



---

## ADMIRE D1.4 – Paper – On the Systematic Design of DMIL

---

Project Title	ADMIRE
Document Title	ADMIRE – On the Systematic Design of DMIL
Deliverable Number	D1.4
Authorship	Peter Brezany, Ivan Janciak, Malcolm Atkinson, Jano van Hemert
Document Filename	ADMIRE-D1.4-paper.tex
Document Version	1.0
Distribution Classification	Public
Distribution List	ADMIRE Project Team
Approval List	Oscar Corcho, Mark Parsons, Project Manager, Executive Board

<i>Personnel</i>	<i>Date</i>	<i>Comment</i>	<i>Version</i>
IJ	03 Aug 2009	First draft	0.1
IJ	10 Aug 2009	Metamodelling	0.2
IJ	20 Aug 2009	Abstract Syntax	0.3
PB	24 Aug 2009	Abstract	0.4
IJ	13 Sep 2009	Addressed review comments	0.5
RMB	15 Sep 2009	Final edit and signoff	1.0



# On the Systematic Design of a Data Mining and Integration Language

Peter Brezany  
and Ivan Janciak  
Institute of Scientific Computing  
University of Vienna  
Email: brezany@par.univie.ac.at  
Email: janciak@par.univie.ac.at

M.P. Atkinson  
and J.I. van Hemert  
National e-Science Centre  
School of Informatics  
University of Edinburgh  
Email: mpa@nesc.ac.uk  
Email: j.vanhemert@ed.ac.uk

**Abstract**—Across a wide variety of fields a lot of large datasets are being collected and accumulated at a dramatic pace. There are major challenges involved in the efficient and reliable storage, integration, fast preprocessing and extraction of descriptive and predictive knowledge from this great mass of data by data mining processes. The EU project ADMIRE aims to extend the state-of-the-art in data mining and integration (DMI) to completely new technologies. DMI applications involve complex processes whose understanding and description requires new approaches. In this paper we introduce a framework for modelling and describing DMI processes using a new DMI Language (DMIL). This paper presents an approach to the design of the new language based on modelling concepts applied in Model Driven Architecture software development. We present the systematic proposal of the abstract syntax, concrete syntax, and graphical models of the language and the mappings between these models.

## I. INTRODUCTION

The European project *Advanced Data Mining and Integration Research for Europe* (ADMIRE) [1] is pioneering architectures, models, languages, and methodologies that will deliver a coherent, extensible and flexible framework to make the best use of a wide range of distributed data resources produced by modern science, business, industrial applications and other domains. Data mining and integration (DMI) is the process of combining data from different data sources and automating the extraction and visualization of patterns representing knowledge. These potentially useful and understandable patterns are implicit in large volumes of data.

The process of specifying a distributed data mining and integration system involves the capture of complex inter-relationships between data mining, data warehousing and workflow management system 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 being developed through such approaches as a Model Driven Architecture (MDA) [2]. 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 adopted MDA and its essential meta-modelling foundation as a base for the development of a new Domain Specific Language (DSL), which we name Data

Mining and Integration Language (DMIL). This language is used for the specification of DMI processes and is a key component of the ADMIRE architecture. Its notation is used to communicate DMI requests and is pivotal for the ADMIRE tools and enactment engines, which can be considered as optimized interpreters of DMIL sentences. Metamodelling provides the foundation for ADMIRE's vision, by enabling a meaningful set of DMIL metamodels to be defined precisely and unified in a consistent framework.

The MDA approach makes models and metamodels primary artefacts of software engineering and thus raises the level of abstraction in the software development process. Nyttun et. al [3] define metamodel as follows: *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 complete and precise specification of 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 is 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) [4], which is intended to support a range of usage patterns and applications by providing a set of standard interfaces to define and manipulate a set of interoperable metamodels and their corresponding models. In addition, the MOF can be combined with the Object Constraint Language (OCL) [5], which is a language that enables us to describe expressions and constraints on object-oriented models and object modelling 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.

This paper is organized as follows. Section II explains the principles of the model-based language design. The model levels considered are: a) abstract syntax, b) concrete syntax and c) graphical form. Their concepts and representations are introduced in this section, after which we introduce both the concrete syntax and graphical models. The concrete syntax

of DMIL is discussed in Section V. It has been understood by the scientific community that a graphical view of DMI processes could have advantages for the end user. Graphs allow easy communication of ideas, especially to novices, allowing much more productive exchanges between the DMI expert developer and the customer, and in general they are an optimal platform for DMI projects conducted in modern collaborative environments. In ADMIRE, graphical forms of DMI process requests are created by a tool called the *Process Designer*. These requests can be improved by applying a set of platform-independent optimizing transformations before conversion into the concrete syntax representation. These issues are discussed in Section VI. Finally, we conclude the paper in Section VII.

## II. A MODEL-BASED LANGUAGE

In this section we introduce an approach for designing a data mining and integration language using a well-defined formalism based on metamodelling that we have adopted for the design of DMIL. Based on Chen et al. [6] and Schmidt [7], 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 well-formed 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 [8]. 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 DMIL metamodel 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 still maintaining the same semantics  $M_S$ . Therefore, a metamodel could be intended as a standard representation of the language notation.

### A. MDA approach

MDA defines an architecture based on four abstraction layers. At the top layer (M3) of the architecture is the metametamodel, which provides a generic language for the definition of domain-specific language. The next layer (M2) is populated by metamodels that represent a metamodel-defined domain-specific language. Layer M1 hosts domain models written in M2-defined domain-specific languages. Finally, layer M0 hosts runtime domain objects that instantiate M1 domain entities. To define the abstract syntax of DMIL we are primarily targeting the M2 level of this layered architecture. An advantage of MDA is that it provides MOF for metamodel specification and OCL for the specification of metamodel constraints. Furthermore, MDA can facilitate the definition of mappings between domain-specific languages using QVT [9] and the exchange of metamodel data using XMI [10].

There are several formalisms (e.g. graph grammars, context free grammars, UML Profiling, etc.) to define 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 of the design of a domain-specific language. While we focus on a concrete domain it is necessary to produce a precise definition for the domain-specific semantics of the language, which is not possible using a generic modelling language such as UML. Further, UML's customization mechanism for defining domain-specific languages, called UML Profiling, is too restrictive for our needs because new concepts need to adapt themselves to other existing concepts. It means that one has to understand all the pre-existing metaclasses to identify the right base metaclasses of a newly defined stereotype. Since the DMIL language introduces new concepts which are not based on any typical process model, UML Profiling is not adequate as a formalism for the language specification. On the other hand, we must say 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.

The process of developing DMIL consists of the following steps. The first step defines a new abstract syntax metamodel, which represents essential concepts and their relations in the DMI domain. This model is based on a metamodel constructed using MOF and expressed using a set of class diagrams organized in packages. In the second step 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 construction of DMI processes based on DMIL.

### III. DMIL — A SOFTWARE DOMAIN-SPECIFIC LANGUAGE

The aim of DMIL is to provide a notation for communication about DMI requests. Its textual form is used to submit requests for specialized DMI task enactment through DMI Gateways [11]. It may also be used as a stored representation of DMI requests by tools, and as the input and output of DMI optimizers. Note that DMIL it is not a language for authoring data mining algorithms.

DMIL is primarily used 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 discussed in details in Section IV;
- a concrete textual (DMIL-t) and graphical representation (DMIL-g) of the language — it supports the design and interchange of the modelled DMI processes. The concrete textual syntax is a result of mapping process of the abstract syntax to the concrete syntax. An overview of concrete language representations is given in Section V;
- language semantics (DMIL-s) — this describes the meanings of the language concepts. These semantics can be implicitly or explicitly defined. The main concepts of DMIL are described in the ADMIRE Whitepaper [12].

The primary features of DMIL are:

- 1) the type system of the 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 organize 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.

### IV. ABSTRACT SYNTAX MODEL

The abstract syntax of DMIL is defined in terms of an object-oriented model, called the DMIL metamodel (DMIL-m). This metamodel characterizes syntax elements together with their relationships and separates the abstract syntax and semantics of the DMIL constructs from their concrete textual notation. Our metamodel is based on the MOF model and UML notation which are used as the modelling language and graphical notation respectively, for defining and representing the complete DMIL-m.

#### A. 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 1. Together, the collection of DMIL-m packages provides the necessary abstractions to model generic representations of DMI processes.

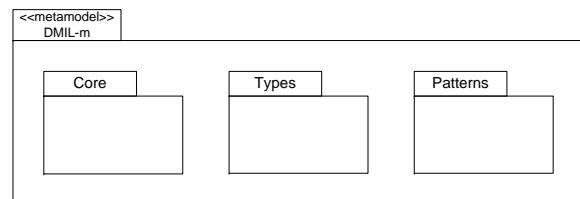


Fig. 1. DMIL-m packages structure

- 1) Core (eu.admire.dmil.core) — this package contains 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; and, c) application domain types that are used in a specific application domain.
- 3) Patterns (eu.admire.dmil.patterns) — this package contains predefined DMI patterns with recurring structure within DMI processes. In DMIL the patterns are defined using functions.

The definition of an abstract syntax by a metamodel is well supported by various metamodeling environments such as the Eclipse Modelling Framework (EMF) [13]. The framework

fully supports model-driven development and offers software engineers powerful tools to improve productivity and enhance quality in the design of systems. In the ADMIRE project Eclipse is used as the main development platform.

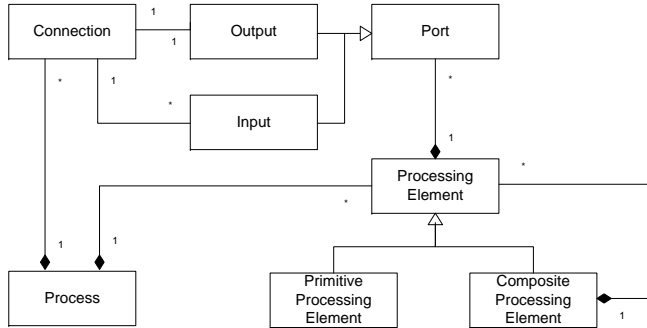


Fig. 2. Core Metamodel

### B. Core Package

The *Core* metamodel package contains the major classes and associations used to define DMI processes as presented in Figure 2.

1) *Processing Element Definition*: The *Processing Element* (PE) is a primitive or composite software component encapsulating a DMI request 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 3. A note on each of these properties is given below.

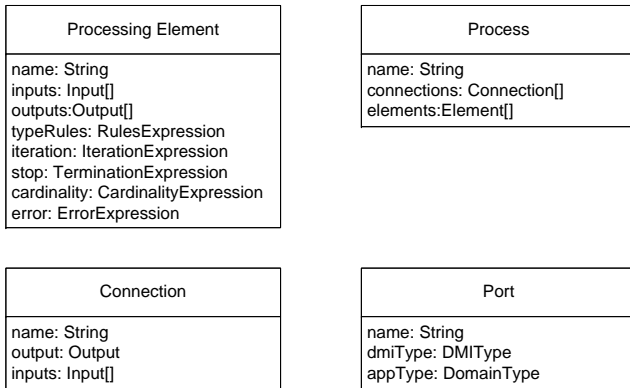


Fig. 3. The main classes of the Core Package

- 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 a 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 the termination that is 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.

```

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))
    
```

Listing 1. Example of Processing Element definition

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, where 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).

2) *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 an input and as an output for a particular PE. Variable numbers of inputs or outputs all with the same purpose are provided by using an array notation.

3) *Process Definition and Functions*: The 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, normalizing, 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 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, parameterize 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. Functions can be used to 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 parallelization targets and so on. The 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 parameterized and activated through a portal;
- the encoding of a pattern, such as repetition, parallelization, all-meets-all, etc.;
- the encoding of an optimization strategy.

### C. 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 PE; these are mathematically based. These types describe the data streaming along connections to and from PE that perform data mining algorithms;
- 3) *Domain Types* describe the data input into and output from PEs, and transmitted through connections that correspond with values in an application domain. There may be many versions of this type system for different application domains.

### D. 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 several PEs. DMIL functions abstract data-flow graphs, contrary to traditional functions which abstract

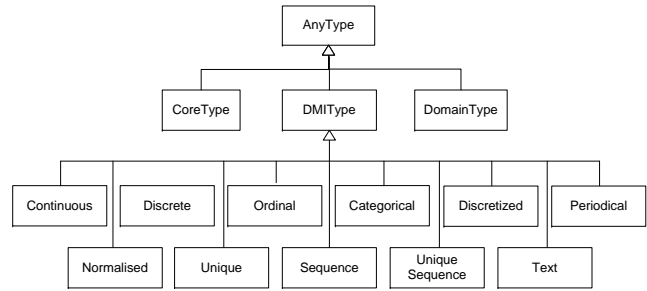


Fig. 4. An overview of DMI Types

a control-flow graph. Every DMIL function is expected to return a data-flow graph, which could be embedded inside an existing 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 for the composition of complex data-flow graphs from well-defined functional abstractions of recurring data-flow patterns.

```

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 2. Example of DMI process represented in DMIL-t

## V. CONCRETE SYNTAX

DMIL provides special textual and graphical notations for communicating about DMI processes. The concrete syntax play a crucial role in the design of DMIL since it is used to encode a DMI process as a request to the enactment engine represented by the Gateway in the ADMIRE architecture.

### A. 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.

```

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>;
  )
}

```

Listing 3. DMI function implementing a DMI pattern

### B. 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 5. The PEs are connected using the connections as shown in Figure 6. As we

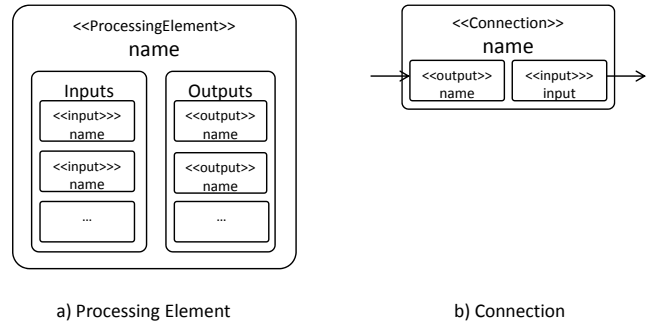


Fig. 5. Graphical notation of DMIL Processing Element and Connection

have already mentioned, DMIL is not designed to encode data mining algorithms but rather to compose graphs consisting of various PEs which implement those algorithms. The graphical notation is especially important for tools such as the Process Designer which can represent a DMI process in visually readable form.

## VI. PROCESS DESIGNER

In order to design DMIL processes we have developed a specialized tool named the Process Designer. The Designer is a graphical environment that allows DMI experts to compose PEs and create complex DMI processes. Additionally, it enables the experts to define new PEs by providing their characteristics, or building new PEs by composing the existing ones and their parameterization. As a result, the tool produces DMIL requests. Since we strictly follow the MDA approach, implementation is based on two major phases. The first one is the modelling phase during which an internal *Ecore* model is developed. The second phase is the code generation and code customization phase.

*Ecore* is a platform-independent metamodel used by Eclipse and its modelling framework (EMF) [13] 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 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), which specifies the visual environment of the Process Designer.

To support the design and implementation of text editing facilities many concrete syntax and model mapping tools have been considered. We have chosen EMFText [14], Eclipse's integrated tool for agile Textual Syntax development. EMFText allows one to define a plain textual syntax for an *Ecore*-based metamodel and generate components to load, edit and store model instances. This feature enables end users to use a special form of concrete syntax specification and generate DMIL textual representation directly from a model instance, and conversely to load model instances from textual representations.

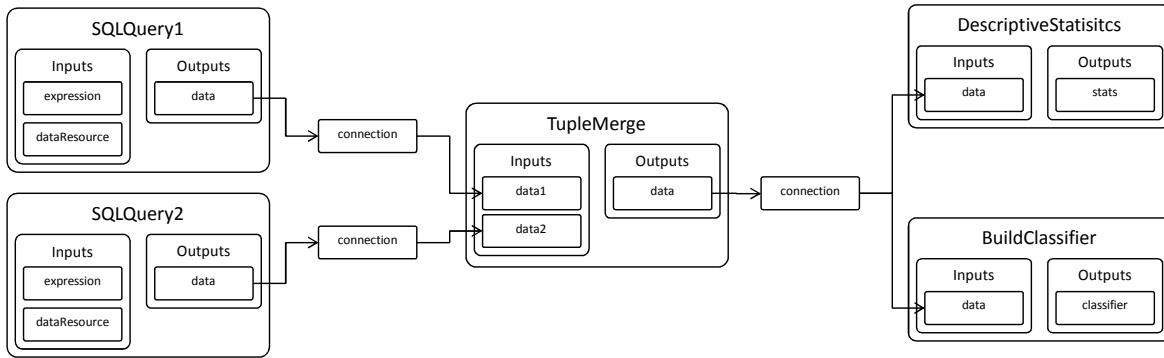


Fig. 6. Example of the DMIL graphical notation

## VII. SUMMARY

In this paper we have presented an overview of the architecture of DMIL and our systematic approach to its design based on metamodeling. Moving away from the traditional ad-hoc design methodology typically applied to the design of all workflow languages, brings two important benefits. Firstly, the design process leads to a more consistent, coherent, and complete language. This is achieved by the prescribed methodologies and their modelling languages. 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, such as Eclipse. This allows for the automatic generation of the target language sentences and delivers more effective development of domain-specific applications, based on DMIL concepts.

## ACKNOWLEDGMENT

The authors gratefully acknowledged the support of the ADMIRE Project and its members in this work which is funded under the European Union's Framework 7 programme as project FP7-ICT-215024. The ADMIRE Project builds on the work of its partners who funded from a variety of national and international sources.

## REFERENCES

- [1] "Advanced Data Mining and Integration Research for Europe," p. 100, May 2007, proposal accepted for EU FP7-ICT-2007-1.
- [2] A. W. Brown, "Model driven architecture: Principles and practice," *Software and Systems Modeling (SoSyM)*, vol. 3, no. 4, pp. 314–327, December 2004. [Online]. Available: <http://dx.doi.org/10.1007/s10270-004-0061-2>
- [3] J. P. Nyttun, A. Prinz, and M. S. Tveit, "Automatic generation of modelling tools," in *ECMDA-FA*, 2006, pp. 268–283.
- [4] Object Management Group, "Meta object facility (mof) 2.0 core specification," OMG, Tech. Rep. formal/06-01-01, 2006, oMG Available Specification. [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>
- [5] —, "Object constraint language (ocl) 2.0," OMG, Tech. Rep. formal/06-05-01, 2006, oMG Available Specification. [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>
- [6] K. Chen, J. Sztipanovits, and S. Neema, "Toward a semantic anchoring infrastructure for domain-specific modeling languages," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM, 2005, pp. 35–43.
- [7] D. A. Schmidt, *Denotational semantics: a methodology for language development*. Dubuque, IA, USA: William C. Brown Publishers, 1986. [Online]. Available: <http://portal.acm.org/citation.cfm?id=6879>
- [8] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of "semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, Oct. 2004.
- [9] W. Bast, M. Belaunde, X. Blanc, K. Duddy, C. Griffin, S. Helsen, M. Lawley, M. Murphree, S. Reddy, S. Sendall, J. Steel, L. Tratt, R. Venkatesh, and D. Vojtisek, "MOF QVT final adopted specification," OMG, Tech. Rep., 2005.
- [10] Object Management Group, "XML Metadata Interchange (XMI), v2.1.1," OMG, Tech. Rep., 2007. [Online]. Available: <http://www.omg.org/technology/documents/formal/xmi.htm>
- [11] M. Atkinson, J. van Hemert, L. Han, A. Hume, and C. S. Liew, "A Distributed Architecture for Data Mining and Integration," Feb 2009, Submitted to DADC'09.
- [12] M. Atkinson, P. Brezany, O. Corcho, L. Han, Jano van Hemert, L. Hluchý, A. Hume, I. Janciak, A. Krause, and D. Snelling, "ADMIRE White Paper: Motivation, Strategy, Overview and Impact," the ADMIRE Project, Tech. Rep. v0.9, January 2009.
- [13] R. C. Gronback, *Eclipse modeling project : a domain-specific language toolkit*, 1st ed. Addison-Wesley, April 2009.
- [14] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. 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*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 114–129.