MM-DSL, support for implementing modelling tools for manufacturing processes

Daniel-Cristian Crăciunean[1,*], and *Daniel* Volovici[2]

[1]Lucian Blaga University of Sibiu, Computer Science and Electrical Engineering Department, Emil Cioran 4, Sibiu, Romania
[2]Lucian Blaga University of Sibiu, Computer Science and Electrical Engineering Department, Emil Cioran 4, Sibiu, Romania

Abstract. Today's competitive conditions call for detailed comparative analyzes of manufacturing processes in order to get competitive products. This analysis involves the development of faithful and robust models for the supervision and management of all organizational and operational activities of companies. Efficient modelling involves the selection and use of appropriate tools for modelling, simulation and analysis of manufacturing processes. The diversity of manufacturing processes often makes it necessary to implement specific modelling tools. MM-DSL is a platform independent language for specifying and implementing specific modelling tools. The core objective of the MM-DSL language is the implementation of the modelling method concept. The paper presents the mechanisms underlying the MM-DSL language as well as its use for building the modelling tools specific to the manufacturing systems.

1 Introduction

Modelling manufacturing systems is one of the most important ways to increase economic efficiency and product quality. Manufacturing systems are hierarchical systems with action-based behavior appropriate to process theory methods.

Through the manufacturing process we understand a set of activities which transform a given set of resources (raw materials, semi-finished goods, energy, labor, equipment) into finished products by aggregating specific actions according to their manufacturing recipes. The activities are the manufacturing processes including materials handling and information processing that must occur to make products. The resources are the humans, machines, raw materials and so on, that are required to perform these activities.

In order to get the desired finished products, we need a collection of resources and an appropriate transformation process that is often complicated.

Resources are of several types, some are not consumed when used such as a software resource, others, such as raw materials, are consumed in the manufacturing process [13-14].

Manufacturing systems are discrete event systems (DES) [2 - 3]. The behavior of manufacturing systems is represented by manufacturing processes. The modelling of these

* Corresponding author: daniel.craciunean@gmail.com

processes is the work of specifying the processes of manufacture by mathematical structures equivalent in a modelling language. Verification is work to prove that the actual behavior of a system is equal to the model behavior.

To use mathematical reasoning in the study of process, it is necessary to describe processes as elements of a mathematical domain [2- 3, 12]. Otherwise the semantics of real process becomes equivalent to semantics of the mathematical elements.

A process specification is a description of the properties of this process as a mathematical object. To describe a process, we need a formalism in which it can be expressed simply and precisely, that is, by a suitable modelling language [4 - 5].

In general, visual process modelling languages (PN, EPC, BPMN, UML) define first the syntactic atomic entities of the processes and then the semantics of the process by the rules of interaction between the syntactic entities. A visual modelling language, therefore, is a collection of atomic entities, to which a set of interconnection rules are added to build processes [4 -5, 12, 14].

Most often, the existing languages for describing manufacturing processes are too abstract, they do not simply and precisely express specific concepts, or are hardly accessible to specialists who model manufacturing processes [9]. The ideal solution is, in this case, the construction of domain-specific modelling tools. But to develop a modelling environment, fast and at low cost, we need powerful metamodelling tools.

MM-DSL is an alternative, platform-independent, way to specify a modelling method [8-9]. The MM-DSL language must facilitate specifications according to the modelling method concept, which extends that of a modelling language. The MM-DSL language was specified at the University of Vienna [9, 10] and the static part of describing the meta-models was implemented. We have implemented the dynamic part of the language, i.e. the one that offers facilities for specifying computational algorithms.

Modelling method is more than just a language, it is a concept introduced by Karagiannis & Kühn, which abstracts the concept of modelling tool [9, 18, 19]. A modelling method consists of two components: (1) a modelling technique, and (2) mechanisms & algorithms working on the models described by a modelling language [7-9].

Mechanisms and algorithms provide generic functionalities that refer to universal modelling method constructions (simulation, visualization, transformation, evaluation, etc.). Mechanisms contribute to increasing modelling efficiency by re-using and standardizing software.

There is a general consensus that the most efficient reuse of code is at the level of components that can be developed separately and can operate in an appropriate environment independently of the other components. A mechanism includes all necessary information to be used by a user without being recompiled. A mechanism is an independent, self-contained, reusable software entity that implements a variety of interfaces for the relationship with the specified model. Thus, a mechanism is an executable code that can be coupled to the code of other components of a model.

This paper summarizes the implementation of the MM-DSL language and presents a practical example of implementing a modelling tool for manufacturing processes using this language.

2 Theoretical foundations and notes

Model Driven Architecture (MDA) is a Model Driven Development (MDD) architectural framework for developing software introduced by Object Management Group (OMG). MDA distinguishes three layers of models [11]: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM).

Multi-level modelling suggests a model organized as a hierarchy in which each level of the hierarchy has a certain degree of connection with the real modeled system. By moving from a given level of this hierarchy to the next level, the model is characterized by increasing the fidelity to the real modeled process. The fundamental concepts of multi-level modelling are of two kinds, linguistic and ontological. The linguistic metamodel is dealing with the languages hierarchy (language syntax and grammar), while the ontological stack appeals to structural hierarchies and the classification of a base domain.

An ontology is a hierarchical representation of the concepts in a particular field with the relationships between these concepts and properties of each concept describing a mechanism of type attribute value. The main components of an ontology are individuals, classes, attributes, relationships, restrictions, rules, axioms, events etc.

We call language[1] a construct of the form L=<M,F,N,P,S,f,g> where: M is a given set of objects called the language semantics; F is a given set of objects called the language syntax; f:F→M is the association of semantics funtion; N is a set of notations; g:M→N is the association of notation function; P is a relation on F called pragmatic and is defined as follows: if x,y∈F then (x,y)∈P if and only if f(x) = f(y) and S is a subrelation S⊆P on the set F.

In order to specify a language, it is necessary to specify the two defining elements of the language, namely: A method for specifying the class of all real M objects that the virtual machine can recognize and manipulate on a symbolic level must be found. This class of objects is called semantics of language. For a given semantics, a mechanism is required to generate the class of all forms or symbolic representations of the objects from M. These symbols make up the syntax of the language.

A special interest in formal language theory as well as in computer science presents those languages for which there is a finite definition mode, that is, a way of defining which can be fully specified by a finite set of words in any language. An important tool for specifying a language through a finite set of words is generative grammar.

A generative grammar [1] is a construction of the form $G=(V_N,V_T,S,P)$ in which V_N is the alphabet of non-terminal symbols, V_T is the alphabet of terminal symbols, $S\in V_N$ is the start symbol or axiom, and P is a finite set of pairs of words from $(V_N \cup V_T)^*$, $P=\{(u_i , v_i) | 1\leq i \leq m\}$, so that any word u_i contains at least a non-terminal symbol. Pairs (u_i , v_i) are called derivation rules or production rules or simpler productions and are denoted $u_i \rightarrow v_i$.

3 Support tools for implementing the MM-DSL language

The specification of a modelling method must ultimately be executed in a metamodelling environment that can produce a useful modelling tool, therefore it must be translated in accordance with the functional and non-functional requirements of the metamodel.

In the process of developing a DSL, there is a need to translate the virtual machine language represented by the DSL we want to construct in the language of another virtual machine, represented by the language that can be executed by it. In our case, the DSL is the MM-DSL language and the second language is the ADO-Script language.

The translation appears as a communication between the two VM1 and VM2 virtual machines specified by the two languages: MM-DSL = (M_1, F_1, f_1, g_1) and respectively ADOScript = (M_2, F_2, f_2, g_2).

In order to be able to communicate between the two VM_1 and VM_2 machines specified by their MM-DSL and ADOScript languages, respectively, $UN(VM_1)=UN(VM_2)$ where UN(VM) is the natural universe of VM, i.e. objects that can be interpreted semantically by an VM machine.

Let us consider the case when VM_1 and VM_2 machines understand the same class of objects of their natural universe, i.e. they can be specified by their natural languages MM-DSL=(M, F_1, f_1, g_1) and respectively ADOScript=(M, F_2, f_2, g_2).

By communicating from VM_1 to VM_2 we understand the process of transmitting programs from VM_1 to VM_2 and executing them by VM_2. For such a program to be received and executed by VM_2 it must appear in its natural language, i.e. ADOScript.

Therefore, in order to be able to communicate from VM_1 to VM_2, the following two functions are required:

$T_1:F_1 \rightarrow F_2$ that converts syntactic representations from F_1 to F_2. This function is called a translator.

$E_2:F_2 \rightarrow M$ that associates the semantics of the syntax F_2. If the F_2 message is syntactically correct, in our case, it already has the semantics associated with the ADOScript language, or f_2: $F_2 \rightarrow M$.

Within programming languages, the word semantic is a technical term that refers to the significance or meaning of a program. In mathematical terms, semantic becomes in this case a function that receives as input a syntactically correct program and outputs a description of the function calculated by the program. On the other hand, the syntax is nothing but a mechanism of symbolic representation (codification) of semantics.

A rule must be specified by which each real object is associated with a form of representation. Operational modelling, is defined in terms of a representation-implementation pair, and thus provides a close link to reality.

Running an imperative program involves executing a series of explicit commands to change the state. The behavior of an imperative program is described by rules that specify how its expressions are evaluated and its commands executed. The rules provide an operational semantics of language in the sense that they provide a language implementation.

Regarding the implementation of the MM-DSL language, the problem of studying the programs from a semantic point of view appears to be the need to meet some very important requirements:

- The program object generated by the translator has to carry with it all the semantic load of the source program, that is, to have the same meaning as the source program.
- The algorithm symbolically represented in the MM-DSL language must correspond to the algorithm that was intended to be specified.
- The function, symbolically correct represented from the syntactically point of view, is a computable function.

There are two types of mechanisms used to model syntax: the mathematical theory of formal languages and the free algebra generated by a finite symbolism. In practice for specifying production rules, the BNF or EBNF language is used which is more accessible, but equivalent to the production rules of a context free grammar. The Xtext framework we use to develop the MM-DSL language accepts EBNF type specifications.

We have seen that if we have the two VM_1 and VM_2 virtual machines that understand the same class of objects M and are specified by their natural languages MM-DSL and ADOScript, then a $T_1:F_1 \rightarrow F_2$ translator is needed to transform syntactic representations from F_1 into F_2.

If the two syntactic representations F_1 and F_2 are given by two generative grammars G_1 and G_2: $G_1=(V_{N1},T_{T1},S_1,P_1)$ and $G_2=(V_{N2},T_{T2},S_2,P_2)$, simplifying things a translator could become partial function $T:\mathcal{P}(P_1) \rightarrow \mathcal{P}(P_2)$, i.e. a partial function that correlates sets of generating rules from the two grammars. To specify this translator function T, we use the Xtend language that provides useful features in the DSL development process.

But we will have to convert the syntax of a MM-DSL language program into the syntax of the same program in ADOScript, and we will first have to find the rules of the G_1 source

grammar that generates our MM-DSL program. This process of calculating the ordered set of rules for generating a program in a given grammar inevitably accompanied by checking the syntactic correctness of the program is called syntactic analysis. Syntactic analysis is performed by two key components of a translator, lexical analyzer and syntactic analyzer.

The lexical analyzer is a phase of the translator who admits the source program at the input and produces a sequence of classes of indivisible entities called lexical atoms. A lexical atom is a terminal in the generating grammar of the syntactic structure of the program and represents a class of lexical elements with precise forming rules.

The syntax of lexical atoms is represented by the use of regular languages or regular expressions. As we know in the case of regular languages, the formal model used to recognize the words of a language is the finite state machine.

Therefore, the lexical analyzer model is the finite state machine improved at the time of designing with several other features to better respond to the needs of lexical analysis.

The Xtext framework [15, 16] includes all these features by using lexical analysis algorithms and LL-type syntactic analysis (ANTLR). These tools (Xtext, Xtend) that we briefly present are integrated into the Eclipse Modelling Framework (EMF) platform.

Eclipse [6] is a software project that provides an integrated software platform, and includes a generic framework for integrating these tools as well as a Java development environment.

The Eclipse Modelling Framework (EMF) [6] is a framework that, based on Eclipse's facilities, supports model building. A model is an abstract formalization of elements from the real world that captures the important aspects from an area of interest.

Modelling in EMF is multi-level, i.e. each model is described on the basis of a meta-model, which contains a set of rules embodied in a model building scheme. The metamodel describes what elements may exist in the model, and how they interrelate. The metamodels are also specified by a meta-metamodel that contains the construction rules for metamodels. In our case, the meta-metamodel is EMF: Ecore.

MOF (Meta Object Facility) is a complex standard specification for meta-metamodels defined by the Object Management Group consortium (OMG) and EMF:Ecore is a MOF implementation that supports other tools to build specific models and can be used without Eclipse[11].

3.1 Xtext, support for syntactic analysis

Xtext is a framework for development of programming languages and domain-specific languages (DSL). Also with Xtext we can create an Eclipse-based development environment. Xtext provides us with modern APIs to specify the different aspects of the programming language. Based on this information, Xtext provides us with a full implementation of the specified language running on JVM. Our language compiler provided by Xtext is independent of Eclipse or OSGi and can be used in any Java environment [15, 16].

It includes actions such as abstract syntax tree (AST) generation, serialization and code formatting, as well as components such as syntax analyzer, lexical analyzer, and code generator or interpreter. These rolling components integrate and rely on the Eclipse Modelling Framework (EMF), which effectively enables us to use Xtext with other EMF frameworks, such as the GMF Graphical Modelling Framework.

In addition to this Xtext-based runtime architecture, we will get an Eclipse-IDE specifically adapted for the specified language. Although the infrastructure itself runs on JVM, we can use Xtext to compile languages on any existing platform. Xtext is a comprehensive framework that helps implement your own DSL language with the appropriate IDE support.

The source language, which we want to translate, will need to be specified through a formal grammar. Xtext is based on Extended Backus-Naur Form (EBNF), which is an extension of BNF (Backus-Naur Form). The MM-DSL language implemented by us is described in the EBNF syntax.

Example 1. The algorithm concept has the following root-level specification:

```
Algorithm:
    'algorithm' name=ValidID '{' stmnt += Statement* '}'
;

Statement:
    selection=SelectionStatement |
    loop=LoopStatement |
    variable=Variable |
    algorithmoperation = AlgorithmOperation |
    insertembedcode = InsertEmbedCode
;

SelectionStatement:
    ('if' '(' ifcondition=Expr ')' '{' ifstatements+=StatementBreakBlock* '}')
    (elseifblock+=ElseIfBlock)* ('else' '{' elsestatements+=StatementBreakBlock* '}')?
;

ElseIfBlock:
    ('elseif'(' elseifcondition=Expr ')' '{' elseifstatements+=StatementBreakBlock* '}')
;
```

3.2 Xtend, support for semantics implementation

The Xtend programming language [17] is tightly integrated with Java. Xtend has a shorter syntax than Java and provides additional features that are useful in development of a DSL. All aspects of implementing a DSL with Xtext can be done in Xtend more easily than in Java because it is easier to use and allows for code to be shorter and easier to read. Xtend is fully interoperable with Java, and therefore we can use all Java libraries; Also, all Eclipse JDT facilities work perfectly with Xtend.

Xtend is a programming language with a static translation to Java source code. Syntactically and semantically, Xtend has its roots in Java programming language, and improves some aspects of java: extension methods, lambda expressions, powerful switch expressions, template expressions and no statements, everything is an expression.

The Xtext framework together with the Xtend language along with the Eclipse Modelling Framework (EMF) form an appropriate tool for implementing a language. It is worth noting that Xtend is implemented in Xtext and is therefore a relevant proof of how language can be implemented with the Xtext framework.

Example 2. The algorithm concept has the following partial implementation:

```
// generates algorithms
def GenerateAlgorithm(Algorithm alg) '''
    # «alg.name»
    PROCEDURE global «alg.name.toUpperCase()»
    {
        «GenerateStatments(alg.stmnt)»
```

```
        }
'''
def GenerateStatments(EList<Statement> statements) '''
        «FOR Statement stmnt: statements»
                «IF stmnt.algorithmoperation != null»
                        «stmnt.algorithmoperation.GenerateAlgorithmOperation»
                «ELSEIF stmnt.insertembedcode != null»
                        «stmnt.insertembedcode.codesnippetname.embeddedcode»
                «ELSEIF stmnt.selection != null»
                        «stmnt.selection.GenerateSelectionStatement»
                «ELSEIF stmnt.loop != null»
                        «stmnt.loop.GenerateLoopStatement»
                «ELSEIF stmnt.variable != null»
                        «stmnt.variable.GenerateVariableStatement»
                «ENDIF»
        «ENDFOR»
'''
// generates selection statement (if-elseif-else)
def GenerateSelectionStatement(SelectionStatement select) '''
IF («getNode(select.ifcondition).tokenText.replaceAll("==","=")») {
        «FOR sbb: select.ifstatements»
                «IF sbb.statements.size != 0 »
                        «sbb.statements.GenerateStatments»
                «ELSEIF sbb.breakcontinue != null»
                        «IF sbb.breakcontinue.break != null»
                                BREAK
                        «ELSEIF sbb.breakcontinue.continue != null»
                                NEXT
                        «ENDIF»
                «ENDIF»
        «ENDFOR»
}
«FOR ifc : select.elseifblock»
        ELSIF («getNode(ifc.elseifcondition).tokenText.replaceAll("==","=")») {
                «FOR sbb: ifc.elseifstatements»
                        «IF sbb.statements.size != 0 »
                                «sbb.statements.GenerateStatments»
                        «ELSEIF sbb.breakcontinue != null»
                                «IF sbb.breakcontinue.break != null»
                                        BREAK
                                «ELSEIF sbb.breakcontinue.continue != null»
                                        NEXT
                                «ENDIF»
                        «ENDIF»
                «ENDFOR»
        }
«ENDFOR»
«IF select.elsestatements.size != 0»
        ELSE {
                «FOR sbb: select.elsestatements»
                        «IF sbb.statements.size != 0 »
```

```
                «sbb.statements.GenerateStatments»
            «ELSEIF sbb.breakcontinue != null»
                «IF sbb.breakcontinue.break != null»
                    BREAK
                «ELSEIF sbb.breakcontinue.continue != null»
                    NEXT
                «ENDIF»
            «ENDIF»
        «ENDFOR»
    }
«ENDIF»
'
```

4 Defining a manufacturing process

We will consider that a manufacturing cell is made up of many workstations, in which there are many operations executing that we call actions. At each moment the manufacturing cell will be in a certain possible state. Realization of events will trigger the execution of some actions, execution that will produce cell state change.

Each workstation therefore contains an input buffer, an output buffer, and a set of actions that are executed on the inputs to convert them into outputs. The triggering of an action will occur if the input and output buffers of the station satisfy the requirements of that action.

A workstation model of a manufacturing cell contains a set of actions defined in the following way:

A transformation action model is a construct of the form a = (PID, Σ, Δ, Ω_I, Ω_O, ℜ) where:

i) PID is a finit set of process instances,
ii) Σ is an alphabet, its elements, we call objects,
iii) Δ is a set of events,
iv) Ω_I is a multiset over the cartestian product PID × Σ × Δ, called the input multiset. Ω_I represents the required configuration in the input buffer so that the action can be executed,
v) Ω_O is a multiset over the cartestian product PID × Σ × Δ, called the output multiset. Ω_O is the outcome resulting from the execution of the action. For the action to be executed, the output buffer capacity must allow it to be stored,
vi) ℜ is a set of data and associated metadata such as duration of action, the minimum allowed stock, costs, technical data about the action, software components, etc.

A transfer action model is a construct of the form a=(PID, Σ, Δ, Ω_I, Ω_O, σ, θ, ℜ) where:

i) PID, Σ, Δ, Ω_I and ℜ are defined as in the foregoing definition,
ii) $\Omega_O = \{(y, r, e)^c \mid (y, r, e)^c \in \Omega_I\}$,
iii) σ is the label of the source workstation,
iv) θ is the label of the target workstation

A workstation model is a construct of the form B=(PID, Σ, E, R_I, R_O, Q, ρ) where:

i) PID is a finit set of process instances,
ii) Σ is an alphabet, its elements, we call objects,
iii) E is a set of events,
iv) R_I is a multiset over the Cartesian product PID × Σ × E which initializes when an instance of the workstation model is created. R_I represents the workstation

configuration that will be initialized on the workstation instantiation. The configuration will be changed by the workstation actions which will consume resources from R_I and will produce as output other resources that will be added to R_O. Also the configuration R_I will be modified by all the transfer actions of other blocks which have as destination the current workstation through adding their output to R_I,

v) R_O is a multiset over the Cartesian product PID × Σ × E called the output multiset of workstation B. R_O represents the set of output resources after the execution of the workstation actions, i.e. the resources transferred outside the workstation by the transfer actions of the workstation,

vi) Q is a set of transformation actions,

vii) ρ is a partial order relation over the set Q of actions called priority relation.

A manufacturing process model is a construct of the form P=(PID, Σ, E, B, R, R_O, ψ, Γ) where:

i) PID is a finite set of process instances and is the set of all possible instances of the process,

ii) Σ is an alphabet, its elements, we call objects,

viii) E is a set of events,

iii) B is a set of workstations B=$\{B_1,B_2,...B_m\}$ marked by 1,2,..m,

iv) R is a set of multisets over the set PID ×Σ × E, associated with the process workstations $R=\{R_1,R_2,...R_m\}$, where R_i is the configuration of the B_i workstation. R is the process configuration and will get an initial value on the process instantiation. The initial configuration will be modified during the process execution accordingly with the evolution of the workstations.

v) R_O is a multiset over the set PID ×Σ × E, whose elements are called output resources and consists of all resources leaving the process workstations during the execution.

vi) ψ is the association function of the initial resources ψ: R→ B. This function distributes to each workstation from B a multiset from R at process initialization, i.e. it distributes the initial configuration of the process to the initial configurations of the blocks.

vii) Γ is a set of transfer actions. We look at set Γ as a function Γ:B→2^B defined as: if the block A∈B then

ΓA={X| there exists a transfer action a=(PID,Σ,Δ,Ω_I,Ω_O,A,X,ℜ)∈Γ}.

If B∈ΓA we say that (A,B)∈Γ, and the pair (A,B)∈Γ we call edge of process P. Therefore Γ={(A,B)|B∈ΓA} and ΓA={B|(A,B)∈Γ}.

viii) The pair (B, Γ) is a connected graph.

We observe that both transformation actions and transfer actions are asynchronous and triggered by events. This flexibility makes them compatible with pull control, push control and Kanban strategies, specific to manufacturing processes. Thus, actions can be implemented as reactive microservices, to the content of input and output buffers, which allows many types of combinations of these strategies in hybrid strategies.

Example 3. Manufacturing_Systems. We will build a modelling tool for manufacturing processes, for a factory, where several products are assembled from components simultaneously [5].

Actions are operations of assembling two or more components to get another component. The process continues until the finished products are obtained. The workstations are of two types, assembling stations and test stations.

Each workstation w_i contains a set of actions, an input buffer B_i and an output buffer B_o, which have limited capacities. Execution of an action is triggered automatically if it has the

necessary components in the input buffer and enough storage space for the resulting component in the output buffer. Operations have a certain duration, and concurrence between actions is resolved on the basis of a priority relation. We will use the graphical notation in Fig. 1.a. for such a station.

A test station X_t contains a set of actions, an input buffer B_i and an output buffer B_o, which have limited capacities. Each test action has a duration in time and automatically triggers if it has the necessary components in the input buffer and sufficient storage space for the component in the output buffer. Competition between actions is resolved on the basis of a priority relation. We will use the graphical notation in Fig. 1.b for such a station.

The processes will also contain a set of initial buffers X_i, a set of final buffers X_o and a set of defective buffers X_d for collecting defective components,. Each initial buffer X_i has a certain capacity and is continuously fed from outside the process. The initial buffers will be symbolized as in Fig. 1.c. The final buffers X_o also have limited capacities and store more types of finished products and are emptied out from outside of the process. The final buffers will be symbolized as in Fig. 1.d. Also, the buffers for collecting defective components X_d have limited capacities and store many types of defective components and are periodically emptied from outside of the process. The defective components buffers will be symbolized as in Fig. 1.e.

The language also contains transfer actions that will be marked with arrows as shown in Fig. 1.f. and represents the transport operations of components from one station to the next. Conveyors also have limited capacities and can carry several types of components in specified quantities. A transport action is of a certain duration and automatically triggers if it has sufficient components in the output buffer of the source workstation and also has enough space in the input buffer of the target workstation.

Fig. 1.a.

Fig. 1.b.

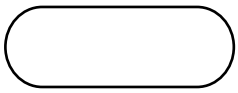

Fig. 1.c.

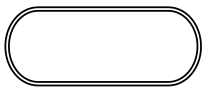

Fig. 1.d.

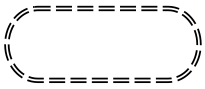

Fig. 1.e.

Fig. 1.f.

Fig. 1. Atomic entities of the modelling language

A process specified in this language can be seen in Fig. 2. The model was implemented in MM-DSL, translated into ADOxx and executed.

In the MM-DSL language, workstations, assembling stations and test stations, as well as initial buffers and final buffers are specified by "class" instructions and the corresponding graphics symbols are specified by "classgraph" instructions. The transfer actions are specified by "relation" instructions and the corresponding graphic symbols with "relationgraph" instructions.

4 Conclusions

The specificity of manufacturing processes implies various modelling concepts that most of the time cannot be specified with existing tools, or can be modeled with too high costs. A suitable modelling tool, with a concept specific to the modelling domain, allows the specification of manufacturing processes quickly and efficiently. MM-DSL provides an efficient solution for specifying and generating such a tool. In this context, much of the process semantics can be implemented at the metamodel level.

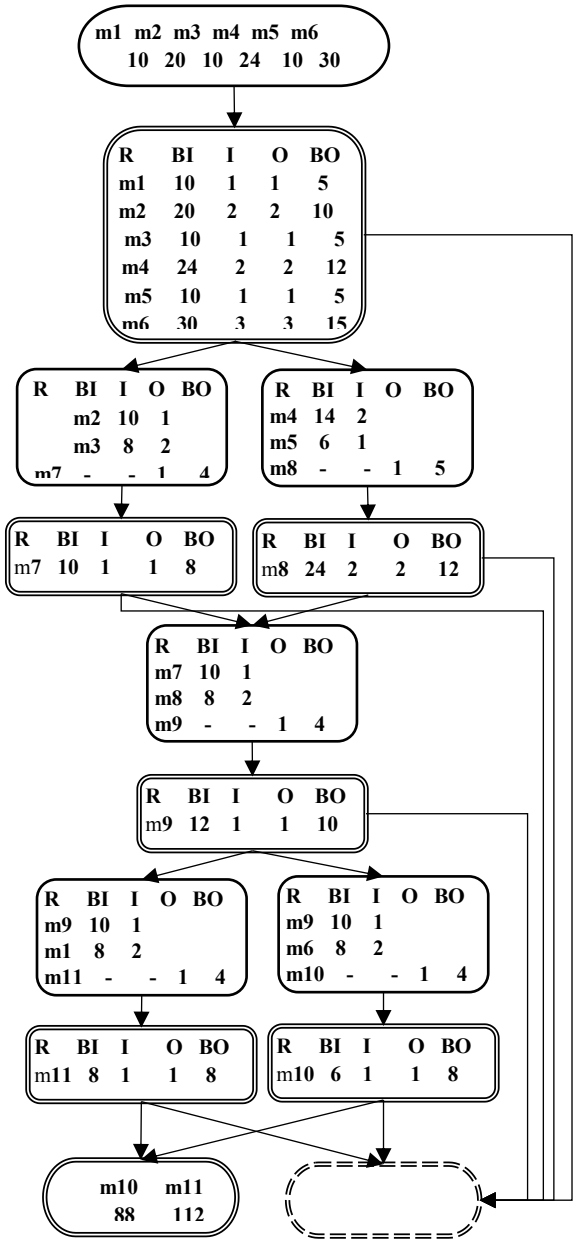

Fig. 2. Example of a MLMS model

In many ways, it is important to represent, for the most part, the semantics of manufacturing processes in the metamodel. If we consider the specification of the metamodel in MM-DSL, then constructions in the metamodel can be implemented as a package of mechanisms and algorithms that will work coherently in all the specified models. This mechanism therefore supports the concept of component-oriented programming and software reuse.

Acknowledgements

This work was partially developed under the ERASMUS+ KA2 project "THE FOF-DESIGNER: DIGITAL DESIGN SKILLS FOR FACTORIES OF THE FUTURE", financing contract no. 2018-2553 / 001-001, project number 601089-EPP-1-2018-1-RO-EPPKA2-KA, web: http://www.digifof.eu.

References

1. A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd edn. Pearson Education, Inc, India (2006)
2. B.P. Zeigler, A. Muzy, E. Kofman, *Theory of Modelling and Simulation Discrete Event and Iterative System Computational Foundations*, Academic Press (2019)
3. C. Gomes, C. Thule, D. Broman, P.G. Larsen, H. Vangheluwe - *Co-simulation: State of the art*, - ACM Computing Surveys, Vol. 1, No. 1, Article 1. Publication date: January (2016).
4. D.C. Crăciunean, D. Karagiannis, *Categorical Modelling Method of Intelligent WorkFlow*. In: Groza A., Prasath R. (eds) Mining Intelligence and Knowledge Exploration. MIKE Lecture Notes in Computer Science, vol 11308. Springer, Cham (2018).
5. D.C. Crăciunean, *Categorical Grammars for Processes Modelling*, International Journal of Advanced Computer Science and Applications(IJACSA), 10(1), (2019)
6. D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, *EMF: Eclipse Modelling Framework*, Addison-Wesley, (2009).
7. D. Karagiannis, R.A. Buchmann, *A Proposal for Deploying Hybrid Knowledge Bases: the ADOxx-to-GraphDB Interoperability Case*, In: System Sciences (HICSS), 51st Hawaii International Conference, pp. 4055-4064 (2018)
8. D. Karagiannis, N. Visic, *Next Generation of Modelling Platforms*, Perspectives in Business Informatics Research 10th International Conference, BIR 2011 Riga, Latvia, October 6-8, (2011)
9. D. Karagiannis, H.C. Mayr, J. Mylopoulos, *Domain-Specific Conceptual Modelling Concepts, Methods and Tools*. Springer International Publishing Switzerland (2016)
10. N. Višić, *Language-Oriented Modelling Method Engineering*, Doctoral Thesis , University of Vienna, Scientific Advisor: o. Univ.-Prof. Prof.h.c. Dr. Dimitris Karagiannis (2016)
11. OMG. Object Management Group, MDA Guide, Version 1.0.1, 12. June 2003. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf, access 30 November (2004).
12. R. Milner, *The Space and Motion of Communicating Agents*, Cambridge University Press, (2009)
13. M. Weske, *Business Process Management - Concepts, Languages, Architectures*, 2nd Edition., pp. I-XV, 1-403, Springer (2012)

14. W.M.P. van der Aalst, *Process Mining Discovery, Conformance and Enhancement of Business Processes*, Springer-Verlag Berlin Heidelberg (2011)
15. Xtext Documentation, https://eclipse.org/Xtext/documentation/
16. Xtext – Community [Online], Available: http://www.eclipse.org/Xtext/community.html, [Accessed: 27-Sep-2014]
17. Xtend Documentation, https://eclipse.org/xtend/documentation/
18. D. Bork, R.A. Buchman, D. Karagiannis, M. Lee, E.T. Miron, *An Open Platform for Modelling Method Conceptualization: The OMiLAB Digital Ecosystem*, Communications of the Association for Information Systems, forthcoming, http://eprints.cs.univie.ac.at/5462/1/CAIS-OMiLAB-final-withFront.pdf (2019)
19. D. Bork, H.G. Fill, *Formal Aspects of Enterprise Modelling Methods: A Comparison Framework*, In: System Sciences (HICSS), 47th Hawaii International Conference, pp. 3400-3409 (2014)