MASTER

VDSEIR - a graphical layer on top of the Octopus toolset

in 't Groen, A.M.

Award date:
2011

VDSEIR - A graphical layer on top of the Octopus toolset

A.M. in 't Groen

July 2011

VDSEIR - a graphical layer on top of the Octopus toolset

A.M. in 't Groen

July 18, 2011

Abstract

This paper introduces a graphical layer on top of the Octopus toolset. This Octopus toolset is a toolset that aims to ease and speed-up the process of designing embedded systems, by combining the power of several existing analysis tools, including CPNTools and Uppaal. The toolset uses the domain-specific language called "Design Space Exploration Intermediate Representation" (DSEIR), inspired by the Y-chart approach (application, platform, and mapping). Specifications in this language can be translated into the formalisms used by the analysis tools. Currently, the language and the toolset itself are implemented as a pure Java API. If users want to make specifications in this domain language they need to be able to program in Java and they need to know the API. In addition, if the users want to address any of the toolset's functionality, they need to know its architecture.

This paper introduces a graphical representation for DSEIR, called Visual DSEIR (VDSEIR). By using VDSEIR, users of the toolset can create specifications in DSEIR by means of creating graphical models, removing the need for those users to know how to program in the Octopus API. Of course, a method is needed to convert a VDSEIR model to a DSEIR model, because otherwise the new graphical representation has no relation with DSEIR. Hence, this paper provides a model transformation from VDSEIR to DSEIR that makes use of an intermediate generator model and a parser that is automatically generated from an annotated JavaCC grammar.

The graphical representation for DSEIR consists of several perspectives and it contains a special form of syntactic sugar, namely hierarchy. It is possible to create hierarchical models in the graphical representation without having support for hierarchy in the original DSEIR language, because these hierarchical models can be translated into non-hierarchical DSEIR models. This way, additional expressiveness is created for the user, without modifying the underlying toolset.

To allow the users of the toolset to create models using this graphical representation, an approach is given to create a graphical editor for VDSEIR with the Eclipse Graphical Modeling Framework (GMF). This approach allows source code for this graphical editor to be automatically generated from a set of models, including meta-models for VDSEIR. In order to get some advanced features in the generated editor, some customizations are made to the GMF generation process, while keeping the generative nature intact.

To let users interact with the toolset, a framework is introduced in which the users can design graphical experiments to create and analyze DSEIR models. This framework is extensible by plug-ins, allowing future contributions to the graphical editor. To allow some core functionality, a base set of plug-ins is already described in this paper.

Concludingly, the graphical editor for VDSEIR in combination with the extensible framework provides a graphical layer on top of the Octopus toolset in which users can create and analyze (hierarchical) DSEIR specifications without any knowledge of the underlying toolset's API.

Contents

1 Introduction

Context
The Octopus toolset ([3]) is a toolset that tries to ease and speed-up the process of designing embedded systems. The following two paragraphs are taken from the Octopus toolset's website and illustrate the goals and motivation of the toolset:

"Today's embedded systems are rapidly becoming more complex. A systematic model-driven design trajectory is needed to provide high-quality, cost-effective systems. The Octopus toolset provides support to model, analyze and select appropriate design alternatives in the early phases of product development."

"An important challenge in the early stages of the design of embedded systems is the many design possibilities that need to be considered. The design spaces usually involve multiple metrics of interest (timing, resource usage, energy usage, cost, etc.) and multiple design parameters (e.g. the number and type of processing cores, sizes and organization of memories, interconnect, scheduling and arbitration policies, etc.). The relation between design choices on the one hand and metrics of interest on the other hand is often very difficult to establish, due to aspects such as concurrency, dynamic application behaviour, and resource sharing. No single modeling approach or analysis tool is fit to cope with all the challenges of modern embedded-system design."

The Octopus toolset aims to leverage existing modeling, analysis, and Design Space Exploration (DSE) tools to support model-driven DSE for embedded systems. This is done using the 'Y-chart' approach, visible in Figure 1.

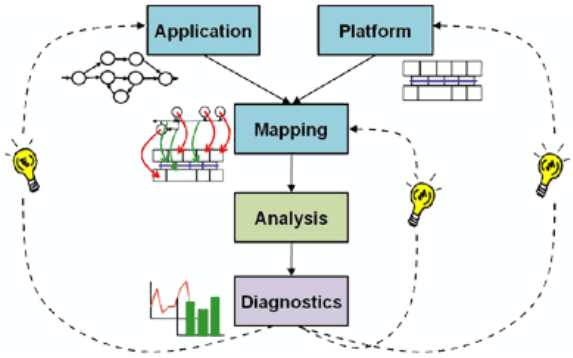


Figure 1: The Y-chart approach; a design pattern on which the Octopus toolset has been built

When designing a system with this approach, three parts of that system have to be specified: an application, a platform, and the mapping of the application onto the platform. Diagnostic information of this system is used to (semi-automatically) improve application, platform, and/or mapping.

Current situation
The Octopus toolset makes use of a language called 'Design Space Exploration Intermediate Representation' (DSEIR). Users of the toolset design their application, platform, and mapping of the application onto the platform in this language (conform the Y-chart). The DSEIR

```
DseirModel dseir = DseirModelFactory.createModel("Example");
Application app = dseir.getApplication();
Platform platform = dseir.getPlatform();
Mapping mapping = dseir.getMapping();

Task taskA = app.createTask("A");
Variable pageSizeVarA = taskA.addParVariable("page_sizeA", ARRAY2);
Port inA = taskA.createPort("page_size", TRUE, cnst(pageSizeVarA), array(ONE, dist(sizeDist)),
    Port.Order.FIFO);

Resource m1 = platform.createResource("M1", 100);
m1.setProcTime(ServiceType.INTERNAL_STORAGE, ZERO);
m1.setProcTime(ServiceType.RESULT_STORAGE, ZERO);
```

Figure 2: (Part of) an example DSEIR specification; knowledge of the toolset's API as well as knowledge of Java is required to create this specification.

language is implemented as a Java library; the supported services are implemented as classes and methods and the Java language itself can be used to define and run experiments. Figure 2 shows a code snipped of a specification in the DSEIR language. We can see that elements have to be created explicitly via a function call (for example `app.createTask("A")`).

The DSEIR model forms the basic input for the Octopus toolset. With it, the toolset can execute different kinds of analysis. The analysis techniques are not directly applied in the toolset, but the analysis is performed by external tools, such as CPN Tools and Uppaal. In order to do that, the Octopus toolset houses translations from DSEIR specifications to the domain languages of these external tools. Figure 3 shows the current architecture of the toolset, where we can see the external tools and their domain language translations within the DSEIR library.

The purpose of DSEIR is to provide an abstraction from the domain languages of the external tools used by Octopus. It also aims at providing some useful modeling constructs that will allow modelers to express things that can be hard to implement or maintain in these languages, for example resources together with their allocation to tasks. These extra modeling constructs will not allow modelers to express things that cannot be expressed in the domain languages of the external tools, so they can be considered syntactic sugar.

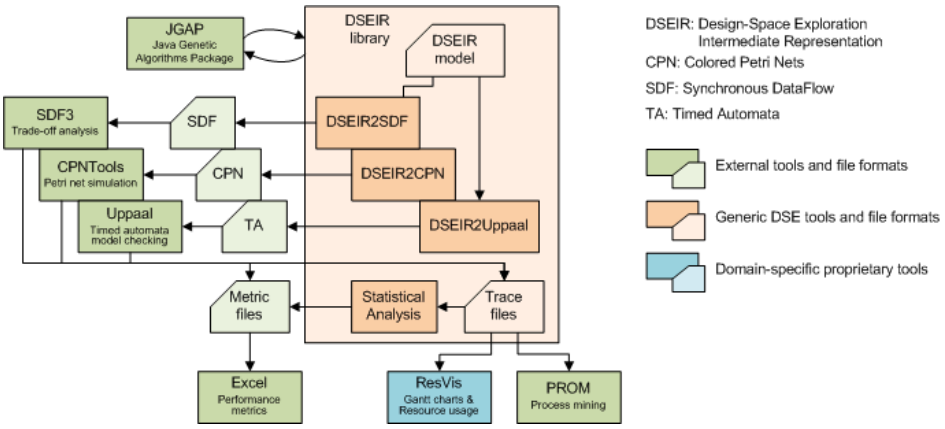


Figure 3: Current toolset architecture; we can see that the toolset addresses different existing tools such as CPN Tools and Uppaal

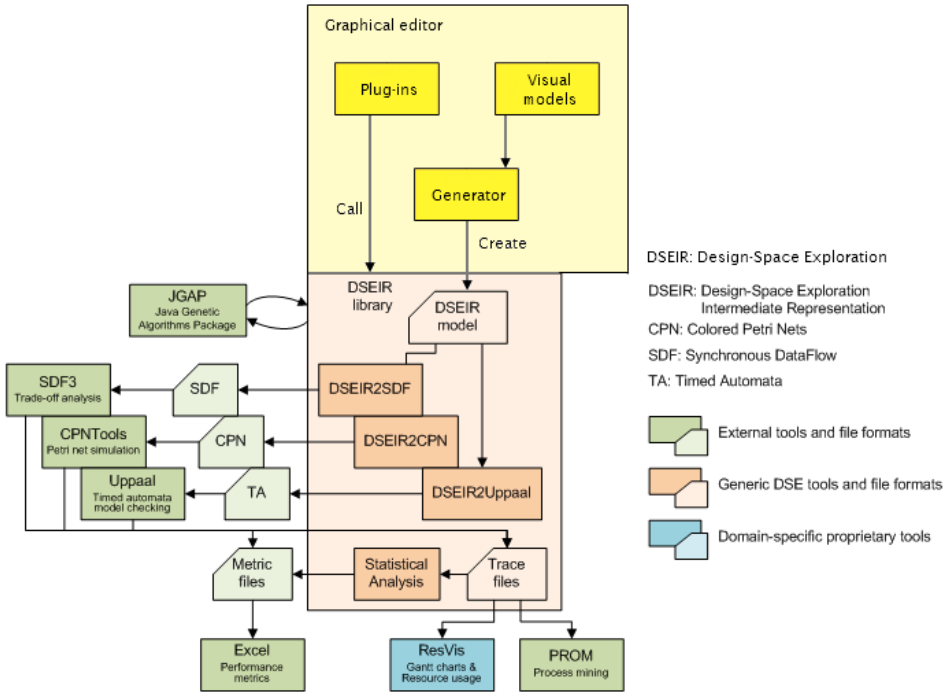


Figure 4: The toolset architecture with the graphical layer on top of it

Problem statement
There are three major issues with the DSEIR language and implementation. The first issue is that *the Java implementation is not suitable for modeling.* Not all potential users of the toolset know how to use Java and even if the users know how to specify DSEIR models in Java, this is not very user friendly and not easy to read, maintain, etc. Therefore *a visual representation of the language is needed* together with a translation from the visual representation to the DSEIR language, so that users of the toolset can create specifications in the DSEIR language in a graphical editor. In this way, the underlying Java implementation is hidden from the users.

The second issue with the DSEIR language is that the language *does not support any hierarchal constructions.* This means that DSEIR models have to incorporate all levels of detail in one place. Hence, a single model's application specification contains the high level workflow together with low level implementation details. Furthermore, the DSEIR language does not allow its users to reuse and combine small parts or building blocks they modeled. To overcome these issues and to make the DSEIR language more readable and maintainable, *its visual representation needs to incorporate hierarchy*, on the application part of the Y-chart.

The final issue is that users of the current implementation of the toolset also have to use Java programming to apply the analysis techniques provided by the toolset. *The user should be able to perform the analysis right from the visual representation.* Therefore, a framework needs to be created in which the user can perform any analysis he or she wants with any specification in the visual representation of the DSEIR language.

General approach
Because the Octopus toolset is still under development, *it is highly likely that the DSEIR language will undergo some changes* that will occur during and after the development of the graphical editor. These changes can possibly influence the visual representation of DSEIR and the translation from this visual representation to the DSEIR language itself. These changes might also influence the set of analysis techniques that are supported by the toolset, mainly by adding extra capabilities. With maintainability in mind, *an approach is chosen that will be robust with respect to these possible changes.*

This robust approach comes in the form of a model-driven approach. Models will be used to specify the visual representation as well as the graphical editor, because models are much easier to create, maintain, and adapt than source code. The language translations will be realized using model-to-model transformations where possible (because the DSEIR language is implemented as a Java library, it is not possible to apply model-to-model transformation techniques to translate any (visual) model into a DSEIR model).

To cope with the changing set of analysis techniques supported by the toolset, *an extensible framework will be designed and implemented.* This framework will allow users of the graphical editor to execute analysis on the DSEIR models. The framework will be able to load so called *plug-in*s, pieces of software that can be added to the finished graphical editor to extend its capabilities. Each of these plug-ins will provide a certain functionality to the user, comparable to the plug-ins in the ProM toolset ([7]). They will have a certain predefined input on which they act (for example a DSEIR model) and they have a predefined output they produce, for example an event trace, a resource occupancy, a modified DSEIR model, a file, etc. Using

this extensible framework, *the graphical editor can easily be changed and improved after its deployment*, again providing a robust solution.

The tool in which everything will be developed is chosen to be the Eclipse Graphical Modeling Project (GMP). This tool is picked, because it is built according to the concept of model-driven approach and it includes a lot of ready-to-use tools for model-driven engineering.

Eclipse GMP includes the Graphical Modeling Framework (GMF). This framework allows a fully-functional graphical editor to be automatically generated out of a meta-model describing the domain language and a set of models describing how the graphical editor should behave. This implies the following property:

> *When the DSEIR language changes, only its visual representation and possibly the models describing the behavior of the graphical editor have to be updated. After these updates, a new graphical editor can be automatically regenerated.*

Because of this property, the GMF framework provides a robust method to create a graphical editor.

Next to the GMF, the Eclipse GMP houses the Eclipse Modeling Framework (EMF). This framework allows its users to easily create meta-models. It also houses tools to generate code for these meta-models, that allows model instances of these meta-models to be created.

In addition to GMF and EMF, Eclipse GMP provides a model transformation engine conforming to the Query/View/Transformation (QVT) standard. This engine makes it possible to easily apply model-to-model transformations.

In [4], a lot of information has been written about the design of domain-specific languages and the tools in GMP to support this.

Step-wise approach
The approach is given by the following steps.

1. First, a visual representation without any hierarchy will be made and the GMF framework will be used to to create a graphical editor for the representation.
2. The transformation from the visual representation to DSEIR will then be designed and implemented.
3. Third, the visual representation will be improved and extended with hierarchy on the application part of the Y-chart.
4. After that, an extensible framework is developed for the graphical editor that allows its users to perform different analysis.
5. Finally, a mechanism is made to report errors to the end-users, originating in the models they create and in the way they use the framework.

This approach can be seen in Figure 5.

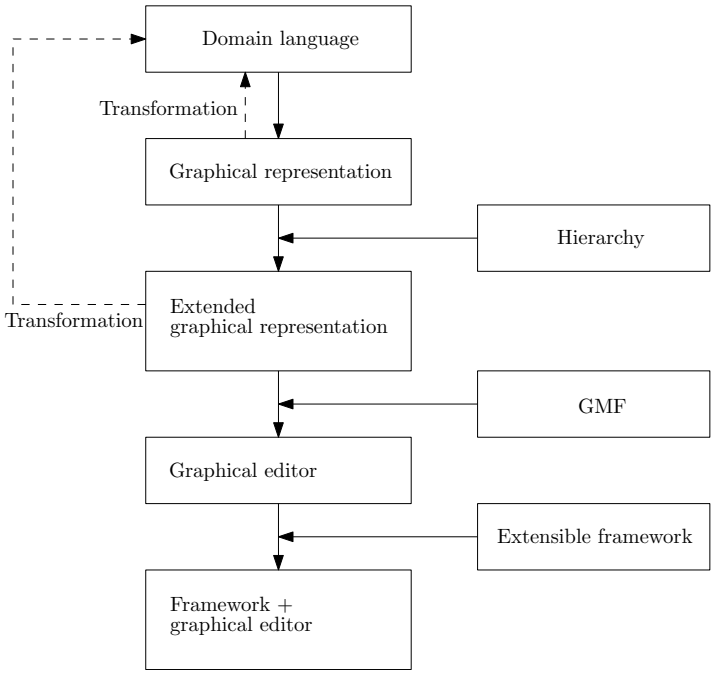


Figure 5: The step-wise approach, starting with de domain language (DSEIR) and ending with the graphical editor with the extensible framework

The visual representation, including the hierarchy concept, will also be tested on a real-life case, to show how this case can be modeled for the Octopus toolset using both the graphical editor and the new hierarchy concepts.

Outline
The outline of this report will follow the step-wise approach, so Chapter 2 will explain the DSEIR language, for which a visual representation is presented in Chapter 3 and Chapter 4. Chapter 5 describes the techniques used to create the graphical editor using GMF and Chapter 6 presents the translation from the visual language to the DSEIR language. In Chapter 7, the visual representation will be modified to support hierarchy on the application level and the resulting visual representation will be compared to known general-purpose languages in Chapter 8. Finally, the analysis framework will be presented in Chapter 9 and will be extended with model parameters in Chapter 10 and error reporting in Chapter 11. Chapter A shows a case study performed on a real-life problem.

2 Domain language

This section gives an overview of the DSEIR language that was introduced in [3], from hereon referred to as the *domain language*, because it is the language that we want to make a graphical editor for. As stated in the introduction, the domain language is implemented as a Java library. Although [6] formally describes the semantics of the underlying concepts of the domain language, it lacks a connection to the domain language itself. Hence, only informal semantics of the language can be given.

First, an overview is given of concepts that can be found in the domain language. After that, the concepts will be described in detail and an example specification is constructed using these concepts. This example will later on serve as an example domain model that will be modeled in the graphical editor.

A system specification in the DSEIR language consists of three main concepts: the application, the platform, and the mapping. The application specifies the control flow using *tasks*, *ports* and *edges*. The platform describes the platform on which the application is executed, for example the hardware of a printer. A platform consists of *resources* such as a CPU, memory, etc. Each resource provides some *services* that can be used by the application. A service can for example be computation, storage, etc. Table 1 shows the list of all supported services. Finally the mapping specifies how a task from the application gets its required resources, by indicating per required service how much of a particular resource is assigned. This last concept is specified via a *scheduler*.

Figure 6 shows the UML class diagram of the current library implementation (semi automatically generated from the source code). Important to note is that there is a common `Expression` class representing an expression tree. The classes in the diagram will now be explained individually.

Table 1: List of all services

Service
`COMPUTATION`
`TRANSFER`
`STORAGE`
`INTERNAL_STORAGE`
`RESULT_STORAGE`
`UPLOAD`
`DOWNLOAD`
`DUMMY`

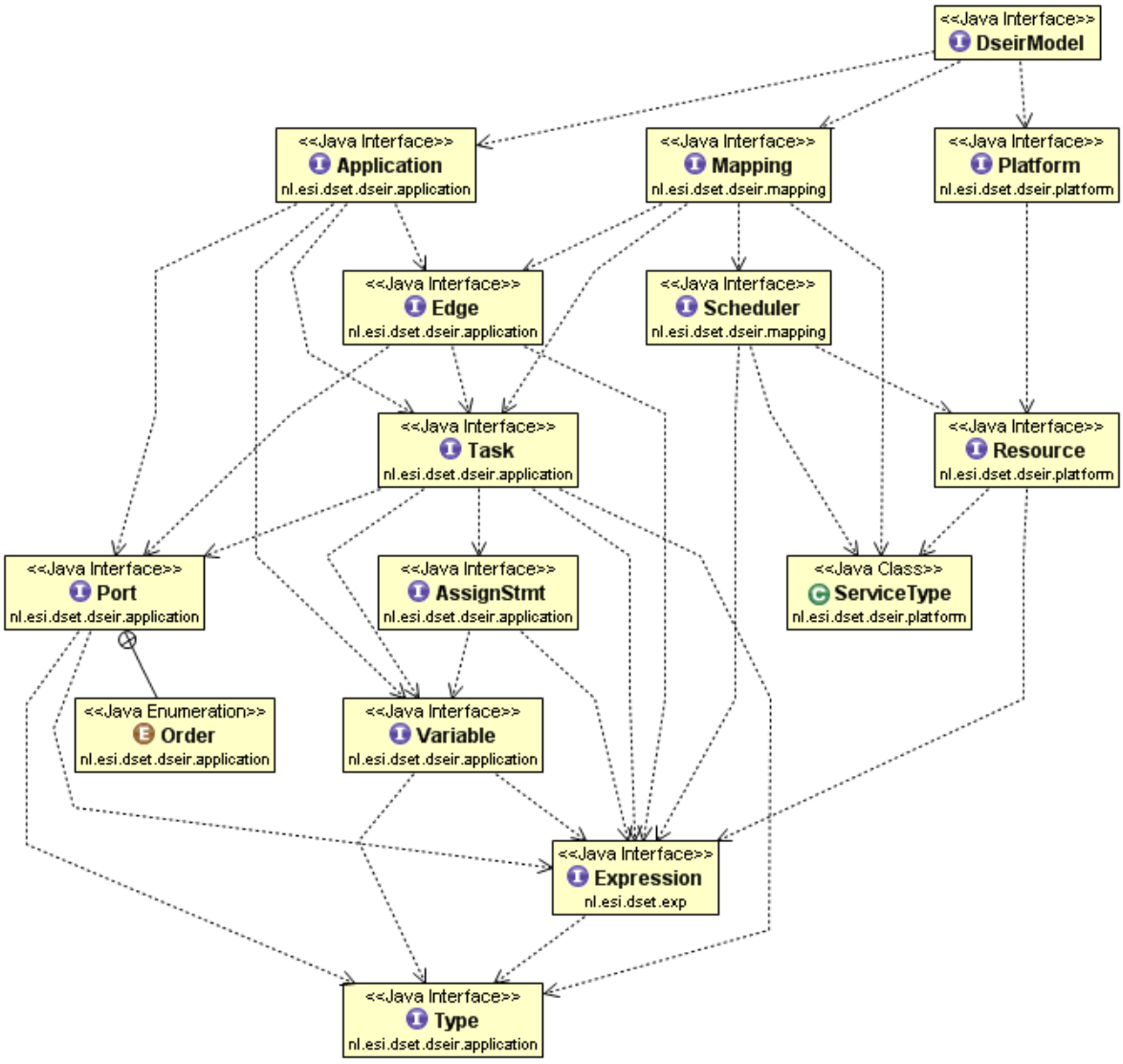


Figure 6: UML class diagram of the domain language

DseirModel
This class represents a model in the domain language. It has a name and it consists of an `Application` object, `Platform` object, and a `Mapping` object. These objects represent respectively the application part of the Y-chart, the platform part of the Y-chart and the mapping part of the Y-chart.

We will now give a simple example DSEIR specification that will be extended in the rest of this section. This specification will create a DSEIR model called 'Example' and will get references to the application, the platform, and the mapping.

```
DseirModel dseir = DseirModelFactory.createModel("Example");
Application app = dseir.getApplication();
Platform platform = dseir.getPlatform();
Mapping mapping = dseir.getMapping();
```

The following sections will describe the example we will use to demonstrate the domain language and will describe the application, the platform, and the resource perspective in more detail.

2.1 Example

To show an example implementation in the domain language we will be specifying a small example. For this small example we will model a single printer. The printer will continuously receive print jobs. When a processing job arrives at the printer, the printer will print the job. This may either succeed or fail. If it fails (5% of all jobs), then the job is again scheduled for printing until it is successfully printed.

Figure 7 shows a conceptual model for this small example.

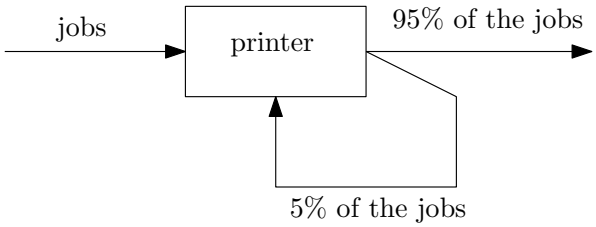


Figure 7: Conceptual model of the example

2.2 Application

The classes in the class diagram that together form the application part can be explained as follows.

`Application`

This class contains all application logic of a model in the domain language, i.e. it describes the control flow. It contains a number of `Task` objects, a number of `Port` objects, and a number of `Edge` objects. Furthermore it has a list of global variables. Note that this class houses a method to instantiate a new `Type` object for convenience only, because it does not contain references to created `Type`.

`Task`

This class represents a task that has to be executed, for example the scanning of a document or the computation of a certain value. A task has a name and furthermore involves the following concepts.

Load: Executing a task may require some services of the platform, hence a task can indicate that it requires a certain amount of units of a particular `ServiceType` (this mapping from a `ServiceType` to an amount of units required of that type is called the *load* of a task).

Ports: A task can have several `Port` objects which serve as inputs for the task.

Variables: It also has *parameter variables* as well as *local variables*, both in the form of `Variable` objects. The parameters of a task define what data is needed to start an instance of that task. When such an instance is started, the parameter variables get a value (which can influence the behaviour of the task). Local variables only exist from the moment the task starts (after parameter variables have been assigned a value) until the moment the task finishes.

Statements: A task can modify local variables and global variables when it starts or ends. A single assignment to a variable is represented with an `AssignStmt` class. A single task can have multiple start statements and/or end statements.

`Port`

A port serves as an input for a task. Entities (tokens) will be stored in ports and will be moved from port to port by the tasks of the application. A port belongs to a specific task and describes how the value of the entity or entities residing in the port can be mapped to the parameter variables of the task, via its *binding expression*.

Besides a binding expression, a port has a name, a type, an order, an initial value and a condition. The type of a port indicates the type of entities that can reside in it and the initial value indicates the entity or entities that reside in the port at the start of execution of the application logic. The order of a port can be either FIFO or Unordered, meaning that respectively entities that first enter a port are the first to come out and that there is no ordering on the outgoing entities of a port. Finally no entity can move out of a port if it has specified a condition that is not met.

Edge
This class represents a flow relation from a task to a port. If the source task of an edge has finished, an entity with the value specified by the edge is moved to the target port of the edge. However, no entity is moved if the condition of the edge is not met. An edge can specify a time for its minimum delay, which indicates that the entity will take that amount of time to move to the target port.

We will now extend the DSEIR model introduced before by creating a model for the small example. We will model this example with two tasks. One task will generate 1000 jobs and one task will print the jobs. The following code shows how these tasks can be created.

```
...
Task jobGenerator = app.createTask("JobGenerator");
Task printer = app.createTask("Printer");
```

The job generator task will have one port of type int (integer), with an initial value of 0. There will be an edge from the generater task to this port that increases this value. The guard of the generator task will become false when this value has reached 1000, so then the generator will stop generating jobs. The following code shows how this can be specified in DSEIR.

```
...
Variable x = jobGenerator.addParVariable("x", int);
Port inJobGenerator = jobGenerator.createPort(
/* name */               "p1",
/* condition */          TRUE,
/* binding expression */ cnst(x),
/* initial value */      ZERO,
/* ordering */           Port.Order.Unordered);

Edge e = app.createEdge(
/* from */       jobGenerator,
/* to */         inJobGenerator,
/* condition */  TRUE,
/* expression */ cnst(x) + ONE,
/* delay */      ZERO) ;

jobGenerator.setGuard(lt(cnst(x), cnst(1000)));
```

For the print task we introduce one port where jobs arrive (jobs will be integers with value zero). To model the success of printing, we introduce a local variable ‘success’ for the print task that is initialized when the task starts. Only if this variable is 1 then the job will be successful. If the print job is not successful, then the job will be put back into the input port of the print task via an edge. The following code shows these additions.

```
...
Variable success = printer.addLocalVariable("success",
gt(uniformDist(ONE, cnst(100)), cnst(5))  /* initial value */);

Port inPrinter = printer.createPort(
/* name */               "p2",
/* condition */          TRUE,
/* binding expression */ ZERO,
/* ordering */           Port.Order.FIFO);

Edge jobInputEdge = app.createEdge(
/* from */       jobGenerator,
/* to */         inPrinter,
/* condition */  TRUE,
/* expression */ ZERO,
/* delay */      ZERO);
Edge jobFailedEdge = app.createEdge(printer, inPrinter,
eq(cnst(success), ONE), ZERO, ZERO);
```

What remains to specify is the load of our two tasks. The job generator task will not require any services, as it is just a dummy task to generate jobs. The printer task will require five units of computation and a certain amount of memory, taken from a uniform distribution between 0 and 50.

```
...
printer.addLoad(ServiceType.COMPUTATION, cnst(5));
printer.addLoad(ServiceType.STORAGE, uniformDist(ZERO, cnst(50)));
```

2.3 Platform

The classes in the class diagram that together form the platform part can be explained as follows.

`Platform`
This class contains only `Resource` objects, which together define the platform of the `DseirModel`.

`Resource`
This class describes a single resource, with a certain name and capacity. A resource can provide several services in the form of a `ServiceType` (for the tasks to use). For each service type a resource provides, a processing time is given that indicates the amount of time needed to process one unit of load imposed on the service type.

For our example domain model, we will use two resources, namely a cpu and a storage unit

called ‘Memory’.

```
...
Resource cpu = platform.createResource(
/* name */     "CPU",
/* capacity */ 1);

Resource memory = platform.createResource(
/* name */     "Memory",
/* capacity */ 100);
```

Given these resource specification, we indicate that the cpu provides the service `COMPUTATION` and that one unit of computation takes two time units. The memory will provide `STORAGE` and storing will take no time.

```
...
cpu.setProcTime(ServiceType.COMPUTATION, 2);
memory.setProcTime(ServiceType.STORAGE, 0);
```

2.4 Mapping

The classes in the class diagram that together form the mapping part can be explained as follows.

`Mapping`
This class specifies the mapping between the application and the platform, using the following concepts. **Schedulers**: The mapping has a number of `Scheduler` objects. A scheduler is used to assign resources to required service types (required by the tasks). For each task, such a scheduler is required. A task can have at most one scheduler, but a single scheduler can be shared between multiple tasks. **Priorities**: The mapping class specifies a priority for each task. A task with a higher priority is always considered for scheduling before a task with a lower priority. **Deadlines**: The mapping maintains a deadline for each task, indicating the maximal (wanted) duration between the start and the end of the task. **Handovers**: The mapping class also indicates *resource handovers*. A resource handover indicates for an `Edge`, for example an edge from a task A to a port of a task B, that an amount of resources claimed by task A of a particular service type are handed over to task B after completion of task A. This can for example be used to hand over memory that contains the result of a computation done by A.

Scheduler
This class represents a scheduler which scheduling indicates which resource has to process a load imposed on a specific service type (by a task). Aside from determining which resource(s) have to process which load, the scheduler also specifies how much units each resource has to process and whether a resource can be taken away from a task (preemption).

In our example we will have two schedulers, one for the generator task and one for the printer task. As the generator task requires no services, its scheduler will have no scheduling rules.

```
...
Scheduler generatorScheduler = mapping.createScheduler();
mapping.addScheduler(jobGenerator, generatorScheduler);
```

Our print task needs scheduling rules for both services it requires: computation and storage. Once a certain amount of units (load) is required from the cpu, we will assign the full capacity of the cpu to it. In case of the memory resource, we will always assign the amount of units required for storage. We will model the cpu as a preemptive resource.

```
...
Scheduler printerScheduler = mapping.createScheduler();
printerScheduler.add(
    /* service type */ ServiceType.COMPUTATION,
    /* resource */     cpu,
    /* amount */       1,
    /* preemptive */   TRUE);
printerScheduler.add(
    ServiceType.STORAGE,
memory,
load(ServiceType.STORAGE
);
mapping.addScheduler(printer, printerScheduler);
```

We will have no deadlines, and for the priorities we will use zero for the generator task and one for the printer task (although this does not matter, because the generator task requires no services).

```
...
mapping.setPriority(jobGenerator, ZERO);
mapping.setPriority(printer, ONE);
```

The last thing that has to be specified is the handovers. In our example system, we do not really need any handovers, but to demonstrate handovers, we add a handover of all storage from the printer task to itself in case of a failed printing task (note that function call to `amount` returns the amount of storage currently assigned to the printer task):

```
...
mapping.addHandover(jobFailedEdge,
    ServiceType.STORAGE,
    amount(ServiceType.STORAGE, 2));
```

3 Visual DSEIR

In Chapter 2 we described a lot of concepts that are present in the domain language. All these concepts need to be present in the visual representation as well. It is not a good idea to let the user specify these concepts in one big model, because this big model can then be very complex to understand. Hence, the concepts are divided into multiple *perspectives*. A perspective is a visual representation of a set of related concepts, for example tasks and ports. All perspectives together form the whole visual representation that can be translated to the domain language. For convenience, we will simply call a model in this visual representation *the model*. The name of the visual representation shall be Visual DSEIR, or VDSEIR in short.

The domain language itself already makes the distinction between application, platform and mapping. For the visual representation, this distinction will be refined. The following sections describe the refined perspectives in detail and the example from Chapter 2 will be modeled using these perspectives.

It is important to note that the provided meta-models of the perspectives are for the visual representation. Because of this, all expressions are represented as strings. In Chapter 4 the translation of such a string to an `Expression` is given.

Throughout all perspectives, a service type is represented in the following way:

`Service`	
	This node represents a service type. The `service` string specifies the name of the service type that this node represents. It can for example be '`COMPUTATION`' or '`STORAGE`'.

3.1 Application perspective

This perspective will represent the control flow of the model as well as the global variables. This means that it incorporates tasks, ports and edges. Figure 8 shows the meta-model of this perspective. This perspective is similar to the application part of the domain language, with the exception that it will not specify task loads.

The application perspective can have at most one `GlobalDeclaration`. This class has a `declaration` attribute to specify global variables. See Chapter 4 for the syntax that is allowed in this attribute together with its interpretation.

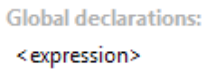
As in the domain language, a `Task` can have a name, a guard, start statements, end statements, local variables and parameter variables. The interpretations of the last four strings are given in Chapter 4 as well, while the guard is a string that represents an `Expression`.

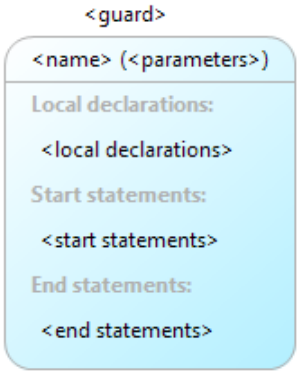
A `Port` has the same properties as its domain version, so it has a name, a binding expression,

an initial value, an ordering, a condition and a type. This interpretation of this type property is given in Chapter 4.

An `Edge` has an expression, a condition and a (minimum) delay, as well as a name. The name attribute is introduced in the visual representation, so that an edge can be referred to by a handover (which will be described in the load perspective).

All these elements will be represented as follows:

`GlobalDeclaration`	
	This node represents the global variables of the application. The variables can be declared in the expression part of the visual representation.

`Task`	
	This node represents a task. It has labels to edit all its properties and it has its child ports attached to its border. Both the name of the task and the parameters of the task are specified via the name label. This is explained in Chapter 4. The labels for the local declarations, start statements, and end statements are only visible if their corresponding property is set. This way, only relevant information is shown in the tasks.

Unordered Port	
	This node represents a port with ordering `Unordered`. The port is attached to the border of its parent task. It has labels to edit its expression and its initial value. The expression of the port consists of either a condition and a binding expression or only a binding expression separated via a delimiter (see Chapter 4). The `ordering`, `type` and `name` attributes are not represented visually, but can be set via the properties view.

FIFO Port	
	This node represents a port with ordering `FIFO`. The only difference with the `Unordered Port` is that this port has vertical lines to represent its ordering.

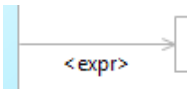
Edge	
	This arc represents an `Edge`. The expression label of the edge is used to specify its condition, expression and a minimum delay, separated with special delimiters (see Chapter 4).

Figure 9 shows the application perspective of the example.

3.2 Load perspective

This perspective will represent the relation between tasks and service types, i.e. it will visualize the service types that are required by the tasks (the load), as well as the service types that are handed over from task to task. The meta-model of the load perspective can be seen in Figure 10.

Tasks will have zero or more `ServiceAmount` children. A single `ServiceAmount` specifies an amount of load of a particular service type. Together, the `ServiceAmount` children of a task describe its load.

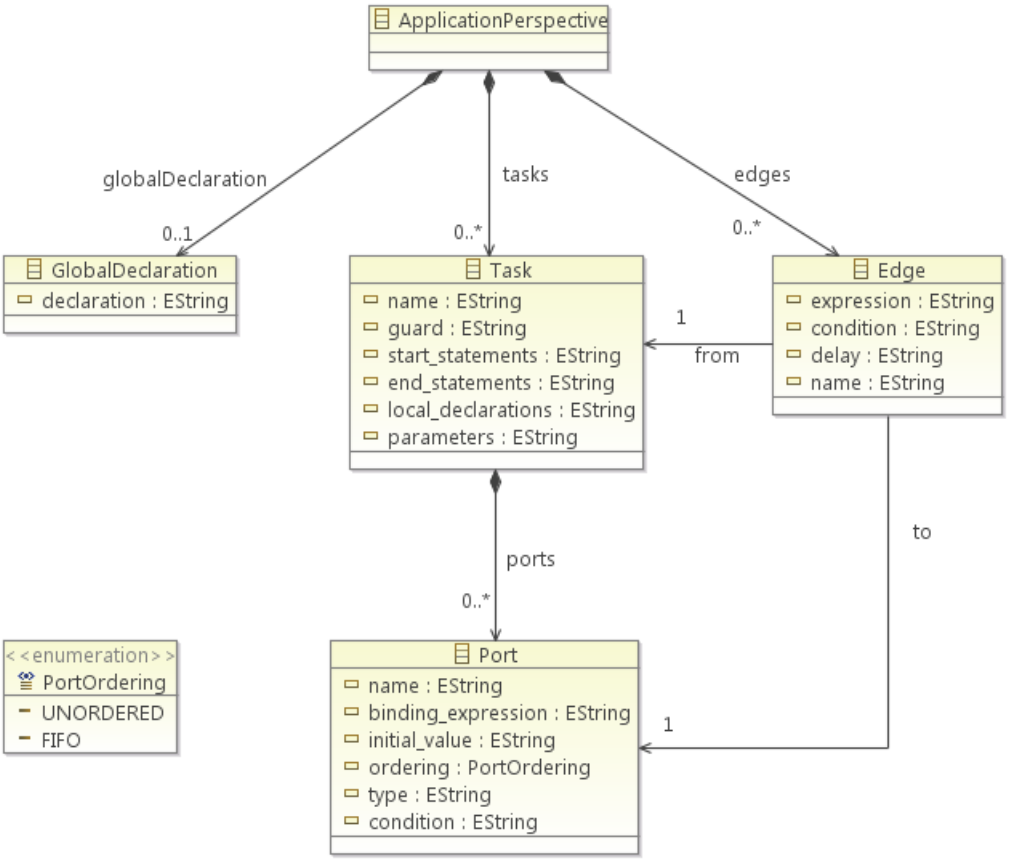


Figure 8: Meta-model of the application perspective

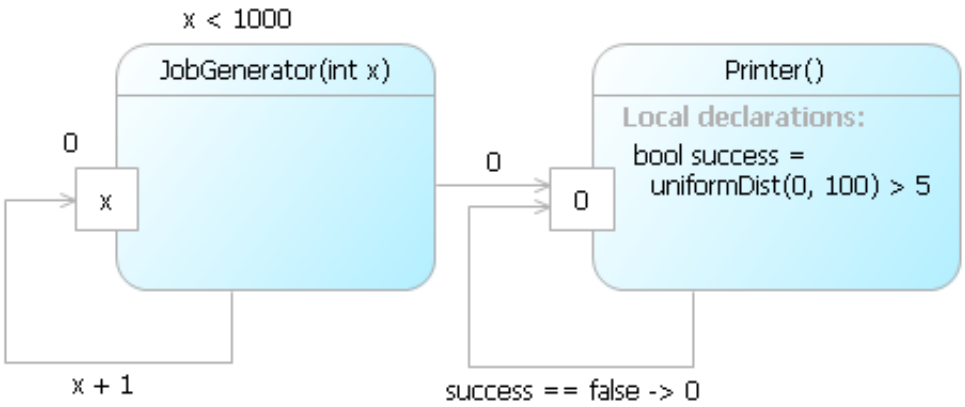
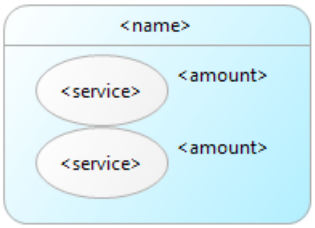


Figure 9: Example application perspective

A handover of certain service types for a specific edge will be specified with a `Handover`. It can contain zero or more `ServiceAmount` children, each one specifying the amount that is handed over of a particular service type. A `Handover` has an attribute `edge` to specify the name of the edge over which this handover occurs. A handover goes from a task (via `Task2Handover`) to a task (via `Handover2Task`). If there is only one edge from the source task to a port of the target task (in the application perspective), then the handover will occur over this edge (and the `edge` is not needed).

These elements will be represented as follows:

`Task`	
	This node represents a `Task` from the application perspective. It can contain zero or more `ServiceAmount` children, each one specifying the load of the task for a particular service type with a particular amount.

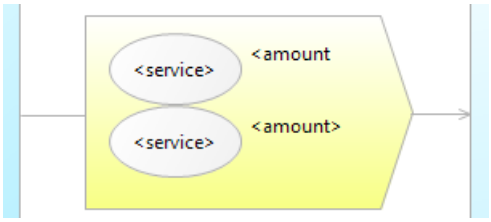
`Handover`	
	This node (together with a `Task2Handover` arc and a `Handover2Task` arc) represents a `Handover`. It can contain zero or more `ServiceAmount` children, each one specifying the amount that is handed over of a particular service type. The `edge` attribute is not represented visually, but can be set via the properties view.

Figure 11 shows the load perspective of our printer example. Because there is only one edge from the printer task to the printer task, the `edge` attribute for the handover is not set.

3.3 Resource perspective

This perspective is similar to the platform part of the domain language, i.e. it describes the resources and what service types the resources do provide at what processing speed. Figure 12 shows the meta-model of this perspective.

As in the domain language, the resource perspective can have multiple `Resource` objects.

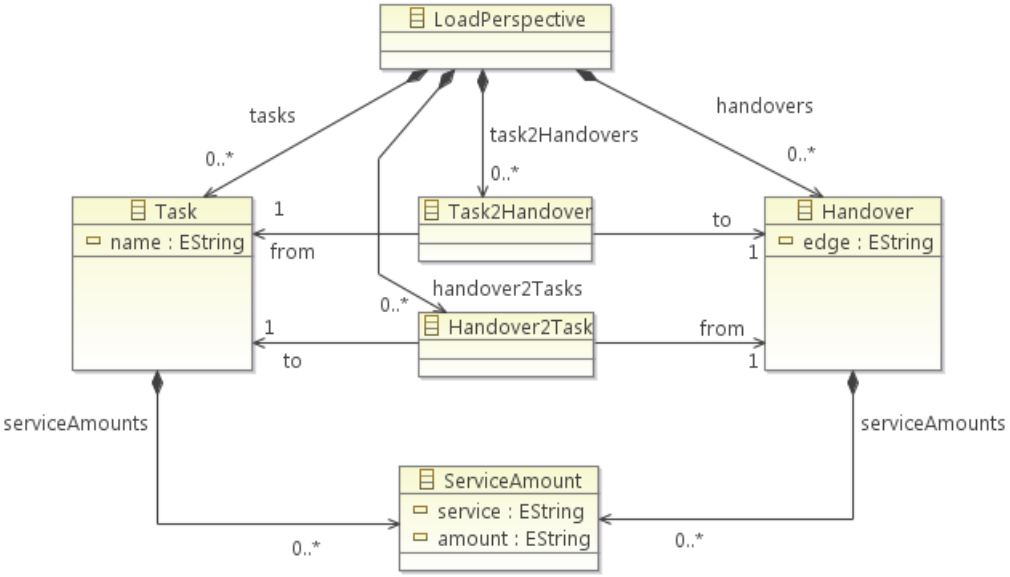


Figure 10: Meta-model of the load perspective

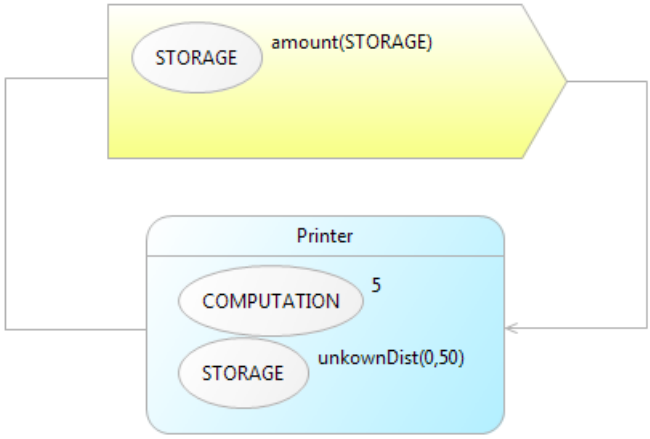


Figure 11: Example load perspective

Each resource has a name and a capacity expression. Each resource can have a `ProvidesArc` to a `Service`, to specify that the resource provides the service type specified via the `Service`. This `ProvidesArc` has a `processingTime` attribute to specify how much time the processing of one time unit of the service type requires.

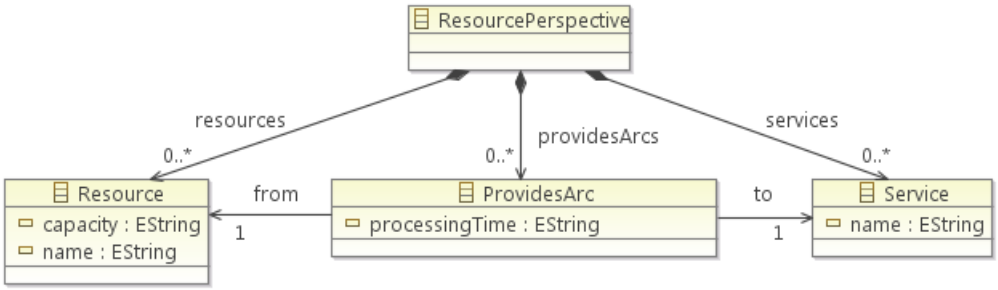
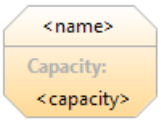


Figure 12: Meta-model of the resource perspective

The elements of this perspective are represented as follows:

`Resource`	
	This node will represent a `Resource`. It has a label for the name of the resource as well as a label for the capacity of the resource.

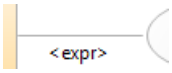
`ProvidesArc`	
	This arc will represent a `ProvidesArc`. It has a label to specify the processing time.

Figure 13 shows the resource perspective for the printer example introduced in Chapter 2.

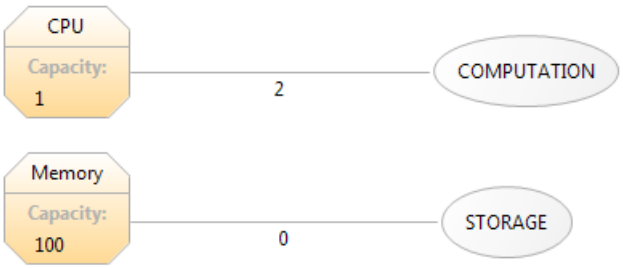


Figure 13: Example resource perspective

3.4 Scheduling perspective

This perspective will represent the scheduling part of the model, i.e. it will visualize the schedulers. In the domain language, the specification of the schedulers belongs to the mapping. Because this concept involves a lot of things (schedulers, services, resources, and scheduling rules), it is not placed alongside other concepts, but instead moved to this new scheduling perspective. Figure 14 shows the meta-model of this perspective.

The main difference between the visual representation and the domain language is the `Knot`. A knot is a child of a `Scheduler` and a knot serves to define a single scheduling rule. It has one or more input services (bound via a `BindingArc`) and one or more output resources (bound via a `AmountArc`). When a task requires some service type that is specified by an input service of a knot, the task gets assigned all output resources of that knot. The amount arcs specify how much of each of these output resources is assigned to the task.

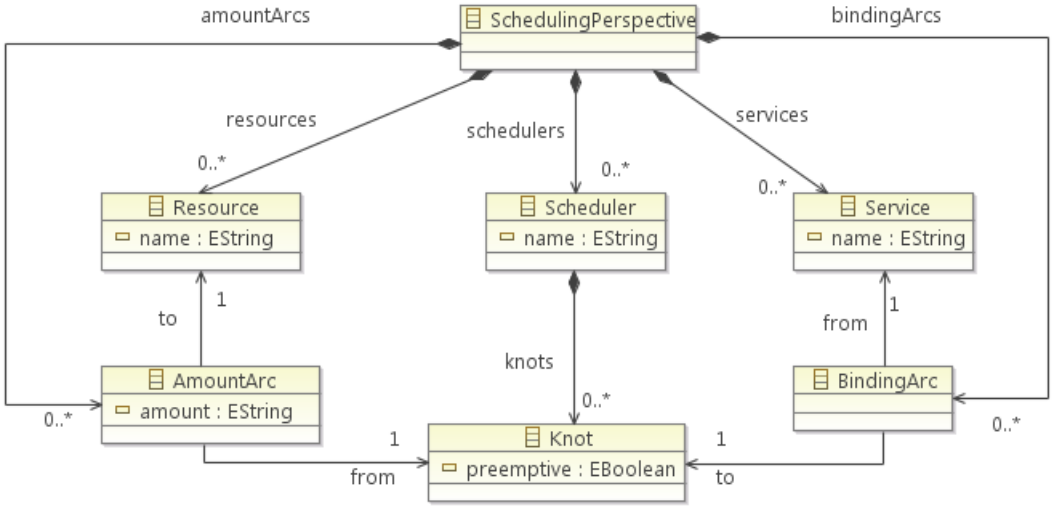
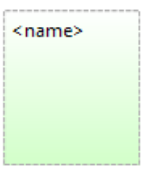
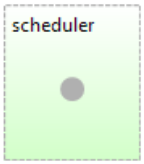
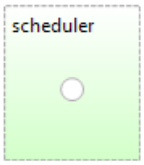


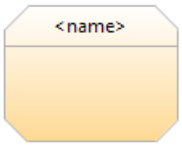
Figure 14: Meta-model of the scheduling perspective

The elements of this perspective are represented as follows:

`Scheduler`	
	This node will represent a `Scheduler`. It has a label to specify its name and it can contain child `Knot` elements.

Non-preemptive knot	
	This child node will represent a `Knot` which is not preemptive. It can be changed into a preemptive knot via the properties view.

Preemptive knot	
	This child node will represent a `Knot` which is preemptive. It can be changed into a non-preemptive knot via the properties view.

Resource	
	This node will represent a `Resource` from the resource perspective. It has a label to specify the name of the resource.

AmountArc	
	This arc will represent a `AmountArc` from a `Knot` to a `Resource`. It has a label to specify the amount of units taken from the resource (and assigned to a task requesting a service bound to knot via a `BindingArc`).

Figure 15 shows the scheduling perspective of the example printer model.

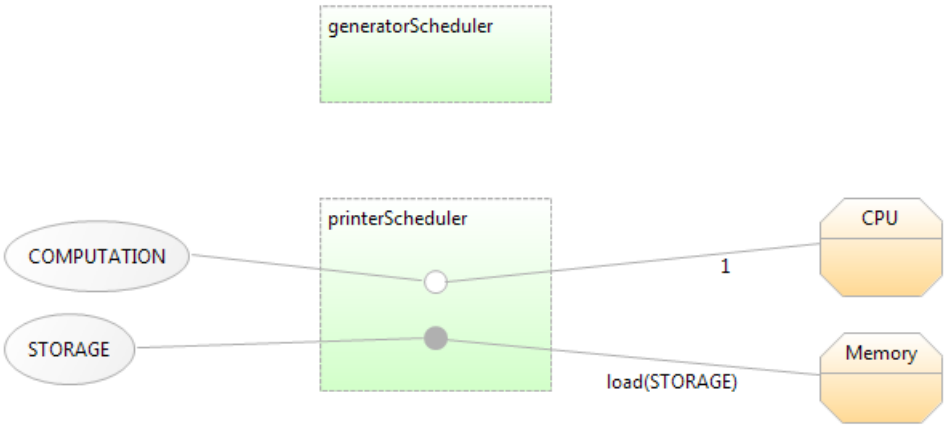


Figure 15: Example scheduling perspective

3.5 Mapping perspective

This perspective will visualize for each task which scheduler it uses, as well as its priority and deadline. These last two attributes are specified within task nodes and the scheduler used by a task is represented using a connection between that task and the scheduler. Figure 16 shows the meta-model of the mapping perspective.

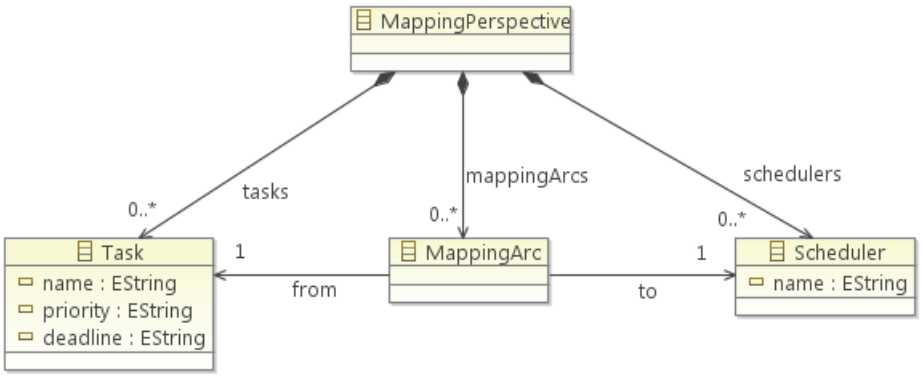
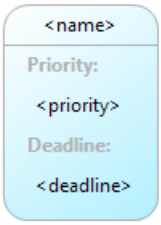


Figure 16: Meta-model of the mapping perspective

The elements of this perspective are represented as follows:

Task	
	This node will represent a `Task` from the application perspective. It has a label to specify its name, priority, and its deadline.

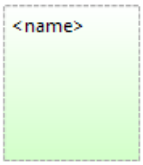
Scheduler	
	This node will represent a `Scheduler` from the scheduling perspective. It has a label to specify its name.

Figure 17 shows the mapping perspective of our printer example.

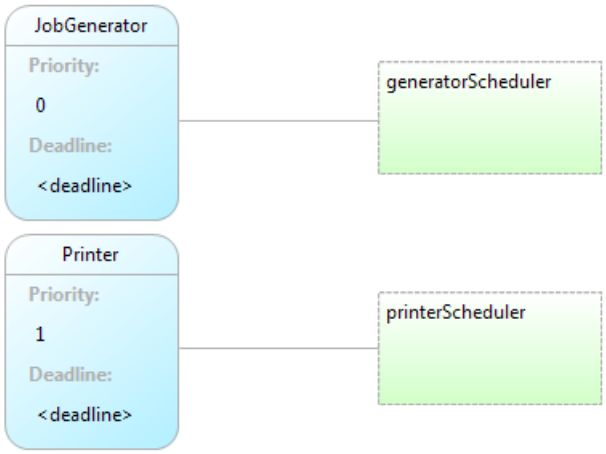


Figure 17: Example mapping perspective

3.6 System perspective

The system perspective will specify how all previously mentioned perspectives together form the system. It will allow the modeler to indicate the models that together form the application perspective, the models that together form the load perspective, etc. (any perspective can consist of multiple models). Without the system perspective there will be no relation between the individual perspectives, so the system can be considered as the glue of the perspectives.

Figure 18 shows the meta-model of the system perspective.

The system perspective is also responsible for the name of the whole system. This name can be specified via the `name` attribute of the `System` (root) class.

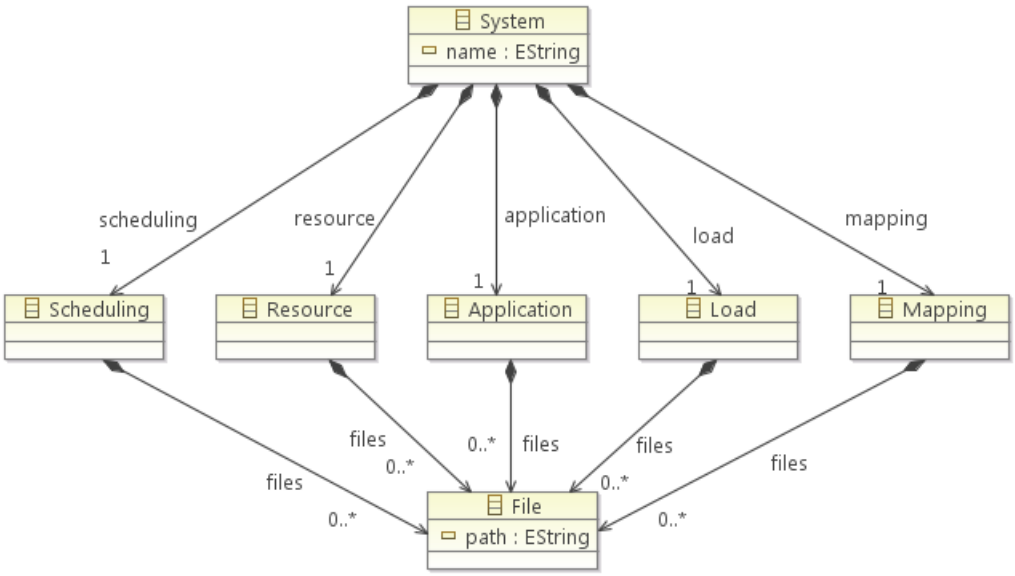


Figure 18: Meta-model of the system perspective

The visual representation of the system is as follows:

Container	
	For each perspective there is a container with zero or more `File` child nodes. These child nodes indicate the relative path of the file that will contribute to the perspective (without the extension). In the picture, only the application perspective container is showed, but the other containers only differ in their header.

Figure 19 shows the system perspective of the printer example. For each perspective we have only one file. The name of the system can only be set via the properties view of the editor, so it is not visible here. Figure 20 shows the name for the example in the properties view.

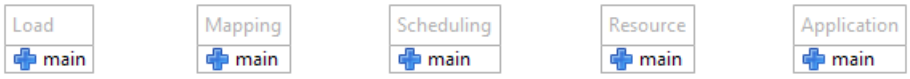


Figure 19: Example system perspective

Property	Value
Name	Printer

Figure 20: Example system perspective name

4 Textual representation

The string labels introduced in the previous section do not yet have any meaning or structure. This section will focus on precisely what can be entered in these labels and if applicable, to what expression in the toolset such an entered string corresponds. Because all name labels have a trivial structure and meaning, they will not be treated in this section.

Most labels represent precisely one expression (for example the capacity of a `Resource`), while some labels can represent multiple expressions (for example the expression label of a `Port` represents both its condition and its binding expression). Next to the expression labels, there are also declaration labels. For example the label of the `GlobalDeclaration` and the local declaration label of a `Task`. Finally, we have statement labels (start and end statements of a `Task`) and some remaining special labels. These special labels are the combination of name and parameters of a task and the type of a port.

The following subsections will explain the textual representations of all these types of labels in detail.

4.1 Expression labels

The `ExpressionBuilder` class that resides in the domain language's implementation is used to specify all expressions. This can be done via methods that create expressions (constants) and methods that combine one or more expressions into a more complex expression (operators).

The following table shows the constants of the domain language together with their textual representation and their explanation.

Constants:

Domain	**Representation**	**Explanation**
`TRUE`	'true'	The boolean constant `true`.
`FALSE`	'false'	The boolean constant `false`.
`ZERO`	'0'	The integer constant `0`.
`ONE`	'1'	The integer constant `1`.
`cnst(int c)`	<string representation of c>	The integer constant `c`.
`cnst(Variable v)`	<name of v>	The value of variable `v`.

The following part of this section explains the operators that are present in the domain language and gives the textual representation for each of them.

Boolean operators:

Domain:	`and(Expression e1, Expression e2)`
Representation:	<e1> '&&' <e2>
Explanation:	The conjunction of expression `e1` and `e2`.

Domain:	`or(Expression e1, Expression e2)`
Representation:	<e1> '||' <e2>
Explanation:	The disjunction of expression `e1` and `e2`.

Domain:	`not(Expression e1)`
Representation:	'!' <e1>
Explanation:	The negation of expression `e1`.

Domain:	`lt(Expression e1, Expression e2)`
Representation:	<e1> '<' <e2>
Explanation:	Expresses `e1` $<$ `e2`.

Domain:	`gt(Expression e1, Expression e2)`
Representation:	<e1> '>' <e2>
Explanation:	Expresses `e1` $>$ `e2`.

Domain:	`le(Expression e1, Expression e2)`
Representation:	<e1> '<=' <e2>
Explanation:	Expresses `e1` $\leq$ `e2`.

Domain:	`ge(Expression e1, Expression e2)`
Representation:	<e1> '>=' <e2>
Explanation:	Expresses `e1` $\geq$ `e2`.

Domain:	`eq(Expression e1, Expression e2)`
Representation:	<e1> '==' <e2>
Explanation:	Expresses `e1` $=$ `e2`.

Domain:	`neq(Expression e1, Expression e2)`
Representation:	<e1> '! =' <e2>
Explanation:	Expresses `e1` $\neq$ `e2`.

Integer operators:

Domain:	`add(Expression e1, Expression e2)`
Representation:	<e1> '+' <e2>
Explanation:	Expresses `e1` + `e2`.

Domain:	`sub(Expression e1, Expression e2)`
Representation:	<e1> '−' <e2>
Explanation:	Expresses `e1` − `e2`.

Domain:	`mult(Expression e1, Expression e2)`
Representation:	<e1> '∗' <e2>
Explanation:	Expresses `e1` $\cdot$ `e2`.

Domain:	`div(Expression e1, Expression e2)`
Representation:	<e1> '/' <e2>
Explanation:	Expresses `e1` div `e2`.

Domain:	`mod(Expression e1, Expression e2)`
Representation:	<e1> '%' <e2>
Explanation:	Expresses `e1` mod `e2`.

Domain:	`min(Expression e1, Expression e2)`
Representation:	'min(' <e1> ',' <e2> ')'
Explanation:	Expresses the minimum of `e1` and `e2`.

Domain:	`max(Expression e1, Expression e2)`
Representation:	'max(' <e1> ',' <e2> ')'
Explanation:	Expresses the maximum of `e1` and `e2`.

Other:

Domain:	`array(Expression... e)`
Representation:	'['<e1> ',' <e2> ',' ... ']'
Explanation:	Expresses the array [`e1`,`e2`,...].

Domain:	`iF(Expression c, Expression t, Expression e)`
Representation:	'if' '(' <c> ')' 'then' <t> 'else' <e>
Explanation:	An if-then-else expression with condition `c`, then-expression `t`, and else-expression `e`.

Domain:	`uniformDist(Expression a, Expression b)`
Representation:	'uniformDist(' <a> ',' <b> ')'
Explanation:	Expresses a sample taken from the uniform distribution ranging from `a` (inclusive) to `b` (exclusive)

Domain:	`dist(Map<Expression, Expression> valueToProb)`
Representation:	'dist(' <name of v> ')'
Explanation:	Expresses a sample taken from a random distribution indicated with the map `valueToProb`. Each entry `<value, probability>` in the map specifies that `value` has a probability of `probability` to be returned as a sample of the distribution. For the visual representation, a variable name of variable `v` has to be given. This variable `v` has to be an array with the values on the uneven indices and a size that is a multiple of two. For each value at index i, its probability is assumed to be at index $i + 1$.

Domain:	`unkownDist(Expression a, Expression b)`
Representation:	'unkownDist(' <a> ',' <b> ')'
Explanation:	Expresses a sample taken from an unknown distribution with inclusive lower bound `a` and exclusive lower bound `b`.

Domain:	`load(ServiceType s, 0)`
Representation:	'load(' <name of s> ')'
Explanation:	In the context of scheduling, this expresses the amount of units of service type `s` required by the task that is currently being scheduled.

Domain:	`load(ServiceType s, 1)`
Representation:	'totalLoad(' <name of s> ')'
Explanation:	In the context of scheduling, this expresses the total amount of units of service type `s` required by all tasks that are being scheduled.

Domain:	`amount(ServiceType s, 0)`
Representation:	'amount(' <name of s> ')'
Explanation:	In the context of processing times for resources, this expresses the amount of units that have to be processed. This allows to specify a processing time of a resource (for a service type) that depends on the number of units that have to be processed.

Domain:	`amount(ServiceType s, 1)`
Representation:	'amount(' <name of s> ')'
Explanation:	In the context of scheduling, this expresses the amount of resources of service type `s` held by the task that is currently being scheduled.

Domain:	`amount(ServiceType s, 2)`
Representation:	'amount(' <name of s> ')'
Explanation:	In the context of resource handovers, this expresses the amount of resources of service type `s` held by the task after it finishes.

Domain:	`free(Resource r)`
Representation:	'free(' <name of r> ')'
Explanation:	In the context of scheduling, this expresses the amount of free units that resource `r` has.

4.2 Declaration labels

The declaration labels are used for the global declarations of an application and for the local declarations of a task. Currently, the domain language only allows variables to be declared. Each variable should either have a type or an initial value (from which its type can be extracted). The supported types are the integer and boolean types. Furthermore integer and boolean arrays can be specified, with a constant size.

The type of a variable will be represented as follows:

Domain:	`INT`
Representation:	'int'
Explanation:	An integer.

Domain:	`BOOL`
Representation:	'bool'
Explanation:	A boolean.

To make variable declarations more clear, the type has to be specified for each variable, even if it has an initial value. A declaration of a single variable can be done in the following ways:

Representation:	<variable type> <variable name> ';'
Explanation:	This declares a variable with the given name and type.
Example(s):	`int x;`

Representation:	<variable type> <variable name> '=' <initial value> ';'
Explanation:	This declares a variable with the given name, type and initial value.
Example(s):	`bool b = true;`

Representation:	<variable type> <variable name> '[' <size expression> ']' ';'
Explanation:	This declares an array variable with the given name and with elements of the given type. The size expression has to be an integer expression specifying the size of the array.
Example(s):	`bool b[6];`

Representation:	<variable type> <variable name> '[]' '=' <array expression> ';'
Explanation:	This declares an array variable with the given name and with elements of the given type. The array will initially be equal to the given array expression, and the size of the array is extracted from the array expression.
Example(s):	`int x[] = [1, 2, 3];`

Multiple variables can be declared by concatenating their declarations.

4.3 Statement labels

The labels that can specify statements are the start statement and end statement labels of a task. The domain language only supports assign statements, which will be represented as follows:

Representation:	<variable name> '=' <expression> ';'
Explanation:	This represents exactly one assignment expressing that the variable with the given name will be assigned the given expression.
Example(s):	`x = 4;`

Multiple assign statements can be given in one statement label by concatenating them. The

order in which the statements are processed is equal to the order in which they appear in the statement label.

4.4 Special labels

4.4.1 Port label

The expression label of a port will represent its condition and its binding expression. The representation will be as follows:

Representation:	[<condition expression> '->'] <binding expression>
Explanation:	The port expression is either a binding expression or a condition followed by a binding expression, separated with '->'.
Example(s):	`x` (a binding expression only) `x > 3 -> x` (a binding expression with a condition)

4.4.2 Edge label

The label on an `Edge` (application perspective) will represent its condition, expression and its delay. Its representation is as follows:

Representation:	[<condition> '->'] <expression> ['>>' <delay>]
Explanation:	The edge label is the expression, possibly prefixed with a condition and a delimiter '->', and possibly followed by a delay delimiter '>>' and a delay expression.
Example(s):	`5` (only an expression) `x == 3 -> 5` (a condition and an expression) `5 >> 10` (a delayed expression) `x == 3 -> 5 >> 10` (an expression with a condition and a delay)

4.4.3 Task name and parameters

A `Task` has a label that represents its name and parameters. To explain its representation, first the representation for a parameter type is given and the representation of the declaration of a single parameter.

A parameter type is represented as follows:

Representation:	<type> ['[' <size> ']']
Explanation:	A parameter type is represented as a type ('int' or 'bool'), optionally followed by an integer size constant in between brackets, expressing an array of the given size.
Example(s):	`int` (an integer) `bool[3]` (a boolean array with three elements)

A parameter variable declaration is represented as follows:

Representation:	<parameter type> <name>
Explanation:	A parameter variable is represented with its type, followed by its name. Because parameter variables can have no initial value (they get assigned a value when the task starts), this is its only representation.
Example(s):	`int[2] x`

The representation for the task name and parameters, based on the previous two representations, is as follows:

Representation:	<name> [‘(’ <parameter> ‘,’ <parameter> ‘,’ ... ‘)’]
Explanation:	This representation consists of a name, possibly followed by a comma-separated list of parameter variable declarations.
Example(s):	`TaskA` (only an name) `TaskA(int x, bool[2] b)` (task name with two parameters)

4.4.4 Port type

A `Port` has an attribute to specify its type. This is a string attribute with the same representation as the parameter type representation from the previous section, so a port type can for example be `bool` to represent a boolean, or `int[2]` to represent an integer array containing two elements.

5 Graphical editor generation

This appendix will explain the approach used to automatically generate the graphical editor for the domain's visual representation. First the GMF framework together with its automatic graphical editor generation will be explained. We will see that by default, this generation is not very flexible and the generated editor is not very adaptable, so we will present our approach to customize the graphical editor to our needs, without losing the ability to automatically generate it. The information found in [4] has been used to come to this approach.

5.1 GMF

Domain model
The core concept in the editor generation process is the *domain model*. The domain model is a meta-model describing the domain language. The modeling language in which this meta-model is developed is called the `Ecore` language. This language is very similar to the UML class diagram language. The meta-model specifies which classes have which attributes and how the classes are related (inheritance, association, containment). The domain model should have a single class representing the root of the domain language. This root will have containment references to other classes, which form the building blocks of the domain language.

Model-code generation
If the domain model has been specified, it is still only a model. The graphical editor has to work with Java code and classes, so the first step in the generation process is to turn the domain model into Java classes. This process is depicted in Figure 21. First, the domain model is converted into a *generator model*. After that, this generator model is converted into plain Java code. The creation of the generator model out of the domain model and the code generation based on this generator model is handled by the EMF framework.

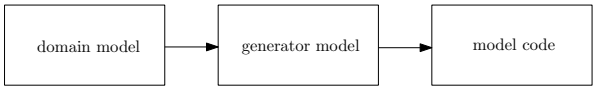


Figure 21: Model code generation process

Graphical definition
The domain model specifies the building blocks that are present in the domain language, but it does not specify the visual representation of these building blocks. Therefore a *graphical definition model* is created. This model describes figures that can be used to represent classes. The expressiveness of this model is limited (we will see how we can overcome this in the next section), but we can specify a rectangle, ellipse, line, arc, etc.

Tooling definition
Apart from the graphical definition, we also need to specify what *tools* we want in the graphical editor. A tool is a entry in the palette that allows us to create elements in the graphical editor. Figure 22 shows an example generated graphical editor. The large pane on the left shows the model as specified in the graphical definition. The pane one the right (the palette) shows the elements we can add to the left pane. This palette is specified via the tooling definition by

specifying groups of tools as well as icons, titles, and descriptions.

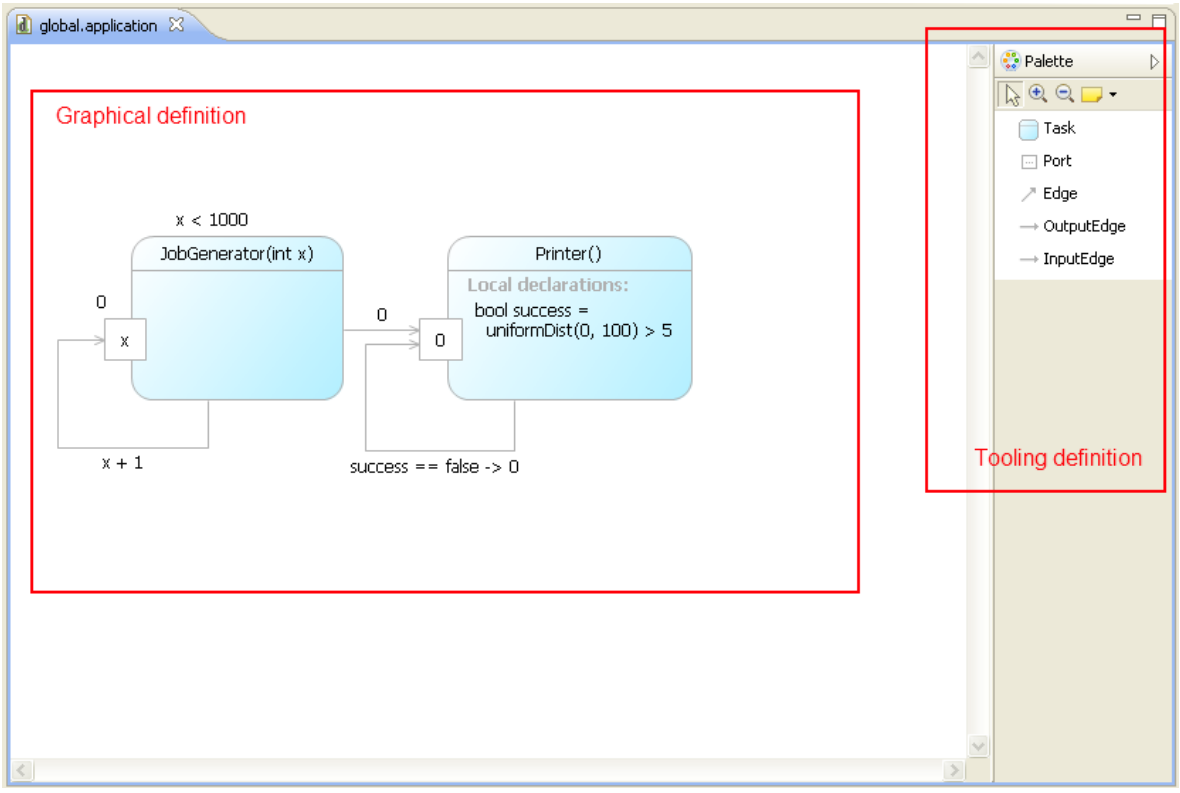


Figure 22: Screenshot of the graphical editor where the areas specified by the graphical definition and the tooling definition are highlighted

Mapping definition

In the domain model we specify the building blocks of or domain language, in the graphical definition we specify some representations and in the tooling definition we specify some palette entries. The *mapping definition* couples these three models together. For each class in the domain model it states the figure that should be used to represent it (taken from the graphical definition), the tool that is used to create it (taken from the tooling definition) and whether it should be interpreted as a node or a link. For each attribute that should be drawn on the canvas, the mapping definition states the figure that should be used to represent it (again taken from the graphical definition). Note that any attribute that is not represented on the canvas will be present in the properties view, as can be seen in Figure 23. The mapping definition also specifies which class from the domain model is the root (so that it knows where to add newly created elements) and it specifies which classes contain which children.

Graphical editor generation

The domain model, the graphical definition, the tooling definition, and the mapping definition together provide enough information for the GMF framework to generate the graphical editor code. Figure 24 shows the code-generation process. First, a generator model is created from the mapping definition (which contains references to the other three models). This generator model combines all information from all mentioned definitions and models into one big model that provides exactly enough information to generate the graphical editor's code. Hence,

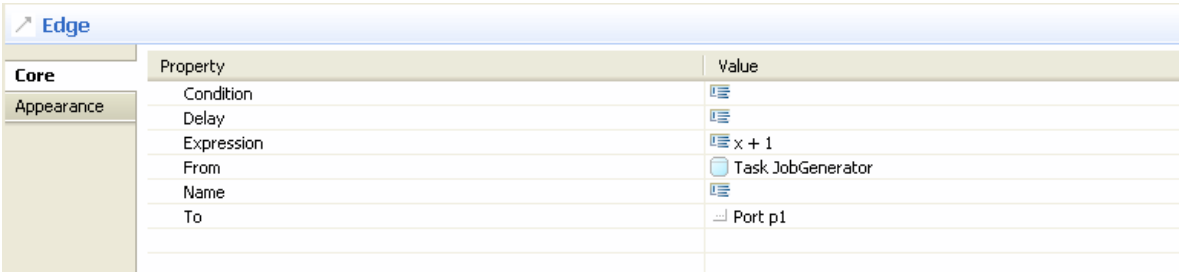


Figure 23: Example properties view

when the generator model is created, the only remaining step is to generate the code for the graphical editor. All these steps are provided by the GMF framework.

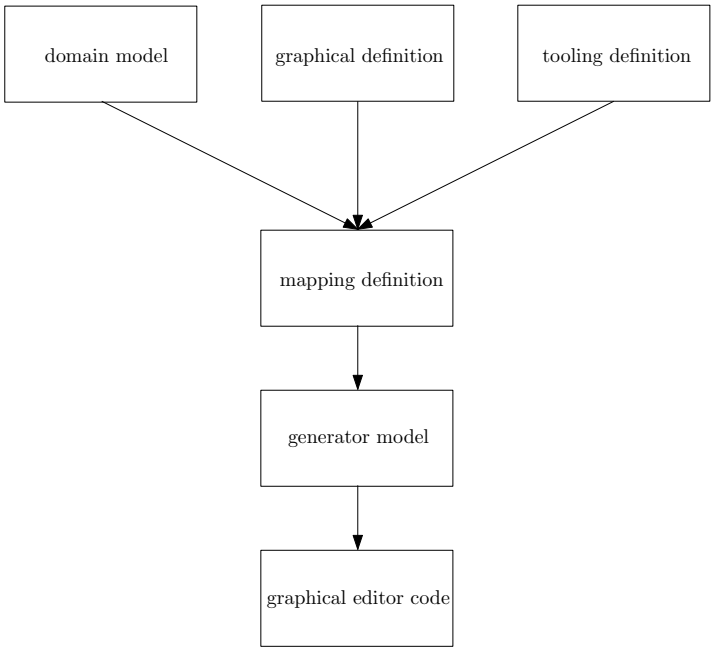


Figure 24: The process of generating graphical editor code

5.2 Customization

The graphical definition does not provide enough expressiveness to describe the figures we want for tasks, resources, etc. Furthermore, the generated graphical editor can only be customized by adapting its source code directly. Ideally, we want the process of creating the graphical editor from the models to be fully automatic and we want to be able to regenerate the graphical editor after some changes have been made. In order to achieve this, we extend the generation flow of the graphical editor in three places, indicated by the darker nodes in Figure 27.

The following part explains each of the three customizations:

- First of all, we add custom figures to the generation process to overcome the lack of expressiveness of the graphical definition. The custom figures will be plain java classes, implementing a common interface used by the GMF editor (to be more precise, they implement the `IFigure` from the `draw2d` library, which is used by the GMF for rendering models). Luckily, the graphical definition model allows us to specify that we want to use such a custom java class as a figure.

  Normally, a figure is specified in the graphical definition by assigning a shape (circle, rectangle, etc) to it. Then, the children of the figure (labels and contained classes, such as ports of a task) are specified. A layout manager can be assigned to the figure to specify how the children of the figure are positioned. This works fine for figures specified in the graphical definition, but we want different behavior for our custom figures. Namely, want our custom figure to take control over the shapes and positions of the child elements and we only want to specify in the graphical definition which child we use for a specific custom figure.

  Therefore, the following customization is implemented. Each custom figure specifies a list of special strings, called 'fields'. These strings represent the name of children that it expects. The custom figure also implements functionality to draw the figure, given the child elements. The top of Figure 25 depicts this situation. For example, the custom figure for a `Resource` expects a label for the name of the resource, specified as 'name'. It also expects a label for the capacity of the resource, specified as 'capacity'. The custom figure might be drawn like Figure 26, where the 'name' label should of course reflect the actual name of the resource and the 'capacity' label should reflect the actual capacity.

  In the graphical definition we now specify a figure by indicating the custom figure's java class. Furthermore we add all necessary child elements of the figure and for each of them we indicate its name. This name is mapped one-to-one to the names specified by the custom figure itself. In this way, we propagate enough information to the the custom figure its children, so that the custom figure can be drawn. Following the resource figure example, the graphical definition of the resource figure points to the custom figure for the resource. The graphical definition will assign two child labels to the rectangle figure. The name label of the resource is given the tag 'name' and the capacity label is tagged as 'capacity'. This is shown in the bottom right of Figure 26.

  To allow this information to be entered in the graphical definition and to expose the children of a figure to its custom figure implementation, a custom layout manager is created. This layout manager collects the children specified in the graphical definition as they are being created by the source code of the graphical editor. When all children required by the custom figure are collected, the layout manager passes this information to the custom figure. The position of this layout manager in the custom figure overview is depicted in the bottom-left of Figure 25.

  Given this customization, we can create custom figures that are in control of their children, which is precisely what we need for the graphical editor.

- The second customization of the graphical editor generation process is the addition of a model transformation on the generator model created by the GMF framework. This generator model has some properties that influence the code generation process, for example the extension used for the model files. We want to be able to change these properties, however the generator model is a generated model. We can of course directly

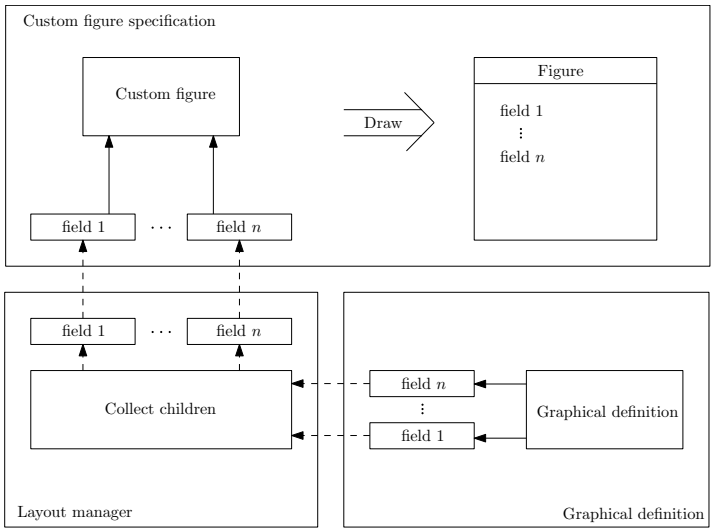


Figure 25: Custom figure specification

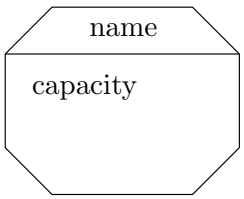


Figure 26: An example figure with two fields: ‘name’ and ‘capacity’

edit the generator model after it has been generated, but for a clean workflow we do not want to adapt generated models (nor source code) manually.

To overcome this problem, we add an in-place model transformation step to the graphical editor generation process. This transformation, specified in the Operational Query/ View/ Transformation language, takes a generator model as input and adapts it to suit our needs. This transformation will form an additional input for the graphical editor generation, so next to the mapping definition, graphical definition, tooling definition, domain model, and custom figures, we also (optionally) have a model transformation.

With this customization, we are able to separate the models required for the generation of the graphical editor from the generated models and source code, resulting in a clean generation process. Figure 27 shows the place of this customization in the generation process.

- The last and most powerful customization of the graphical editor generation process is the use of *dynamic templates*. If we would change the source code that is created by the GMF framework, we lose the option to regenerate the graphical editor, because changes made to the source code are lost when regenerating. To be able to adapt the source code without losing the regenerative property of the generation process, we use dynamic templates. These dynamic templates allow us to modify the way in which the source code is generated. The following two paragraphs briefly explain these dynamic templates.

  The GMF framework uses `Xpand` to generate the source code of the graphical editor. `Xpand` is a model-to-text engine that comes with the GMP project. The GMF framework houses the `Xpand` definitions (called templates) that can generate the source code for the graphical editor from the generator model.

  Custom versions or adaptations of these templates can be made to customize the code generation process. These custom templates are called dynamic templates and can be plugged in in the GMF code generation framework. Any desired behavior that cannot be specified in the domain model, graphical definition, tooling definition, mapping definition, or generator model (via a transformation), is expressed in a dynamic template.

  So, with the use of dynamic templates, we can totally customize the graphical editor to our needs, without modifying any generated source code. Furthermore, with the use of dynamic templates we make sure that the graphical editor can be regenerated from scratch when some changes have occurred. Of course, all dynamic templates form another input for the generation process. Figure 27 shows the place of this customization in the generation process.

Given these customizations, the graphical editor can be automatically generated out of a mapping definition, graphical definition, tooling definition, domain model, (optionally) a model transformation, dynamic templates, and custom figures. The additional inputs for the generation process provide a lot of expressiveness, making the graphical easy customizable.

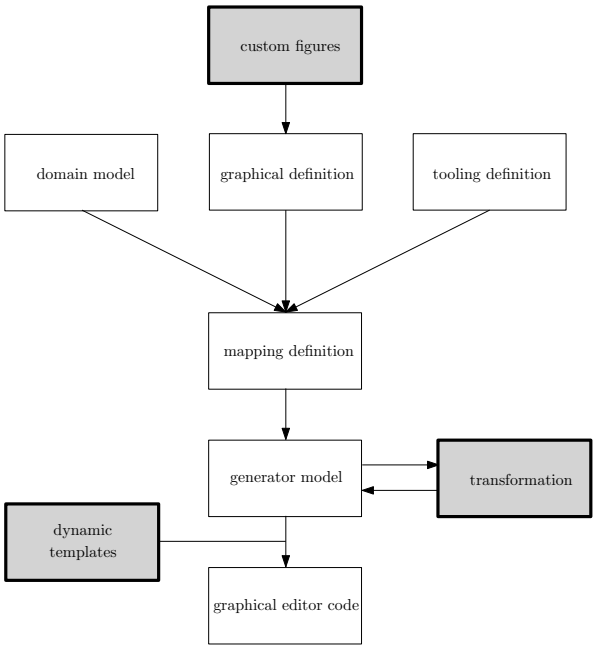


Figure 27: Customized process of generating source code for the graphical editor.

5.3 Implemented customizations

The following list gives a small overview of what has been done with these customization techniques:

- **Custom figures**
  Custom drawn figures with nice visual features such as gradients. Note that most customizations have been made to make this possible.
- **Context-sensitive figures**
  The port figure looks different based on its type. See visual representation for the port in Chapter 3.
- **Multi-line labels**
  When a user presses Control + Enter, he or she can add extra lines to labels. This is especially useful for large declaration blocks.
- **Arc endpoints**
  By default, GMF attaches the endpoints of arcs only to points lying on the axis-parallel bounding box of figures. This behavior is bad for non-rectangular figures, such as ovals and rounded rectangles. Hence, the behavior has been modified so that the endpoints of an arc are always correct.
- **Ports attached to tasks**
  The GMF framework does support nodes that are attached to parent nodes, however a special customization had to be made to allow the ports of a task to overlap a bit the the task (see visual representation for the port in Chapter 3).

- **Removed bugs in the GMF framework**
  Several bugs originally in the GMF framework have been removed, for example there was an issue with not being able to create arcs from sub-nodes such as scheduling knots.

- **Delete semantics**
  Customizations have been made to make it impossible to remove the global declaration node. In addition, the delete behavior of labels is changed. Originally, when a user deletes a label, the label was not actually deleted, but only hidden. This behavior has been changed so that the text in the label is set to the empty string instead.

- **Initial models**
  When we create an application perspective, we want the global declaration node to be present in the new model automatically. To allow this behavior, a customization to the GMF code generation was made.

6 Domain specification generation

This section will show how the perspectives introduced in section Chapter 3 and the textual representations introduced in Chapter 4 can be used to generate a specification in the domain language.

Such a specification can be either an instance of the `DseirModel` class, as explained in Chapter 2, or it can be plain Java source code that, when run, creates the instance of the `DseirModel`. The first approach is taken because of the following reasons:

- No intermediate layer is needed to create the `DseirModel` instance.
- The source code will not be visible to the end-user anyway, so possible extra work to generate the source code will not be worth it.
- Although source code can be persisted (opposed to an `DseirModel` instance), it is invalidated when the domain language changes.

The process of generating an instance of the `DseirModel` class consists of three steps, which can be seen in Figure 28.

1. First, the system perspective is used to find all input models of the system. Then, all input models are combined into one *generator model* and errors in any of the input perspectives are detected. This generator model contains enough information to actually generate a domain specification and it contains some extra information to ease this generation process.
2. In the second step, the generator model is checked for errors which can only be found after combining the perspectives. For example, we need information from both the application perspective and the mapping perspective to check whether each tasks has a scheduler associated with it (to be more specific, each task that requires at least one service type).
3. Finally, the actual domain specification is generated using the generator model. During this step all strings are parsed and the domain elements are initialized.

First, the generator model will be explained and then each step will be described in more detail.

In the first step, all individual perspective files are combined into one big generator model that contains all information. The meta-model of this generator model is visible in Figure 29 (note that it shows only the elements with their attributes and references as a tree, because a diagram would contain too much crossing links).

The root of this meta-model is the `DseirGen` class. The meta-model is designed in such a way that it is very close to the domain language, while still using strings. This has the following advantages:

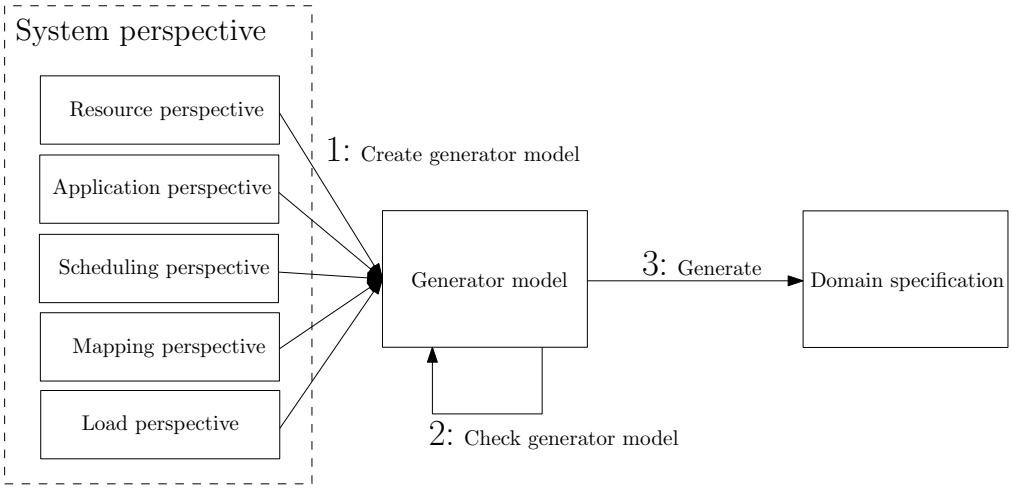


Figure 28: Domain specification generation process

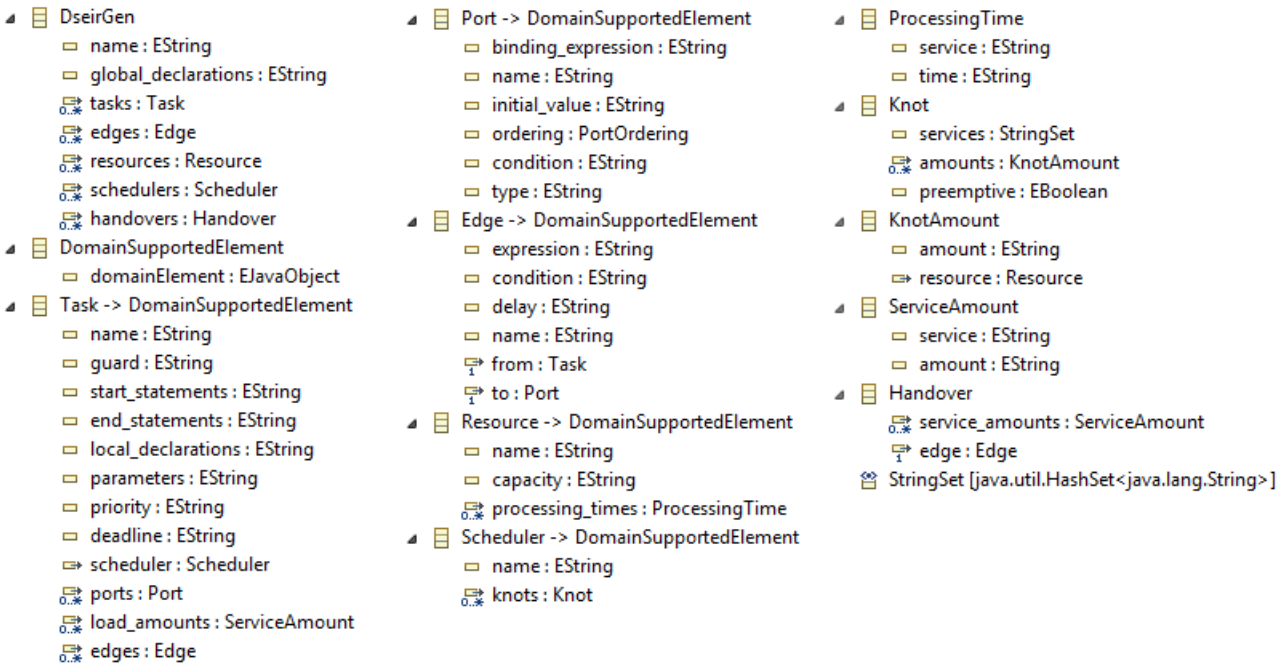


Figure 29: Meta-model of the generator model

- Generation of the domain specification becomes relatively straightforward. The main challenge will be parsing all strings.
- When new syntactic sugar (such as hierarchy) is added, the only thing that has to change is the process of combining the perspectives into the generator model. The generation of the domain specification does not have to be touched.

The disadvantage of the use of the generator model is that it would be more efficient to generate the domain specification directly when combining the perspectives, but the advantages given above outweigh this efficiency loss. Especially because we know for sure that syntactic sugar will be added.

Most elements in the generator meta-model are already explained in Chapter 2 and Chapter 3. Apart from a convenience reference `edges` pointing to the outgoing edges of a task, the main new thing is the `DomainSupportedElement` class.

`DomainSupportedElement`
This is a superclass of all elements in the generator model that can also be constructed in the domain language. It has an attribute `domainElement` that is used for generation of the domain specification. This attribute provides a link between the generator model and the domain specification being constructed. The specific use of this attribute is explained further on.

The generator meta-model does not contain separate elements for label-less connections used in the perspectives. Instead, these connections are represented using attributes and references. For example all service types that are bound to a knot are represented using the `services` attribute of the `Knot` class.

Finally, the `Handover` class is somewhat different than the one found in the load perspective. The edge over which the handover occurs is referenced using the `edge` reference. The handover class in the load perspective does not directly contain this information, so this edge is discovered while creating the generator model.

6.1 Combining the perspectives

This step starts by finding all models that contribute to the system, as indicated by the system perspective. Then, an empty generator model created and each perspective is handled. While handling each single perspective model, all significant nodes are added to the generator model, so finally the generator model contains information about all perspectives.

The perspective files cannot be handled in arbitrary order. For instance, the mapping perspective has references to tasks defined in the application perspective and schedulers defined in the scheduling perspective. If we would start by handling the mapping perspective, we would not be able to resolve these references. Figure 30 shows an overview of the dependencies between the perspectives. If there can be a reference from perspective A to an element in perspective B, an arc is drawn from A to B.

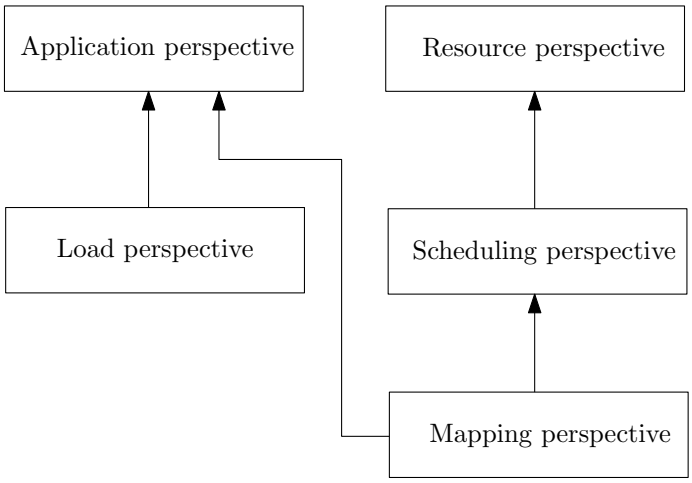


Figure 30: Dependencies between perspectives

From these dependencies, the following order is used to handle the individual perspectives.

1. Resource perspective
2. Application perspective
3. Load perspective
4. Scheduling perspective
5. Mapping perspective

In order to be able to resolve referenced elements, a central mapping (implemented using a hash-table) is maintained while filling the generator model. This mapping has methods to register elements with their name and to retrieve registered elements given their name, i.e. it has keys of type `String`. For instance, when handling the application perspective, a `Task` instance is added to the generator model. This task instance is then registered in the central mapping, using its name as a key. When handling the load perspective later on, this central mapping is used to lookup the `Task` instance in the generated model.

If we would not use this central mapping and we want to find the `Task` instance in the generator model for a task with a given name, all `Task` instances in the (partial) generator model have to be inspected. This is not needed if we use the central mapping, so we gain a more efficient implementation.

Handling the individual files is pretty straightforward. It mainly involves creating a task for each task in the application perspective, a port for each port in the application perspective, a scheduler for each scheduler in the scheduling perspective, etc. The only difficulties are caused by the connections. When processing a connection from a perspective, the `from` and `to` attributes of the connection point to elements in the perspective, so not in the generator model. However, when creating the corresponding connection in the generator model, we want the `from` and `to` to point to the corresponding elements in the generator model.

To overcome this problem the general schema given in Algorithm 1 is used for each perspective, where *generatorModel* is the partial generator model that is being filled and *perspective* is the perspective that is being handled. First all nodes (elements which are not a connection) are handled and a local mapping is maintained, remembering the corresponding node in the generator model for each node in the perspective. For example, in the resource perspective, for each resource r, a resource node r' is created in the generator model. This pair r, r' is then stored in the local mapping, such that given r we can obtain r' again when we need it. If the node has any child nodes (such as ports for tasks), they are handled and remembered as well. After this, all connections in the perspective are handled and the local mapping is addressed to find the source and target of the connections in the generator model.

Algorithm 1 HandlePerspective($generatorModel, perspective$)

```
localMapping ← new mapping
for all node ∈ perspective do
    node′ ← new generator model node for node
    generatorModel.add(node′)
    localMapping.put(node, node′)
    for all child ∈ node.children do
        child′ ← new generator model node for child
        generatorModel.add(child′)
        localMapping.put(child, child′)
    end for
end for
for all connection ∈ perspective do
    from ← localMapping.get(connection.from)
    to ← localMapping.get(connection.to)
    Handle connection
end for
```

This general scheme that is not optimal for all perspectives. First of all, not all nodes have a corresponding node in the generator model. Second, when a certain node is never referenced in a connection later on, it is still stored in the local mapping. Therefore only the necessary nodes are stored in the local mapping in our implementation.

Note that during the handling of each perspective, errors such as empty attributes (for example no name for a task, not type for a port) are reported as well as errors that are caused by unresolvable references (a load specified for a task which does not exist). Opposed to the first error type, this last type of error depends on more than one perspective. The general approach was to check these types of errors during the validation of the generator model, however the generator model simply cannot be constructed if these errors are present. Hence, they are already reported in this step.

6.2 Generator model validation

If the generator model is successfully created, it can still contain (semantic) errors. For instance, each task which requires some services should have a scheduler associated with it.

This requirement involves information from multiple perspectives, so it could not be checked during creation of the generator model. Hence, this step checks whether the generator model does not violate any of these type of requirements. These checks are performed simply by iterating over the elements regarding the requirement.

6.3 Domain specification generation

After the generator model is created and checked, it will be transformed into a specification in the domain language. Because the structure of the generator model is closely related to the domain language, each element in the generator model that is supported in the domain (it inherits from `DomainSupportedElement`) will become precisely one element in the domain specification.

The general approach is as follows. First, an empty `DseirModel` instance is constructed. Then, all elements of the generator model are traversed and for each element, the corresponding element in the domain is created.

The hardest part in this approach is the interpretation of all string attributes. As shown in Chapter 4, a great deal of the strings represent expressions. Other common strings are statement and declaration strings.

For the expression strings, a parser has been made that can parse a given string into an instance of the `Expression` class. The following part will explain how this parser is created.

Expression parsing
Ideally, the expression parser takes a `String` and turns it into an `Expression`. If we for example take the string `"5 * 3 + 2"`, we want it to be converted to `add(mult(cnst(5), cnst(3)), cnst(2))`.

However, this is impossible to do for any expression. Lets assume we have a global variable named `"a"`. To parse the string `"a - 2"`, we need information about the global variables. Because a task can have local variables and parameters that can be used in an expression string, we need information about these variables as well.

Furthermore, we can have expressions referring to resources (for example `"free(CPU)"`). To make things even worse, the string `"amount(COMPUTATION)"` has three different meanings, depending on the context in which the string is parsed (scheduling, processing times, or resource handovers).

To solve these issues, a context is defined that is used together with a string to create an `Expression` instance.

Context	
A context has the following attributes:	
`model`	This is a `DseirModel` instance from the domain language in which global variables and resources are located.
`task`	This can be either a `Task` instance from the domain language or `null`. If it is a task instance, variables are first sought in this task's local variables and parameters. If the task does not have the required variables or when the attribute is `null`, the variables are sought in `model`.
`contextType`	This can be either `RESOURCE_PROCESSING_TIME`, `SCHEDULING_AMOUNT`, `HANDOVER_AMOUNT`, or `OTHER`. This indicates how an amount expression should be interpreted. The value `OTHER` indicates that it does not matter how an amount expression should be interpreted.

Given this definition of the context, an annotated grammar is created to parse a string into an expression, given the context. This grammar is then converted into a recursive-descent parser with JavaCC ([2]).

The obtained parser is used to parse all expression strings found in the generator model. The `model` of the context is the instance of the `DseirModel` that is under construction. Hence, we have to make sure that all referred elements are present in this model before we parse the expressions referring to these elements. The dependencies between the expression strings used in the generator model's elements are depicted in Figure 31. The schedulers can refer to resources (with a `"free(RESOURCE)"` expression), while the ports, edges, and handovers can refer to variables used in their associated task (the parent task for the ports, the source task for the edges and the task from which resources are handed over for the handovers).

In order to not violate these dependencies, the resources are handled first and then the schedulers. After that the tasks are handled together with their ports. After initializing the task in the domain language, the `task` attribute of the context is set to this task in the domain language. After the tasks and ports are generated, the edges and handovers are handled with `task` attribute of the context set to the associated task in the domain specification (using the `domainElement` attribute to get the task in the domain specification given the task in the generator model).

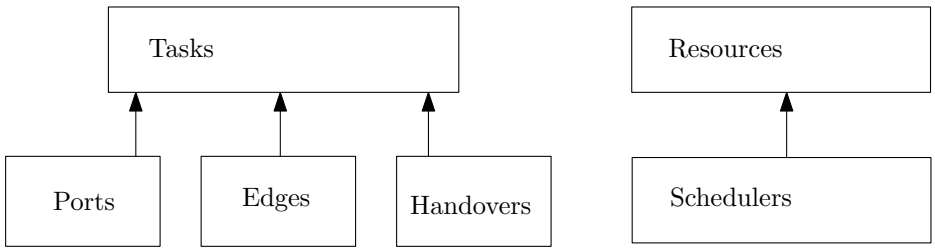


Figure 31: Dependencies of expressions in generator model elements

Declaration and statement parsing

The declarations and statements have to be parsed as well. Opposed to the parsing of expressions, declarations and statements should not generate a data structure (there is no such thing as a `Declaration` or a `Statement` in the domain language).

Instead, when a variable is discovered during parsing, it should be added to the correct part of domain specification and when a statement is discovered, it should be added either to the start statements of a task of the end statements of a task. To achieve this, again a context is supplied while parsing. This time, the context has methods that are called by the parser when it discovers a variable (with its type and possible initial value), or a statement.

`Context`

This context has the following methods:
 	
`onVariable(String name, Type type)`
This method indicates that a variable has been discovered with the given type and no initial value.
 	
`onVariable(String name, Expression initialValue)`
This method indicates that a variable has been discovered with the given initial value.
 	
`onAssignment(Variable variable, Expression value)`
This method indicates that an assign statement has been discovered. This assign statement assigns `value` to `variable`.

Again an annotated grammar is created that calls the appropriate methods in the supplied context. This context is converted into a recursive-descent parser with JavaCC.

Because a declaration or statement can contain expressions, the grammar is set up as follows. When the declaration and statement parser expects an expression while parsing, the control is passed over to the expression parser. The expression parser will then try to parse as much as possible starting where the declaration and statement parser has stopped. At some point the expression parser will stop parsing. This either means that the remaining input has been completely parsed into a single expression or that the expression parser constructed an expression up until the point it encountered an error in the remaining input (for example when it encounters a character that is not valid in an expression, but that is part of the declaration or statement, e.g. ';'). In both cases the expression is returned and the declaration and statement parser will continue where the expression parser stopped parsing.

7 Task hierarchy

This section describes the extension to the visual representation of the domain language to incorporate *task hierarchy*. The aim of the task hierarchy is to allow a single task at a high abstraction level to represent a complete application perspective at a lower abstraction level. This allows us to create models with different levels of abstraction.

Before explaining the hierarchy implementation, two notions are introduced. We will call a task that represents a complete application perspective a *super task* and we will call the application perspective its *sub-application*.

Our design goals of the hierarchy are as follows:

- It should be possible to let information flow from the super task to its sub-application.
- After a sub-application has been executed, any results of the execution should be accessible at the super task.

The hierarchy will be implemented as syntactic sugar only and in such a way that any model incorporating task hierarchy can be translated into a model without hierarchy. In the remainder of this section, the general idea of the hierarchy is sketched and the meta-model of the application perspective is extended. Then, the visual representations of a super task and a sub-application are given. After that, the transformation from a representation using hierarchy to the representation without hierarchy is explained. Finally, our example from Chapter 2 is refined using hierarchy.

7.1 Extensions to the application perspective

The general idea of the hierarchy is as follows. We want to allow a sub-application to be represented by a super task and we want information to flow from this super task to the sub-application. Therefore we introduce the concept of *application parameters*. Each application perspective will have the possibility to specify parameters, just like a task. The semantics of these parameters will be given later on in this section, however the idea behind the parameters is that a super task can start its sub-application with a particular valuation of these parameters. If we have for example a sub-application that will print a number of pages, say n pages, we give this sub-application a parameter `int n`. The sub-application can then access this parameter to know how many pages should be printed (how this parameter is accessed will be explained later).

The `GlobalDeclaration` class in the application perspective's meta-model is extended with a `parameters` property for the parameters of the application perspective. The extended meta-model can be seen in Figure 32.

Given these application parameters, we can define how a super task refers to its sub-application. Each super task has a *call* to its sub-application. A call to a sub-application involves specifying which sub-application to use and supplying values for all parameters of the sub-application

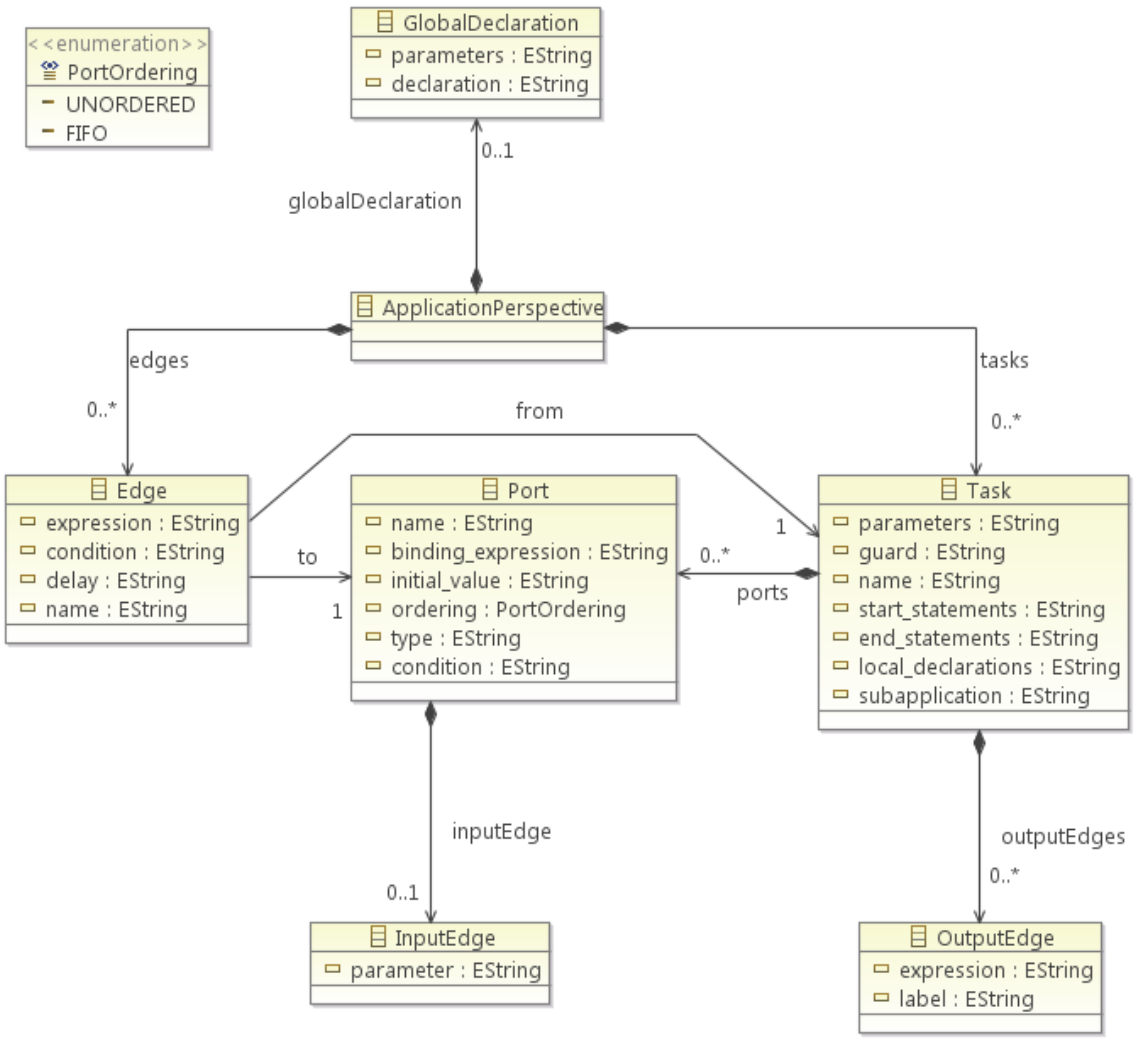


Figure 32: Meta-model of the hierarchical application perspective

(similar to a function call). A call is of the form

```
<sub-application name>(<parameter value>, <parameter value>),
```

where the name of the sub-application is the name of the file in which it resides and the parameter values can be arbitrary expressions. Via these parameter values, information can flow from the super task to the sub-application.

The `Task` class is extended in the application perspective's meta-model, to support an additional property for sub-application call. We will call any task that has this property set a super task. Tasks that do not have this property set will remain regular tasks.

What remains to define is how the sub-application can refer to its parameters and how information can flow from a sub-application to the super task. To allow a sub-application to use its parameters, we introduce the element `InputEdge` in the application perspective's meta-model.

`InputEdge`
An `InputEdge` can be attached to a `Port`. Its purpose is to allow parameter passed to the sub-application to be redirected to the `Port` it is attached to, called the *target port*. It has a property that specifies the name of one of the parameters of the sub-application, say parameter p. When the sub-application is called with a value v for this parameter p, the value v will be put into the target port.

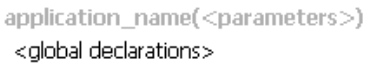
To allow information to flow from the sub-application to its super task, we use the following mechanism. We allow a task in the sub-application to *publish* some expressions. An expression that is published in the sub-application can be used in the super task. Expressions will be published under a name. For example, say we have a sub-application that does some complex calculations and these calculation can either be successful or unsuccessful. We publish the boolean expression indicating whether the calculations were successful in the last task of the calculations. We publish this expression under the name 'success'. We can then use 'success' in expressions in the super task to check whether the sub-application has successfully executed its calculations, in the same manner that we use variables.

`OutputEdge`
An `OutputEdge` can be attached to a `Task`. Its purpose is to publish a single expression. Its `label` property is the name under which the expression is published and the `expression` is the expression that is published.

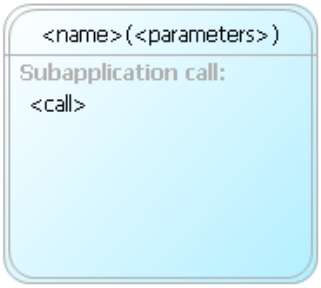
Note that a sub-application is assumed to be finished after a task which publishes one or more expression is executed.

7.2 Visual representation

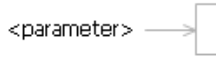
The visual representation for the application perspective needs to be adapted because of these changes. First of all, an application perspective can now have parameters.

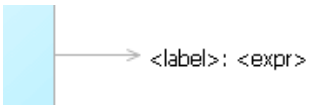
`GlobalDeclarations`	
	Instead of 'Global declarations', the header now shows the name of the application perspective (the name of the file it resides in), together with the parameters in between brackets.

We also introduce a new representation for a super task, to make it easy for the modeler to separate super tasks and normal tasks.

`Task` (which is a super task)	
	The visual representation of a super task is similar to a normal task. The only difference is the double border and the label that allows the call to the subapplication to be made.

Beside of these changes to the visual representation, two new elements have been introduced, namely the input edge and the output edge.

`InputEdge`	
	The visual representation of the input edge is similar to an edge without a source. It has a label to specify the parameter whose value will be redirected to the target port.

OutputEdge	
	The visual representation of the output edge is similar to an edge without a target. It has a single label that represents both the label property (the name under which the expressions is published) and the expression property. These two properties are separated by a semicolon.

7.3 Transforming a hierarchical model

Because the hierarchy is syntactic sugar only, we need a way to convert an arbitrary model with hierarchy into a model without hierarchy. In order to do this, all super tasks have to be replaced by their sub-application, respecting the information flow inside the sub-application and the information flow out of the sub-application. We will now explain how to do this for a single super task and its sub-application.

We will use a super task in a generalized context, visible in Figure 35. The super task, annotated with the letter `S`, has an arbitrary number of input ports and an arbitrary number of outgoing edges. It makes a call to its sub-application with expressions `<par_1>` ... `<par_n>` as parameter values.

The generalized sub-application corresponding to `S` is visible in Figure 36. The sub-application has parameters `<var_1>` ... `<var_n>`. It has a single task with input edges (task `A`) and a single task with output edges (task `B`). Task `A` has several ports with input edges and task `B` has several output edges. The other tasks of the sub-application are not specified and they can form an arbitrary sub-application logic. The only thing we know about this sub-application logic is that there is no task in it with input edges and there is no task in it with output edges. Note that we say a task has input edges if and only if it has at least one port with an input edge.

When we now replace `S` by its sub-application, the following steps occur:

1. All tasks of the sub-application are copied and inserted at the level of `S`. For each task, its name is concatenated with '_sub_' followed by the name of `S`. This concatenation is done to ensure unique names for all tasks after the sub-application is inserted.

2. The global declarations of the sub-application are added to the global declarations of the level of `S`. However, instead of using the variable names as defined in the sub-application, the names are again concatenated with '_sub_' followed by the name of `S` to ensure unique names for all global variables.

3. We replace `S` with a dummy task that is an exact copy of super task `S`, without the sub-application call and with it's name appended with '_start'. The purpose of this dummy

task is to start the sub-application by redirecting the parameter values supplied in the sub-application call to the correct ports of the sub-application.

4. For each input edge of the sub-application we identify the variable `v` that is on the label of the input edge. Then we find out the index `i` for which `v = <var_i>`. Now that we know `i`, we know which parameter value in the call to the sub-application should be redirected to the target port of the input edge, namely `<par_i>`. Hence, we replace the input edge with a real edge from the dummy task to the target port of the input edge. The expression of the edge is set to `<par_i>`, so the correct parameter value will be put in the correct port of the sub-application after execution of the dummy task.

5. For each outgoing edge of `S` we change the source task to `B`. This ensures that the control flow at the level of `S` is continued after the sub-application has finished. Furthermore, the expression `<expr>` on each outgoing edge is replaced by `<expr>'`, which is a transformed version of the expression. The transformation is obtained by replacing each occurrence of `<label_i>` by `<out_i>`, for $0 \leq$ `i` $\leq$ `n`.

The model that is obtained after performing all these steps is visible in Figure 37.

7.4 Example

We will now extend the printer example introduced in Chapter 2 and modeled in Chapter 3. Suppose that we want three different kinds of jobs to arrive at the printer, all with an equal likelihood of arriving. Each job will have a different chance of success, given by the following table:

Job type	**Chance to fail**
0	5
1	3
1	8

Furthermore, we will refine the 'Printer' task by making it a super task with a 'printer' sub-application. This sub-application will be responsible for printing a job of a given type and returning whether the print job was successful.

To do this, we change the 'Printer' task into a super task, as can be seen in Figure 33. We also change the expression on the edge from the generator task to the printer task, so that it generates a random job type each time a job is created.

The printer sub-application will be kept very simple. It will contain two tasks. One task identifies the success percentage for the given job type and the other task will print the job, with the calculated success percentage. The sub-application can be seen in Figure 34. Note that the sub-application has only one parameter, namely the job type, and that the sub-application publishes the 'success' expression as well as the job type (needed for the edge that is taken when the print job failed).

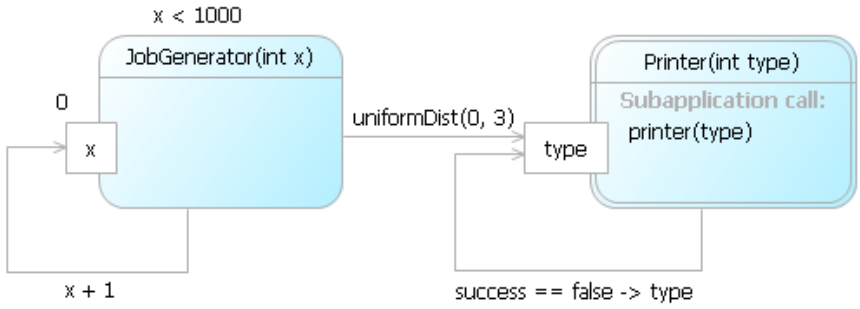


Figure 33: Printer example with hierarchy

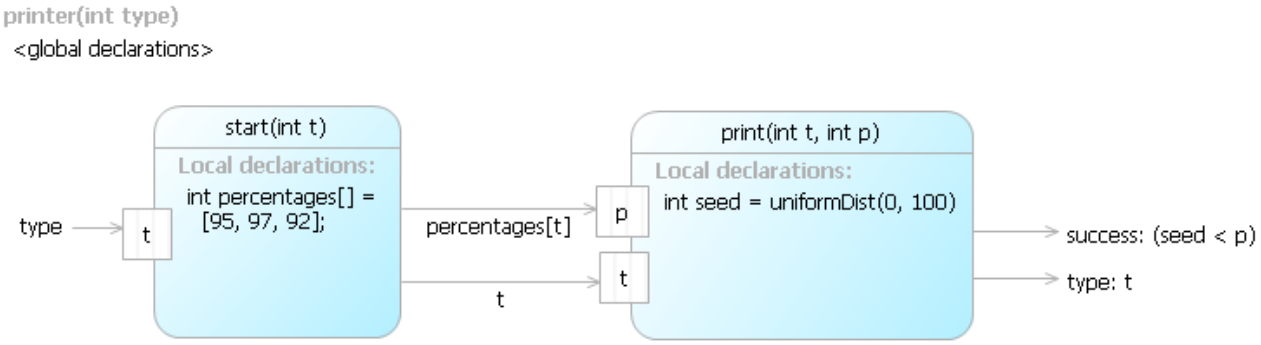


Figure 34: The 'printer' sub-application

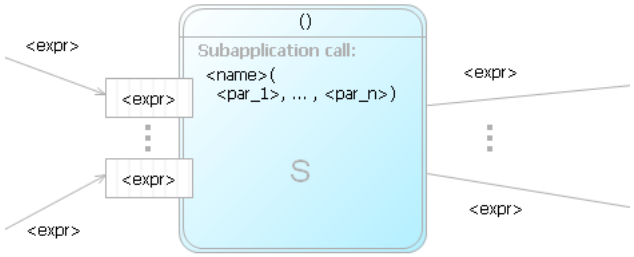


Figure 35: Super task S in a generalized context

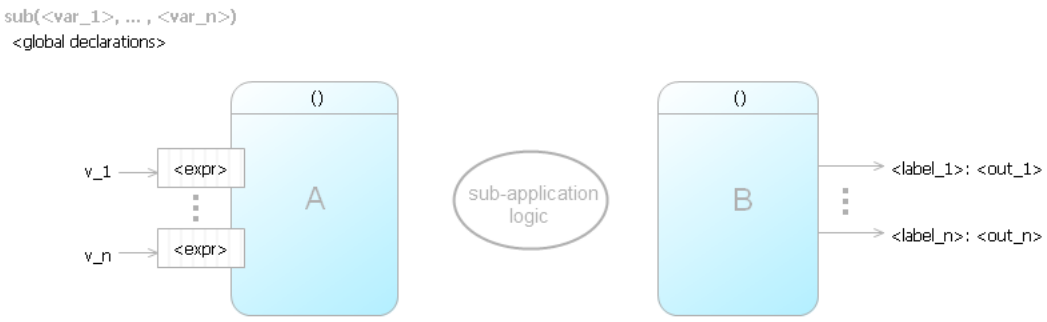


Figure 36: Generalized sub-application

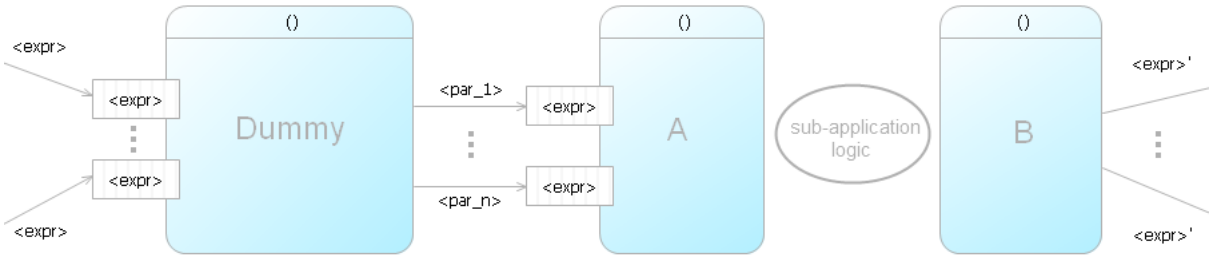


Figure 37: Generalized model without hierarchy

7.5 Consequences

The addition of hierarchy does not come without any consequences. The following two sections describe problematic consequences of the introduction of hierarchy together with their solutions.

No parameters
First of all, the steps to replace a super task with its sub-application only work if the sub-application has parameters. If a sub-application with no parameters is called from a super task, step 4 will not add any edges from the dummy task to the entry point of the sub-application (as there can be no ports with input edges referring to a parameter). To overcome this problem and allow parameterless sub-applications, we introduce the following solution.

In a sub-application without parameters, we allow an input edge to have an unset `parameter` property. This input edge will mark the entry point of the sub-application. We require that this input edge is attached to a port of the boolean type (because we will be adding an edge to this port with expression `true` when removing hierarchy). We allow a super task to call a sub-application with no parameters simply by supplying no arguments in the sub-application call.

To allow correct removal of hierarchy, we modify step 4 of replacing a super task by its sub-application as follows:

4. If the sup-application has parameters, executed step 4 as before. Otherwise, let p be the port of type boolean with an input edge that has no parameter attached to it, e.g. the entry point of the sub-application. Add an edge from the dummy task (created in step 3) to port p, having expression `true`.

Task references
Second, tasks are referenced in both the load perspective and the mapping perspective. With the new hierarchy concept, we need a way to reference tasks within a sub-application. Therefore we introduce a dot-notation for the names of tasks in the load perspective and mapping perspective. Such a name is now equal to the following:

<application_perspective> '.' <task>

In words, the name of a task reference now consist of the name of the application perspective in which the referenced task resides, followed by a dot and the name of the referenced task. For example, a task reference can now be `application3.task2`, referencing `task2` from application perspective `application3`.

The conversion from the visual perspectives to the generator model (see Chapter 6) is modified in such a way that all tasks are registered according to this scheme when handling the application perspectives.

Using this new notation, individual tasks in sub-applications can be referenced in the load perspective and mapping perspective.

Multiple occurrences of a single sub-application
The third problem that is caused by the introduction of hierarchy is that special action has to be taken to allow multiple super tasks to call the same sub-application. The steps taken to remove hierarchy have no problem with multiple calls to the same sub-application, because step 1 of replacing a super task with its sub-application makes sure that a fresh (and uniquely named) copy of the sub-application is created for each super task calling it.

However, this causes the situation where single task in a sub-application is converted to multiple tasks in the generator model. Take the hierarchical model depicted in Figure 38 for example. It has two super tasks that call the same sub-application. This sub-application contains one task ('Sub'). When we remove the hierarchy from this hierarchical model, we get the model depicted in Figure 39. We can see that the single task 'Sub' occurs twice in the resulting model, one time as 'Sub_sub_SuperTask1' and one time as 'Sub_sub_SuperTask2'. Note that this example has a sub-application with no parameters.

When we refer to 'sub.Sub' from either the load perspective or the mapping perspective, we have to make sure that this reference influences *both* 'Sub_sub_SuperTask1' and 'Sub_sub_SuperTask2'. If we for example specify a load for 'sub.Sub', the specification should be applied to both 'Sub_sub_SuperTask1' and 'Sub_sub_SuperTask2'.

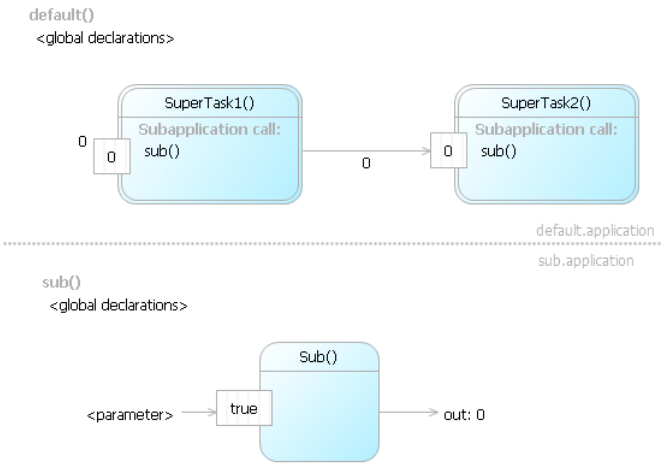


Figure 38: Example model with two super tasks calling the same sub-application

To do this, we modify the central mapping introduced in Chapter 6 to store multiple tasks for a

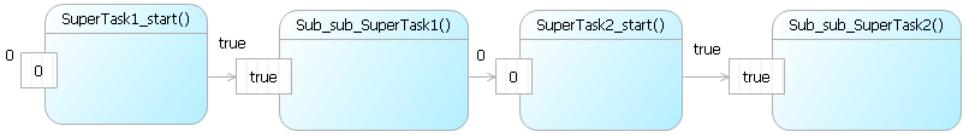


Figure 39: Example model with two super tasks calling the same sub-application, where hierarchy has been removed

single key. When processing the application perspective in the example, both 'Sub_sub_SuperTask1' and 'Sub_sub_SuperTask2' are stored in the central mapping under the key 'sub.Sub'. The following changes are made to the generation of the generator model.

- When adding a scheduler to a task referenced by the name s in the mapping perspective, this scheduler is added to all tasks registered under the key s in the central mapping.
- When adding load to a task referenced by the name s in the load perspective, this load is added to all tasks registered under the key s in the central mapping.
- When encountering a handover from a task referenced by the name s_1 and a task referenced by the name s_2 in the load perspective, this handover is processed as if it were from task t_1 to t_2, for each t_1 in the central mapping for key s_1 and each t_2 in the central mapping for key s_2.

8 Comparison with other languages

This section presents a comparison of the graphical language for DSEIR with a well-known general-purpose modeling languages. The reason that the language is compared to general-purpose languages, is because it is the first existing Y-chart-based language, so there are no other languages in the same domain to compare it to. The visual representation of the domain language will be compared to the language for Colored Petri Nets (CPN) ([5]), the Unified Modeling Language (UML) and (Uppaal?, EPCs?).

A tool is picked for each of the languages that the graphical editor is compared to. Of course, it is unfair to compare the tools developed by teams of people over several years with a tool developed in less than six months by one person, so the main focus of the comparison will be the language supported by the tool.

Because we are comparing a domain-specific language with general-purpose languages, the focus is on how models can be made for this specific domain.

8.1 CPN

For the comparison to the CPN language, the 'CPN Tools' tool is used as a reference. This tool is picked because it is free, it has been around some time, it is under active development, and it provides a large set of features. It is also used by the Octopus toolset for simulations.

The following advantages of the CPN language over the VDSEIR language are identified:

- **Clear semantics**
  The CPN language has very clear semantics. There are only two key elements, places and transitions, and their interaction is made very explicit.

- **Hierarchy expressiveness**
  A sub-net in a CPN model can have multiple entry points and multiple exit points. Opposed to task hierarchy in VDSEIR, input tokens can arrive at a sub-net at arbitrary moments in time and output tokens can leave a sub-net at arbitrary moments in time.

- **Expressiveness**
  The use of a functional language to annotate CPNs allows for a lot of expressiveness. The functional language has support for multiple high order data structures, such as lists, sets, bags, records, tuples, etc. It also allows the user to specify custom data structures that can be combinations of other data structures. Furthermore, CPN Tools allows users to specify functions that can be called from expressions in CPNs.

  Unfortunately, the expressions used in the domain language are not as expressive. The domain language can of course be extended with custom data structures and custom functions, but this requires that the core toolset should be changed as well, which is outside the scope of this thesis.

Concludingly, the CPN language is very expressive, with only a small number of concepts.

The following list show some advantages of the VDSEIR language over the CPN language:

- **Perspectives**
  A VDSEIR model consists of five key perspectives. These perspectives provide a guidance in the modeling process of the users of the language. The users are motivated to think of the application and the platform as two independent entities and the perspectives allow the user to follow the Y-chart approach very closely.

- **Concepts with a higher abstraction level**
  The VDSEIR language has a lot more concepts than the CPN language, with resources, services, schedulers, etc. This can be both good and bad. It does give the language a more steep learning curve for people not used to these concepts, but users that are used to these concepts will find that the language is more closely related to their point of reference. Independent of the learning curve, the concepts with a higher abstraction have their advantages. By having resources for example, users of the toolset are not forced to model each resource using tasks and places, allowing the user to focus more on the important aspects of the model they are creating.

  Furthermore, the abstract concepts can also help making the model more readable. Take for example a CPN net, describing two tasks that have to be executed. The tasks both require the same two resources. A typical CPN model would look like the one shown in Figure 40. As we can see, this model requires two arcs for each task using a resource, requiring a lot of arcs, even in this small example. Now imagine that more tasks are added, requiring more resources. This will cause a lot more arcs to be added and this will reduce the readability of the model significantly. In the VDSEIR language, the readability of such an example would be much better, because of the 'resource' concept. The VDSEIR application perspective, load perspective, and resource perspective for the same example can be found in Figure 41, Figure 42, and Figure 43 respectively.

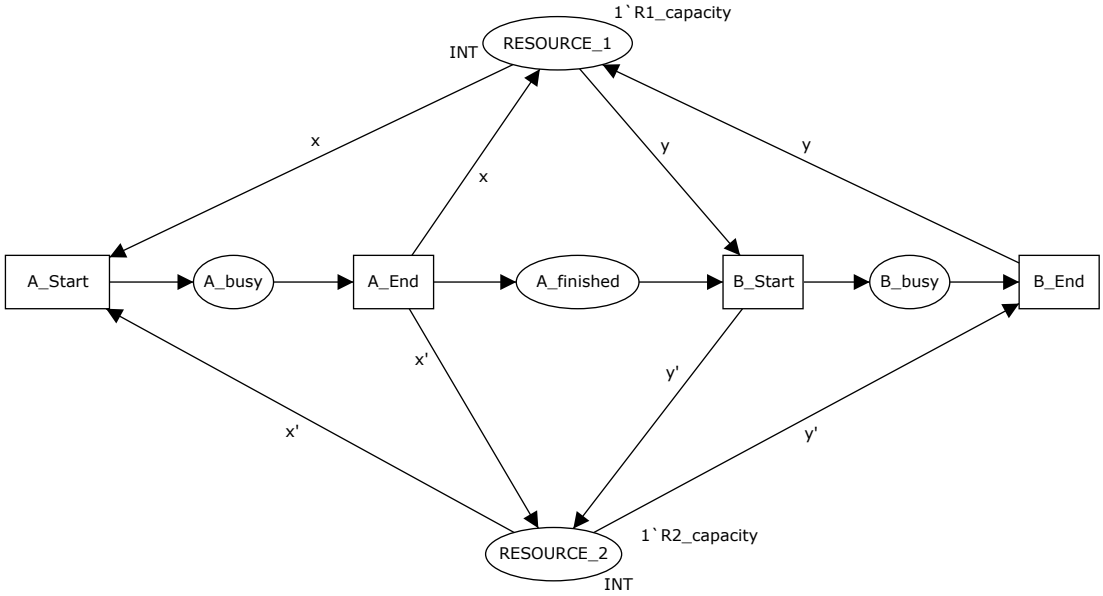


Figure 40: Typical CPN model of two tasks using two resources

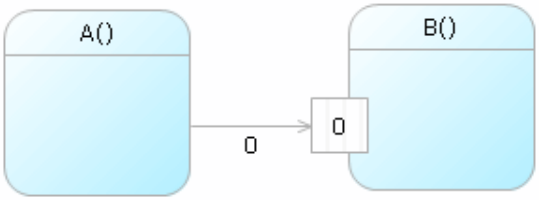


Figure 41: Application perspective of two tasks using two resources

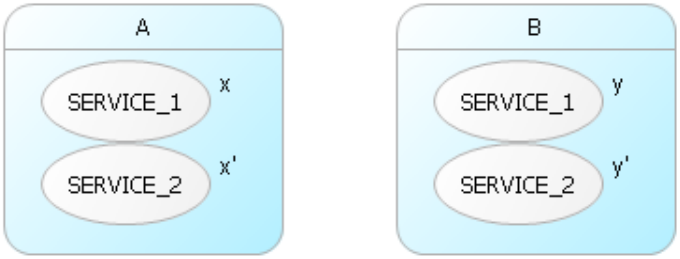


Figure 42: Load perspective of two tasks using two resources

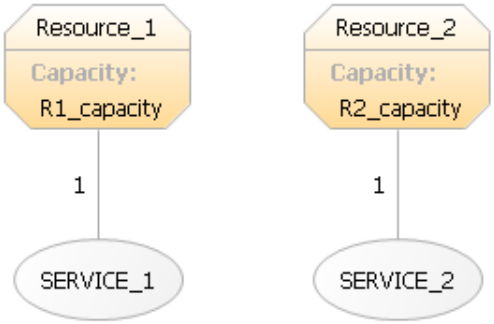


Figure 43: Resource perspective of two tasks using two resources

- **Hierarchy more suitable for workflows and pipelines**
  Although the hierarchy in the CPN language is very expressive, it also has disadvantages. The semantics of the CPN hierarchy do not allow to treat a sub-net as a 'function call', which is what happens in VDSEIR. Having such 'function call' semantics allows to clearly specify the information that flows in a sub-application and the information that flows out of the sub-application. In CPN Tools this information flow is not even visible at one abstraction level above the sub-net, as can be seen in Figure 44. The arc inscriptions are only visible in the sub-net.

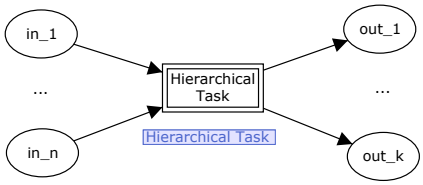


Figure 44: Top level view of a hierarchal CPN model does not show arc inscriptions

- **Ability to swap perspectives and sub-applications**
  Because the VDSEIR language uses decoupled perspectives, it is very easy to change the set of perspectives that together define the VDSIER model. One could for example model multiple alternatives for the load perspective and easily switch between those perspectives when desired. The same holds for sub-applications. One could model multiple alternatives of a sub-application and then switch between the alternatives simply by changing the sub-application call(s) in the super tasks that use the sub-application.

8.2 UML

As a reference tool for UML, the free 'ArgoUML' tool is picked ([1]).

Characteristics of UML:

- **Diagrams**
  Similar to the perspectives in VDSEIR, UML has several diagrams. Each diagram contains a piece of the information about the whole system that is being modeled. Opposed to VDSEIR, there exists some overlap in what the different perspectives express. For instance, we have an activity diagram and a state machine diagram, which can both describe application flow and state information.

- **Semantics**
  Although there is a standard that defines the UML modeling language (managed by the Object Management Group), different users across industry have been using their own semantics for UML models. Hence, differences in interpretation exist, making it hard to spread UML models across companies or even across departments of the same company.

- **Code generation**
  Most UML tools support automatic code generation based on the created models, in-

cluding ArgoUML. This functionality is not yet available for VDSEIR models (although it is certainly possible to add this feature). For our domain it might be hard to make good use of this feature, because we can have different application logic running on different computational units. For instance, we could have a task running on a dedicated piece of software, opposed to a task running on a normal CPU.

Advantages of VDSEIR:

- **Design space exploration**
  The big advantage of the VDSEIR language over the UML language is that it allows simulations to be run on it. UML also has some support for simulations, but only for certain diagrams. We can use the simulation information to fine tune the VDSEIR models themselves, making VDSEIR a good choice if the architecture of the system is not yet determined. UML is better at specifying a system once it's architecture is known.

Now concerning hierarchy, the UML 'state machine' diagram can be used to model a hierarchical state machine. It can have normal states and transitions, but it can also have states that contain other states, see Figure 45, where we have a state called 'Super state' that contains two other states.

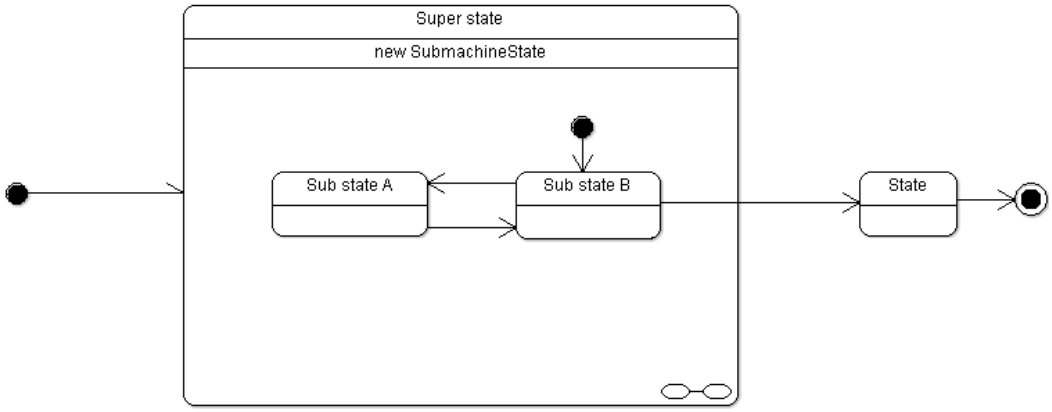


Figure 45: Hierarchy in a state machine diagram in UML, using the ArgoUML tool

As we can see, the sub-states of a super state are depicted directly in the graphical representation. This has the advantage that arbitrary sub-states can easily be coupled to other states outside of the super state. On the other hand, because there is no way to visually collapse a super state, we still have information about all abstraction levels inside the same model. Furthermore, this mechanism does not allow to re-use sub-state configurations, like having multiple calls to the same sub-application.

8.3 Uppaal

The Uppaal formalism is only supported by the Uppaal tool, so this tool is picked for reference. The Uppaal tool is also used by the Octopus toolset and VDSEIR models can be converted

to Uppaal models.

Advantages of Uppaal:

- **Parallel components**
  Uppaal allows to model multiple parallel components with communication channels in between them. When expressing applications or platforms with a similar architecture this can be really convenient. If the application logic for instance is not a pipeline or workflow, but a number of computational units communicating with each other, Uppaal can be better suited.

- **Templates**
  A model drawn in Uppaal is merely a template, depending on some parameters. In the global system declaration one needs to instantiate templates by supplying values for these parameters. So for instance, one can model a single resource and then instantiate it multiple times, each time with a different capacity. This behavior can be slightly mimicked by calling sub-applications in multiple places, each time with different parameters. Later on in this paper (when we talk about model parameters in Chapter 10, we see how we can even further mimic this behavior.

Advantages of VDSEIR:

- **Higher abstraction level**
  As with the comparison with CPN, the constructs in VDSEIR are of a higher abstraction level. The same advantages (and disadvantages) hold in comparison with Uppaal as well.

  Furthermore, while modeling in automata, systems can quickly become very complex, because everything is modeled at the level of states. In VDSEIR, we can model elements in a more abstract manner, hiding the small details.

- **Support for stochastic processes**
  Uppaal does not allow its user to model stochastic quantities, but VDSEIR does.

9 Analysis framework

In the previous sections we have seen the perspectives that are used to generate specifications in the domain language and we have seen how these specifications in the domain language are actually created. This means that users of the toolset will now be able to create graphical models and generate DSEIR models from these graphical models. However, the users are not yet able to use these generated DSEIR models for analysis. This sections will introduce an extensible framework in which users can specify what they want to do with the models they create. The first part of the section will explain the design goals of the framework, followed by the explanation of the framework itself.

9.1 Design goals

The introduction already introduced the concept of an extensible framework to allow users to apply analysis techniques offered by the toolset on their models. The design goals of this framework are identified as follows:

- The framework should be extensible, meaning that users (and developers for) the toolset can add functionality to the framework, even after deployment of the toolset. We want the framework to be extensible for two major reasons. The first reason is that the toolset is still under development and new analysis techniques and methods are likely to be added to the toolset, so we want a way to add support for this new content easily. The second reason is that we want to give advanced users of the toolset the possibility to extend the toolset themselves, so that they can perform any analysis they want.
- The framework should be housed in the graphical editor. We want users that do not have any experience with programming to be able to access all features of the toolset, so the users should be able to invoke any functionality they want from the toolset right from the graphical editor.
- The framework should allow analysis techniques to be chained, i.e. we want the user to be able to specify that the results of one analysis technique should be used for another analysis technique. For example, we want the user to be able to specify that first a simulation has to be performed and that the results of this simulation have to be used to create a performance report.

9.2 The framework

General idea
The chosen framework satisfies all three design goals from previous section. The idea of the framework is as follows. We have a set of *plug-ins* that are linked with *transport arcs*. In our framework, a plug-in is a piece of code that can produce a set of output objects, given a set of input objects. For example, we can have a simulator plug-in that takes a DSEIR model as input and that produces an event trace. A transport arc allows an output object produced by a plug-in to be offered to another plug-in as an input object.

To be able to steer the behavior of a plug-in, a plug-in acts on a set of *properties*. These properties are also an input of the plug-in when producing output objects. Each plug-in also provides a means for the user to set its properties. See Figure 46.

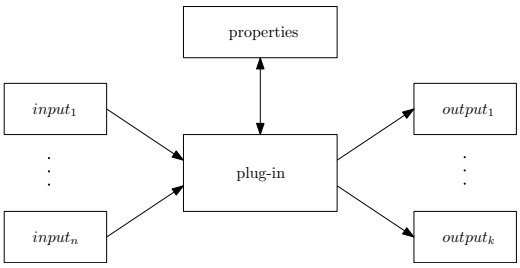


Figure 46: A plug-in

Now that we know the general idea of a plug-in, we define the most important things a plug-in should offer to the framework:

- A number of input *connectors*. A connector is a tuple consisting of a name and a type, which are both strings. The name is for the users of the toolset, so they know what that particular connector does. The type of the connector is a string describing what type of data it acts on. A connector can have an empty string as its type, indicating that it acts on data of *any* type. The input connectors describe the objects that are required for the plug-in to produce its output objects. The previously mentioned simulator plug-in for example has one input connector with name 'Model' and type 'Model'.

- A number of output connectors. These output connectors describe the objects that are produced by the plug-in. For example, the simulator plug-in has one output connector with name 'Trace' and type 'Trace'. Now, if another plug-in has an input connector with type 'Trace' as well, a transport arc can be created between this input connector and the output connector of the simulator.

- Code that, given the input objects, produces the output objects of the plug-in. This code can be seen as the 'functionality' that is added to the framework by the plug-in. In case of the simulator plug-in, this code will execute a simulation on the input model. After this simulation is finished, its event trace will be extracted and the output object of the plug-in will be produced. This code can also have side-effects, such as a dialog that pops up or a file that is created (which is especially relevant for plug-ins with no outputs).

- A set of default properties as well as code to create a dialog for the user to modify the properties of the plug-in. The default properties are used to create new instances of the plug-in. The code that creates the dialog is used when the user wants to modify the properties of a plug-in (instance). When executed, the code should create a dialog with text fields, radio buttons, etc. When the user makes changes in this dialog, these changes should be propagated to the properties of the plug-in.

- Code to validate the current properties of the plug-in, to be able to check whether the current properties of the plug-in are making sense. A file-loading plug-in with a

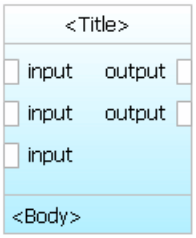
property indicating the path of the file can for example have code to check whether the path is valid.

- Code to check whether the plug-in can act on a set of input objects. With this code, the plug-in can accept or reject a set of input objects, based on information about these objects (e.g. their type) and information that resides in the input objects.

Note that the type of the connectors is only there to make sure that a transport arc can only be created between an output connector and an input connector if they have the same type or if either of them has no type.

Concludingly, we have plug-ins, which provide some properties and functionality and we have transport arcs, transporting output objects from one plug-in to the inputs of other plug-ins. We will use the name *experiment* to refer to a collection of plug-ins and transport arcs.

Visual representation
Because the framework should be housed in the graphical editor, a visual representation of a The visual representations of an experiment will be as follows:

Plug-in	
	This node represents a single plug-in instance. On the left it has its inputs and on the right it has its outputs. It has a title so that the users of the toolset know which plug-in it is and it has a *body*. The body of a plug-in is a string representing its current properties. Double clicking this node brings up a dialog to modify the properties of this plug-in.

Transport arc	
	This arc represents a transport arc from an output of a plug-in node to an input of another plug-in node.

Execution
Now that we know what an experiment is and how it looks, we will explain how an experiment can be executed. By executing, we mean that we want the plug-ins in the experiment to start working. If we for example have the experiment shown in Figure 47, we want `Plugin_1` to start (because it requires no input). After `Plugin_1` finishes, we want `Plugin_2` to be executed with the output of `Plugin_1`. Finally, if `Plugin_2` is finished, we want `Plugin_3` to start with

the output of `Plugin_2`.

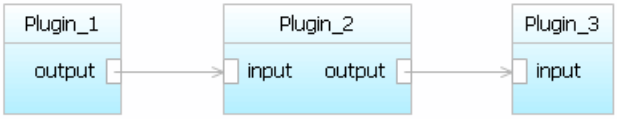


Figure 47: An example experiment

In order to execute an experiment, we first accumulate all input connectors specified by all the plug-ins that are present in the experiment. We call this set of input connectors I. Second, we create an empty 'First In, First Out' (FIFO) buffer that can hold any data object for each $i \in I$ and we will use buffer(i) to denote the buffer that was created for input connector i.

We will now present the execution semantics of an experiment. Executing an experiment is done using the following algorithm (Algorithm 2). Note that by executing a node with certain input objects, we mean executing the code it has to produce its output objects from these input objects.

Algorithm 2 ExecuteExperiment()

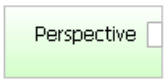
```
Start execution (with no input objects) of all plug-ins with no input connectors.
while plug-ins are being executed do
  Wait for the first plug-in to finish; let p be the this plug-in.
  updatedPlugins ← ∅
  for all output connectors o of plug-in p do
    for all transport arcs from o to an input connector i do
      Add the output object produced by p for output connector o to buffer(i)
      Let p' be the plug-in to which i belongs
      updatedPlugins ← updatedPlugins ∪ {p'}
    end for
  end for
  for all p' ∈ updatedPlugins do
    if buffer(i) is not empty for each input connector i of p' then
      input ← ∅
      for all transport arcs from o to an input connector i do
        Move the first object in buffer(i) to input
      end for
      Start execution of p' with input
    end if
  end for
end while
```

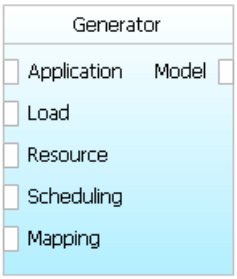
The algorithm consists of an initialization phase, followed by a main loop. In the initialization phase, all plug-ins that do not require any input, i.e. those without input connectors, are started. After that, the algorithm keeps looping until no plug-in is running anymore. Each time a plug-in finishes, its produced output objects are moved along the transport arcs, ending up in a buffer for an input connector. Then it can be the case that a plug-in can be executed because of this newly produced and moved data. Each plug-in that has data in the buffer of

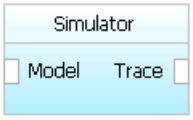
each of its input connectors then consumes the first object out of each of these buffers and starts execution with these objects.

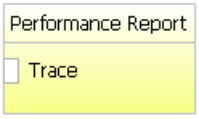
9.3 The plug-ins

We have described the extensible framework, what an experiment is, what an experiment looks like and how an experiment can be executed, but we haven't shown any plug-ins that can be added to an experiment. To provide some basic functionality to the users, a basic set of plug-ins is developed.

Perspective	
	In order to do anything with VDSEIR models, we need a way to load a perspective. The perspective plug-in allows the user to specify a perspective file (i.e. an application perspective, a load perspective, etc.). Upon execution of the plug-in, the selected file is loaded and forwarded through the perspective output connector.

Generator	
	The generator plug-in converts a combination of an application perspective, load perspective, resource perspective, scheduling perspective, and mapping perspective into a specification in the DSEIR language (see Chapter 6). Important to note is that no system perspective is required. Instead, an implicit system perspective is used. This implicit system perspective simply contains the perspectives connected to the input connectors of the plug-in (so only one file per perspective). It has a property to set the name of the generated model.

Generator	
	The simulator plug-in takes a DSEIR model and simulates it, creating a *trace*. A trace contains all information about a run of the model, including information about the start times and end times of the tasks, as well as resource occupation.

PerformanceReport	
	This plug-in takes a trace and creates a human-readable performance report of it. The performance report is stored in a file.

Given these plug-ins, we can for example have the experiment indicated in Figure 48. This example experiment creates a DSEIR model, simulates it and then creates a performance report of the trace generated by the simulator.

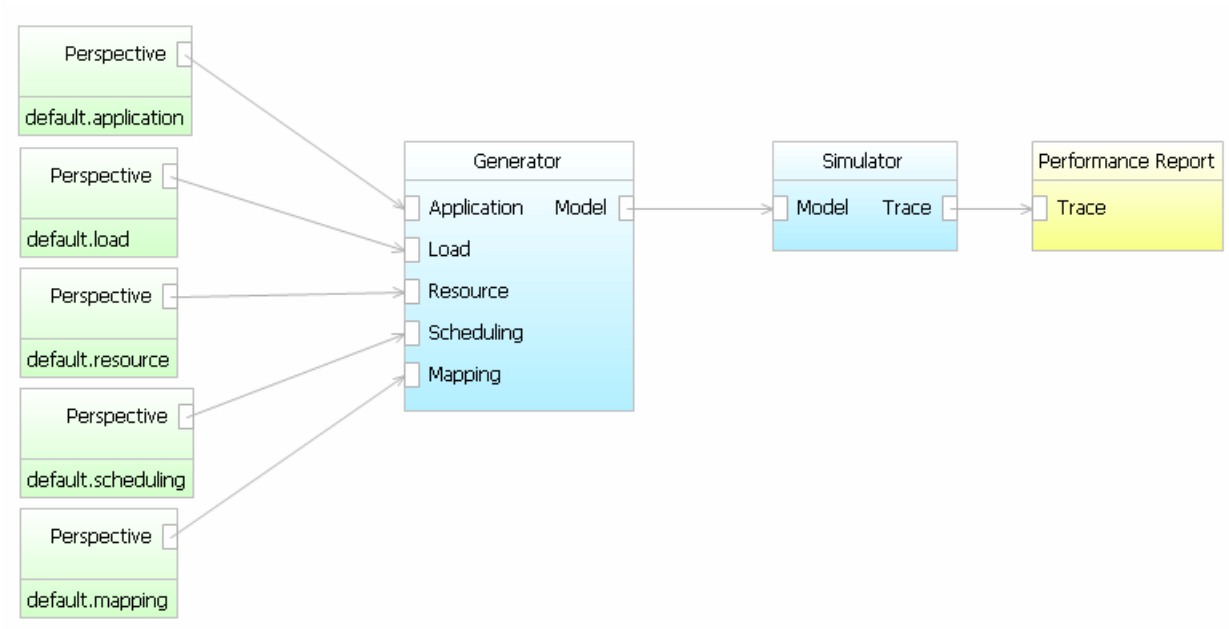


Figure 48: An example experiment

10 Model parameters

The analysis framework described in section Chapter 9 allows us to easily generate a model in the domain language based on a particular set of perspectives and to apply analysis on the generated model. However, the framework does not allow us to use simulation to for instance find the best capacity of a resource or the best resource to use in the platform. If a user for example wants to find out the best capacity for a resource, he or she has to set the capacity to a particular value, run the experiment, and check the results. After that, the user changes the capacity to another value and repeats the process to see whether this new value is better.

To ease this process of having to manually change certain model values, we introduce the concept of a *model parameter*. The idea behind a model parameter is as follows. We have a set of parameters that we allow all perspectives to refer to in expressions. In our experiment, we assign values to these parameters and when we generate a domain specification for the perspectives, we replace all references to the parameters by their values.

10.1 Domain specification generation

To implement this behavior, we resolve the model parameters when we convert the perspectives to a generator model (see Chapter 6). Because we resolve the parameters in this step, we do not have to change the graphical representations and we do not have to change the transformation from the generator model to the domain model. The only thing that happens is for each expression set in the generator model, we take the expression from the visual model and replace all occurrences of parameters by their values. See Figure 49.

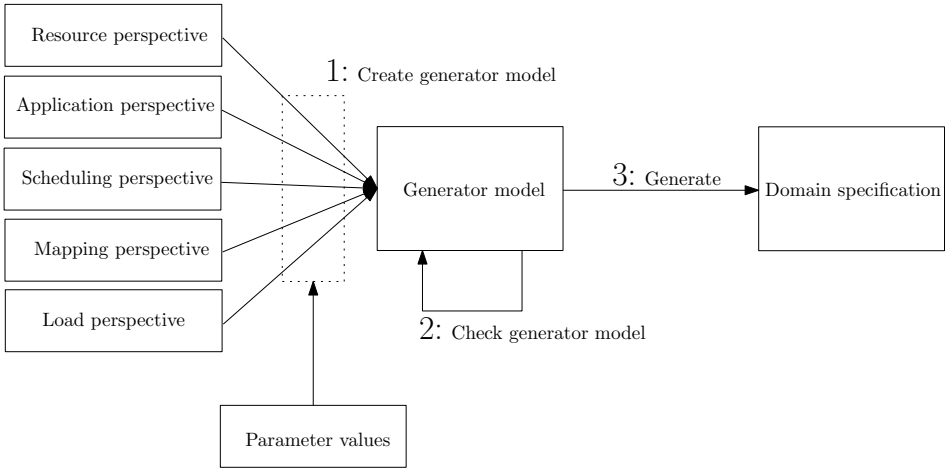


Figure 49: Domain specification generation with model parameters

10.2 Generator plug-in

Now we know *how* the model parameters are resolved, but we do not have a place *where* the parameters and their values come from. This place will be the generator plug-in. We change

this plug-in in such a way that we can specify model parameters in its properties dialog, see Figure 50.

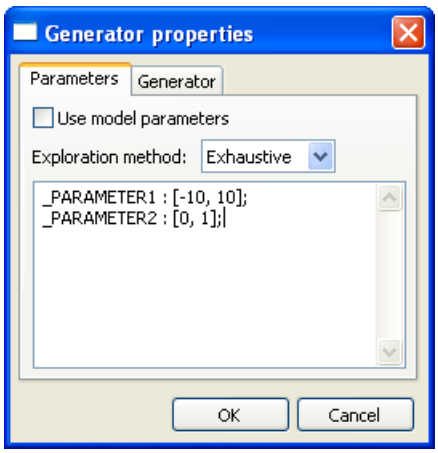


Figure 50: Properties dialog of the generator plug-in

We add a property to indicate whether we want to use model parameters and we add a property specifying wether we want to generate a single DSEIR model with random parameter values (the 'Random' exploration method), or that we want to generate a DSEIR model with all possible combinations of parameter values (the 'Exhaustive' exploration method').

There is also a text area in which the users can specify the model parameters together with the interval describing the possible values of the parameter. The textual representation consist of one or more concatenated *model parameter declarations*.

Representation:	<name> ':' '[' <min> ',' <max> ']' ';'
Explanation:	This declares a single parameter with `name` as its name. The values of this parameter all come from the interval [`min`, `max`] (both inclusive).
Example(s):	`X: [0, 2];` (parameter `X` with possible values 0, 1, and 2) `Y: [4, 4];` (parameter `Y` with only one possible value: 4)

In order to allow the users of the toolset to distinguish the generated models, the name of any generated DSEIR model is changed into the concatenation of the name provided by the user (as a property of the generator plug-in) and a string showing the parameters with their values. If 'model' is the name provided by the user and we have parameters 'A' and 'B' with values '0' and '3' respectively, the resulting name of the DSEIR model will be 'model[A=0,B=3]'.

Now, if the user indicates that he or she wants to use random parameter values, a random value is picked for each of the model parameters and these values are used to generate a single domain specification.

If the user indicates that he or she wants to use an exhaustive exploration, we want the generator plug-in to keep generating DSEIR models with permutations of the parameter values, until all of them have been generated. We do not want all the models to be created at once, because it is highly likely that not all models fit in memory. In addition, we want

the plug-ins that come after the generator to be able to already start working once the first model has been generated. Because the analysis framework allows plug-ins to produce output one time per input, we need to change the execution semantics of the framework a bit.

10.3 Analysis framework

In order to allow a plug-in to produce multiple outputs based on a single input only, we introduce the notion of a *streaming* plug-in. A streaming plug-in indicates to the framework that it can produce multiple outputs for each input.

The execution semantics of the framework change as follows. When input I arrives at a streaming plug-in, the plug-in is added to a global set of streaming plug-ins. The plug-in is then executed with input I. After execution of the streaming plug-in, all plug-ins are executed normally, until no plug-in can be executed anymore. After that, if there are streaming plug-ins in the global set, one of those streaming plug-ins is executed again and the process repeats. Once a streaming plug-in does not produce any data (it returns `null`), it is removed from the global set of streaming plug-ins. Using these semantics, the framework aims to do everything possible with the available data, before executing streaming plug-ins to generate new data.

Algorithm 3 shows an adaptation on Algorithm 2. This new algorithm is able to cope with streaming plug-ins.

Algorithm 3 ExecuteStreamingExperiment()

```
streaming ← ∅
Start execution (with no input objects) of all plug-ins with no input connectors.
while true do
  while plug-ins are being executed do
    Wait for the first plug-in to finish; let p be the this plug-in.
    if p is a streaming plug-in then
      streaming ← streaming ∪ {p}
    end if
    Process the output of p and start plug-ins that can be started.
    if p is a streaming plug-in, but it did not produce data then
      streaming ← streaming \ {p}
    end if
  end while
  if streaming ≠ ∅ then
    Let p be a plug-in such that p ∈ streaming.
    Start execution of p
  else
    Finish the experiment
  end if
end while
```

Using these new execution semantics, the generator plug-in can exhaustively generate DSEIR models for all permutations of the model parameters, based on a single input.

11 Error reporting

When creating VDSEIR models, an error can easily be made, for example one could make a typo in one of the names of the tasks in the load perspective. The plug-ins in the experiment framework can also cause validation errors or runtime errors. To give good feedback to the users of the toolset, an error reporting mechanism is implemented.

Basically, all errors originate in plug-ins, because all functionality of the toolset is addressed from an experiment. Hence, errors can be found only when running an experiment. A plug-in can cause two types of errors, namely *validation errors* or *runtime errors*. Validation errors occur when a plug-in has an ill-configured set of properties or when it is unable to run on its given input data. Runtime errors occur in between the start and end of a plug-in.

We extend the framework in such a way that the plug-ins can create both these errors via exceptions. When a plug-in creates an error (validation or runtime), we create an error using the resource marker mechanism designed by Eclipse, so an error will be created in the standard problems view of the graphical editor, see Figure 51.

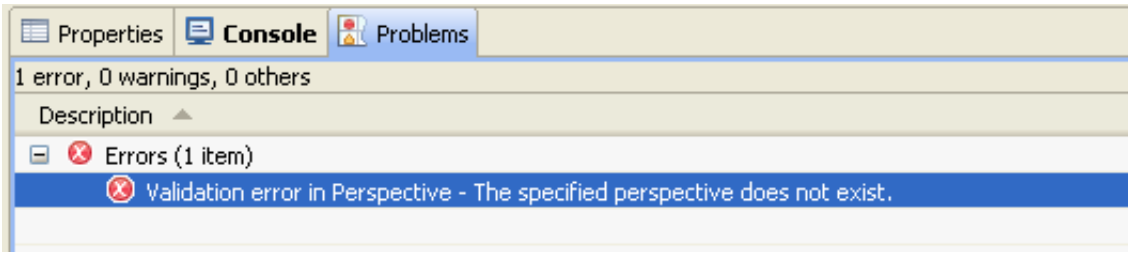


Figure 51: A validation error in the perspective plug-in

Now, when the user sees such an error, he or she needs to go trough all the perspective plug-ins in the experiment to find out which one has a wrong file reference. To allow the user to conveniently navigate to the source of this error, we do something special. When a plug-in creates an error, we capture this error in the framework and we add information about the originating plug-in in the error (e.g. the id of the visual node belonging to the plug-in). When the user then double clicks the error, this information is used to open the experiment with the faulty perspective plug-in and then this faulty plug-in is automatically selected. So when we double click the error from Figure 51, we get the situation depicted in Figure 52.

For these types of errors this mechanism works good, however we run into problems when the error is not really in the experiment. This applies especially to the generator plug-in. If the generator plug-in encounters and error in for example the load perspective while generate a domain model from the five perspectives, the user should be forwarded to the load perspective when double clicking the error; he or she should not be forwarded to the generator plug-in.

So, say we have a load perspective with a reference to a task that does not exist. This will give the error shown in Figure 53.

What we want to see after double clicking the error is that the load perspective is opened and that the cause element is selected, as depicted in Figure 54.

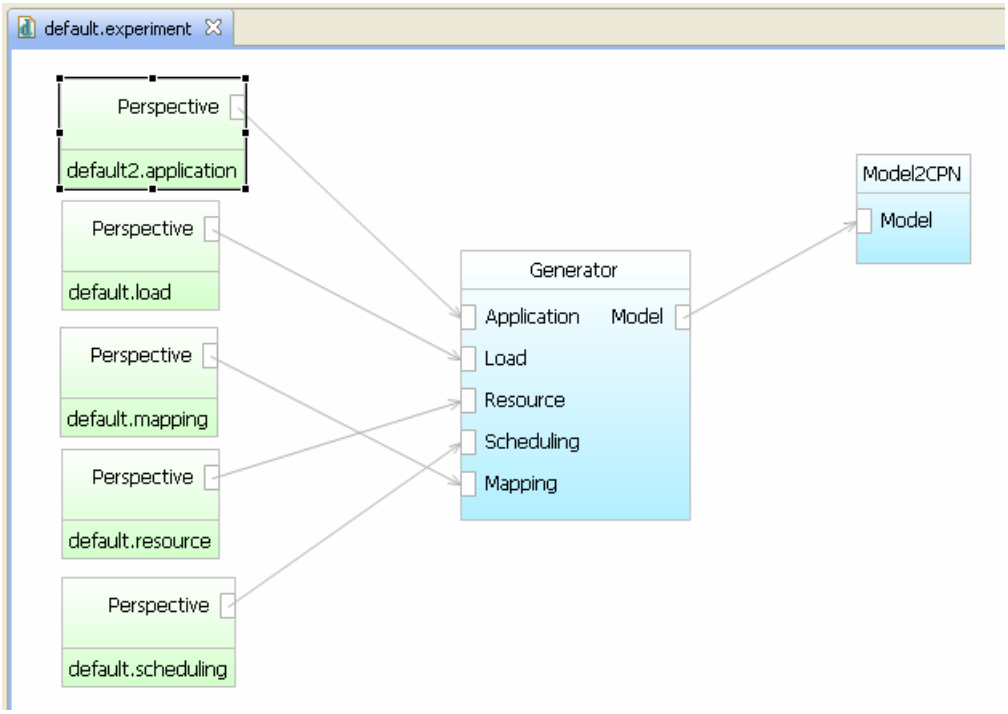


Figure 52: The result of double clicking the error from Figure 51; the user is forwared to the faulty perspective plug-in

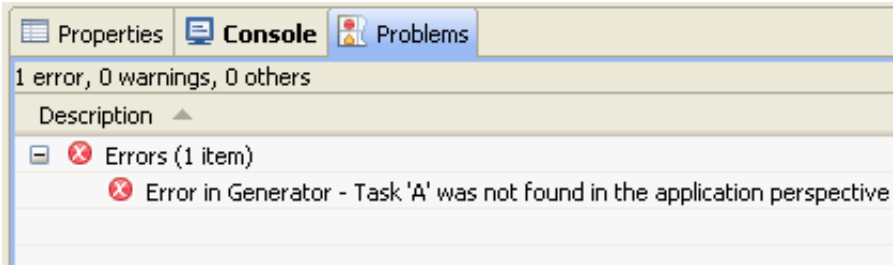


Figure 53: A load perspective (runtime) error, caused by the generator plug-in

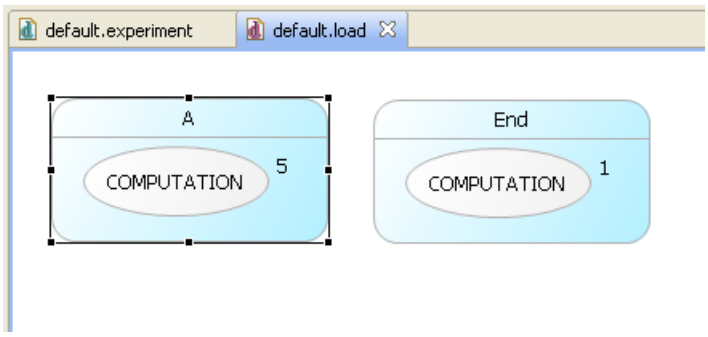


Figure 54: The result of double clicking the error from Figure 53; the user is forwared to the faulty load perspective and the faulty element is automatically selected

To enable this behavior, we allow plug-ins to supply additional information about the location of the error when the cause an error (this information is equal to the file in which the error occurs, together with the id of the faulty visual element). We extend the double-clicking behavior of the errors to automatically navigate to this location if it is supplied.

This way, the plug-ins can forward errors to the end-user in a nice manner and by supplying location information, they can let the users navigate automatically to the error's cause. The only problem is that information about the location of the cause is not always present (for example when encountering errors in places where the visual models are not known anymore, i.e. when encountering an error in the generator model described in Chapter 6), so if that is the case, the error will not have a intuitive double-click behavior; it will only point to the plug-in that caused the error. Hence, it is up to the plug-ins to make sure that they can always supply enough information in their errors.

12 Conclusion

Accomplishments
Using model-driven engineering and by making use of the GMP framework in Eclipse, we have been able to automatically generate a complete graphical editor for the DSEIR domain language. The visual representation of the DSEIR models has been developed in such a way that we have several perspectives which together define a single DSEIR model, following the Y-chart approach. Each individual perspective is independent of the other perspectives and can therefore easily be swapped for another perspective and interchanged between different users of the editor.

Because of the generative nature of the approach, any changes in the visual representation can be incorporated in the graphical editor simply by regenerating its source code. Hence, any future changes to the underlying toolset and any extensions to the visual representation can rapidly be added to the graphical editor, making it a robust solution.

The visual representation of the DSEIR models does not only allow to express everything that can be expressed in the domain language, but it also adds additional expressiveness in the form of task hierarchy. The system with super tasks and sub-applications allows us to build models with different abstraction levels at different levels in the hierarchy.

The extensible framework that has been developed add a easy way for users to interact with the toolset. Its extensible nature allows developers to easily add additional expressiveness to the framework.

Impact
Because a visual representation has been developed for the domain language, users of the toolset are no longer required to have programming skills to use the toolset. This opens up the toolset to a broader range of users. This can imply that the expressiveness of the toolset has decreased, but because of the extensible framework, advanced users that want to use programming to perform special actions are still able to do so. Hence, the toolset offers everything it had to the advanced users, while offering a low step-in visual representation for less advanced users.

A second advantage of existence of a visual representation for the domain language is communication. Before any visual representation existed, users of the toolset communicated their models by means of source code, which is way harder to read than a model. Even when communicating a certain case, it is much clearer to provide a model than to provide a textual description. As a side note, the visual representation for the toolset is already used for explanation purposes in reports of colleague students. So we see that the presence of the visual representation already eases the way users of the toolset communicate their models to other users.

Future work
The toolset and the graphical editor can benefit from some improvements, including the following ones:

- More plug-ins - The extensible framework has only a small set of plug-ins to generate

DSEIR models and to do something with the generated DSEIR models. Not all functionality of the toolset is captured in the form of a plug-in, so adding more plug-ins can lift the expressiveness of the graphical editor to the level of the toolset. Later on, as new functionality comes to the toolset, this functionality can be added to the framework in the form of a plug-in as well.

- Experiment hierarchy - An experiment currently is a flat view of plug-ins linked via transport arcs. It would be nice if we can have a plug-in that consists of several other plug-ins connected with transport arcs, so that we can combine low-level plug-ins into plug-ins with a higher abstraction level.

- Model-to-code generation for plug-ins - Right now, a plug-in definition has to be created manually by the user of the toolset. This definition has information about the plug-in's name, input connectors, output connectors, etc. This information can perfectly be visualized by a model (for example a single plug-in node to which input and output connectors can be attached and properties of this node can be set). If we have such a visual model of a plug-in's signature, we can also have a model-to-code transformation that creates skeleton code for the plug-in.

- Error reporting - Not all errors that occur while running an experiment, are clear to the end-user, especially those from the generator. Ideally, when a user gets a notification of an error, he or she should be able to jump to the place in the visual representation where the error is found. The problem, however, is that the information about the visual representation is lost when we convert the perspectives to a generator model (Chapter 6). When we encounter an error in the generator model (for example a parsing error), we do not know the visual element corresponding to the element in the generator model. To be able to trace the problem origin back to the visual model, we can for example maintain a mapping from model element's names to their visual element.

- Model-time consistency checking - Right now, the consistency between the perspectives is only checked when we combine the perspectives via the generator plug-in. The users of the toolset could benefit from model-time consistency checking, which would alert them immediately if they for example have a reference to a non-existing task in the load perspective.

Reflection

The model-driven-engineering paradigm is well-suited to design a domain-specific language and to create a graphical editor for this domain-specific language. Eclipse provides a fully functional and expressive framework to do this. A simple graphical editor can be generated for a particular domain model in a short amount of time. The drawback, however, is that this simple graphical editor is not good enough and some special actions have to be taken to customize it.

The EMP framework houses facilities to create a meta-model of the domain-specific language, which is a good way to describe the elements and their constraints of a visual representation. The GMF framework takes this meta-model of the domain-specific language to generate a graphical editor, which is intuitive, but there is one drawback. The GMF framework *only uses the meta-model of the visual representation.* In our case, the real domain differs from

it's visual representation, but there is no way in GMF to have a separate domain model and a visual representation model. Therefore the transformation from the visual models to the domain models had to be manually made (Chapter 6). The domain-specific language expertise field would greatly benefit from a clear separation of the domain model and its visual representation.

Finally, the framework lacks a central place with good documentation and its own editors (especially the editor for the graphical definition model, see Chapter 5) are not what one would expect from a framework specialized in graphical editors. The automatic code generation on the other hand heavily outweighs these downsides.

All in all, the model-driven-engineering approach for domain-specific languages is supported by existing tools in a sufficient enough way to create real graphical editors, but the tools themselves could still be improved.

A Case study

For confidentiality reasons, this chapter is hidden from public view.

References

[1] ArgoUML. `http://argouml.tigris.org/`.

[2] JavaCC. `http://javacc.java.net//`.

[3] T. Basten et al. Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In T Margaria and B Steffen, editors, *4th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2010, Heraklion, Crete, Greece, Proceedings, Part I*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010.

[4] R.C. Gronback. *Eclipse modeling project: a domain-specific language toolkit.* The Eclipse series. Addison-Wesley, 2009.

[5] Kurt Jensen, Lars Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9:213–254, 2007.

[6] N. Trcka, M. Voorhoeve, and T. Basten. Parameterized Timed Partial Orders with Resources: Formal Definition and Semantics. Technical report esr-2010-01, Eindhoven University of Technology, Department of Electrical Engineering, Eindhoven, The Netherlands, 2010.

[7] B. van Dongen, A. de Medeiros, H. Verbeek, A. Weijters, and W. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 1105–1116. Springer Berlin / Heidelberg, 2005.