other side,

there are clear indications that considering concurrency on higher levels of abstraction yields performance speed-ups that are more significant than speed-ups gained from lower abstraction levels [PJT09]. Eventually, the programming languages themselves have to incorporate higher-level concurrency models 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.

The goal of this work is to address the compliance to higher-level concurrency models by regarding the coordination model of a concurrent program as the first artefact in the software development process, and by providing conformance to the widely-known Business Process Model And Notation 2.0 (BPMN 2.0) specification as a vehicle for dissemination for different stakeholders [ABB+10]. The overall approach is based on combining coordination modelling with model-driven development techniques. The contributions are (1) the *Coordination-First* approach, (2) the high-level coordination language *SCOPE*[1] which conforms to the BPMN 2.0 and differentiates between the space-based choreography of multiple concurrent process components and the orchestration of fine-grained activities within a single process component, and (3) the *SCOPE workbench*, an implementation of *SCOPE* based on the Xtext framework[2] to show the feasibility of our approach.

2 The SCOPE Coordination Model

A higher-level concurrency model can be considered a *coordination model*. A coordination model describes the interaction of active and independent components by defining the coordination laws that specify how the components coordinate themselves through the given coordination media [Cia96]. The work of Wegner suggests that models of coordination have a significant impact on the engineering of complex systems [Weg97]. Today this is shown by coordination-based programming-in-the-large approaches that are employed in industry and academia, see for example [SHG+09]. Most modern coordination models coordinate fine-grained components denoted as *activities* within single composite components denoted as *workflows*. Coordination is specified by control flows that define the order of activities via small pieces of data treated as control information. This kind of coordination modelling is called *orchestration* [Mel07].

Introduced by the Linda model [Gel85], space-based systems (SBS) represent a class of coordination models that employs a data-sharing approach based on so called *spaces* as active data management components. The activity of components is defined by the availability of passive data structures as pre- or postconditions. Consequently, we identify space-based coordination as a kind of *choreography*, since the interaction between black-box components is emphasised prior to the fine-grained coordination of activities within a single component [Mel07]. SBS show appealing features for concurrent programming: (1) SBS assume explicit indirect coordination amongst components. Instead of requiring software developers to cut away unwanted non-determinism by fine-grained synchronisation,

[1] `http://scope-dsl.sourceforge.net/`
[2] `http://www.eclipse.org/Xtext/`

they introduce non-determinism on a higher level of abstraction by the space operations to publish, consume, or read data objects. (2) While residing in spaces, data objects are immutable. To modify a data object, components must explicitly remove it from a 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. (3) SBS ease concurrent programming by abstracting from the location of components in space and time. Components do not have to exist at the same time, and always remain anonymous to each other. (4) SBS abstract from the computational model. They are orthogonal to widespread general purpose programming languages (GPLs) such as C++, C#, and Java, since the latter are based on the computational model in principle.

Workflows and SBS represent abstractions of complex system behaviour that emphasise different aspects of coordination, namely orchestration and choreography. We suggest these to be combined into one single coordination model: SCOPE (Space-based COncurrent Process Engineering). This combination has three fundamental advantages: First, Space-based choreography decouples components in space and time. Second, orchestration separates fine-grained activities of domain-specific computation from the coordination primitives to publish, read, or consume data from spaces. Third, the overall model is significantly higher level than GPL-based implementations of SBS and workflows. It provides a viewpoint on the architecture of a concurrent program that is even understandable by non-programmers. As such, SCOPE can be regarded as a representation of the problem space instead of the solution space.

In [Gud11] we presented a programming library based on SCOPE named PROCOL (PROcess COordination Library) that scales reasonably well on multi-core architectures, and showed that the performance overhead that SCOPE imposes is a reasonable trade-off for the ease of programming provided. Using our PROCOL programming framework, we created an application that computes the Mandelbrot set concurrently and shows it on the complex plane [Gud11]. The application represents a benchmark for massive concurrency since each point of the complex plane can be computed independently. Also, it highlights two kinds of concurrency: concurrent execution of the *same* work, and concurrent execution of *different* work. Figure 1 shows an informal overview of the coordination model of the application represented as a BPMN 2.0 collaboration diagram. The pools represent the participants in the collaboration: ImageSpace represents a *space* and Mandelbrot represents the only *client*. Firstly, the Mandelbrot's process publishes (out) a configuration data object which is constructed from the command line parameters. Secondly, all application-specific sub-processes are started concurrently using a Parallel Sub-Process. These coordinate themselves along the data flows of the space operations. ImageProvider reads the configuration (rd) and constructs and publishes a set of unrendered image slices (outg). These are consumed (in) by as many concurrent Renderer instances (a multiple-instance sub-process) 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 reads the configuration (rd), consumes the rendered image slices (looping in), puts them together to a single image, and saves it into a file. As shown in [Gud11], the application scales well on different multi-core machines.

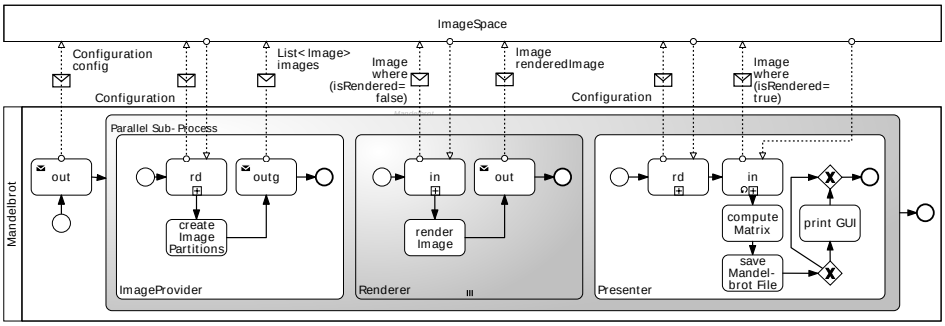

Figure 1: Coordination model of the space-based Mandelbrot application; concurrent components are grey-shaded.

3 The Coordination-First Approach

Coordination-First consists of the development of a coordination model based on SCOPE in an early phase of the development process, see Figure 2. Firstly, *scoping* describes the parallel software system and its environment informally. The phase encourages the use of architectural patterns. In the case of the Mandelbrot application, the applied pattern is the master-worker pattern. Secondly, *collaboration* specifies concurrent components that collaborate in a program based on the initial problem specification (the BPMN participant pools in Figure 1). Thirdly, *process definition* defines the processes that implement the behaviour of each participating component (the sub-processes in Figure 1). 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. The resulting model represents the specification of the concurrent program. The approach can be regarded as a refinement of the *coordination* design phase in the parallel software design method described by Ortega-Arjona [OA10]. Thereby, Coordination-First puts the emphasis on *synthesis*: It provides a "well-defined structure for the software system [...] described in terms of software components in sufficient detail to support the required partition of algorithm and/or data and to enable an analysis of its performance and cost properties." [OA10, p.318]. We omitted *documentation* as a separate phase since we consider it as a cross-cutting activity that builds up across all phases in the choreography-first approach.

Since the overall approach is oriented towards defining the expected behaviour of a software system, we prospect *Coordination-First* as a basis for *architecture-centric model-driven software development* (AC-MDSD) [VS06]. The benefits are (1) Knowledge capture: SCOPE models provide a basis for communication between software architecture modellers, transformation developers, and programmers of application domain-specific logic. (2) Reuse and portability: Reference models and transformations can be reused, providing a basis for software product families. Different target platform transformation sets can also be applied to the same coordination model. Reusing models and transforma-

tions can also save development time. (3) 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 generated source code. (4) Information hiding: Transformations can encapsulate platform-specific coordination implementation, thus relieving software architecture modellers and application domain-specific programmers.

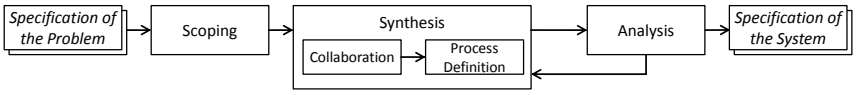

Figure 2: Coordination-first approach.

Table 1 subsumes the requirements that we considered for a realisation of Coordination-First. To do so, we suggest BPMN 2.0 as a "host" language for SCOPE (requirement **R1**). It provides three benefits for Coordination-First: (1) BPMN 2.0 is well-known by a variety of roles in the IT industry and is widely supported by tools. It provides a promising vehicle for the rapid dissemination and acceptance of the proposed approach. (2) BPMN provides a comprehensive graphical outline view on behavioural architecture models, even across non-technical stakeholders. (3) BPMN 2.0 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.

Unfortunately, BPMN 2.0 also has some drawbacks: Firstly, the BPMN 2.0 metamodel defines over 150 entities. Industry practice recommends to identify and use only a meaningful subset of BPMN elements to cope with this complexity [SFS11]. However, subset adherence is in the responsibility of the modeller and typically not supported by BPMN tools. Secondly, SBS-specific aspects can make it necessary to extend BPMN 2.0. Although BPMN 2.0 provides an explicit extension mechanism [ABB$^+$10, pp.57-61], this mechanism has several drawbacks: (1) domain-specific validation is not considered (requirement **R5**), (2) custom graphical notation elements mapped to extensions are not guaranteed to be handled uniformly in tool chains (requirement **R4**), and (3), the extensions themselves are not guaranteed to carry over in BPMN tool chains by definition [ABB$^+$10, p.57] (requirement **R2** and **R3**). Finally, since the BPMN 2.0 specification defines the operational semantics informally, a formal specification of SCOPE that conforms to the BPMN 2.0 specification is still desirable.

As a consequence, we suggest an external Domain-Specific Language (DSL) for SCOPE and subsequent model transformations. First, an external DSL based on an explicit BPMN 2.0 sub-metamodel guarantees BPMN subset adherence by nature. Also, it can support metamodel and notation extensions directly, and can provide elaborate domain-specific validation. Second, interoperability can be provided by a model-to-model transformation that preserves extensions at least as BPMN annotations which can be handled uniformly across tool chains. Iterative development processes can be supported by model transformations (like QVT [CCD$^+$11]) that use trace information of subsequent transformation executions to provide update behaviour. This means that only model changes are carried

over (updated) with each transformation run. Third, the BPMN 2.0 specification defines its operational semantics in terms of token flow. A model transformation that maps SCOPE models to token flow-based formalisms such as Petri nets can provide an evident way to provide a BPMN-conformant semantics definition.

Key	*Requirement*
R1	BPMN 2.0 conformance: *SCOPE* implementations should conform to the BPMN 2.0 specification in order to benefit from its wide use and tool support.
R2	BPMN 2.0 interoperability: *SCOPE* implementations should be interoperable with BPMN 2.0 tools in order to establish seamless development tool chains.
R3	Iterative development process: *SCOPE* implementations should support an iterative development process to reduce cascading change effects when existing SCOPE models are manipulated.
R4	Usability: *SCOPE* implementations should be as comfortable as possible and meet current expectations to contemporary tooling.
R5	Validation: *SCOPE* implementations should support the structural and domain-specific validation of *SCOPE* models. Errors and warnings should be indicated instantaneously and corrected automatically, if possible.

Table 1: SCOPE implementation requirements.

4 Development of the SCOPE DSL

To realise a DSL for SCOPE, we employed a development process that is based on the phase model of Mernik et al. [MHS05]. Figure 3 shows an overview of the activities taken and their results. Regarding the question *when* a DSL is developed, we consider the rationale given in Section 1 as a positive answer to the necessity assessment.

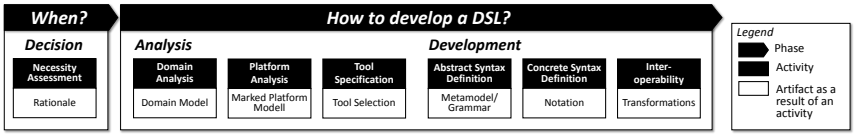

Figure 3: SCOPE DSL development process.

4.1 Analysis

First, we conducted a *domain analysis* to answer the question of how to develop the DSL. The SCOPE model from Section 2 represents the resulting domain model. Second, we conducted a platform analysis that delivered an overview of the relevant BPMN 2.0 metamodel fragments. We considered a minimal subset of BPMN 2.0 notation elements to model space-based concurrent systems, and identified the related metamodel fragments (marked platform model) in the BPMN 2.0 metamodel. These form the basis for the design of the abstract syntax of the SCOPE DSL (requirement **R1**) as well as for the model

transformations that are required for BPMN 2.0 interoperability (requirement **R2**). Finally, we conducted the *tool specification*. We selected the Eclipse Xtext framework to implement the SCOPE DSL since it supports the development of language infrastructure for textual languages, including compilers and interpreters, domain-specific validation and quick fixes based on the Extended Backus-Naur Form-like (EBNF) grammar language Xtext. In the following, we give a rationale for the elements of the BPMN subset. For the sake of brevity, we do not discuss the given conformity of the selected elements to the informal operational semantics of the BPMN 2.0 specification. We reference pages instead, so that one can re-check conformity if desired.

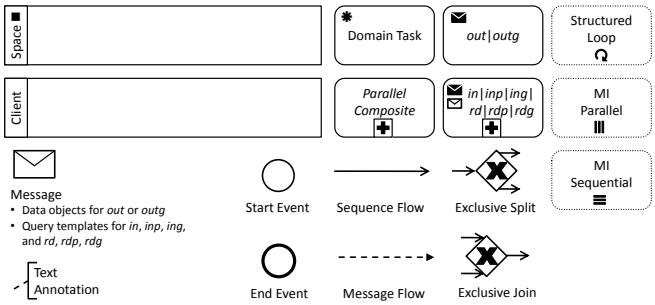

Figure 4: BPMN subset.

Spaces: In SBS, the operational semantics of a space is determined by its implementation. Spaces are considered to be black-box components in SCOPE models. Spaces can be represented as BPMN Participants. Participants itself represent concrete entities or roles that interact with each other and can execute individual processes [ABB+10, p.114]. The use of BPMN Participants for spaces has several benefits: (1) There is no constraint on the locality of Participants. 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. (2) BPMN Participants do not necessarily have to define their processRef property [ABB+10, p.116]. Also, they are represented graphically by (black-box) Pools that can be defined without a Process [ABB+10, p.114]. As a consequence, the behaviour of space Participants does not have to be explicitly modelled in BPMN diagrams. (3) Data objects can be represented as messages that are exchanged between client processes and spaces. This lets us consider the message exchange protocol between clients and spaces as BPMN 2.0 Choreographies. We advise to indicate space Pools with a black-box marker in its upper left corner to emphasise their black-box execution semantics.

Client processes and intra-process activities: As a consequence of modelling spaces as Participants, client processes are also modelled as Participants. A natural candidate for activities that embody domain-specific logic is the BPMN Task element: It represents an atomic activity that cannot be broken down to a finer level of detail [ABB+10, p.156]. We introduce the Domain Task as a concrete Task type that represents atomic domain-specific logic. The Domain Task can be indicated by an asterisk marker in its upper left corner.

Different concurrent activities 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. [...] This usage of expanded Sub-Processes for 'parallel boxes' is the motivation for having Start and End Events being optional objects." [ABB+10, p.174]. The concurrent execution of the *same* activity can be modelled with the Multiple Instance Loop Characteristics [ABB+10, p.432]. Thereby, the number of instances to be generated is either specified by its feature loopCardinality or as the cardinality of a collection data item (property) of the respective activity.

Space coordination primitives: Publishing a data object to a space can be modelled using a Send Task. 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). 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 a matching data object can be received. Also, it must be possible to differentiate between the blocking space coordination primitives *in* (consume) and *rd* (read) and their non-blocking siblings *inp* (non-blocking), *ing* (non-blocking group), rdp, and rdg. 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 5). 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.

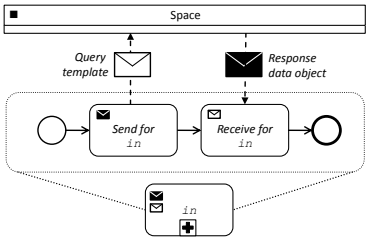

Figure 5: BPMN space query idiom.

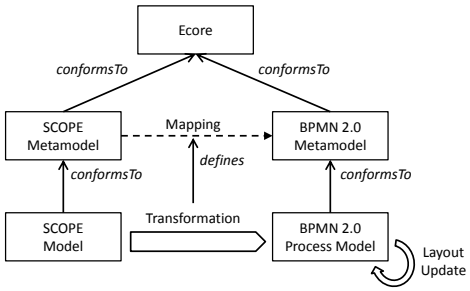

Figure 6: Interoperability by QVT transformation.

Data objects and inter-process data exchange: Much of the benefits of the SCOPE model relies 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 of the latter is bound to the lifecycle of their parent Process or Sub-Process [ABB+10, p.207].

Instead, Messages are a natural choice to represent SBS data objects [ABB+10, 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 [ABB+10, p.95]. Also, Messages can be used in a BPMN Choreography [ABB+10, p.93]. The BPMN standard differentiates between initiating and non-initiating Messages. Since we cannot guess the responses from space Participants, we regard only initiating Messages in SCOPE. These are Messages that represent query templates for the *in*, *inp*, *ing*, *rd*, *rdp*, and *rdg* coordination primitives, and Messages that represent data objects for the *out* and *outg* space coordination primitives. Inter-process data exchange is represented as Message Flows [ABB+10, p.120].

Intra process control flows: We use the BPMN Start Event to model the instantiation of a client process [ABB+10, p.238]. Analogously, we use the BPMN End Event to model the termination of a client process. In contrast to the BPMN standard, we restrict processes to only allow one single Start Event and one single End Event within a Process. [ABB+10, pp.246,426]. Control flows are represented in BPMN in terms of Sequence Flows [ABB+10, p.97]. In the BPMN subset, we use Sequence Flows to model the flow of control between Start 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: (1) Do not use arbitrary control flow loops. (2) Do not use Gateways to control concurrency. Instead use Sub Processes and Loop Activities. Structured loops in the control flow of a Process can be modelled with conventional Standard Loop Characteristics [ABB+10, p.432]. Thereby, its testBefore feature determines if the loopCondition is evaluated before (true) or after (false) the Activity is executed. loopMaximum determines the maximum number of executions, including unbounded. For structured loops in the BPMN subset, we assume that the testBefore attribute is always true. Control flow decisions can be represented in the BPMN by Exclusive Gateways, Inclusive Gateways, Parallel Gateways, Event-based Gateways or Complex Gateways. From these we only consider the Exclusive Gateway in the BPMN subset since its semantics are uncomplicated [ABB+10, 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 [CCH10] is executed. We also refrain from using Event-based Gateways since the subset currently does not consider Events except for Start and End Events. 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 [ABB+10, p.437].

4.2 Implementation

Abstract syntax: The grammar of a language is defined by its abstract syntax. The abstract syntax can be represented by a metamodel. Figure 7 presents an overview of the SCOPE metamodel that was developed based on the results of the analysis phase. The actual metamodel development can be considered as *language piggybacking*: Domain-specific elements were added to parts of a host language (the previously identified BPMN

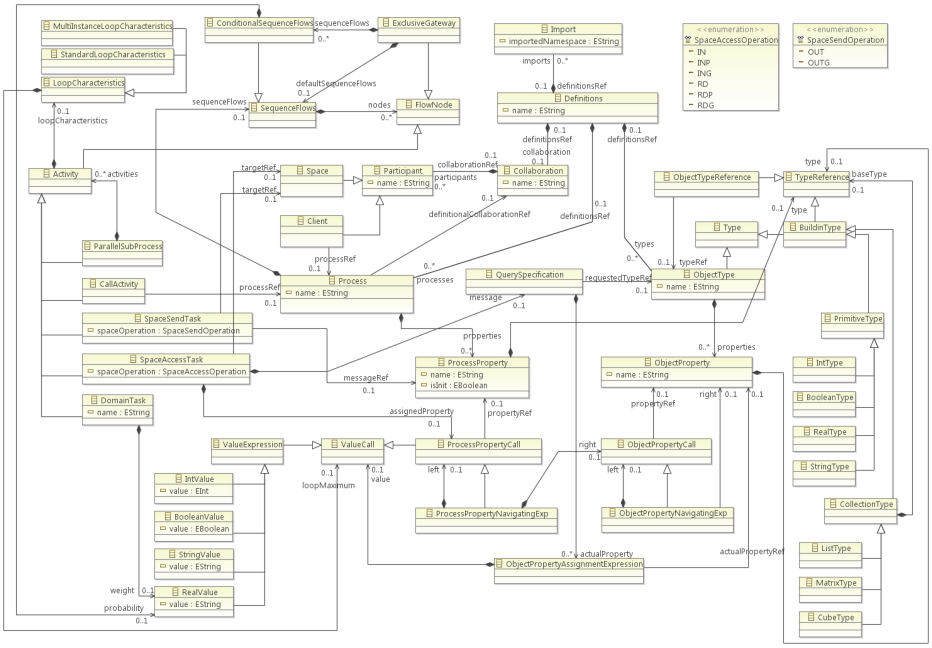

Figure 7: SCOPE metamodel.

2.0 subset) [MHS05].

First, we discuss which 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 [ABB+10, p.51]. The Import class is used to reference elements contained in other Definitions [ABB+10, 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 ProcessProperties) 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 nodeRefs property of the Sequence Flows or Conditional Sequence Flows elements. Start 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 instantiation. The Call Activity can be used to invoke other Processes (considered as sub-processes in the BPMN subset) from the current one.

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 the subsequent *analysis* phase (see Sec. 3). 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. The Space Access Task represents the BPMN 2.0 idiom for consuming or reading data objects from spaces (see Sec. 4.1). 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 Value Expressions to the Object Properties of the requested data object. Finally we model the conditions of Conditional Sequence Flows as probabilities using Real Values instead of providing a complete expression system. It 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 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 SCOPE was developed on basis of its abstract syntax metamodel. We selected a textual notation as the primary concrete syntax. The following code listing illustrates the implementation of the Mandelbrot example from Section 2 in SCOPE. Firstly, the collaboration defines the interacting participants. Secondly, the individual processes are defined. Therefore, the model addresses the phases *collaboration* and *process definition* in the Coordination-First approach. MandelbrotImpl represents the executable main process as it calls all other processes and is referenced by the Mandelbrot client. Domain Tasks are initially weighted equally. The notation employs an SQL-like rendering for the Object Property Assignment Expressions: “where”-clauses greatly simplify the specification of templates to be sent as queries to spaces (see, for example, lines 10 and 22 in the listing).

```
definitions mandelbrot.processes {
  import mandelbrot.types.*
  import mandelbrot.types.Configuration.*
  collaboration Mandelbrot {
    space ImageSpace
    client MandelbrotClient : MandelbrotProc }
  process ImageProvider attends Mandelbrot {
    Configuration config init;
    list<Image> images;
    config = read Configuration from Mandelbrot.ImageSpace;
    createImagePartitions weighted 1.0;
    publish-group 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 = consume 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 = loop(config.numberOfImagePartitions) consume Image
      where (Image.isRendered = true) from Mandelbrot.ImageSpace;
    computeMatrix  weighted 1.0;
    saveMandelbrotImageAsFile  weighted 1.0;
    xor {
      case 0.7 : printGui  weighted 1.0; } } }
definitions mandelbrot.types { <...> }
```

In contrast to a graphical notation, the textual notation supports the iterative collaboration among stakeholders (requirement **R3**) 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 notations profit from broad tool support (requirement **R4**). This is in particular the case with version management systems. Figure 8 illustrates a part of the Mandelbrot scenario within the SCOPE workbench that was developed using the Xtext Framework. To increase user acceptance we put particular effort into domain-specific validation and appropriate IDE-supported quick fixes (see the error message at the bottom and the quick fix pop-up to the left, requirement **R5**). The domain-specific outline view (right) and the dynamically searchable quick outline (bottom) provide additional user support (requirement **R4**).

Interoperability between the SCOPE workbench and third party tools that conform to the BPMN 2.0 specification is guaranteed by a *Query View Transformation* (QVT) transformation (requirement **R2**). QVT is a standard of the Object Management Group for model-to-model transformations [CCD+11]. We chose medini QVT[3] to execute the transformation since it is integrated in the Eclipse IDE environment and since it supports both the BPMN 2.0 metamodel as well as the SCOPE metamodel using EMF Ecore, see Figure 6. The transformation rules map the SCOPE metamodel to the complete BPMN 2.0 metamodel. Graphical BPMN models require additional layout information. These are generated after the transformation. We used 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. Only changes to the 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 (requirement **R3**). Overall, the QVT transformation maps all SCOPE elements and their relations to the corresponding BPMN elements and establishes conformance of SCOPE to BPMN 2.0

[3] http://projects.ikv.de/qvt

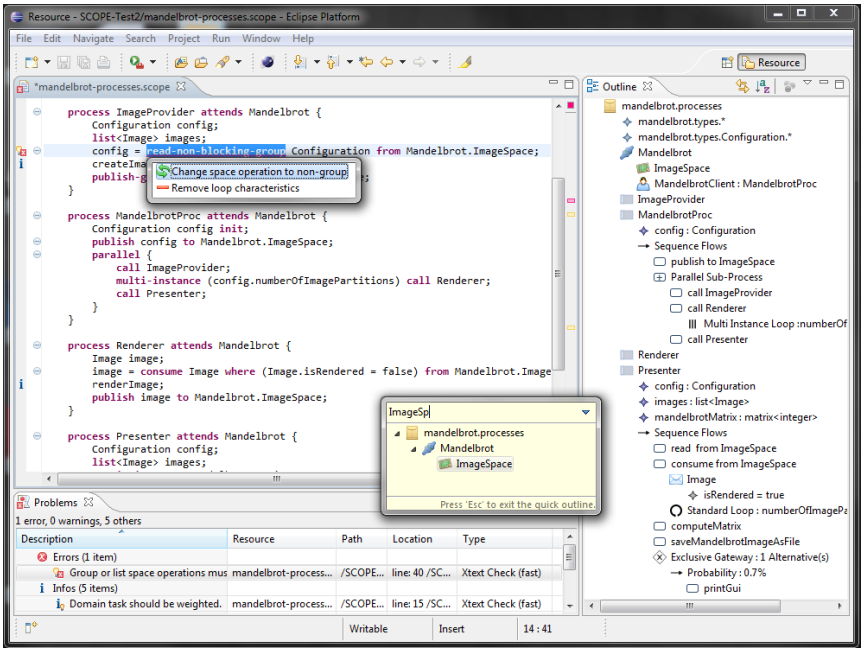

Figure 8: SCOPE workbench for coordination engineering.

(requirement **R1** and **R2**).

Regarding an explicit operational semantics definition of SCOPE, we already identified a model transformation that maps SCOPE models to Petri nets as a way to provide a BPMN-conformant semantics definition. We consider such transformation as future work. Also, we currently develop a transformational mapping of SCOPE to the PROCOL programming framework, since both are based on the SCOPE model abstraction. Finally, we will address evaluation by considering practical application scenarios developed with the Coordination-First approach and the SCOPE DSL.

5 Related Work

Mernik et al. describe patterns for the analysis, design, and implementation of a DSL [MHS05]. These comprise design patterns to exploit an existing host language. The authors denote the exploitation of parts of a host language, which are subsequently extended by domain-specific elements, as *piggybacking*. Because we used parts of the BPMN 2.0 specification and extended them by elements specific to SBS, our SCOPE DSL can be categorized as language piggybacked.

There are many examples for the extension of the BPMN for domain-specific aspects. For

instance, Awad et al. describe how BPMN 1.1 process diagrams can be extended by constraints for resource assignment [AGMW09]. 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. developed their own metamodel based on BPMN 1.0 process diagrams, and extended it by security requirements aspects [RFMP07]. 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. describe an extension of the BPMN 2.0 metamodel [SLSW10]. They use the extension mechanism of the BPMN 2.0 specification to add so-called *compliance scopes* to process diagrams. In contrast to the piggybacking approach, the approach is limited in interoperability. The BPMN 2.0 extension mechanism and custom graphical notations for extensions are not widely and uniformly supported across tool chains, and means for higher-level validation of BPMN models extended by domain-specific elements are often missing.

Regarding our prospects of using SCOPE models for AC-MDSD, there are few comparable approaches. Tan et al. present an infrastructure to use design patterns to generate parallel code for distributed and shared memory environments [TSS+03]. 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. present an approach of model-driven development of multi-core embedded software [HLC+09]. 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 library [Rei07], 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. Coordination-First is instead applicable to different target platforms. In [PBM+09], Pllana et al. 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. 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 also intent Choreography-First and SCOPE to be applicable to distributed system development by providing appropriate model-to-platform transformations.

6 Conclusion

In this paper we introduced the Coordination-First approach. 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 the space-based coordination language SCOPE, and realised a workbench for SCOPE. Both, the language and the workbench, conform to the BPMN 2.0 specification and are interoperable with BPMN 2.0-

conformant tools. Additionally, SCOPE models can be validated on domain-level rather than only on the level of the underlying BPMN 2.0, and support iterative software development processes.

References

[ABB+10] 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.

[AGMW09] Ahmed Awad, Alexander Grosskopf, Andreas Meyer, and Mathias Weske. Enabling Resource Assignment Constraints in BPMN. BPT Technical Report 04-2009, Ahmed Awad, Alexander Grosskopf, Andreas Meyer, Mathias Weske, 2009.

[AMD11] AMD. Multi-Core Processing with AMD. http://www.amd.com/us/products/technologies/multi-core-processing/Pages/multi-core-processing.aspx, August 2011.

[CCD+11] Codagen Technologies Corp, Compuware, DSTC, France Telecom, IBM, INRIA, Interactive Objects, Kings College London, Object Management Group, Softeam, Sun Microsystems, Tata Consultancy Services, Thales, TNI-Valiosys, University of Paris VI, and University of York. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1. Technical report, Object Management Group, Inc. (OMG), 2011.

[CCH10] David Raymond Christiansen, Marco Carbone, and Thomas 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.

[Cia96] Paolo Ciancarini. Coordination Models and Languages as Software Integrators. *ACM Computing Surveys*, 28(2):300–302, 1996.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[Gud11] Stefan Gudenkauf. Space-Based Multi-Core Programming in Java. In Christian Wimmer and Christian W. Probst, editors, *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, 41–50, New York, 2011. The Association for Computing Machinery, Inc.

[HLC+09] Pao-Ann Hsiung, Shang-Wei Lin, Yean-Ru Chen, Nien-Lin Hsueh, Chih-Hung Chang, Chih-Hsiong Shih, Chorng-Shiuh Koong, Chao-Sheng Lin, Chun-Hsien Lu, Sheng-Ya Tong, Wan-Ting Su, and William C. Chu. Model-Driven Development of Multi-Core Embedded Software. *Multicore Software Engineering, ICSE Workshop on*, 0:9–16, 2009.

[Int11a] Intel. Intel Many Integrated Core Architecture. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html, August 2011.

[Int11b] Intel. Parallel Programming. Multicore processors TODAY, many-core co-processors READY. Technical report, Intel, 2011.

[Lee06] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[Mar07] Ami Marowka. Parallel Computing on any Desktop. *Commun. ACM*, 50(9):74–78, 2007.

[Mel07] Ingo Melzer. *Service-orientierte Architekturen mit Web Services*. Elsevier, München, 2nd edition, 2007.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[OA10] Jorge Luis Ortega-Arjona. *Patterns for Parallel Software Design*. Wiley Series in Software Design Patterns. John Wiley and Sons, Ltd., Chichester, West Sussex, UK, 2010.

[PBM+09] Sabri Pllana, Siegfried Benkner, Eduard Mehofer, Lasse Natvig, and Fatos Xhafa. Towards an Intelligent Environment for Programming Multi-core Computing Systems. In César Eduardo, Michael Alexander, Achim Streit, Jesper Larsson Träff, Christophe Cérin, Andreas Knüpfer, Dieter Kranzlmüller, and Jha Shantenu, editors, *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415, pages 141–151, Berlin / Heidelberg, 2009. Springer.

[PJT09] Victor Pankratius, Ali Jannesari, and Walter F. Tichy. Parallelizing Bzip2: A Case Study in Multicore Software Engineering. *IEEE Software*, 26(6):70–77, November 2009. Download: http://dx.doi.org/10.1109/MS.2009.183.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[RFMP07] Alfonso Rodríguez, Eduardo Fernández-Medina, and Mario 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.

[SFS11] Gerardo Navarro Suarez, Jakob Freund, and Matthias Schrepfer. Best Practice Guidelines for BPMN 2.0. In Layna Fischer, editor, *BPMN 2.0 Handbook*, chapter 2. Future Strategies Inc., Lighthouse Point FL, USA, 2011.

[SHG+09] Guido Scherp, André Höing, Stefan Gudenkauf, Wilhelm Hasselbring, and Odej 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.

[SLSW10] Daniel Schleicher, Frank Leymann, David Schumm, and Monika 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.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[TSS+03] Kai Tan, Duane Szafron, Jonathan Schaeffer, John Anvik, and Steve MacDonald. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. *ACM SIGPLAN Notices*, 38(10):203, October 2003.

[VS06] Markus Völter and Thomas Stahl. *Model-Driven Software Development*. Wiley & Sons, 1 edition, May 2006.

[Weg97] Peter Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5):80–91, 1997.