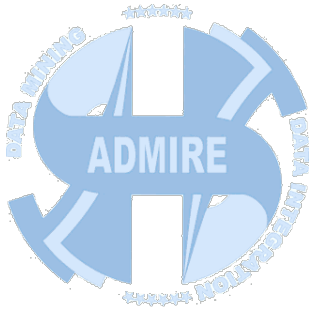
ADMIRE D1.5 textendash Report defining an iteration of the model and language: PM3 and DL3



ADMIRE D1.5 – Report defining an iteration of the model and language: PM_3 and DL_3

Project Title	ADMIRE
Document Title	Report defining an iteration of the model and language: PM_3 and DL_3
Deliverable Number	D1.5
Authorship	Peter Brezany, Ivan Janciak, Alexander Woehrer Carlos Buil Aranda, Malcolm Atkinson, Jano van Hemert
Document Filename	ADMIRE-D1.5-model-language-ontology.tex
Document Version	1.0
Distribution Classification	Project Internal
Distribution List	ADMIRE Project Team
Approval List	Amy Krause, David Snelling, Project Manager, Executive Board

Personnel	*Date*	*Comment*	*Version*
IJ	03 Aug 2009	First draft	0.1
CB	19 Aug 2009	Platform ontology	0.2
AW	24 Aug 2009	Process optimization	0.3
IJ	25 Aug 2009	Metamodelling	0.4
PB	26 Aug 2009	Executive Summary	0.5
IJ	11 Sep 2009	Review comments addressed	0.6
IJ	14 Sep 2009	Additonal comments addressed	0.7
RMB	16 Sep 2009	Final edit and signoff	1.0

Contents

1 Executive Summary

This document is the third deliverable to report on the progress of the model, language and ontology research conducted within Workpackage 1 of the ADMIRE project. Significant progress has been made on each of the above areas. The new results that we achieved are recorded against the targets defined for project month 18 and are reported in four sections of this document as follows.

Extend CRISP-DMI Model (Section 2). The first section briefly outlines a small extension of the CRISP-DMI model, originally presented in D1.1 [6]. This update to the model involves a new look at the provenance phase, plus we have added a new optimisation phase to the lifecycle of a DMI project.

In D1.1 we presented ADMIRE's high-level model for the data mining and integration process, CRISP-DMI, which specifies the phases that have to be applied in a typical DMI project. Based on our experience from recent data mining projects, we claim that the DMI system must support mechanisms for the optimization of DMI processes in each phase of the CRISP-DMI process to conduct high-productivity data exploration. Therefore, we have explicitly included this activity into CRISP-DMI and elaborated our first, target platform independent optimization ideas in Section 4.

Systematic Definition of DMIL from the DMI Model (Section 3). The kernel of this section is devoted to a comprehensive presentation of our approach to the systematic design of the ADMIRE data mining langauge DMIL. This is a significant increment to the initial ideas presented in D1.2 [5], where the formalisms for defining DMIL where based on UML profiling. Now we move to a pure *metamodelling approach*, which allows us to define the complete language specification including abstract and concrete syntax and semantics. Here we also discuss how metamodelling is implemented within the Process Designer, a tool for the graphical design of DMI processes, their (optional) improvement through optimizing transformations and the generation of DMIL sentences.

DMI Process Optimisation (Section 4). Basically in ADMIRE we consider the improvement of the execution performance and storage efficiency of data flows specified by DMIL through two kinds of optimizations: platform independent and platform dependent. This section discusses a set of platform independent optimizing transformations, appropriate dataflow analyses and additional requirements put on the processing element descriptions in the ADMIRE registries to enable such optimizations.

Define Revised Ontologies (Section 5). Another key aspect of WP1 is ontology research. D1.1 outlined how semantic technologies could be used in the DMI processes. The first version of the ADMIRE DMI ontology was based on CRISP-DMI and the existing data mining standard PMML. D1.2 extended the ontology concept from D1.1 and introduced a new Data Mining Ontology. This section describes the Platform Ontology and how it interacts with the other ADMIRE components, especially the ADMIRE Registry and the Process Designer. Moreover, we describe how the processing elements are described and used in the ontology.

2 Extended CRISP-DMI Model

The initial version of the CRISP-DMI Model describing a methodology for realizing DMI processes was presented in D1.1 [6]. There are seven main phases of the model that cover tasks typically appearing in DMI processes, namely *Business understanding, Data understanding, Data processing, Data integration, Modelling, Evaluation* and *Explotation*. These phases are illustrated in Figure 1. In addition to these essential phases, there are two other phases that are related to the whole lifecycle of a DMI project. The first one is the *Provenance* phase, which tracks each task performed during the project to ensures the reproducibility of the processes. The second one is the *Optimization* phase, which is applied on each performing task in order to ensure optimal utilisation of available resources. These two phases are not isolated from the other phases but naturally support the tasks involved in these main phases. Moreover, the provenance information can be used in the optimization phase and in this way they contribute to each other. For example, at the platform level a monitoring service can provide such information about the load of the underlaying system that this information can be used in the decisions for optimal distribution and scheduling of computational tasks.

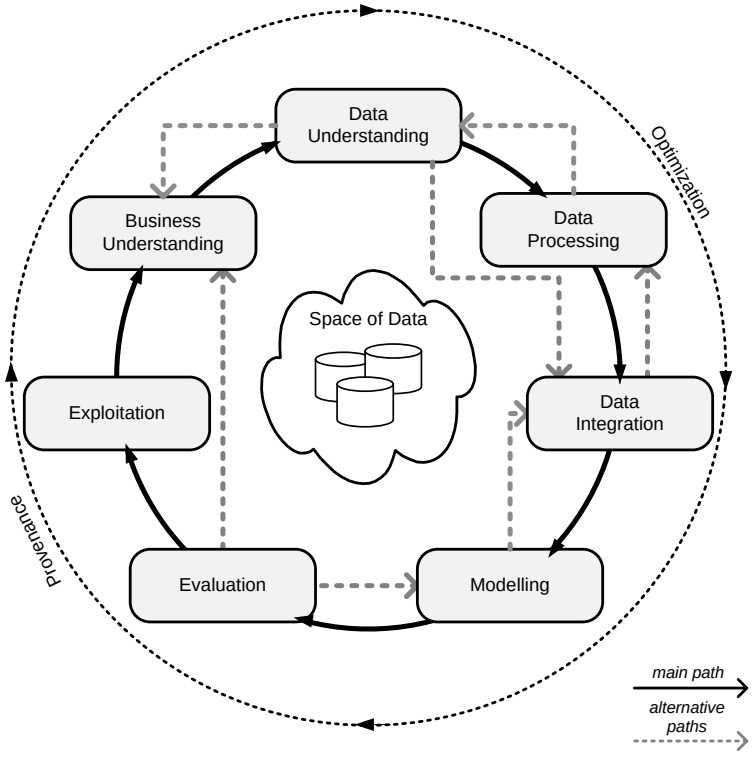


Figure 1: CRISP-DMI Model

2.1 Provenance

The provenance phase is concurrent with the other seven phases of the CRISP-DMI model, meaning that the information acquisition tasks are performed continuously. The DMI system must, therefore, support mechanisms for collecting provenance information so that when a user specifies that some data is saved to a data resource, sent to a third party, or delivered

to a visualisation mechanism, it is possible to obtain the provenance information and send it to a user-chosen destination. The hierarchical structure of the provenance phase and related generic and specialised tasks follows.

- Provenance Collection
  - Execution system information acquisition
  - Resource monitoring
  - Data transformation monitoring
- Provenance Presentation
  - Provenance data analysis
  - Report generation
  - Visualisation

Our first attempt at provenance collection, processing and visualisation is described in the paper "Provenance Support for Grid-Enabled Scientific Workflows" [18].

2.2 Optimisation

The DMI processes can be optimized at any stage of their life cycle. We recognize two types of optimization as follows:

a) Platform independent optimization

b) Platform specific optimization

Our approach to the platform-independent optimization of DMI processes is discussed later in this deliverable in Section 4. Within the ADMIRE project platform-dependent optimization is done at the level of the execution environment represented by USMT and OGSA-DAI.

3 Systematic Design of DMIL based on Metamodelling

The process of specifying a distributed data mining and integration system involves capturing complex interrelationships between the data mining, data warehousing and workflow management systems domains and a domain used to describe the environment in which the system will be implemented. Developers increasingly turn to domain-specific modelling techniques to manage the complexity of systems under development through such approaches as a Model Driven Architecture (MDA) [7]. The goal of domain-specific modelling is to increase the productivity of software engineers by abstracting away from low-level code details.

In our approach, we have adopted the MDA and its essential metamodelling foundation as a base for the development of a new Domain Specific Language (DSL), named Data Mining and Integration Language (DMIL). This language is used for specification of DMI processes in the form of workflows.

DMIL is a key component of the ADMIRE architecture. Its notation is used to communicate information about DMI processes and it is crucial for ADMIRE tools and enactment engines, which in this context can be considered as optimized interpreters of DMIL codes.

The metamodelling approach we use here aligns with the following definition given by Nytun et. al [24]:

> *A metamodel is a model that defines a language completely including the concrete syntax, abstract syntax and semantics.*

In other words metamodelling is the process of completely and precisely specifying a domain-specific modelling language which in turn can be used to define models of that domain. Put simply, a metamodel is a model that is used to define a language. Typically, a complete language specification has an abstract syntax model, a concrete syntax model and a semantic domain model, each of which is a model used to define a language and therefore a metamodel. The modelling languages can be executable and typically share a common structure. In order to define the language and its semantics we use the Meta Object Facility (MOF) [25] combined with the Object Constraint Language (OCL) [26]. The MOF is intended to support a range of usage patterns and applications providing a set of standard interfaces to define and manipulate a set of interoperable meta-models and their corresponding models. The OCL is a language that enables us to describe expressions and constraints on object-oriented models and object modeling artifacts.

So far, to the best of our knowledge, no research on a systematic approach to the design of a language for the specification of DMI processes has been reported.

3.1 A Model-based DMIL

In this section we introduce an approach for the design of a DSL using a well-defined formalism based on metamodelling that we have adopted for the design of DMIL. Based on Chen et al. [8] and Schmidt [29], we define a language L as a six-tuple:

$$L = \langle A, C, S, P, M_C, M_S \rangle$$

consisting of abstract syntax A, concrete syntax C, semantic domain S, pragmatics P, syntactic mapping M_C, and semantic mapping M_S.

- The abstract syntax A defines the language concepts, their relationships, and the well-formedness rules available in the language. The abstract syntax can be defined using context-free grammars or metamodels.
- The concrete syntax C defines a specific notation used to express models, which may be graphical, textual, or mixed.
- The semantic domain S defines the language semantics using a semantic mapping $M_S : A \rightarrow S$, which relates syntactic concepts to those of the semantic domain [13]. The semantic domain S and the mapping M_S can be described in varying degrees of formality, from natural language to rigorous mathematics.
- The pragmatics P deals with the usability of the language. This includes the possible areas of application of the language, its easy of implementation and use, and the language's success in fulfilling its stated goals. The pragmatics is typically described in natural language.
- The syntactic mapping M_c assigns syntactic constructs to elements in the abstract syntax $M_c : C \rightarrow A$.

In a model-based approach, the abstract syntax A of L is described by a metalanguage ML, which itself has an abstract syntax A_{ML}. A is called the metamodel of L, while A_{ML} is the metametamodel.

In our approach, we focus on a language whose abstract syntax is defined in terms of a metamodel. The metamodel of a language describes the vocabulary of concepts provided by the language, the relationships existing among those concepts, and how they may be combined to create models of DMI processes.

A metamodel-based abstract syntax definition has the great advantage of being suitable to derive (through mappings or projections) different alternative concrete notations C_i and their syntactic mappings M_{C_i} , textual or graphical or both, for various scopes such as graphical rendering, model interchange, standard encoding in programming languages, and so on, while maintaining the same semantics M_S. Therefore, a metamodel could be intended as a standard representation of the language notation.

3.1.1 Why Metamodelling ?

There are several formalisms (e.g. graph grammar, context free grammar, UML Profiling, etc.) for defining languages but not all of them define the complete language specification (e.g. abstract, concrete syntax and semantics). The reason for choosing metamodelling as the formalism to specify our software language is that it offers enough abstraction and understandability for the design of a domain-specific language. While we focus on a concrete domain (i.e. DMI) it is still necessary to produce a precise definition for the domain-specific semantics of the DSL, which is not possible using a generic modelling language such as UML.

In addition the UML customization mechanism for defining DSLs, called UML Profiling, is too restrictive for our needs because new concepts need to adapt themselves to other existing concepts. This means that one has to understand all the pre-existing metaclasses to identify the right base metaclasses of a newly defined stereotype. Since our language introduces new concepts which are not based on any typical process model (although there are some similarities) UML Profiling is not an adequate formalism for our language specification. On the other hand, we must note that an advantage of UML is that it offers a well-defined general purpose modelling notation and has a strong tooling support and therefore we use its graphical capabilities to represent our model.

3.1.2 Metamodelling Example

The following example is taken from [19] to illustrate how an abstract syntax model defined using a metamodel can be used to represent a language. The model in Figure 2 is a specification of the abstract syntax for a simple mathematical expression. The model contains no hints on what a mathematical expression should look like; that is it does not contain information about a concrete syntax. The concrete syntax in various forms is shown in Figure 3. Figure 4 shows the abstract form as a tree of instances of the classes in the abstract syntax model.

3.1.3 MDA approach

MDA defines an architecture based on four abstraction layers. At the top layer (M3) of the architecture is the metametamodel (i.e MOF), which provides a generic language for the definition of a DSL. Layer M2 is populated by metamodels that represent MOF-defined

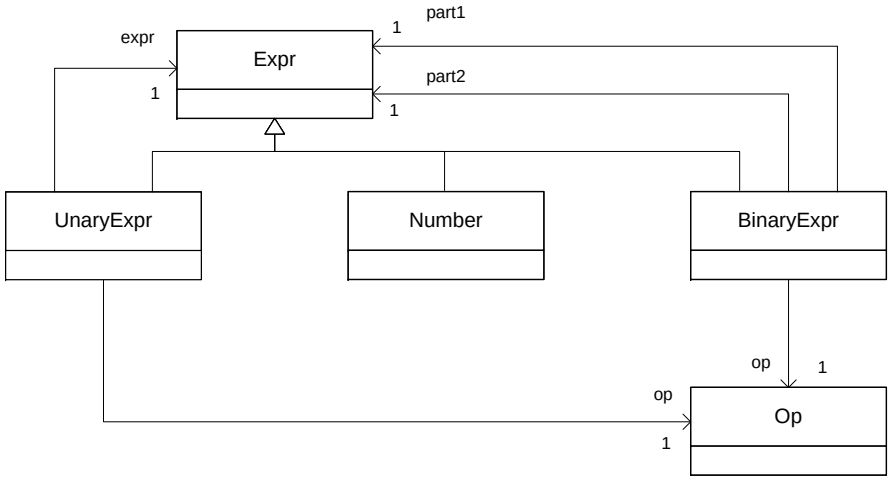


Figure 2: The abstract syntax for a simple mathematical expression.

$-32/4 + 5 * 3$	(infix)
$(32)4/53 * +$	(postfix/Polish)
$+/(-32)4 * 53$	(prefix/reverse Polish)

Figure 3: Various forms of the concrete syntax for the expression in Figure 2.

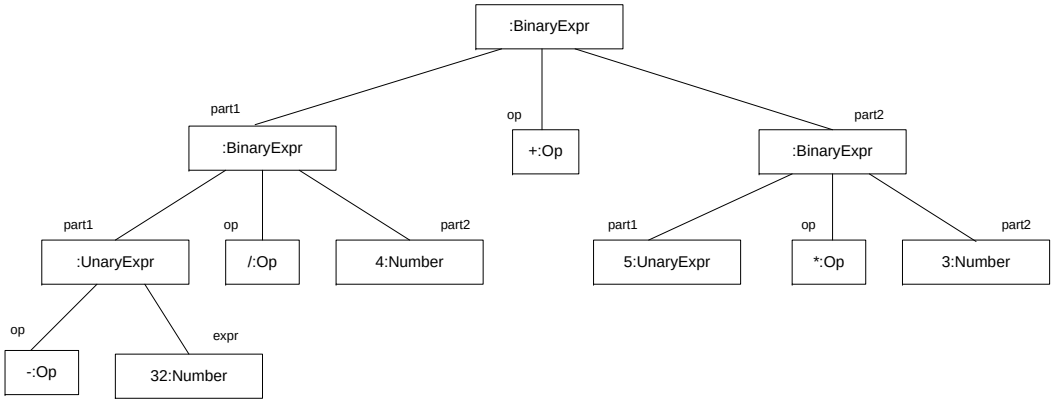


Figure 4: The abstract form as a tree of instances of the classes in the abstract syntax model.

DSL. Layer M1 hosts domain models written in M2-defined DSLs. Finally M0 hosts run-time domain objects that instantiate M1 domain entities. To define the abstract syntax of the DMIL language we target primarily the M2 level of the MDA layered architecture. The advantages of MDA are that it provides both MOF for metamodel specification and OCL for the specification of metamodel constraints. Furthermore, MDA can facilitate the definition of mappings between DSLs using QVT [4] and the exchange of metamodel data using XMI [27].

The process of developing DMIL consists of the following steps. Firstly, we define a new abstract syntax metamodel which represents essential concepts and their relations in the DMI domain. This model is based on a metamodel using MOF and expressed using a set of class

diagrams organised in packages. Secondly, we define a concrete syntax of the language, which is a mapping of the elements of the abstract syntax model to their textual representation. In addition, we also define a graphical notation for the all these elements. Then we describe the semantics of the defined concepts and validate the metamodel. Finally, we develop a tool based on the models to allow the construction of DMI processes based on DMIL.

3.2 DMIL — A Software Domain Specific Language

DMIL provides a notation for communicating about DMI processes, but it is not a language to be used for coding data mining algorithms. Its textual form is used to submit requests for DMI process enactment through DMI gateways. It may also be used as a stored representation of DMI processes by tools, and as the input and output of DMI process optimizers.

DMIL is used primarily to define graphs and functions that generate these graphs dynamically. The nodes of graphs are *Processing Elements* (PE) that perform tasks such as extracting data from databases and files, transforming data and performing data mining algorithms. The directed arcs, called *Connections*, denote a data flow from a specified output of one PE to the specified inputs of one or more PEs. A literal data stream notation in DMIL denotes a sequence of values to be delivered to a connection or specified input. DMIL can be also used to define DMI patterns and to define or redefine DMI processes by composing PEs.

DMIL is delivered as:

- an abstract syntax language defined by a metamodel (DMIL-m) — its purpose is to define concepts and their relations in the DMI domain which are as far as possible platform independent. DMIL-m is presented in Section 3.3;
- a concrete textual (DMIL-t) and graphical (DMIL-g) representation of the language, to support the design and interchange of the modelled DMI processes. The concrete textual syntax is a result of mapping the abstract syntax to the concrete syntax. An overview of concrete language representations is given in Section 3.4;
- a language semantics (DMIL-s) to describe the meanings of the language concepts. These semantics can be implicitly or explicitly defined. The DMIL-s is presented in Section 3.5.

The primary features of DMIL are:

1. the type system of this language — the structural types — is kept separate from the type system identifying the semantics of data-mining or application domain values — the DMI and domain types respectively — so that it can organise DMI for application domains that use different types of data without itself having a complex type system;
2. the interconnection of elements are described in terms of connections that, conceptually at least, transmit data as a stream;
3. the language supports the incremental design and installation of (libraries of) components that support DMI, so that the computational context can be incrementally manipulated to better serve a community's needs during its operation.

DMIL encodes the following:

- requests for information about the services, data resources, data collections, defined components (PEs, functions and named types) and supported DMI libraries;
- the definition, redefinition and withdrawal of any of the above, i.e. the capabilities of a gateway can be dynamically tailored;
- the submission of requests to enact a specified DMI process.

3.3 DMIL Abstract Syntax Model

In our model-based language definition, the abstract syntax of the DMIL language is defined in terms of an object-oriented model called DMIL Metamodel (DMIL-m). This metamodel characterizes the syntax elements together with their relationships and separates the abstract syntax and semantics of the DMIL constructs from their concrete textual notations.

Our metamodel is based on the MOF model, UML notation and OCL, used as the modelling language, graphical notation, and constraint language, respectively, for defining and representing the complete DMIL-m .

3.3.1 Metamodel Packages

The DMIL-m uses packages to control complexity, promote understanding, and support reuse of defined classes and their relations. The DMIL-m consists of three conceptual areas represented by the metamodel packages as illustrated in Figure 5. Collectively, the collection of DMIL-m packages provides the necessary abstractions to model generic representations of DMI processes.

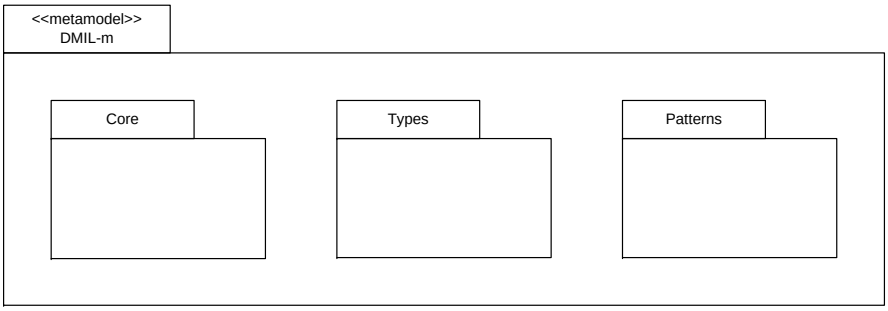


Figure 5: DMIL-m Package structure

1. Core (eu.admire.dmil.core) — this package contains the basic metamodel classes and associations used by all other DMI metamodel packages. This includes DMI process, processing element and patterns definitions.
2. Types (eu.admire.dmil.types) — this package contains three type systems:
   a) structural types defining the representation of DMI-process definitions;
   b) DMI types that describe values that are used in DMI domain;
   c) application domain types that are used in a specific application domain.

3. Patterns (eu.admire.dmil.patterns) — this package contains predefined DMI patterns, which encode recurring structure within DMI processes. In DMIL the patterns are defined using functions.

The definition of the DMIL abstract syntax by a metamodel is well supported by metamodelling environments such as the Eclipse Modelling Framework [11]. The framework fully supports model-driven development and offers software engineers powerful tools to improve productivity and enhance the quality of system design.

3.3.2 Core Package

The *Core* metamodel package contains the major classes and associations used to define DMI processes as presented in Figure 6.

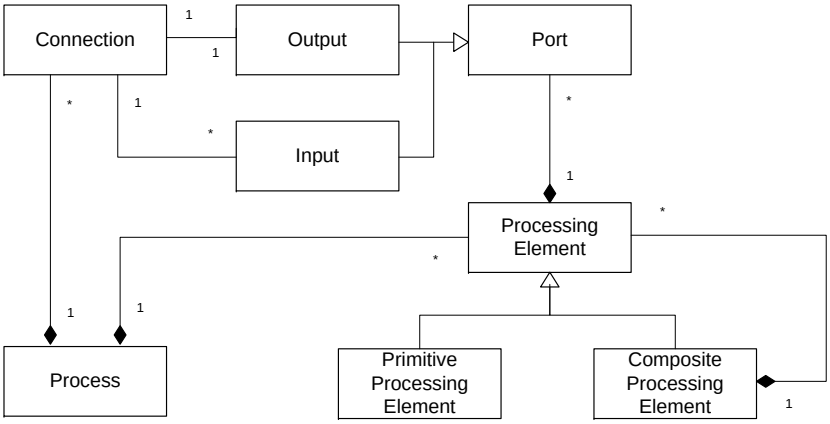


Figure 6: Core metamodel

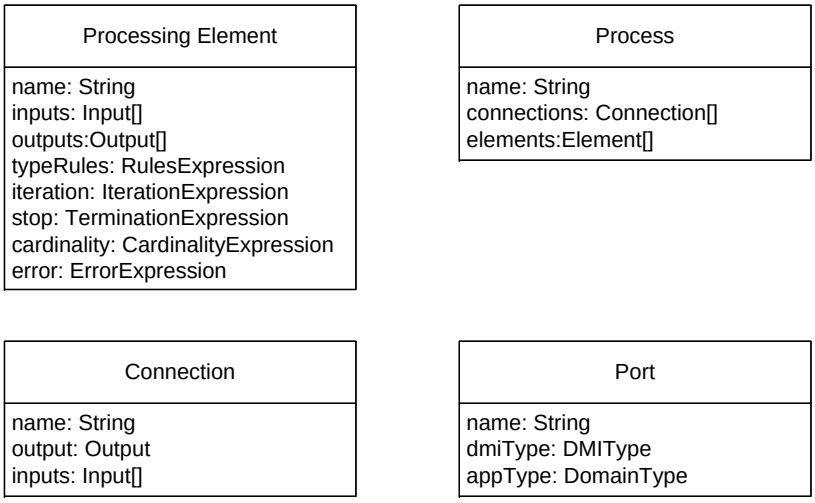


Figure 7: The main classes of the Core Package

Processing Element Definition

The *Processing Element* (PE) is a primitive or composite software component encapsulating a DMI task and providing for its use in DMI Processes. Multiple instances may be used in one process. It has a specified structure of inputs and outputs. An instance of the element can be executed during a DMI Process enactment. Each PE has a list of properties which characterize the PE, as shown in Figure 7. A note on each of these properties is given below.

- Name (*name*) — this is a name that persists over development versions as it indicates a consistent intended behaviour.
- Tuple of Inputs/Outputs (*inputs/outputs*) — each PE can have zero or more tuples of inputs and outputs. Each input and output is characterized by a structural type and optionally by a domain type.
- Type constraints and transfers (*typeRules*) — this shows constraints and relationships between type variables introduced in the structural types above. If optional application-domain interpretation identifiers have been placed on these types, then these follow the transfer of the type variables.
- Iteration behaviour (*iteration*) — this describes the order in which the inputs are consumed and the outputs produced.
- Termination behaviour (*stop*) — this describes the circumstances under which this PE stops iterating and cleans up (other than termination applied by the execution container's termination).
- Cardinality effects (*cardinality*) — this describes volume relationships between inputs and outputs. It may use selection expressions to assert relationships about substructures in the inputs and outputs.
- Error behaviour (*error*) — this describes known reasons for the PE signaling an error.

The example definition shown in Listing 1 states there are two inputs and one output and gives their application-domain interpretation identifiers as `SQLquery`, `RDBidentity` and `ResultSet` respectively. It consumes the data first from `dataResource` and then from `expression` and it always produces a value, with a structural type of list-of-tuples, the tuples having at least one element, with any identifier and any type. The PE stops when either input is exhausted, but it is an error if they are not both exhausted (note the interaction with the semantics of repeat enough).

Connections

Each PE has named input and output connections. The names of the inputs and outputs of a PE must form a set, i.e. the same name may not occur as both an input and an output for a particular PE. Variable numbers of inputs or outputs all with the same purpose are provided by using an array notation.

Listing 1: PE definition example

```
name "SQLQuery"

inputs <
    expression : String : SQLquery
    dataResource : EPR : RDBidentity
    >

outputs <
    data : [<anyId: any(T1), ... >] :ResultSet
    >

iteration lockstep(dataResource; expression) -> data

stop empty(dataResource) OR empty(expression) OR notWanted(data)

cardinality card(dataResource) = card(expression) = card(data)

error (empty(dataResource) AND NOT empty(expression)) OR
      (empty(expression) AND NOT empty(dataResource))
```

Process Definition and Functions

Descriptions of DMI processes are compositions of existing PEs that already perform tasks such as querying a data resource, transforming each tuple in a sequence, merging two input streams, removing anomalies, normalising, classifying, etc. New composite PEs can be defined by composing other PEs and can be registered for future use. Registered components can be collected together to form a Composite Processing Elements (CPEs) that support a particular data integration, data mining or domain-specific class of processing steps.

Functions may be used to name, parameterise and encapsulate any sequence of DMIL sentences optionally ending with a return expression. They can yield an encapsulation, be used to program patterns of composition and to represent DMI patterns. Using functions one can encode repetitive and data-adaptive process patterns.

Parameters to functions can specify other functions, PEs, PE instances, data resources, data collections, controls for generating literal data streams, sampling rates, repetition and parallelisation targets and so on. Functions simplify the abstract machine by serving purposes perceived as different by users and interaction tools, e.g. they represent:

- a composite PE, built by connecting other PEs, where they hide internal information and prevent ambiguity over naming multiple PE instances;
- a packaged DMI-process definition that can be parameterised and activated through a portal;
- the encoding of a pattern, such as repetition, parallelisation, all-meets-all, etc.;
- the encoding of an optimization strategy.

3.3.3 Types Package

There are three type systems that are accommodated in the *Types* metamodel package, corresponding to the three conceptual domains of interest:

1. *Core Types* (or graph-construction types) are used to constrain all of the operations used in describing, constructing and manipulating the graphs of PE nodes interconnected by streaming connections. This type system is the same whatever DMI-application domain — it uses structural type equivalence;

2. *DMI Types* are used to specify the inputs and outputs of the data-mining specific PEs; these are mathematically based. These types describe the data streaming along connections to and from PEs that perform data mining algorithms;

3. *Domain Types* describe the data input into, and output from, PEs and transmitted through connections that correspond to values in an application domain. There may be many versions of this type system for different application domains.

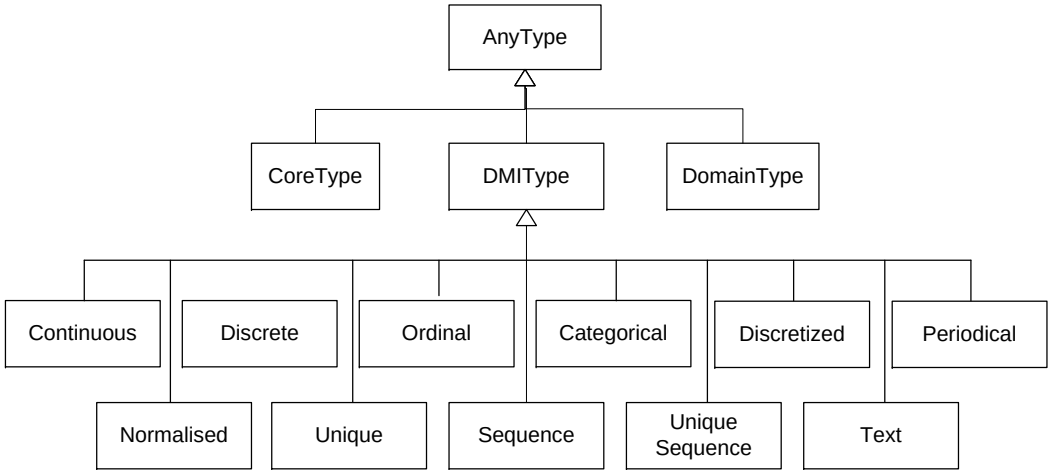


Figure 8: DMI Types

3.3.4 Patterns Package

The *Patterns* metamodel package depends on the *Core* and *Type* metamodels. This package contains a set of typical DMI pattern definitions that can be directly instantiated as DMI processes. In DMIL, a DMI pattern is defined using a function which encapsulates PEs. DMIL functions abstract data-flow graphs, contrary to traditional functions which abstract a control-flow graph. Every DMIL function is expected to return a data-flow graph, which could be embedded inside an exiting data-flow graph, or may be connected to other data-flow components to form a bigger data-flow graph. In other words, DMIL functions allow the composition of complex data-flow graphs from well-defined functional abstractions of recurring data-flow patterns.

3.4 DMIL Concrete Syntax

DMIL provides special textual and graphical notation for communicating about DMI processes. This concrete syntax plays a crucial role in the DMIL design since it is used to encode a DMI process as a request to the enactment engine.

3.4.1 Textual Syntax

MDA supports the transformation of higher-level models into platform-specific models that can be used to generate implementation-level models. The transformation of the source model into a target model is based on transformation rules. There are different methods that can be used for defining the transformation rules. The concrete textual syntax (DMIL-t) is directly generated from the abstract syntax model. The syntax for model elements is described using the Extended Backus-Naur Form (EBNF) notation. Listing 2 shows an example of a DMI process in the DMIL concrete textual syntax. Listing 3 shows an example of a DMI pattern definition in DMIL .

Listing 2: Example DMI process in DMIL

```
use eu.admire.pe.SQLQuery;
use eu.admire.pe.MergeTuple;
use eu.admire.pe.DescriptiveStats;
use eu.admire.pe.BuildClassifier;

Connection sqlQuery1;
Connection sqlQuery2;
Connection resourceEPR;

/* Initialise */

SQLQuery query1 = new SQLQuery();
SQLQuery query2 = new SQLQuery();
TupleMerge merge = new TupleMerge();
BuildClassifier buildClassifier = new BuildClassifier();
DescriptiveStats descriptiveStats = new DescriptiveStats();

/* Query resources */

sqlQuery1 => query1.expression;
sqlQuery2 => query2.expression;
resourceEPR => query1.dataResource;
resourceEPR => query2.dataResource;
sqlQuery1 => query1.expression;
sqlQuery2 => query2.expression;

/* Merge data */

query.data => merge.data1;
query.data => merge.data2;

/* Compute statistics */

merge.data => stat.data;

/* BuildClassifier */

merge.data => buildClassifier.data;
```

Listing 3: DMI function implementing a DMI pattern

```
use eu.admire.pe.SQLQuery;
use eu.admire.pe.MergeTuple;
use eu.admire.pe.DescriptiveStats;

function ComputeDescriptiveStats ()
PE (<Connection query1, query2, resource> => <connection statistics>) {

Connection sqlQuery1;
Connection sqlQuery2;
Connection resourceEPR;

/* Initialise */

SQLQuery query1 = new SQLQuery();
SQLQuery query2 = new SQLQuery();
TupleMerge merge = new TupleMerge();
BuildClassifier buildClassifier = new BuildClassifier();
DescriptiveStats descriptiveStats = new DescriptiveStats();

/* Query resources */

sqlQuery1 => query1.expression;
sqlQuery2 => query2.expression;

resourceEPR => query1.dataResource;
resourceEPR => query2.dataResource;

sqlQuery1 => query1.expression;
sqlQuery2 => query2.expression;

/* Merge data */

query.data => merge.data1;
query.data => merge.data2;

merge.data => descriptiveStats.data;

return PE(<Connection query1 = sqlQuery1;
          Connection query2 = sqlQuery2>;
          Connection resource = resourceEPR> =>
          <Connection statistics = descriptiveStats.stats>);
}
```

3.4.2 DMIL Graphical Notation

DMIL has its own graphical representation that supports comprehension of the DMI processes and facilitates the design phase of the preparation of DMIL requests. In order to define a graphical modelling language it is necessary to define the graphical notation of the language. Since DMIL has only a few elements for describing a DMI process (as defined by the metamodel) we provide the graphical notation only for the PE and Connection as shown in Figure 9. The PEs are connected using the connections as shown in Figure 10. As we have noted, DMIL is not designed for encoding data mining algorithms but rather for composing graphs consisting of various PEs which provide those algorithms. The graphical notation is specially important for tools such as the ADMIRE Process Designer, which can represent a DMI Process in visually readable form.

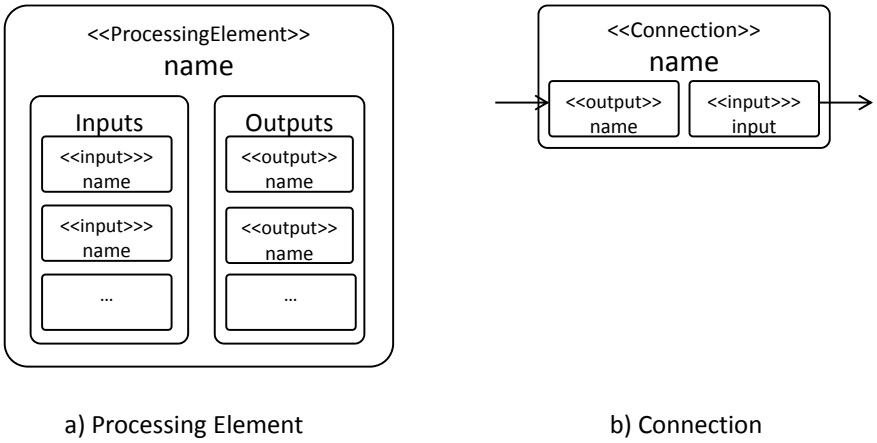


Figure 9: Notation DMIL Processing Element and Connection

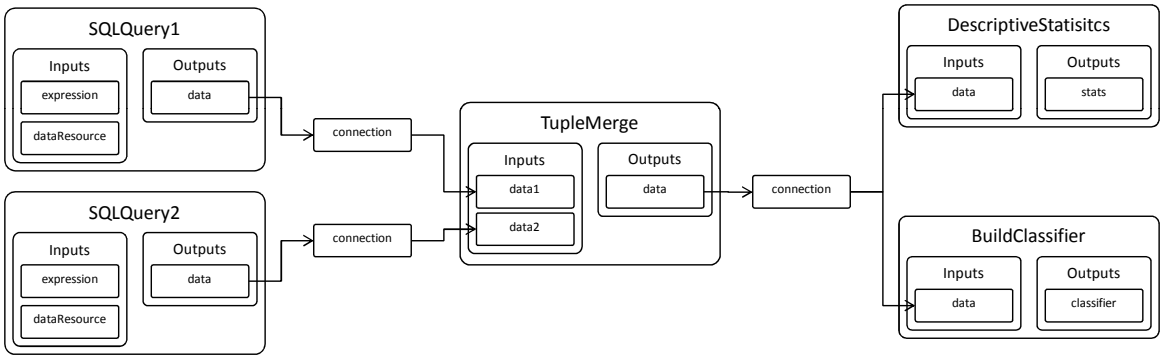


Figure 10: Example of the graphical notation

3.5 DMIL semantics

The language semantics describes the meaning of language concepts. The main concepts of DMIL are described in the ADMIRE Whitepaper [3].

3.6 The Process Designer

In order to design DMIL processes we have developed a specialized tool named the Process Designer (PD). The PD is a graphical environment that allows DMI experts to compose PEs and create complex DMI processes. It enables one to define new PEs by providing their characteristics or to build new PEs by composing existing PEs and their parameterisation. As a result, the tool produces DMIL requests. Since we follow strictly the MDA approach, the implementation is based on two major phases. The first is the modelling phase during which an internal *Ecore* model is developed; the second is the code generation and code customization phase. The *Ecore* is a platform-independent metamodel used by Eclipse and its modelling framework (EMF)[1] to design other models. The *Ecore* model is based on the elements defined in the *Core Metamodel Package* (eu.admire.dmil.core) and contains a set of

[1] http://www.eclipse.org/emf

classes for representing designed DMI processes. This model is combined with a *graphical definition model* and *tool definition model* in order to produce the final Java code. The graphical model is created using the Eclipse graphical modelling framework (GMF)[2], which specifies the visual environment of the Process Designer. The current version of the PD is fully integrated into the Eclipse platform. More details about implementation of the tool can be found in ADMIRE D5.3 [22].

To support the design and implementation of text editing facilities many concrete syntax and model mapping tools have been considered. We have chosen EMFText [15], Eclipse's integrated tool for agile textual syntax development. EMFText allows us to define a plain textual syntax for Ecore-based metamodels and to generate components to load, edit and store model instances. This feature enables us to use a special form of concrete syntax specification and generate the DMIL textual representation directly from a model instance, and conversely to load model instances from textual representations.

3.7 Summary

In this section we have discussed the architecture of DMIL and our systematic approach to its design based on metamodelling. Moving from traditional ad-hoc design methodology which was, to the best of our knowledge, applied to the design of all workflow languages up until now, brings two important benefits. Firstly, the design process leads to a more consistent, coherent, and complete language. Secondly, the language metamodels associated with different levels of the language architecture and DMI process representations can be mapped to existing integrated program development environments, like Eclipse. This allows the automatic generation of target language sentences and enables us to build domain-specific applications, based on DMIL concepts, effectively.

[2] http://www.eclipse.org/gmf

4 DMI Process Optimisation

The full potential of large-scale DMI platforms, which are typically geographically distributed and involve multi-level concurrency, can only be reached when the DMI processes to be enacted by them are programmed effectively. So far this has proved to be a difficult task. The handcrafting of data processing graphs, having arbitrary data operators as nodes and their producer-consumer interactions as edges, to make optimal use of available resources and satisfy other constraints, is a daunting task [10]. The efficiency of the enactment depends critically on the proper utilization of architectural features of the underlying hardware, making automatic support for DMI process development highly desirable. Work in the field of programming environments for DMI platforms spans several broad areas, including the design of very high-level specification tools, e.g. workflows [9], and the development of new data processing languages, e.g. Athena Distributed Processing Query Language [10].

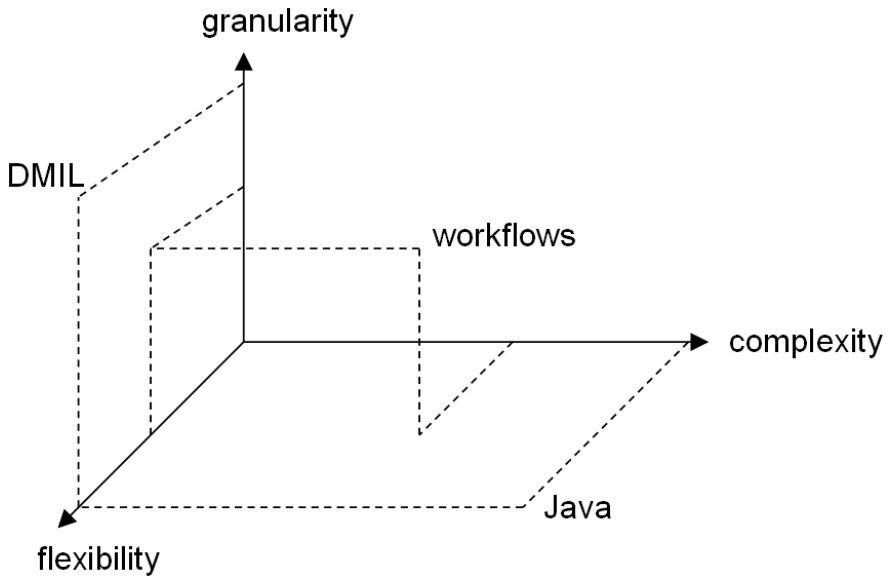


Figure 11: Characteristics of DMIL, workflows and Java.

Different languages have different characterisations with respect to their complexity, flexibility and granularity, as pictured in Figure 11 for DMIL, Workflows and Java. This influences the methods used for their optimization. The three dimensions are defined as follows:

1. granularity/abstraction: the level on which programs are defined;
2. complexity/functionality: the coverage of typical programming language features, e.g. control structures, etc.;
3. flexibility/extensibility: the applicability to new architectures, data types and domains.

Java is a fully-fledged programming language with a rich type system and control structure, defining programs at quite low level. Recent workflow languages [30] for distributed environments typically support some forms of control structure, operate on a predefined level (mostly Web services) which in turn restrict their flexibility to new architectures and domains, particularly with respect to optimization possibilities. DMIL is a language for data-intensive

systems process engineering operating on the high abstraction level of Processing Elements (PE) where a PE is a unit of data processing functionality implemented in whatever form, e.g. Web services or OGSA-DAI activities. Each PE has inputs and outputs with connections defining the data flowing between them. DMIL does support a limited set of typical programming language features, e.g. control structures, but focuses primarily on the extensibility and flexibility of the description of data processing functionality (patterns), data flows, and associated domain types for optimization.

Our research goal is to extend our work on DMIL by a novel research area called *DMI process optimization engines (DMI-POEs)* that transforms specifications (codes) produced by tools such as the ADMIRE Process Designer or written directly by a domain expert into semantically equivalent codes that can be efficiently executed on a target machine. This functionality can be also called as automatic DMI process restructuring; it concentrates on the transformation of DMI process specifications for large-scale scientific and engineering DMI applications. The original code produced by the Process Designer or written by the user can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations, although the term *'optimization'* is a misnomer because there is rarely a guarantee that the resulting code is the best possible [1].

The emphasis of this section is on DMI platform-independent optimizations, shown in the upper part of Figure 12, which are program transformations that improve the target code without taking into account any specific properties of the target platform such as dead code elimination and similar [14].

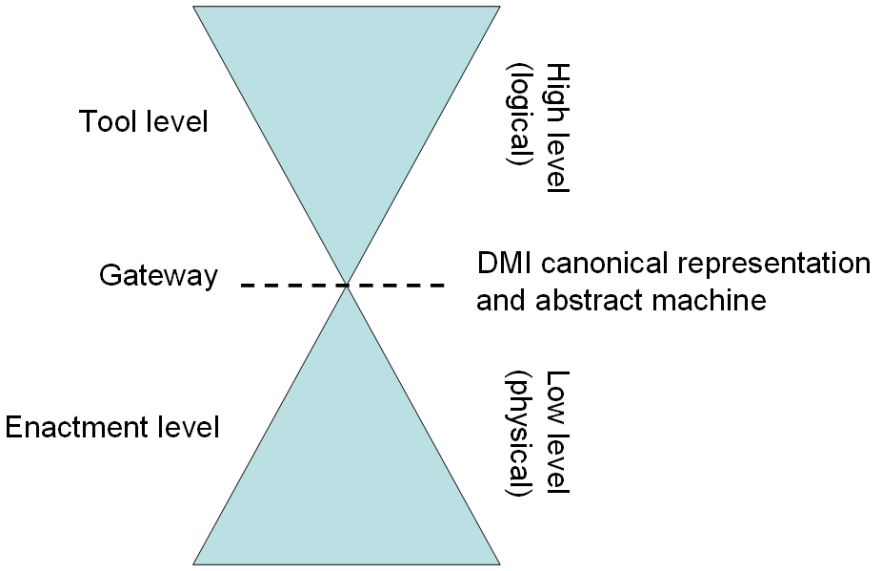


Figure 12: Optimization possibilities of DMI processes.

Low-level platform-dependent optimizations take into account the physical environment of any given DMI process execution and handle such things as Process Element placement and binding (e.g., late binding for choosing among concrete service implementations at workflow runtime [9]) as well as pipelining [12].

Optimizing transformations are based on the pre-execution analysis of DMI process spec-

ifications or workflows usually referred to, in the compiler optimization domain, as flow analysis. Flow analysis [14] consists of both control flow analysis and data flow analysis and is a fundamental prerequisite for many important types of code improvement. In general, control flow analysis precedes data flow analysis. Control flow analysis is the encoding of pertinent, possible program control flow structures or flows of control, usually in the form of one or more graphs. Data flow analysis is the process of ascertaining and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or various attributes of variables) in a computer program. Flow analysis is a prerequisite for data dependence analysis, which is at the heart of our restructuring system. It computes a relation between activities which essentially determines whether or not they can be executed concurrently (thus exposing potential parallelism), and that means the removal of any unnecessary order between the activities of the DMI process. Another important aspect of DMI process optimization has close relations with traditional query optimization [16] and execution in traditional databases, with the differences that component Process Elements may represent arbitrary operations on data with unknown semantics, algebraic properties and performance characteristics and are not restricted to come from a well-known fixed set of operators, e.g. a relational algebra.

4.1 Components of a DMI Process Optimization Engine

The following characteristics of DMIL reduce the application space of typical optimizations [14, 2, 1] regularly performed by compilers:

- DMIL is a high-level language for specifying data processing flows in an abstract and reusable manner;
- PEs may represent arbitrary data processing with unknown semantics, algebraic properties and performance characteristics;
- control structures are used just for connection setting (see Listing 1).

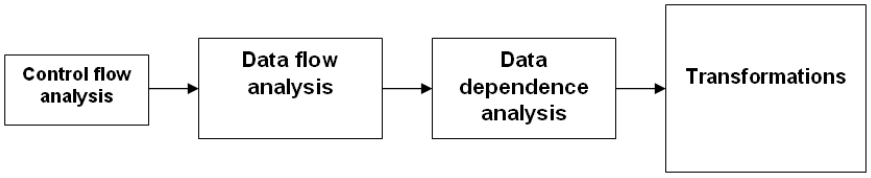


Figure 13: Components of a DMI Process Optimization Engine.

These facts puts the focus on data flow/dependence analysis and subsequent possible transformations, resulting in Figure 13 showing the components of a DMI Process Optimization Engine where the size of the boxes represent their importance to our research direction.

The definition of data dependence [2]:

There is a data dependence from statement S1 to S2 (statement S1 depends on S2) if and only if (1) both statements access the same memory location and at least one of them stores into it, and (2) there is a feasible run-time execution path from S1 to S2.

The best program transformations are those that yield the most benefit for the least effort, as stated by [1]. The transformations provided by an optimizing engine should have several properties:

1. a transformation must preserve the meaning of programs. That is, *'an optimization'* must not change the output produced by a program for a given input, or cause an error that was not present in the original version of the source program. More formally, a valid transformation is defined in [2] as:

> *A transformation is said to be valid for the program to which it applies if it preserves all dependencies in the program.*

2. a transformation must, on the average, speed up programs or reduce the memory space used by the running program by a measurable amount. If it does not produce a suitable efficient program, application developers will abandon attempts to use the language [2];

3. A transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code-improving transformations, and to have the compiler expend the additional time compiling source programs, if this effort is not repaid when the target programs are executed.

4.2 DMI POE within the ADMIRE Architecture

Including the DMI POE into the overall creation of DMI process can happen at different places in the architecture, depending on whether optimization is seen as a reoccurring phase suggesting improvements in an iterative process to define ones DMI process, or as a one-shot effort performing fully automated unsupervised improvements on the final version of a DMI process. If we prefer the former, the DMI POE is invoked by the Process Designer continuously, needing access to the same Registry for assessing possible improvements. If we go for the latter, the DMI POE might be located close to or even inside the Gateway as a phase after the initial syntax and correctness checking of submitted DMIL [3]. In both cases, as DMI POE is focusing on high level optimizations, no additional information has either to be included or submitted with the updated DMIL as all the (logical) improvements are again expressed in DMIL .

If an optimization task takes too long (e.g. due to weak DMI POE performance) it should not be included in the designed process; and if the confidence in the suggested improvements is low it should not be allowed at the Gateway level. This relation is depicted in Figure 14. Note that there could be more than one DMI POE with different characteristics, e.g. one with fast but low confidence optimizations usable at Process Designer level and one with slower but high confidence optimizations applicable at Gateway level, making a hybrid solution possible as well.

4.3 Optimization Examples

In Figure 15, we give an example of a data mining and integration workflow [17]. The data flow between activities has been annotated with the composite type of the data.

In the context of DMI the composite types mentioned have the following meaning:

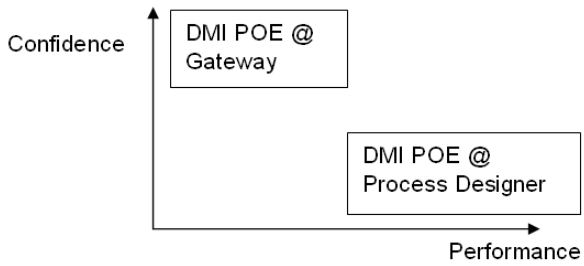


Figure 14: Depending of the characteristics of a DMI POE component with respect to confidence and performance restricts its application at different architectural places.

- Set: is a set of tuples without any ordering on the tuples, whose members are distinct;
- Bag: can contain duplicate tuples without any ordering on the tuples. This provides the most flexible options for making activities parallel as we need not to be concerned with any duplicates or ordering of tuples;
- Sequence: is a bag of tuples that is ordered.

When a transformation specified on a composite type is performed on data, the semantics of the data may change. Furthermore, it is possible that data are lost in the transformation. A transformation t is neutral if it transforms c into c' without losing any data. Of course, a neutral transformation can still change the semantic interpretation of data. The following transformations are all neutral:

- Set2Bag
- Set2UniqueSequence
- UniqueSequence2Sequence
- Set2UniqueSequence
- Set2Sequence
- Bag2Sequence

For every composite type identified above a neutral transformation to Sequence exists, so we can always safely transform any composite type to it.

One sub-workflow called for data cleaning can be identified as well as one for model creation and validation. The aim of the overall workflow is to construct a classification model of the data from sources 1 and 2 in the form of a decision tree. This model shall then be deployed on different data using sources 3 and 4. In the workflow we know from our table of activities that a classification algorithm requires a Set as the composite input type. This workflow uses n-fold validation to prevent the over-fitting of the model to the data, which requires a Set as input. A Seq2Set activity provides this functionality as the PE *'Fill missing values C'* before n-fold validation outputs a Sequence.

If we optimize the workflow by hand to incorporate some parallelism, some changes are obvious. The result is shown in Figure 17. Three important changes were made. As each of

the activities can equally handle a Set, we can bring forward the Seq2Set and then use a Set as the composite type all throughout the cleaning process as well. This will allow us to run activities in the cleaning process in parallel by splitting up the data and creating multiple instances of the cleaning activities. This is possible as we know the composite type required at the n-fold validation step requires only a Set as its input. Another important but not so obvious change is the re-ordering of the PE *'Quantize a'* after *'Filter a>5'* as this PE reduces the cardinality in contrast to the other.

So let us take a close look what is required in order to automate this optimization.

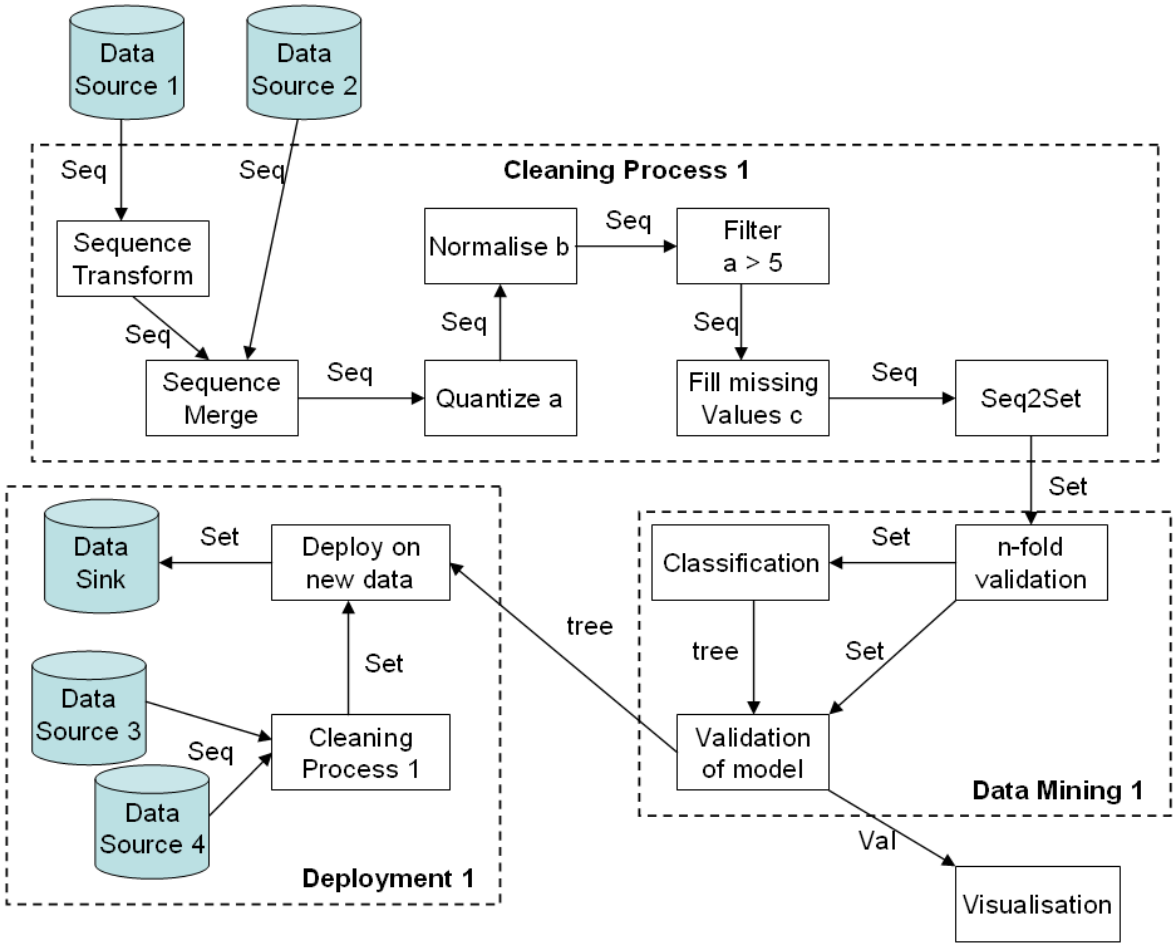


Figure 15: An example of a data integration and mining process that depends on multiple data sources and a cleaning process before the classification phase can start.

4.3.1 Composite Input Type Propagation

First, we need to know if any of the PEs really needs a sequence as input or not. Three quickly identified by name are *'Sequence Merge'*, *'Seq2Set'* and *'Sequence Transform'*. This leads us to the first description requirement of PEs.

Requirement 1: *The required composite input types of a PE and their relation to the com-*

posite output types have to be defined.

But this doesn't resolve the fact that *'Sequence Merge'* and *'Seq2Set'* both need a sequence as input and are part of the cleaning process. We address this issue by the second description requirement of PEs.

Requirement 2: *Standard ADMIRE PEs fulfilling a defined function, e.g. merge or CompositeTypeX2CompositeTypeY, on specific composite types have to be labeled as such in order to be replaceable by the appropriate equivalent, or omitted altogether.*

Applying the descriptions above results in acceptable composite input types *'Seq'*, *'Set'*, *'Bag'* for all the PEs of the cleaning process until the PE *'n-fold validation'* which requires a Set. Set is typically preferred over a Sequence as one has not to maintain order. Applying now the Seq2Set transformation as early as possible leads to one Seq2Set PE after each data source, requiring replacement of the *'Sequence Merge'* PE with a *'Set Merge'* PE as well.

Via forward type propagation [14] the outputs' composite types are re-calculated until n-fold cross validation, where Requirement 2 allows the optimizer to omit the *'Seq2Set'* PE before it as the output type of the PE before the n-fold cross validation already provides a Set.

4.3.2 Transformations

We now have to define what has to be known about a PE and composite input types in order to parallelize parts of a workflow. In order to decide if a sequence of PEs can be executed in parallel, we need to know on which part of their composite input/output types they really operate on (read/write). This leads us to the third description requirement of PEs.

Requirement 3: *PEs have to define which parts of the input type they read (typically columns of tuples contained in the composite types) and which part of the output type they write in order to be able to generate a data dependence graph. These parts of a composite input/output type are called 'primary', as the PE really operates on them.*

An example annotation of the *'Normalise'* PE regarding this additional metadata has to include the following:

- Composite Input Type: Set
- Parts actually read: PARAM
- Composite Output Type: Set
- Parts actually written: PARAM

where PARAM denotes an identifiable part of a composite type. Note that in our concrete example PARAM needs to identify just one column in a tuple. Although this information is enough for dependency analysis on a high level, for later low level optimizations (e.g. deciding about PE distribution) it will be important to know the *'secondary'* parts of a composite type flowing between PEs as well.

Having this additional description for PEs we can infer that this sequence is in fact parallelisable since all data dependences are still fulfilled if we operate on them in parallel and composite data types persist. Operating just on known parts of the data we can introduce

a Split/Merge PE combination before/after the sequence to parallelize. This leads us to the first transformation heuristic for DMI optimization.

Transformation Heuristics 1: *Provide just as much data as necessary to a PE to operate in parallel. The original data input and output of the sequential process transformed via Split/Merge PEs must remain the same in order to maintain the original data flow semantics (this is also important on the lower level, e.g. columns in tuples).*

Note that we distinguish transformation from re-ordering. The former is the application of a certain pattern on a sub-graph, changing its structure, while the later is a simple re-ordering of the PEs which does not change the structure of a sub-graph.

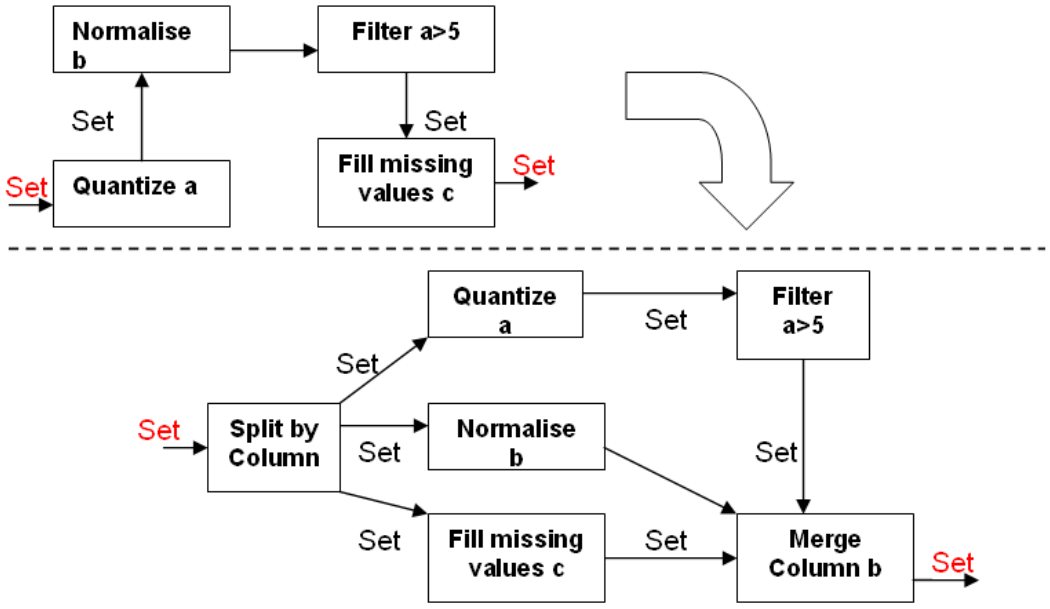


Figure 16: Applying Split/Merge transformation heuristics on part of cleaning process.

4.3.3 Re-Ordering

In order to allow the optimizer to perform PE re-ordering, another description requirement of PEs has to be fulfilled.

Requirement 4: *PEs have to define the relation of input cardinality and output cardinality.*

Having this information at hand we can introduce another transformation heuristic for DMI optimization.

Transformation Heuristics 2: *Re-order PEs in such a manner that cardinality-reducing ones are applied on the data flow as early as possible whilst preserving data dependencies and intended data flow semantics.*

Note that in Figure 16 we just re-ordered the PEs in the sequence which we parallelized, while in fact it might be possible to apply the *'Filter'* PE much earlier, e.g. after the custom PE *'Set Transform'*, thereby reducing the number of tuples even earlier in the data flow processing.

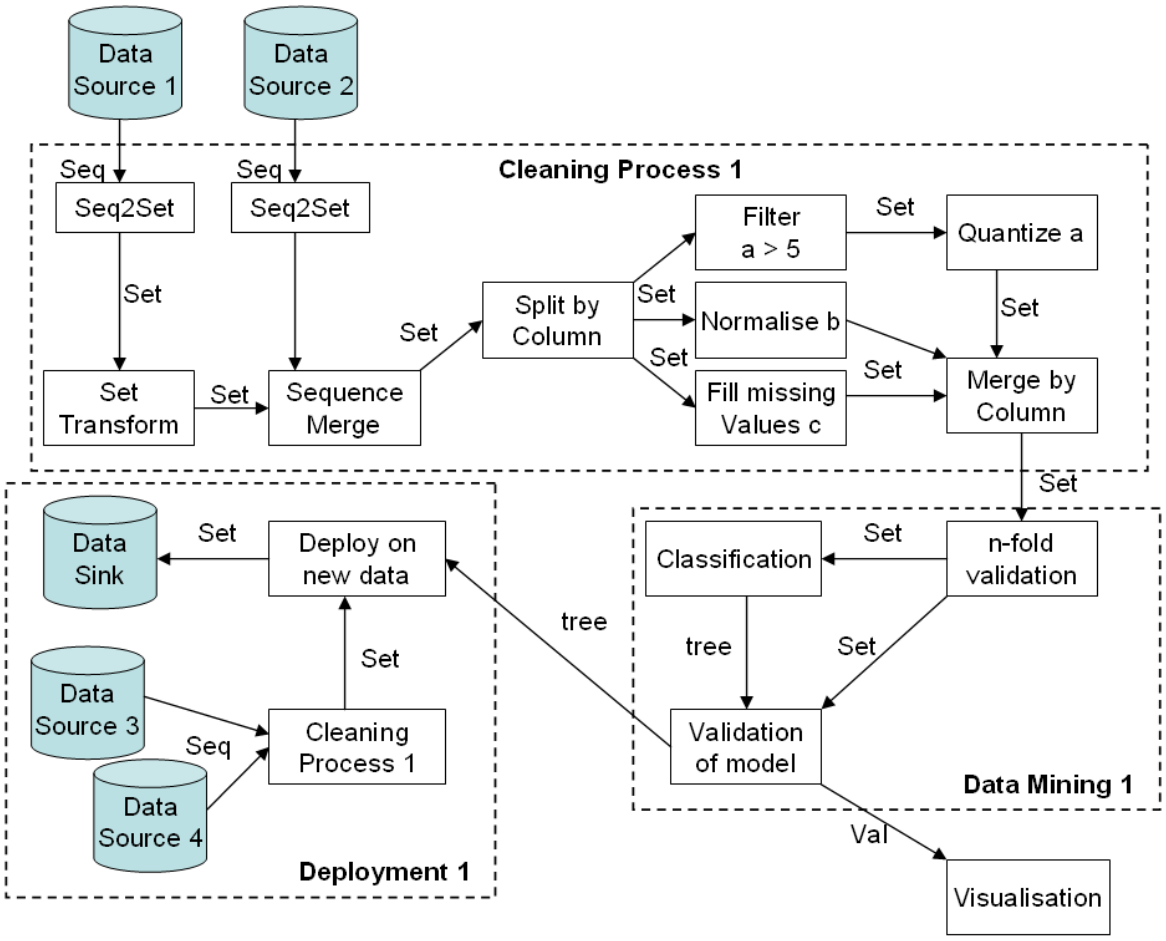


Figure 17: An equivalent data integration and mining process to that shown in the Figure above where the activities are rearranged to improve the performance by allowing activities to execute in parallel.

4.3.4 Handling Loops

The second dashed box in Figure 15 and 17 are a re-usable pattern named k-fold cross validation. K-fold cross validation [20] is used in data mining to determine how accurately a mining algorithm will be able to predict data that it was not trained on. When using the k-fold method, the training dataset is randomly partitioned into k groups. The learning algorithm is then trained k times, using all of the training set data points except those in the k-th group. Listing 4 shows a representation of the k-fold cross validation pattern in DMIL using loops for setting up the connections between the PE arrays. A simplified graphical representation is given in Figure 18 for $k = 2$.

From the graphical representation it can easily be observed that the DMIL explicitly specifies a parallel execution of the k-fold cross validation pattern where the sub-graph in each dashed box represents one iteration of the for-loop. It uses problem (data) partitioning [14], where the same data processing graph is applied independently on different parts of the data, achieved by the *'ListRandomSplit'* PE.

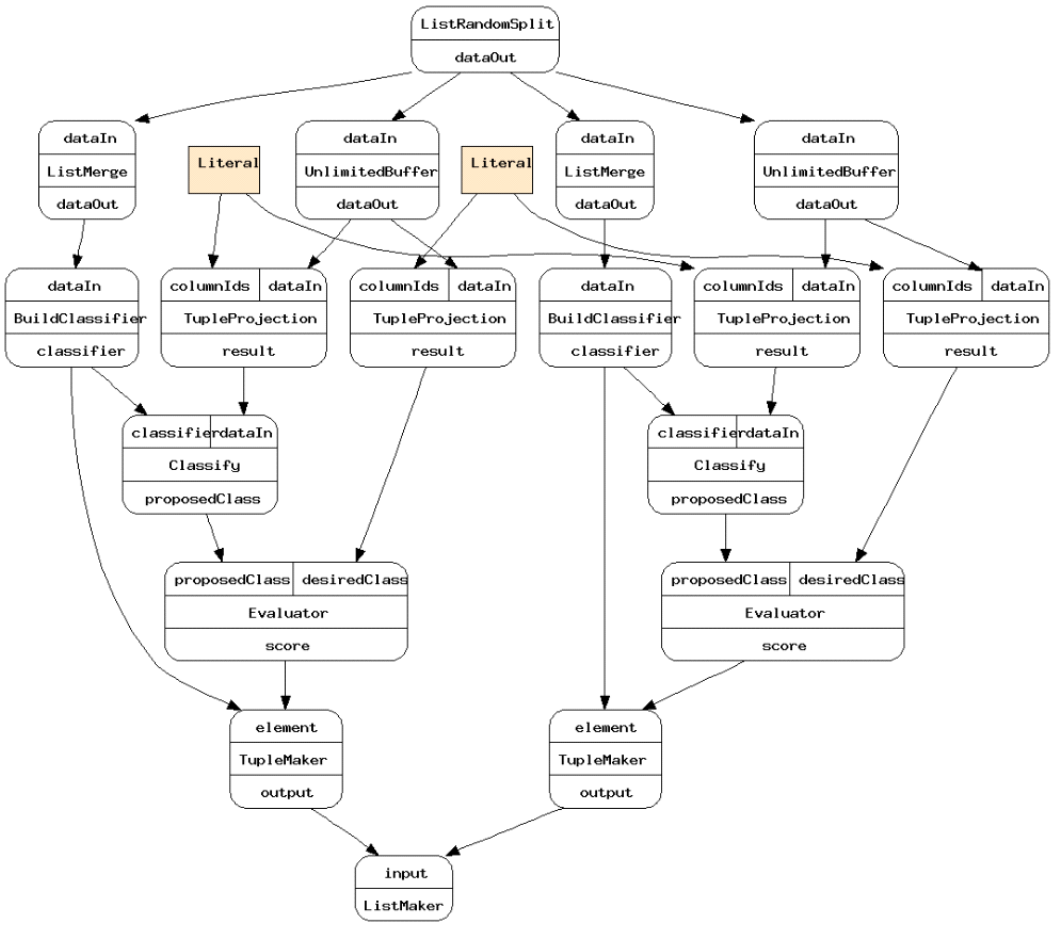


Figure 18: Simplified graphical representation of k-fold cross validation for k=2.

4.4 Cleaning Process: Graphical Design and Optimization

4.4.1 Graphical Representation

The visualisation of the unoptimized process using the graphical notation described in Section 3.4 and used by the Process Designer is shown in Figure 19 on page 31.

4.4.2 DMIL Representation Before Optimization

The graphical representation depicted in Figure 19 is automatically converted by the Process Designer into the form shown in Listing 5 on page 33.

4.4.3 DMIL Representation After Optimization

Basically, a DMIL code after parallelising optimisations proposed in this document is just a serial code with directives called pragmas placed at appropriate points.

A pragma is a special instruction to the compiler. Also called a pseudoinstruction, the pragma does not change the meaning of a program. It simply passes information to the DMIL compiler and interpreter processing the DMIL code in the Gateway. In DMIL a pragma

(directive) takes the form:

//@DMIL:...

In the following code, the directives are expressed in the spirit of the OpenMP standard [28]. The construct

//@DMIL:PARALLEL
code-block

where *code-block* denotes a DMIL statement or a list of DMIL statements enclosed in { and }, specifies that a team of processes or thread can execute *code-block*.

The **SECTIONS** directive is conceptually a work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the processes (threads) in the team specified by //@DMIL:PARALLEL.

Example of an appropriate code pattern:

```
//@DMIL:PARALLEL
   {
      //@DMIL:SECTIONS

         {
            //@DMIL:SECTION
                 code-block
            //@DMIL:SECTION
                 code-block
              ...

            //@DMIL:SECTION
                 code-block
          }
    }
```

For example, if there are 10 sections, they can be executed by one, two, ..., ten process, or more if there is a nested parallelism; the work is appropriately scheduled and balanced. The optimized code with the inserted pragmas is shown in Listing 6 on page 34. In this form it is passed to the Gateway.

Remark: Directives PARALLEL, SECTIONS and SECTION can include optional parameters specifying e.g., which variables (datasets) shall be considered as private for executing processes (threads).

4.5 Summary

This section on DMI process optimization identified characteristics of DMIL and their implications for an automatic DMI Process Optimization Engine. By elaborating on the example of a typical data integration and mining process, requirements on Process Element descriptions have been defined as well as initial transformation heuristics on top of them. The importance of the well-defined handling of composite types has been illustrated via composite type propagation as a basis for later transformations and PE re-ordering. The implications of loops for setting up connections between arrays of PEs were studied.

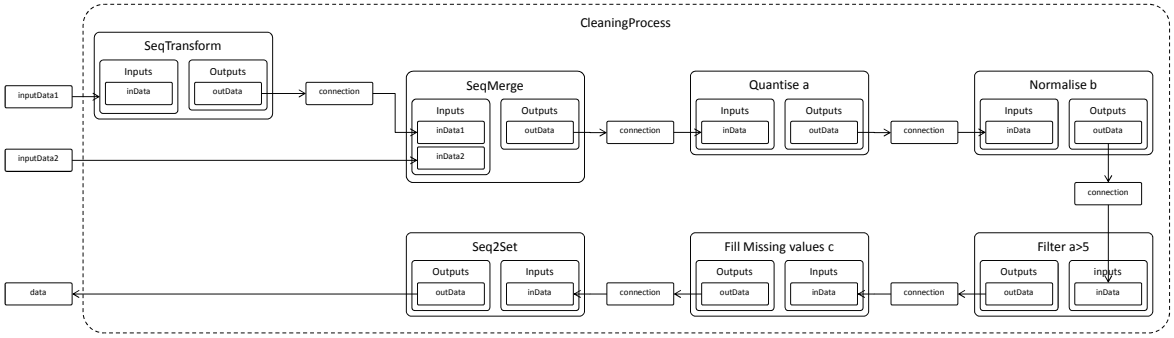


Figure 19: Representation of the cleaning process using DMIL graphical notation

Future work will include detailed elaboration of all three sub problems: type propagation, transformations patterns and PE re-ordering; as well as developing DMI POEs implementing them and their evaluation with respect to confidence and performance.

Listing 4: k-fold cross validiation pattern in DMIL.

```
use eu.admire.TupleProjection;
use eu.admire.TupleMaker;
use eu.admire.Classify;
use eu.admire.UnlimitedBuffer;
use eu.admire.ListMerge;
use uk.org.ogsadai.ListRandomSplit;
use eu.admire.ListMaker;
use eu.admire.BuildClassifier;
use eu.admire.Evaluator;

Connection inputVariables;
Connection outputVariables;
Integer k = 2;
ListRandomSplit lrs = new ListRandomSplit(k);
UnlimitedBuffer[] buffer = new UnlimitedBuffer[k];
ListMerge[] listMerge = new ListMerge[k];
TupleProjection[] projectInputVariables = new TupleProjection[k];
TupleProjection[] projectOutputVariables = new TupleProjection[k];
BuildClassifier[] buildClassifier = new BuildClassifier[k];
Evaluator[] evaluator = new Evaluator[k];
Classify[] classify = new Classify[k];
TupleMaker[] tupleMaker = new TupleMaker[k];
ListMaker listMaker = new ListMaker();
for (Integer i = 0; i < k; i = i+1) {
   for (Integer j = 0; j < k; j = j+1)  {
      if (i == j)  {
         lrs.dataOut[j] => buffer[i].dataIn;
      }
      else  {
         lrs.dataOut[j] => listMerge[i].dataIn[j];
      }
   }
   listMerge[i].dataOut => buildClassifier[i].dataIn;
   inputVariables => projectInputVariables[i].columnIds;
   buffer[i].dataOut => projectInputVariables[i].dataIn;
   buildClassifier[i].classifier => classify[i].classifier;
   projectInputVariables[i].result => classify[i].dataIn;
   classify[i].proposedClass => evaluator[i].proposedClass;
   buffer[i].dataOut => projectOutputVariables[i].dataIn;
   outputVariables => projectOutputVariables[i].columnIds;
   projectOutputVariables[i].result => evaluator[i].desiredClass;
   buildClassifier[i].classifier => tupleMaker[i].element[0];
   evaluator[i].score => tupleMaker[i].element[1];
   tupleMaker[i].output => listMaker.input[i];
}
```

Listing 5: Cleaning process in DMIL.

```
use eu.admire.Seq2Set;
use eu.admire.SeqTransform;
use eu.admire.SeqMerge;
use eu.admire.Quantize;
use eu.admire.MergeColumn;


function MergeAndCleanPattern-Seq (

PE (<Connection inData> => <Connection outData>) Filter,
PE (<Connection inData> => <Connection outData>) Normaliser,
PE (<Connection inData,Connection replacer> => <Connection outData>)
    MissingValuesFiller
) PE (<Connection inData1, inData2> => <connection outData>) {


Connection inputData1;
Connection inputData2;


/* Initialize phase */


SeqTransform transform = new SeqTransform();

SeqMerge merge = new SeqMerge();

Quantize quantize = new Quantize();

Filter filter = new Filter();

Normaliser normaliser = new Normaliser();

MissingValuesFiller filler = new MissingValuesFiller();

Seq2Set seq = new Seq2Set();


/* Transform phase */

inputData2 => transform.inData;

/* Merge phase */

inputData1 => merge.inData1;
transform.outData => merge.inData2;

/* Cleaning phase */

merge.outData => quantize.inData;

quantize.outData => normaliser.inData;

normaliser.outData => filter.inData;

filter.outData => filler.inData;

filler.outData => seq.inData;


return PE(<Connection inData1 = inputData1; Connection inData2 = inputData2> =>
                                        <Connection outData = seq.outData>);
}
```

Listing 6: Cleaning process in DMIL after optimization.

```
use eu.admire.Seq2Set;
use eu.admire.SetTransform;
use eu.admire.SetMerge;
use eu.admire.SplitColumn;
use eu.admire.Quantize;
use eu.admire.MergeColumn;

function MergeAndCleanPattern-Par (

PE (<Connection inData> => <Connection outData>) Filter,
PE (<Connection inData> => <Connection outData>) Normaliser,
PE (<Connection inData,Connection replacer> => <Connection outData>)
    MissingValuesFiller
) PE (<Connection inData1, inData2> => <connection outData>) {

Connection inputData1;
Connection inputData2;

/* Initialize phase */
Seq2Set seq1 = new Seq2Set();
Seq2Set seq2 = new Seq2Set();
SetTransform transform = new SetTransform();
SetMerge mergeSet = new SetMerge();
SplitColumn split = new SplitColumn();
Filter filter = new Filter();
Normaliser normaliser = new Normaliser();
MissingValuesFiller filler = new MissingValuesFiller();
Quantize quantize = new Quantize();
MergeColumn mergeColumn = new MergeColumn();

/* Transform phase */
inputData1 => seq1.inData;
inputData2 => seq2.inData;
seq2.outData = transform.inData;

/* Merge phase */
transform.outData => merge.inData1;
seq2.outData => merge.inData2;

/* Split phase */
merge.outData => split.inData;

/* Clean phase */
//@DMIL:PARALLEL
   {
      //@DMIL:SECTIONS
          {
             //@DMIL:SECTION
                 {
                   split.outData1 => filter.inData;
                   filter.outData => quantize.inData;
                 }
            //@DMIL:SECTION
                 split.outData2 => normaliser.inData;
            //@DMIL:SECTION
                 split.outData3 => filler.inData;
          }
    }

/* Merge phase */
quantize.outData => mergeColumnt.inData1;
filler.outData => mergeColumnt.inData2;
normaliser.outData => mergeColumnt.inData3;

return PE(<Connection inData1 = inputData1; Connection inData2 = inputData2> =>
        <Connection outData = mergeColumn.outData>);
}
```

5 DMI Ontology

In this section we describe the Platform Ontology and how it interacts with the other ADMIRE components, especially the ADMIRE Registry [23] and the Process Designer. Later we describe how the processing elements — key components in the ADMIRE architecture —n are described and used in the ontology.

5.1 Platform Ontology

The Process Designer offers to users a set of operators from which to build DMI workflows. These operators are basically Processing Elements. The users choose the Processing Elements that they need and try to build a new DMI workflow [6]. The development of a DMI workflow is a complex task — it is necessary to know what these PEs do and how to connect them. Users need information about what the PEs do and what their inputs and outputs are. The *Platform Ontology* represents the lowest elements of the CRISP-DMI model that the users access when designing a new DMI process. In this ontology we represent the most common PEs, the "Structural Types" (ST), which are the representations of the elements that implement the PEs and their inputs and outputs. These descriptions are complemented with descriptions of logic axioms to allow the Process Designer to provide the users with guidance in the process of designing DMI workflows. Figure 20 shows the main classes of the Platform Ontology; we describe them below.

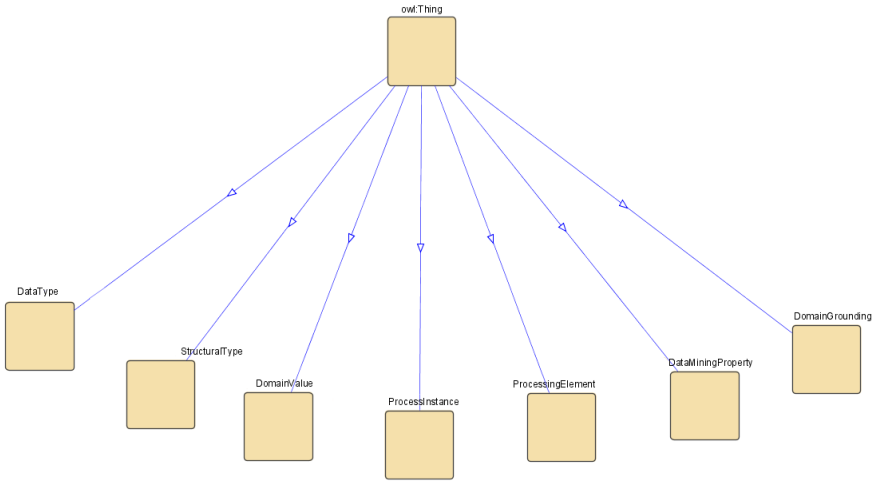


Figure 20: Platform Ontology Main Classes

- class "ProcessingElement"

  In ADMIRE a Processing Element is "a primitive or composite software component encapsulating a DMI algorithm and providing for its use in DMI processes. Multiple instances may be used in one process. It has specified structure of inputs and outputs". Thus in the Platform Ontology PEs are described as specified. There are two subclasses of "ProcessingElement" which are "PrimitiveProcessingElement" and "CompositeProcessingElement" [3]. Figure 21 shows some of the PEs represented in the Platform Ontology.

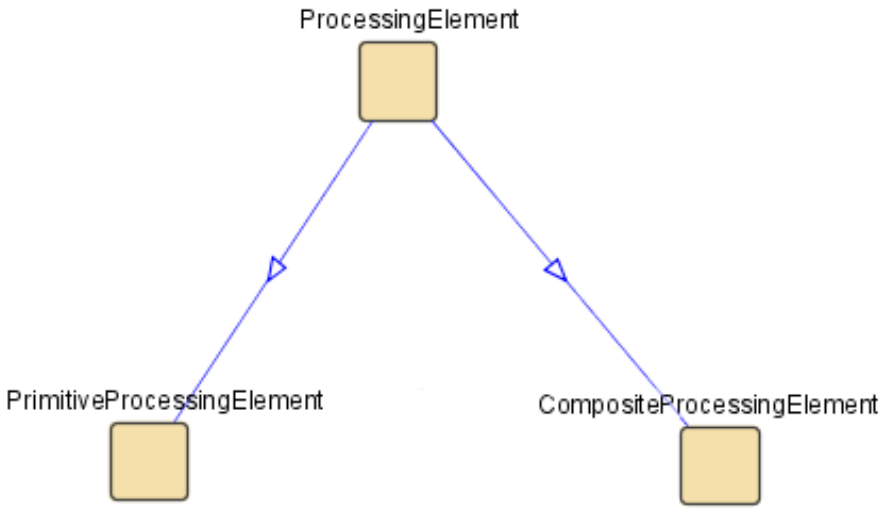


Figure 21: Platform Ontology Processing Elements representation

  - subclass "PrimitiveProcessingElement"

    This class represents the PEs which are not composed by other PEs. These PEs represent primitive functions that may be enacted and perform some data mining operation. The subclasses of the "PrimitiveProcessingElement" class are descriptions of the data processing steps that users may need in the development of their DMI workflows. Some of these subclasses are "BuildClassifier", "Classify", "Merge", "DataAccess", "SQLQuery" or "SQLUpdate". Each "PrimitiveProcessingElement" has its corresponding implementation class which is implemented by a "StructuralType".

  - subclass "CompositeProcessingElement"

    This class represents the PEs which are composed by two or more PEs. The definition of this type of PE is based on Description Logics axioms and it has no subclasses. In the class definition we specify that a "CompositeProcessingElement" contains some PEs and the implementing ST will define the PEs that composes it.

- class "StructuralType"

  This class represents the PE implementation functions which can be executed. In this class the inputs and outputs of these functions and methods are represented. Typically these STs are named using the name of the PE and adding "Service" at the end. This "Service" word emphasizes the implementation status of the ST. It does not mean that it is necessarily implemented by a Web service — it is simply the notation selected for naming them. The subclasses of "StructuralType" are "BuildClassifierService", "ClassifyService", "MergeService", "DataAccessService", "SQLQueryService", "SQLUpdateService", etc. Figure 22 shows some of the Structural Types represented in the Platform Ontology which implement the PEs.

  - The "StructuralType" class contains three relations, "composedOf", "hasInput" and "hasOutput" (which are inherited by its subclasses). The relation "com-

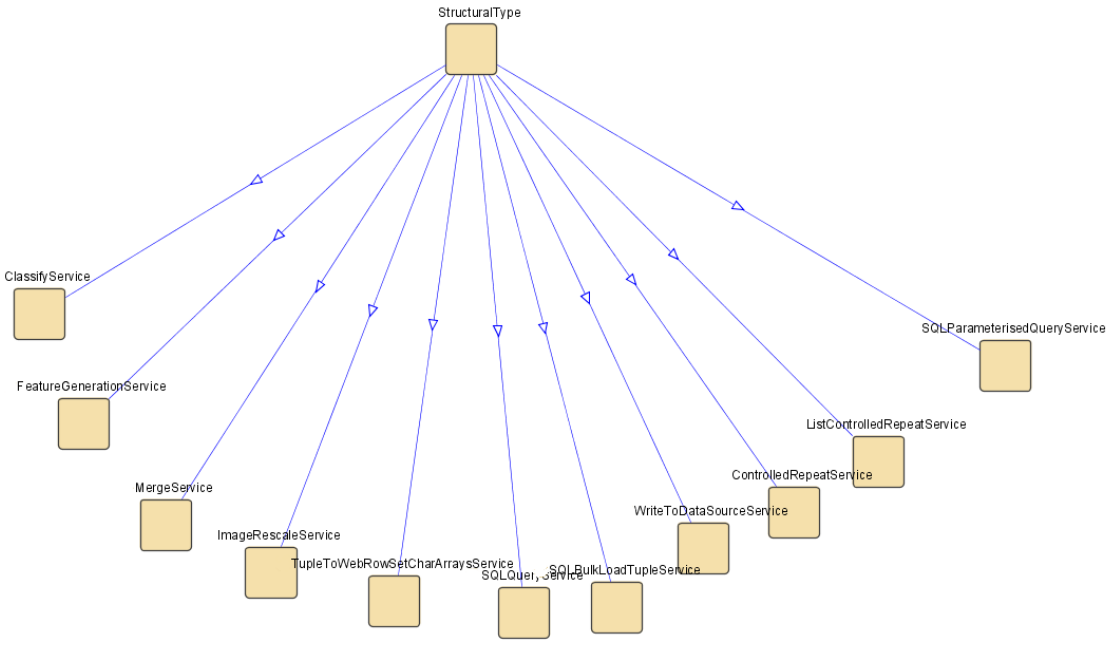


Figure 22: Platform Ontology Structural Types Representation

posedOf" is used to indicate that an ST is composed of several STs. This represents a "CompositeProcessingElement" described before. The relations "hasInput" and "hasOutput" represent respectively the inputs and outputs that an ST has. These relations have as their domain the "DataType" class which represents the parameters of a function/method implementing a PE.

– DL Axioms

The DL axioms used in this class are two. First an axiom that specifies the disjoint subclasses that the ST has. This is used in the PE for providing feedback about which ST can be used at a certain point at design time. The second axiom is a "covering axiom", which is used to make explicit the need of a ST to be one of its subclasses. This also allows the PE to launch a warning should a user do something wrong when designing a DMI workflow.

- class "DataType"

  The "DataType" class represents the data types used by the inputs and outputs represented in the "StructuralType" class. The subclasses of "DataType" are "EPR" (including "DataEPR" or service EPRs), "Histogram", "Integer", "SQLSentence" and "TupleList". These are the initial data types so represented but more will be added through extended representation of the PEs in the ontology. Figure 23 shows the current data types represented in the ontology.

Other classes in the ontology are "DataMiningProperty", "DomainGrounding", "DomainValue" and "ProcessInstance" which complete the description of the data mining domain and

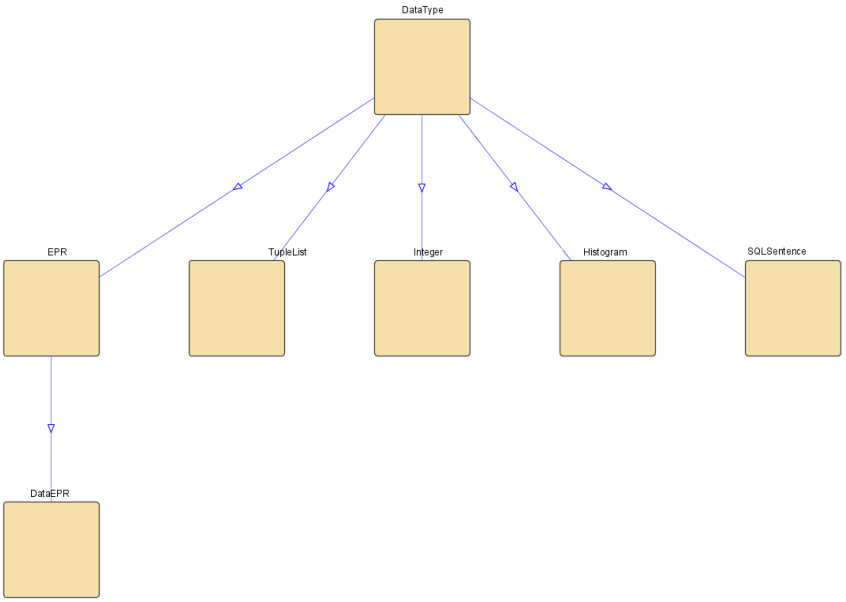


Figure 23: Platform Ontology Data Types representation

provide the link between this ontology and the others.

5.2 Relation of the Platform Ontology with the ADMIRE Components

The Platform Ontology is integrated with other components of ADMIRE and interacts with them to provide guidance to the user in the creation of DMI workflows. It also helps the Process Designer in locating the PEs stored in the ADMIRE system. The components with which the Platform Ontology interacts are the Process Designer, the ADMIRE Registry and the Semantic Data Descriptor Assistant (SDDA) [21].

The Process Designer. Interaction with the Process Designer occurs through 'recommendations' at design time. The user interacts with the ontology at design time, creating instances of it. When the user is designing their DMI workflow, they create instances in the Platform Ontology at the same time. When these instances are created consistency checking is done at the ontology level. This checking allows the Process Designer to validate what users are doing and provide suggestions to them.

The ADMIRE Registry. With the Registry, interactions happen when a user creates a DMI workflow. At some point in time the user will need to access existing PEs developed by other users; the Process Designer will communicate with the Registry, asking for these PEs. The Registry accesses the ontology in which the PEs are described and wherein are stored the existing instances of these PEs. Additionally, the ADMIRE Gateway uses the Registry to look up supported processing elements to assist in the interpretation of submitted DMIL requests.

The Service and Data Description Assistant (SDDA). The interaction of the Platform Ontology with the SDDA is rooted in the interaction of the users with the Process Designer. When users create their DMI workflows they also create instances of the classes in the Platform Ontology *through the agency of the SDDA* (as implemented as an Eclipse plugin). These instances represent the PEs that are being created/accessed by the users.

5.3 Processing Elements

As it is described in [3] Processing Elements are "A primitive or composite software component encapsulating a DMI algorithm and providing for its use in DMI processes. Multiple instances may be used in one process. It has a specified structure of inputs and outputs". The initial list of ADMIRE PEs is defined by the ADMIRE use cases; here we present some examples of existing PEs, together with their OWL representation in the ontology:

- *SQLQuery* — SQLQuery states there are two inputs and one output and gives their application-domain interpretation identifiers as SQLquery, RDBidentity and ResultSet respectively. It consumes data first from 'dataResource' and then from 'expression' and it always produces a value, with a structural type of 'list of tuples'. The tuples will have at least one element, with any identifier and any type. The PE stops when either input is exhausted, but it is an error if they are not both exhausted (note the interaction with the semantics of 'repeat enough').

```
name "SQLQuery"

inputs <
    expression : String : SQLquery
    dataResource : EPR : RDBidentity
    >

outputs <
    data : [<anyId: any(T1), ... >] :ResultSet
    >

iteration lockstep(dataResource; expression) -> data

stop empty(dataResource) OR empty(expression) OR notWanted(data)

cardinality card(dataResource) = card(expression) = card(data)

error (empty(dataResource) AND NOT empty(expression)) OR
      (empty(expression) AND NOT empty(dataResource))
```

The OWL representation of the SQLQuery PE:

```
<owl:Class rdf:about="#SQLQuery">
    <rdfs:subClassOf rdf:resource="#PrimitiveProcessingElement"/>
      <rdfs:subClassOf>
          <owl:Restriction>
            <owl:onProperty
                rdf:resource="&CRISP-DMIOntology;hasStructuralType"/>
            <owl:allValuesFrom rdf:resource="#SQLQueryService"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>
```

```
<!--
     http://www.admire-project.eu/ontologies/PlatformOntology.owl#SQLQueryService
    -->

<owl:Class rdf:about="#SQLQueryService">
   <rdfs:subClassOf rdf:resource="&CRISP-DMIOntology;StructuralType"/>
</owl:Class>
```

- *Merge* — The Merge PE is a simple merge that takes data from both of its inputs in arbitrary order and puts the same values out onto the stream *data*. The two input types must be the same, but can be any type; they determine the type of the output. It consumes all of the available data from each of its inputs and then stops.

```
name "Merge"

inputs <
  input1 : any(T)
  input1 : any(R)
  >
outputs <
  data : any(S)
  >
type_rules T = R = S

iteration random(input1 | input2) -> data

stop (empty(input1) AND empty(input2)) OR notWanted(data)

cardinality card(input1) + card(input2) = card(data)

error T != R
```

And in OWL:

```
<owl:Class rdf:about="#Merge">
    <rdfs:subClassOf rdf:resource="#PrimitiveProcessingElement"/>
    <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty
              rdf:resource="&CRISP-DMIOntology;hasStructuralType"/>
          <owl:allValuesFrom rdf:resource="#MergeService"/>
        </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<!-- http://www.admire-project.eu/ontologies/PlatformOntology.owl#MergeService
    -->

<owl:Class rdf:about="#MergeService">
     <rdfs:subClassOf rdf:resource="&CRISP-DMIOntology;StructuralType"/>
</owl:Class>
```

- *SortedListMerge* — The PE SortedListMerge takes two input streams and incrementally consumes a value from both inputs. These are each sorted lists. It merges the values

from these two lists and incrementally produces an output list on *data*. When both lists have been consumed (from *input1* and *input2*) it terminates the current output list and starts on the next pair of input lists.

```
name "SortedListMerge"

inputs <
   input1 : [any(T)]
   input2 : [any(R)]
   >
outputs <
   data : [any(S)]
   >
type_rules T = R = S

iteration lockstep(incrementally(input1);incrementally(input2)) ->
    incrementally(data)

stop empty(input1) OR empty(input2) OR notWanted(data)

cardinality
  card(input1) = card(input2) = card(data)
  card(elem(input1)) + card(elem(input2)) = card(elem(data))

error (T != R) OR (NOT ascending(input1)) OR (NOT ascending(input2)) OR
       (empty(input1) AND NOT empty(input2)) OR
       (empty(input2) AND NOT empty(input1))
```

In OWL:

```
<owl:Class rdf:about="#SortedListMerge">
     <rdfs:subClassOf rdf:resource="#PrimitiveProcessingElement"/>
     <rdfs:subClassOf>
          <owl:Restriction>
             <owl:onProperty
                  rdf:resource="&CRISP-DMIOntology;hasStructuralType"/>
              <owl:allValuesFrom rdf:resource="#SortedListMergeService"/>
          </owl:Restriction>
     </rdfs:subClassOf>
</owl:Class>

<!--
    http://www.admire-project.eu/ontologies/PlatformOntology.owl#SortedListMergeService
    -->

<owl:Class rdf:about="#SortedListMergeService">
     <rdfs:subClassOf rdf:resource="&CRISP-DMIOntology;StructuralType"/>
     <rdfs:subClassOf>
          <owl:Restriction>
             <owl:onProperty rdf:resource="&CRISP-DMIOntology;hasInput"/>
             <owl:onClass rdf:resource="&CRISP-DMIOntology;DataType"/>
             <owl:cardinality
                  rdf:datatype="&xsd;nonNegativeInteger">2</owl:cardinality>
          </owl:Restriction>
     </rdfs:subClassOf>
</owl:Class>
```

5.4 Summary

In this section we have described the Platform Ontology which is used in ADMIRE for describing the Processing Elements that the users will create and the those that already exist. The creation of these Processing Elements will generate instances that will be stored in RDF foramt and will be available for the users via the ADMIRE Registry. In the ontology we described the relations and restrictions that each PE has, among others their inputs and outputs. We showed the relation of the ontology with the other components of ADMIRE as well.

The next steps in the development of the ontology are to link the elements of this ontology with the other ontologies (integrating all the descriptions of the ADMIRE project, which are in the different ontologies i.e. CRISP-DMI Ontology or DM Ontology) and test the Platform Ontology by means of the SDDA creating instances of PEs and the ADMIRE registry, which will access these instances using SPARQL.

6 Future Work

In this final section we describe future work for the next six months. This work can be divided into two steps. The first one is a finalization of the DMIL language and its components and the second one is a tight integration of the DMI ontologies. In order to accomplish the first step we will need to describe DMIL semantics as well as to complete definitions of the abstract and concrete syntaxes. The semantics is currently being described as a part of the ADMIRE WhitePaper [3], and in the next months we will focus on a more systematic description of all the main concepts of DMIL . To cover the whole abstract syntax of DMIL we will present a complete set of models for each metamodel package. This will include taxonomies and modeling of concepts relations. The complete version of the concrete textual syntax will be released during the next six months together with the graphical notation. The concrete textual and graphical syntax will also be included in the next version of the Process Designer, which will be released at the of PM24. Regarding the second step, we will integrate the already developed ontologies (i.e. CRISP-DMI, Data Mining and Platform ontologies) and form the ADMIRE ontology.

Thus, we can summarise our research goals for PM24 as:

DMIL Semantics — complete a detailed description of the language concepts;

Abstract syntax — specify DMIL-m + taxonomies + types;

Concrete syntax — complete versions of DMIL-t + DMIL-g;

ADMIRE ontology — integrate CRISP-DMI, Data Mining and Platform ontologies.

References

[1] R. Sethi A. V. Aho and J. D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1988.

[2] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures.* Morgan Kaufmann, 2002.

[3] Malcolm Atkinson, Peter Brezany, Oscar Corcho, Liangxiu Han, Jano van Hemert, Ladislav Hluchý, Ally Hume, Ivan Janciak, Amy Krause, and Dave Snelling. ADMIRE White Paper: Motivation, Strategy, Overview and Impact. Technical Report v0.9, the ADMIRE Project, January 2009.

[4] Wim Bast, Mariano Belaunde, Xavier Blanc, Keith Duddy, Catherine Griffin, Simon Helsen, Michael Lawley, Michael Murphree, Sreedhar Reddy, Shane Sendall, Jim Steel, Laurence Tratt, R. Venkatesh, and Didier Vojtisek. MOF QVT final adopted specification. Technical report, OMG, 2005.

[5] Peter Brezany, Carlos Buil, Ivan Janciak, and Sabri Pllana. ADMIRE – DMI Model, Language and Ontology. Deliverable report D1.2, the ADMIRE Project, Feb 2009.

[6] Peter Brezany, Ehtesham ul Haq Dar, Ibrahim Elsayed, Yuzhang Han, Ivan Janciak, Fakhri Alam Khan, Sabri Pllana, Yuan Tian, Alexander Wöhrer, Malcolm Atkinson, Jano van Hemert, Oscar Corcho, Carlos Buil Aranda, Marian Babik, and Rob Baxter. ADMIRE – Towards the High-Level DMI Model, Language and Ontology. Deliverable report D1.1, the ADMIRE Project, August 2008.

[7] Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling (SoSyM)*, 3(4):314–327, December 2004.

[8] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 35–43, New York, NY, USA, 2005. ACM.

[9] Y. Huang D. W. Walker, O. F. Rana and L. Huang. Workflow optimisation for e-science applications. In *International Conference on Information Technology Interfaces*, 2005.

[10] M. Tsangaris et al. Dataflow processing and optimization on grid and cloud infrastructures. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2009.

[11] Richard C. Gronback. *Eclipse modeling project : a domain-specific language toolkit.* Addison-Wesley, 1 edition, April 2009.

[12] L. Han, C. Liew, J. Hemert, M. Atkinson, and A. Hume. Facilitating data mining and integration using pipeline. In *International Conference on e-Science*, 2009.

[13] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, Oct. 2004.

[14] M.S. Hecht. *Flow Analysis of Computer Programs.* North Holland, 1977.

[15] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *ECMDA-FA '09: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Y. Ioanidis. Query optimization. In *ACM Computing Survey*, 1996.

[17] M. Atkinson J.V. Hemert and P. Brezany. Composite types in the context of parallel service-oriented architectures for data mining and integration. In *ADMIRE DELIVERABLE XY*, 2009.

[18] Fakhri Alam Khan, Yuzhang Han, Sabri Pllana, and Peter Brezany. Provenance support for grid-enabled scientific workflows. *Semantics, Knowledge and Grid, International Conference on*, 0:173–180, 2008.

[19] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison Wesley Pub Co Inc, 1 edition, January 2009.

[20] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence*, 2009.

[21] Amy Krause, Carlos Buil, Rafał Gąsiorowski, Branislav Simo, Michal Laclavik, Ivan Janciak, and Rob Baxter. ADMIRE – Tools Development Report and Requirements Analysis. Deliverable report D5.2, the ADMIRE Project, Feb 2009.

[22] Amy Krause, Ivan Janciak, Michal Laclavik, Branislav Simo, Maciej Jarka, Andrzej Biernacki, and Marek Lenart. ADMIRE – Tools Development Progress Report. Deliverable report D5.3, the ADMIRE Project, Aug 2009.

[23] Vivian Lee and work package partners. ADMIRE – Development and Deployment Report for USMT V2: capabilities of USMT V2. Deliverable report D4.2, the ADMIRE Project, Feb 2009.

[24] Jan Pettersen Nytun, Andreas Prinz, and Merete Skjelten Tveit. Automatic generation of modelling tools. In *ECMDA-FA*, pages 268–283, 2006.

[25] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. Technical Report formal/06-01-01, OMG, 2006. OMG Available Specification.

[26] Object Management Group. Object Constraint Language (OCL) 2.0. Technical Report formal/06-05-01, OMG, 2006. OMG Available Specification.

[27] Object Management Group. XML Metadata Interchange (XMI), v2.1.1. Technical report, OMG, 2007.

[28] OpenMP. http://openmp.org/wp/openmp-specifications/.

[29] David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[30] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. In *SIGMOD*, 2005.