

# META-MODELLING OF DATA FLOW PROCESSES WITH META-MODELLING TOOL ATOM<sup>3</sup>

ANDRIY LEVYTSKYI and EUGENE J.H. KERCKHOFFS

*Faculty of Information Technology and Systems, Mediamatica Department  
Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands  
Email: a.levytskyi@cs.tudelft.nl*

**Abstract:** this paper illustrates how meta-modelling is used to support designing and executing data flows in a web-based simulation environment (in our case the home-made so-called NCSE environment [Levytskyi and Kerckhoffs, 2000a]). Although simple, the data flow considered has a significant leverage in the real-world scenarios typical for web-based environments. Based on the definition of Data Flow Diagrams (DFD), we specify a DFD metamodel in the Entity-Relationships formalism with the meta-modelling tool ATOM<sup>3</sup> [de Lara and Vangheluwe, 2002a] and use it to generate a visual modelling tool tailored according to the proposed DFD metamodel. Finally, the paper illustrates how a DFD model created with this modelling tool is transformed into a textual code, a job description for the NCSE execution controller.

*keywords:* Data Flow Diagrams, Metamodel, Transformation, Code Generation, Web Environment

## 1. INTRODUCTION

The emergence of the world-wide web (WWW) and its popularity in the simulation community gave birth to the concept of *web-based simulation* [Fishwick, 1996], which now includes (among others) activities that deal with the use of the WWW as infrastructure to support distributed simulation execution and encompass research in tools, environments and frameworks that support the distributed, collaborative design and development of simulation models [Page, 1998].

Within this domain, several years ago we started a Collaborative Simulations project in which a generic web environment is developed to support simulation and modelling components in multidisciplinary collaborative projects [Levytskyi and Kerckhoffs, 2000a]. The environment's functionality is similar to that of the DLR-IMF Virtual Laboratory [DLR-IMF]. The practical application of our prototyped environment lies in the so-called NanoComp project, which investigates computing systems based on quantum devices; therefore the environment is named NanoComp Simulation Environment (NCSE).

NCSE is *based* on two major types of remote objects called *resources* [Levytskyi and Kerckhoffs, 2001]: conventional tools and models, which are maintained by the collaborative groups that own them. The environment *provides*: (i) an infrastructure that connects remote resources to their respective web-façades (proxy objects accessible from the web) via a distributed object middleware; (ii) centralised access control (via a controller) to remote resources; and (iii) on-line

services, such as registration, discovery and processing of resources (i.e. simulation of a registered model with an integrated simulation tool). These web-façades are containers for metadata that describe properties of the remote counterpart tools and models, thus enabling the above-mentioned services. Since 2002, NCSE includes meta-modelling capabilities with the assistance of ATOM<sup>3</sup> (A Tool for Multi-formalism and Meta-Modelling).

ATOM<sup>3</sup> is a visual tool for meta-modelling and model-transforming. Meta-modelling refers to modelling formalism concepts at a meta-level, and model-transforming refers to automatic converting, translating or modifying a model of a given formalism into another model of the same or different formalism [Vangheluwe et al, 2002]. The tool's meta-layer allows a high-level description of models, based on which ATOM<sup>3</sup> can automatically generate a tool tailored to the family of those models.

In NCSE, ATOM<sup>3</sup> is used as Meta-CASE Tool (and the topic of this paper is an example of such a use) to develop meta-models for various formalisms supported by the environment. Given these metamodels, ATOM<sup>3</sup> can be used as a conventional modelling tool for the supported formalisms. Finally, we employ the model-transforming capabilities (a) to generate job descriptions for the NCSE controller (which is discussed in this paper) and (b) given a formalism's metamodel, to synthesize code for the formalism's components of the NCSE environment.

In this paper we illustrate how meta-modelling is used to support designing and executing data flows in the NCSE environment. Although simple, the data flow considered has a significant leverage in the real-world scenarios typical for web-based environments. In section 2 we provide a definition of Data Flow Diagrams that will serve as specification for the DFD metamodel presented in section 3. Based on this metamodel, AToM<sup>3</sup> can generate a completely new DFD modelling tool. Section 4 describes how to construct a transformation that, given a DFD model, generates a respective textual code for an external solver: the NCSE controller. An example of model-transforming is given in section 5. We conclude the paper with final remarks.

## 2. DFD DEFINITION

Data Flow Diagrams (DFD) present the flow of data through a system [Gane and Sarson, 1979]. The focus is on how data is processed by a system in terms of inputs and outputs. The building constructs of Data Flow Diagrams are *Data Flow*, *Data Store*, *Process* and *External Entity*. Figure 1 shows their respective graphical notations as proposed in [Gane and Sarson, 1979].

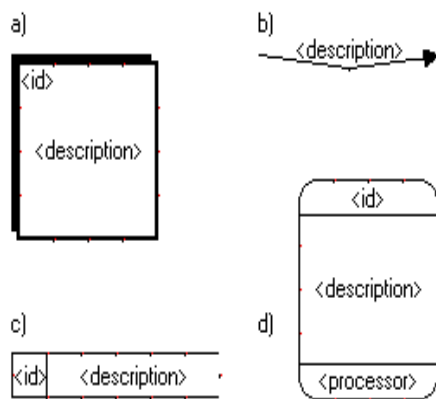


Figure 1: DFD notation.

*External entities* (Figure 1a) are data objects outside the context of the modeled system. External entities are sources and sinks (destinations) of the system's inputs and outputs. Each is given an alphabetic identifier.

*Data flow* (Figure 1b) is a pipeline through which packets of data of known composition flow. The arrowhead indicates the direction of the data flow. Each data flow must have a label describing the data.

*Data stores* (Figure 1c) are repositories of data inside a system. It is a data queue as opposed to

data flow. Each is identified by "D" and an arbitrary number.

*Process* (Figure 1d) transforms an incoming data flow into an outgoing data flow. Each is given a numerical identifier, physical reference (in the lower part of the process box) and is described with an imperative sentence containing an active verb e.g. "CONVERT data".

Additionally, there are general rules that a valid DFD diagram should comply with. Some of them are:

- Data flow connects other DFD constructs.
- No alteration of data can take place within a data flow.
- An external entity cannot be connected to another external entity.
- Data stores receive inputs and outputs only from processes.

There is much more to say about DFD (levels and types of Data Flow Diagrams, more rules and recommendations), but the definition provided here is sufficient for our purposes.

## 3. METAMODEL

A metamodel of a given formalism specifies the syntax aspect of the formalism by defining the language constructs and how they are built-up in terms of other constructs.

To construct a DFD metamodel we used Entity Relationships (ER) diagrams extended with constraints, a default meta-formalism of AToM<sup>3</sup>. Constraints provide a view on how a construct can be connected to another construct to be meaningful, and thus specify static semantics of the formalism. In this paper constraints are expressed in Object Constraint Language [OCL, 1997].

Important properties of each construct are *Cardinality*, *Attributes*, *Constraints*, and *Appearance*. Cardinality determines the possible number of incoming and outgoing connections of a construct. Additionally, we employ Constrains to control what constructs can connect to what constructs. We populate each construct's Attributes property (of collection type) with a minimum set of regular attributes that supports the semantics of the construct alone and in combination with other constructs. Finally, we define the Appearance property of each construct in accordance with the notation presented in Figure 1.

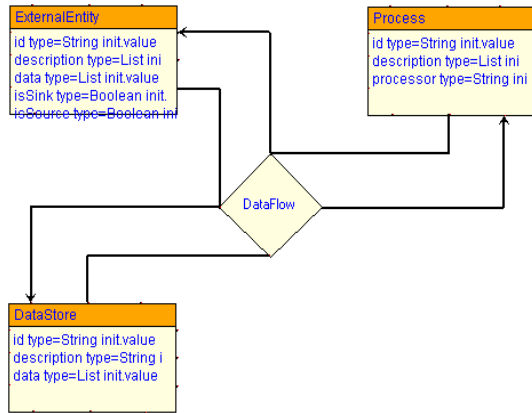


Figure 2: DFD Metamodel.

The metamodel in Figure 2 was constructed in AToM<sup>3</sup> according to the DFD definition provided above and shows how DFD constructs can be combined together. In the following, we describe each element of the metamodel in more details:

#### EXTERNALENTITY:

##### Cardinality:

self-to-dataflow: (1: 0..\*)  
dataflow-to-self: (1: 1)

##### Attributes:

id: string = 'a'  
description: string  
data: sequence  
isSource: boolean  
isSink: boolean

#### DATASTORE:

##### Cardinality:

self-to-dataflow: (1:1..\*)  
dataflow-to-self: (1:1)

##### Attributes:

id: string = 'D'  
description: string  
data: sequence

#### DATAFLOW:

##### Cardinality:

self-to-destination: (1:1)  
source-to-self: (1:1)

##### Attributes:

description: string  
data: sequence

##### Constraints:

```

DataFlow :: CONNECT(...)
post: self.Source.metaclass ->
  forAll(s |
    self.Destination.metaclass ->
      forAll(d |
        not s = d =
          'ExternalEntity'))
post: self.Source.metaclass ->
  forAll(s |
    self.Destination.metaclass ->
      forAll(d |
        Set{s,d} =
          Set{"Process", "DataStore"}))
  
```

#### PROCESS:

##### Cardinality:

self-to-dataflow: (1:1..\*)  
dataflow-to-self: (1:1)

##### Attributes:

id: string  
description: string  
processor: string

Along with the properties defined for each DFD construct, we also extend the global properties for the metamodel itself with attributes, such as *title*, *subject*, *description*, *author* and *version*. They can be used for basic documentation of models specified in this DFD formalism.

All global properties and regular attributes are to be filled-in by the end-user of the DFD modeling tool to be generated at the lower meta-level.

Finally, the flexibility and elegance of the meta-modeling concept allows us to easily adapt the DFD formalism as defined in section 2 to our needs. For example, to match the capabilities of the controller, we introduce two new general rules for our DFD:

- Do not allow branching.
- Do not allow loops on the same Process.

AToM<sup>3</sup> allows preventing branching by tuning the Cardinality property of elements. The loops are avoided by the following constraint:

```

DataFlow :: CONNECT(...)
post: self.Source -> forAll(s |
  self.Destination -> forAll(d |
    s <> d implies s.id <> d.id))
  
```

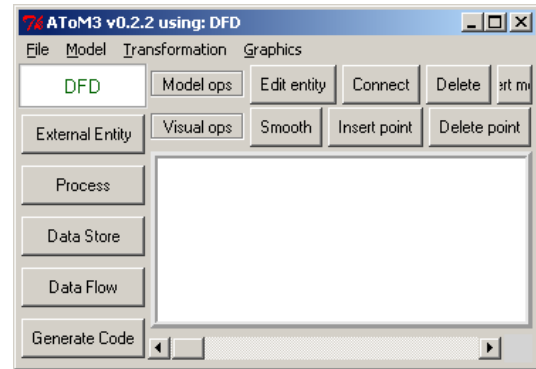


Figure 3: Generated DFD modeling tool.

Given our metamodel, we can now generate in AToM<sup>3</sup> a meta-specification, which, when loaded into the meta-level of AToM<sup>3</sup>, turns it into a new modeling environment for the modeled DFD formalism. A part of this meta-specification is a specification of the User Interface. This specification is a model in its own right and can be edited in AToM<sup>3</sup> at any time under a so-called "Buttons" formalism. By default, this specification creates a button for every construct of the formalism. In addition, we created one extra button, which on click applies the code generation transformation to the model on the tool's canvas. An instance of the generated DFD modeling tool is shown in Figure 3.

#### 4. CODE GENERATION TRANSFORMATION

Model transformation is related to dynamic semantics of a formalism, which defines the meaning of well-formed constructs. This meaning can be described in a number of ways, e.g.: formalism transformation, model optimization, code generation and simulator specification.

This section describes a code generation transformation that, given a DFD model, generates a corresponding textual job description for the NCSE controller. The controller is a custom built Process-Interaction (PI) solver based on the operational semantics of  $\pi$ Demos [Birtwistle and Tofts, 1994].

In ATOM<sup>3</sup> model transformations are specified through Graph Grammars, and consist of *Initial Action*, *Final Action* and *Transformation rules*. Each rule consists of *Left Hand Side* (LHS) and *Right Hand Side* (RHS) graphs, and *Condition*, *Action* and *Priority* properties.

The *Initial Action* of the transformation iterates through all the elements of the current model (objects on the tool's canvas) to augment them with temporary attributes to be used in the conditions specified below. Attribute *isVisited* helps to distinguish the elements that have been already processed from those that have not yet. Attribute *isCurrent* is used to mark a DataFlow that leads to the element whose code has to be generated next. It also creates the job data structure:

```
{'source': '', 'sink': '', 'body': [ ]}
```

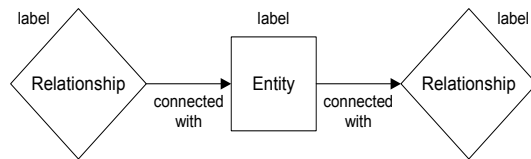


Figure 4: Subgraph match pattern.

We designed the rules to match a pattern shown in Figure 4, where the relationship element is a DataFlow, and the entity can be an instance of any other DFD component. Either the left or right relationship can be omitted. Present elements are labelled with consequent numbers. In the following we briefly describe each rule:

**RULEPROCESS** (priority 1) locates a Process and rewrites the model as shown in Figure 5. Its *action* generates code using proper controller commands (**get**, **hold**, **put**) to access, use and release the physical entity implementing the process, and marks element 1 as not current, element 2 as visited, and element 3 as current.

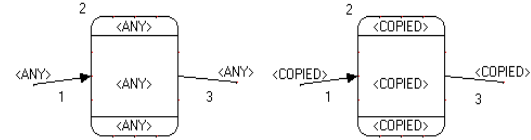


Figure 5: LHS and RHS subgraphs for processes.

**Action**  

```
pre: LHS.element1.isCurrent = 1
      and LHS.element2.isvisited = 0
post: RHS.element1.isCurrent = 0
      and RHS.element2.isvisited = 1
      and RHS.element3.isCurrent = 1
```

**RULESOURCEEXTERNAL** (priority 2) locates a source ExtEntity and rewrites the model as shown in Figure 6. Its *action* updates the 'source' field of the job description with the URL value of the data object of element 1 and marks element 1 as visited and element 2 as current.

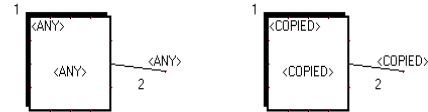


Figure 6: LHS and RHS subgraphs for source externals.

**Action**  

```
pre: LHS.element1.isSource = 1
      and LHS.element1.isvisited = 0
post: RHS.element1.isvisited = 1
      and RHS.element2.isCurrent = 1
```

**RULEDATASTORE** (priority 3) locates a DataStore and rewrites the model as shown in Figure 7. As semantics of this entity in the controller's context is currently not defined, the *action* only marks element 1 as not current, element 2 as visited, and element 3 as current.

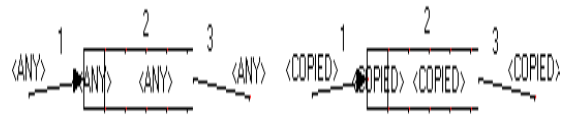


Figure 7: LHS and RHS subgraphs for data sources.

**Action**  

```
pre: LHS.element1.isCurrent = 1
      and LHS.element2.isvisited = 0
post: RHS.element1.isCurrent = 0
      and RHS.element2.isvisited = 1
      and RHS.element3.isCurrent = 1
```

**RULESINKEXTERNAL** (priority 4) locates a sink ExtEntity and rewrites the model as shown in Figure 8. Its *action* updates the 'sink' field of the job description with the URL value of the data object of element 2 and marks element 1 as not current and element 2 as visited.

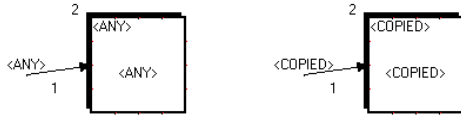


Figure 8: LHS and RHS subgraphs for sink externals.

**Action**  
**pre:** LHS.element1.isCurrent = 1  
 and LHS.element2.isSink = 1  
 and  
**post:** LHS.element2.isVisited = 0  
 RHS.element1.isCurrent = 0  
 and RHS.element2.isVisited = 1

The *Final Action* prints the job data structure into an output file. As the last step, it iterates through all the elements on the tool's canvas removing temporary attributes *isVisited* and *isCurrent*.

## 5. MODEL-TRANSFORMING

Figure 10 shows a DFD model created with the generated modeling tool.

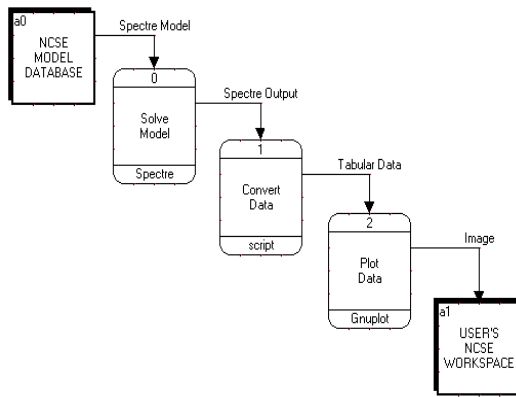


Figure 9: A model in the DFD formalism.

Source external entity **a0** contains a reference to a model registered in the NCSE model base. Process **0** refers to a simulation tool that can solve the model concerned. Process **1** is a script that converts the output of process **0** into the input for process **2**. Process **2** refers to a visualization application that produces diagrams from the input data. Finally, sink external entity **a1** refers to the modeler's workspace at NCSE.

Model-transforming in AToM<sup>3</sup> can be launched in a variety of ways, e.g. by clicking the button, which we created in the graphical user interface for the code generation transformation.

During execution of a model transformation, AToM<sup>3</sup>'s Graph Rewriting Processor (GRP) iterates through the list of rules sorted by their priority in an ascending order and tries to apply the current rule to the model. If the rule makes a match (LHS pattern is found and conditions are met), it is

executed and the GRP repeats trying each rule again from the beginning of the list. This continues until there are no rules anymore that can be applied, then GRP considers the model transformation as completed [de Lara and Vangheluwe, 2002].

The result of our model-transforming is a valid job description for the controller (see Figure 10).

```
#
# This code is automatically generated.
#

__version__ = 'Revision: 0.01 $'[11:-2]
__author__ = 'A. Levytsky'

# A job description for NCSE controller
job = {
  'body': ["getR ('spectre')",
    'hold ()',
    "putR ('spectre')",
    "getR ('script')",
    'hold ()',
    "putR ('script')",
    "getR ('Gnuplot')",
    'hold ()',
    "putR ('Gnuplot')",
    "close()"
  ],
  'source': 'scheme://host:port/sourcepath',
  'sink': 'scheme://host:port/sinkpath'
}
```

Figure 10: Generated textual code for execution.

At this point the synthesized code can be passed to the NCSE controller for execution. The controller will create a new job (and add it to the pool of already existing jobs) that will provide the data as input for process **0**, and so on until the output of process **2** is placed in the environment's cash and the output's URL is stored in the user's workspace. More details on the controller and job execution can be found in [Levytsky and Kerckhoffs, 2000b].

## 6. FINAL REMARKS

In this paper we demonstrate how the concept of meta-modelling could be used to easily extend an existing simulation environment with new functionality, namely dataflow modelling and execution. The meta-modelling tool AToM<sup>3</sup> plays an important role in this (even though currently AToM<sup>3</sup> can only be used locally and not from the web) and is primarily used as Meta-CASE Tool to develop meta-models for various concepts used in the environment, and as code generator. The most important is that meta-modelling and AToM<sup>3</sup> indeed enable us to adjust NCSE to different situations.

## ACKNOWLEDGEMENT

The research reported in this paper is done in the framework of the NanoComp project, sponsored by TU-Delft.

We would like to thank the Modelling, Simulation and Design Lab (MSDL) of the School of Computer Science of McGill University (Montreal, Canada), and especially Hans Vangheluwe and Juan de Lara, for providing and helping us with AToM<sup>3</sup>.

## REFERENCES

- Birtwistle G. and Tofts C. 1994, An operational semantics of process-oriented simulation languages: Part 1 pDemos. Trans. Soc. Comput. Simul., 10(4), Dec. 1994, pp. 299-333.
- de Lara J. and Vangheluwe H. 2002, "AToM3: A Tool for Multi-Formalism Modelling and Meta-Modelling". In: *European Conferences on Theory And Practice of Software Engineering ETAPS02, Fundamental Approaches to Software Engineering (FASE)*. Lecture Notes in Computer Science 2306, Springer-Verlag, pp. 174 - 188.
- DLR-IMF. Virtual Laboratory, a repository of online-executable scientific software: [http://vl.nz.dlr.de/VL/S\\_qb4Mm21B/portal/](http://vl.nz.dlr.de/VL/S_qb4Mm21B/portal/)
- Fishwick P.A. 1996, "Web-Based Simulation". In: *Proceedings of the 1996 Winter Simulation Conference*, pp. 772 - 779.
- Gane C. and Sarson T. 1979, *Structured Systems Analysis: Tools and Techniques*. Prentice-Hall, Englewood Cliffs, USA
- Levytskyy A. and Kerckhoffs E.J.H. 2000a, "Towards a Prototype Web-Based Collaborative Simulation Environment", SCS: paper of the 5th Euromedia Conference, May 2000, pp. 60 - 66.
- Levytskyy A. and Kerckhoffs E.J.H. 2000b, "A simulation-based controller for a distributed collaborative environment". In: *D.F. Moeller (ed.): Simulation in Industry, Proceedings of ESS2000 (12th European Simulation Symposium, Hamburg, Germany, September 28-30, 2000)*, pp. 88-95.
- Levytskyy A. and Kerckhoffs E.J.H. 2001, "Integration of Simulation Tools and Models in a Collaborative Environment". In: *Proceedings of 2001 European Simulation Interoperability Workshop* (London, UK, June), Simulation Interoperability Standards Organisation, pp. 407-415.
- OCL (1997) Object Constraint Language Specification, version 1.1, September 1.
- Page E. H. 1998, "The rise of Web-based simulation: implications for the high level architecture". In: *Proceedings of 1998 conference on Winter simulation* (Washington, D.C., United States), pp. 1663 - 1668.
- Vangheluwe H., de Lara J. and Mosterman P.J. 2002, "An introduction to multi-paradigm modelling and simulation". In: *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, Lisboa, Portugal, April 2002, pp. 9 - 20

## AUTHORS' BIOGRAPHIES



**Andriy Levytskyy** graduated from Chernivtsi State University, Ukraine and holds an MSc-degree in Computer Science. Currently, he is a PhD student at Delft University of Technology, Faculty "Information Technology and Systems", Department "Mediamatica", Group "Knowledge-based Systems".



**Eugene J.H. Kerckhoffs** holds an MSc-degree from Delft University of Technology (1970, Physical Engineering, thesis on analogue and hybrid computer simulation) and a PhD-degree from the University of Ghent (1986, Computer Science, thesis on parallel continuous simulation). Currently, he is an associate professor at Delft University of Technology (Faculty "Information Technology and Systems", Department "Mediamatica", Group "Knowledge-based Systems"). He was also chairholder of the SCS Chair in Simulation Sciences at the University of Ghent, Belgium.