Forward engineering and early model validation with Smalltalk

A. Plantec F. Singhoff V. Gaudel V. Ribaud

LISyC, Université de Bretagne Occidentale, UEB, Brest, France

Abstract

Since more than two decades, a lot of work has been achieved around *Model Driven Engineering* (MDE). One of the main goals of MDE is to improve the software development process productivity thanks to a generative approach.

Many MDE environments are available today. These environments allow a designer to specify object oriented meta-models. Meta-models structure can be specified through the definition of statically typed entities and entities relationships. Even if some of these environments also enable specifying meta-model invariants and behavior, two important issues remain unsolved:
– regarding meta-model execution, only limited capabilities are exhibited and the development process still relies on code generation into a mainstream language for validation;
– because of the generative approach, no causal connection is possible between a meta-model and its conforming models.
To deal with these issues, we decided to use Smalltalk as a MDE environment mainly because it provides us with a way of implementing meta-model execution. A Smalltalk system can be used as a modeling system providing out-of-the-box an infrastructure for meta-modeling, browsing, and testing. It also provides a rich system which enables implementing realistic prototypes. Moreover Smalltalk implements a Meta Object Protocol which maintains a causally connected modeling architecture.

Smalltalk is known as very powerful for the agile implementation of systems, for continuous testing and finally for early validation through prototyping. Unfortunately, Smalltalk is not a mainstream language and can be rarely used as the final platform. This article shows that it is possible to successfully use Smalltalk as a MDE environment even if the target system is not implemented in Smalltalk. Indeed, the software development process can be adapted to benefit from early validation and to be able to automatically produce or update a target system. For validation purpose, we have developed the Platypus tool, which enables static typing in Smalltalk and makes this MDE approach possible.

Key words: Model Driven Engineering, Smalltalk, Early validation, Prototyping, Code generating, Interoperability

1. Introduction

Since more than two decades, a lot of work has been achieved around *Model Driven Engineering* [1] (MDE). One of the main goals of MDE is to improve the software development process productivity. Despite a lot of work and a huge number of publications, it remains very difficult to conclude on the concrete benefits of the MDE [2, 3].

Models quality and accuracy are very important issues. Because, predominantly, the cost of the software implementation comes from validation, the software development process must be adapted to enable early validating of models. In [4], Osterweil considers that verification and validation can consume from 50% to 60% of the software production cost and naturally, these values can increase a lot for critical systems. A part of this cost is due to the inaccuracy of test and validation infrastructures [5, 2]. In [6], the authors argue that 60% of development efforts are related to models design and that an half of these efforts are due to models simulation implementation. Thus, improving simulation implementation as well as model validation remain vital stakes.

MDE development process is technically based on a generative approach. The executable system is the final result of the process. From the high level models to the final code, the production chain can be time and resource consuming. This development process does not facilitate agile implementation [7], continuous testing and early validation through simulation and prototyping.

The lack of velocity of the MDE software process is pointed-out by an increasing number of authors. They exhibit the benefit of executable meta-models [8, 9, 10, 11, 12, 13, 14] to fulfil the software developers need for an agile

Email addresses: `alain.plantec@univ-brest.fr` (A. Plantec), `frank.singhoff@univ-brest.fr` (F. Singhoff), `vincent.gaudel@univ-brest.fr` (V. Gaudel), `vincent.ribaud@univ-brest.fr` (V. Ribaud).

development process.

Many MDE environments are available today, but they do not offer very early validation because they are mainly based on a generative approach. Examples of such environments are EMF/Ecore [15], MetaGME [16], AMMA/KM3 [17] and Kermeta [9]. All these environments are based on an object oriented approach for meta-models specification and rely on a standard for the specification of meta-models. The MOF [18] and ECORE [15] are the most used standards. This kind of language mainly allows a designer to specify the structure of meta-models through the definition of statically typed entities and entities relationships. As an example, with the Kermeta environment [9] it is possible to specify entities behavior with the help of *aspects*. These environments are most often integrated within Eclipse [19].

These specifications can be enriched with the definition of models and entities invariants and with the expression of entities behavior.

Two important issues remain unsolved in these environments :

– regarding meta-model execution, they exhibit only limited capabilities and the development process still relies on code generation in a mainstream language for validation; for instance, assuming we expect to design a mail client specific system language, it will be very difficult to prototype and test it with these environments because its validation requires a network protocol stack;
– there is no causal connection [20] between a meta-model and its conforming models : changes in a meta-model do not automatically cause the adaptation of conforming models; this implies a long delay between a change and a test [12].

To deal with these issues, we decided to use Smalltalk as a MDE environment mainly because it provides us with a way of implementing meta-model execution. A Smalltalk system can be used as a modeling system providing out-of-the-box an infrastructure for meta-modeling, browsing, and testing them. It also provides a rich system which allows a designer to implement realistic prototypes. Moreover Smalltalk implements a Meta Object Protocol [21] which maintains a causally connected modeling architecture.

This article presents our iterative software design and coding process together with our tool Platypus allowing early validation of meta-models within an MDE environment implemented with Pharo [22].

The particularity of our process is to consider the MDE environment and the target system as tightly interconnected. Models built within the target system are used to further validate the meta-models within the MDE environment and with realistic conforming models.

While a target system is automatically generated from the MDE environment, meta-models must be statically typed to be able to build code generators with Smalltalk. Because Smalltalk is a dynamically typed language, a mean to statically specify types of meta-models elements must be integrated. We also present the modeling tool Platypus[23, 24] that we use to statically type our meta-models.

The paper is structured as follows. A glossary is given in Section 2, then our approach is presented in Section 3. The approach is illustrated with an end-to-end simple example in Section 4. The implementation of the core component, Platypus, is explained in Section 5. We discuss related work in Section 6 and we conclude in Section 7.

2. Glossary

In this Section we give some explanations of terms that we use in the paper. The following definitions must be understood in the context of Model Driven Engineering.

2.1. *Model*

Despite there is no clear and universal definition of what a model is, a model can be considered as a simplification of a system built with an intended goal in mind [25]. A model needs to meet three criteria [26]: (1) a model is a mapping based on a subject :the system that the model simplifies; (2) a model is a reduction or a simplification of a system because some aspects of the system are omitted; (3) a model must be usable in some context in place of the system. The software engineering lifecycle is made of consecutive steps, each step aims at producing one or several models. The goal of a model at a step N is to describe what we expect to produce at step $N + 1$. For example, an early step of the lifecycle is the requirement analysis step which aims at producing the requirement specification model. This model describes what we expect to produce at the Architecture specification step (see [26] to get more details about what is a model).

2.2. *Meta-model*

A meta-model is itself a model which describes the language used to specify a model. A meta-model specifies constructs and rules within a domain of interest. A well formed model is said as *conforming-to* its meta-model. It means that the model is well specified according to the authorized constructs and rules of its meta-model.

A meta-model is also specified using a particular language which is itself specified with a model. The model that specifies the meta-modeling language is called a meta-meta-model.

2.3. *The four level architecture*

A model and its meta-model are specified at two levels of a conceptual architecture.

This architecture is often represented as a four level model architecture [25, 27]. In this architecture, the system is at the M0 level. The model which is describing the system is at the M1 level. The M2 and the M3 levels hold respectively the meta-model and the meta-meta-model. Generally, the meta-meta-model is auto-described.

2.4. *Model transformation*

A model transformation is an operation that can be applied to a model to produce another model. A model transformation is also a model which is conforming to a model transformation meta-model. In other word, a model transformation is a program written in a model transformation language. Two kinds of model transformation can be programmed. (1) A model to model transformation is intended to transform a source model conforming to a source meta-model in a target model conforming to a target meta-model. (2) A model to text transformation is intented to transform a source model in a textual representation. Model to text transformations are usually implemented with code generators.

2.5. *MDE workbench*

A MDE workbench is a tool based on the implementation of a modeling language. It allows a designer to specify models within the workbench's modeling language. Technically, a workbench is based on the implementation of the meta-model that is specifying the workbench's modeling language. Usually, a workbench enables model to text transforming using a built-in set of code generators. A workbench may also provide the user with the implementation and the running of user defined model transformations. In that case, it implements a model transformation language and is able to run model transformations written in its model transformation language.

2.6. *MDE meta-workbench*

A MDE meta-workbench is a tool which provides facilities to build and run specialized MDE workbenches. It is based on the implementation of a meta-meta-model which is specifying the meta-modeling language. *MetaEdit+* [28] is an example of a meta-workbench using visual programming. Primarily, it enables specifying domain graphical languages. For a particular domain language, it allows a designer to specify conforming models and enables running of model transformations.

2.7. *Model Driven Engineering*

Main goals of Model Driven Engineering (MDE) are to improve the software development process productivity, to facilitate software portability and interoperability and finally to improve maintenance and documentation of software components. As an example, the Model Driven Architecture [29] (MDA), defined by the Object Management Group (OMG), is a well-known MDE framework. The MDA development life cycle is globally defined as a classical development life cycle. The originality lies on the nature of the artifacts that are always models. These models can be automatically understood and transformed by computers. Software systems are made of models and early defined meta-models describe them at a very high level of abstraction. As such, these models are independent of any implementation technology and of any execution platform. These high level models are then automatically transformed in more specific models, taking care of a particular platform and of a set of target technologies. Finally, platform specific models are automatically transformed in code which can be compiled, executed and tested in the target environment.

2.8. *The STEP standard and the EXPRESS language*

STEP, namely the STandard for the Exchange of Product model data is the *ISO 10303* standard [30]. Its main goal is to facilitate product data models exchange between heterogeneous systems. To define data models and to implement data exchange, a rich set of technical standards has been implemented. Mainly, the EXPRESS data modeling language is used for data model specification and the STEP neutral data encoding format is used for data exchange.

EXPRESS is the ISO-10303 standard data modeling language [31]. With EXPRESS, a data model is specified with a set of schemas that can reuse each-other. A schema is made of types, entity hierarchies and constraints declarations. Functions and procedures can also be defined for the formal specification of constraints. EXPRESS makes very precise cardinality constraint specifications possible with the help of inverse relation declarations. A constraint can be either global to a model or local to an entity.

Regarding the four levels architecture, a data model and a set of STEP encoded data instances are at the M1 level. The EXPRESS language and the STEP neutral formatting language are at the M2 level. The EXPRESS language is used to specify itself as well as the STEP neutral formatting language. Thus, EXPRESS constitutes also the meta-meta-model at level M3.

2.9. *Modelling an example*

Figure 1 shows an EXPRESS example which is specifying domain entities for a very simple mail sender. More accurately, the schema *St11MailBox* declares which data are to be exchanged in the system, two kind of entities in our case. The *St11Mail* specifies a mail to be sent, represented by a target mail address, a subject, and the content of the mail. The *St11MailBox* entity contains the sender mail address and a list of *St11Mail* instances (mails to be sent).

```
SCHEMA St11MailBox;

  ENTITY St11MailBox;
    owner : STRING;
    mails : LIST [0:?] OF St11Mail;
  END_ENTITY;

  ENTITY St11Mail;
    target : STRING;
    subject : STRING;
    content : STRING;
  DERIVE
    sender : STRING := box.owner;
  INVERSE
    box : St11MailBox for mails;
  WHERE
    adrRule : (sender > '') and (target > '');
  END_ENTITY;

END_SCHEMA;
```

Fig. 1. An EXPRESS example

The *St11Mail* entity is declared also with two computed attributes. The attribute *sender* is a derived attribute: an expression is given to compute its value. The *box* attribute is an inverse attribute. It links a mail to its sender and specifies what is the inverse cardinality for the association between an *St11MailBox* and its *St11Mail* instances. In this example, a *St11Mail* belongs to one and only one *St11MailBox*.

```
#100=ST11MAIL('apl@lob.fr','demo','test1');
#110=ST11MAIL('apl@lob.fr','demo','test2');
#300=ST11MAILBOX('vr@re.fr',(#100,#110));
```

Fig. 2. STEP encoded data instances

Figure 2 shows a set of data instances encoded with the STEP neutral format. These data instances are valid according to the specification given in Figure 1. Each instance is encoded with an unique identifier (e.g. #100) followed by the name of the entity and the attribute values. If a value is a reference to an instance, then the encoded value is made of the referenced instance unique identifier (e.g. the #100 list element value in the #300 encoded instance).

3. An approach for early verification and validation of meta-models

Our approach is intended to make easier the verification and the validation of a meta-model as soon as it has been changed. This capability is called early verification and validation. We will use the Unified Process [32] terminology to describe our approach.

This idea is not new. Recent software life cycles are using iterations to adapt the target system as soon as a source artifact has been updated. For instance, while a program is being coded, design and requirement problems can be found. Once the problem has been detected, an iterative cycle should to correct the design or update the requirements.

After a brief presentation of the Unified Process, in this Section we presents our iterative meta-modeling process allowing early validation with the help of the target system under construction.

3.1. *Verification and validation during an Unified Process iterations*

This Section describes the operations to yield for verification and validation during an Unified Process iteration. We first give the terminology, and then, we briefly explain what has to be done, how and when.

3.1.1. *The unified Process iteration*

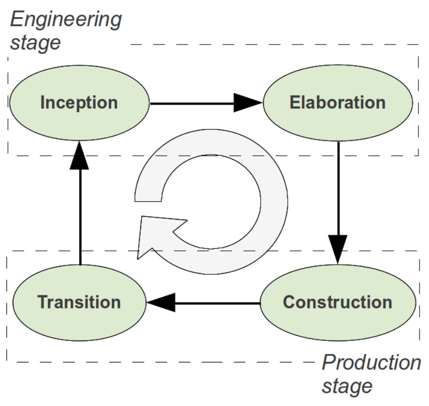

Fig. 3. The unified process iteration

Figure 3 depicts an Unified Process iteration. Each iteration is managed as a software project and is made of two main stages, called the Engineering and the Production stages:

– the Engineering stage consists of the inception and elaboration phases; schedule and technical feasibility are the main artifacts of the inception phase whereas the analysis and design are produced during the elaboration phase;
– the Production stage consists of the construction and transition phases; implementation, integration and test of meta-models are achieved during the construction phase and the transition phase includes code generation and integration of the generated code in the target system.

3.1.2. *Verification and validation*

For a given iteration, implementation or evolutions of meta-models are performed during the production stage. Especially during the construction phase, models conforming to the meta-models under elaboration are intensively used for the evaluation of the meta-models. The goal is to enhance the reliability and the usability of meta-models for the domain and to put on the test functional requirements under consideration. Two kinds of work are performed.

– A two-step verification. A first step verifies the structural characteristics of the meta-models. The designer evaluates their capacities according to the preliminary analysis and according to the domain rules. In a second step, the designer evaluates the invariants or constraints which are specified in the meta-models;
– Validation. Validating the meta-models according to the functional requirements; it might use testing, type checking or prototyping.

3.1.3. *An usual approach using a meta-workbench for verification and validation*

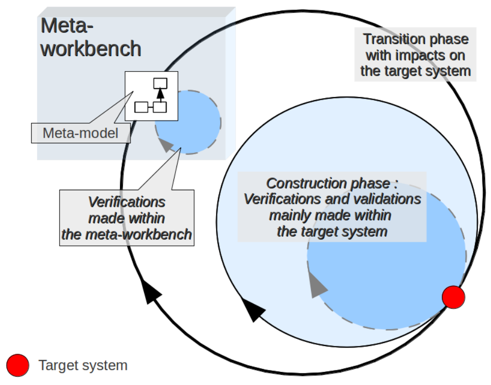
Meta-workbench
Meta-model
Transition phase with impacts on the target system
Verifications made within the meta-workbench
Construction phase : Verifications and validations mainly made within the target system
Target system

Fig. 4. Classical approach for verification and validation

With the usual MDE approach, the first step is the build of the target system. The target system is generated from meta-models that are specified within a meta-workbench (transition phase).

As it is shown in Figure 4, verification and validation (construction phase) mainly occur after the target system is ready to run. This implies the following four operations: (1) the target code is generated from the source meta-models, (2) the target code is compiled, (3) the target system is initialized and configured with appropriate instances and (4) the system runs.

Depending on the application size, these operations can be very tedious and time consuming. Thus, several minutes to hours can be necessary before to run verifications and validations. For sure, a few part of early verification is achieved in the meta-workbench. For instance, when the modeling language is statically typed, static type verification might be performed in the meta-workbench.

In next Section we describe our meta-modeling process which enables early checking of meta-model constraints and early validation with tests and prototypes within the meta-workbench.

3.2. *Outline of our meta-modeling process*

A part of verifications and validations do not need a complete target system. We consider the two following cases:

(i) verifications that may require to verify types but also constraints that are specified within the meta-model.
(ii) validations that may not need the target system infrastructure to be programmed or generated then executed.

If the meta-workbench provides the user with executable meta-models, a part of verifications and validations can be performed in the meta-workbench before the code generation.

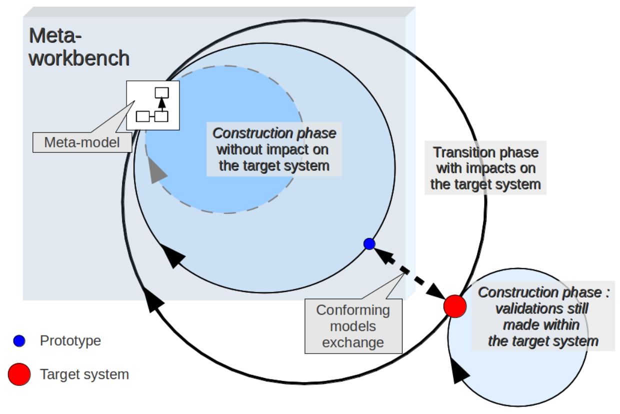
Meta-workbench
Meta-model
Construction phase without impact on the target system
Transition phase with impacts on the target system
Conforming models exchange
Construction phase : validations still made within the target system
Prototype
Target system

Fig. 5. Construction and transition phases

Figure 5 represents an iteration. From the meta-workbench point of view, inner circles represent the construction phase that is achieved in the meta-workbench without any direct impact on the target system. The outer circle represents the transition phase. It is also implemented in the meta-workbench but results in an update of the target system.

Very important validations have to be performed in the target system. The target system code might be only partially generated. The integration of the generated code as well as the functional requirements have to be validated in a running target system. Indeed, the generated code remains the main artifact and final validations should be achieved on it.

3.3. *Using the target system for verification and validation within the meta-workbench*

Although some verifications and validations can be achieved within the meta-workbench, the target system remains very important. Indeed, for the verification and the validation of a meta-model, it is desirable to be able to get realistic and rich models. But, producing such models in the meta-workbench might be very difficult and imply a waste of time. Compilers or advanced user interfaces might be necessary to produce them.

Often, facilities to produce realistic models are already available in the target system. Thus, it might be very helpful to benefit in the meta-workbench from the models that will be produced for the target system. These models can be exchanged from the target system to the meta-workbench by file exchange. In our implementation, we are using the STEP technology to automatically generate a model exchange component from the domain meta-model. This exchange component implements an automatised serialization/materialization of conforming models into/from a file. Given a meta-model, such exchange component is integrated into the meta-workbench as well as into the target system. Then, a conforming model built into the target system can be serialized into a exchange file. This file can be read from the meta-workbench to materialize the exchanged conforming model.

Using the exchange component, a target system of the version N can always produce and send to the workbench conforming models which are used for the evaluations done during the elaboration of the $N + 1$ release.

4. Using Platypus

In this Section, after a brief presentation of what is Platypus, we are illustrating our approach with an end-to-end small example.

4.1. *Basic principles behind Platypus*

Smalltalk is a dynamic language [33], without any static typing capability. Regarding the MDE context of our works, static types are useful for documentation, type checking and code generators building.

The idea behind Platypus is to offer to designers the enrichment of Smalltalk models with static type declarations. Then, dedicated tools can use type declarations. Types are considered as additional and optional annotations which are used only when they are required. The immediate benefit of optional types is that it is possible to freely elaborate a meta-model in Smalltalk, to implement tests and validations and to evaluate them immediately into the meta-workbench.

When types are specified, type declarations can be used to improve the validation with type checking or to implement code generators. When a new release is to be implemented, the Smalltalk meta model can evolve without any constraint regarding the type descriptions which was given during the previous iteration. Of course, when the new version is considered as stable again, it is possible to update the type descriptions to fit the new release of the meta-model.

Another benefit is that types are only required where they are useful.

Platypus is a Pharo tool that enables the static typing of Smalltalk models with the help of the EXPRESS language [31]. The STEP data encoding standard [34] is also implemented for the exchange of conforming models.

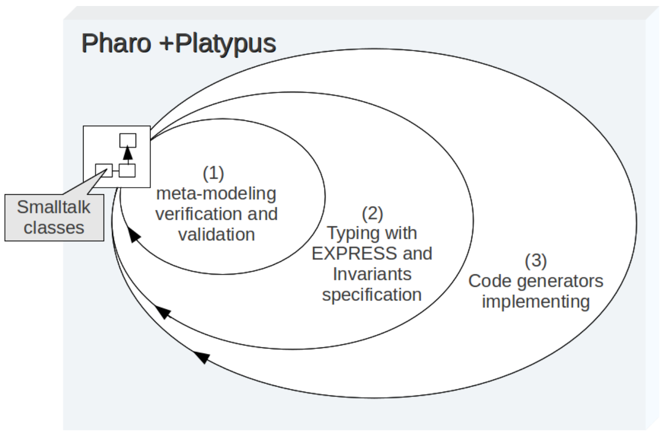
Pharo +Platypus
Smalltalk classes
(1) meta-modeling verification and validation
(2) Typing with EXPRESS and Invariants specification
(3) Code generators implementing

Fig. 6. Using of Pharo with Platypus

Figure 6 depicts how Pharo and Platypus are used for the verification and the validation of a meta-model. From the inner to the outer cycle:

(i) according to the domain, meta-models are implemented using Smalltalk classes; prototypes and tests are implemented regarding the functional requirements;
(ii) when the meta-model is mature and stable enough, property types, association cardinalities and invariants can be declared using the EXPRESS language;
(iii) additional static type checking tools can be implemented to enforce the meta-model validation and code generators can be built upon the meta-model.

4.2. *A first meta-model in Smalltalk*

As a starting point, we are building a *St11MailBox* meta-model with the unique class *St11Mail*. The meta-model is declared into Pharo as the *St11MailBox* package. The declaration of the *St11Mail* class within the *St11MailBox* package is shown in Figure 7. A mail consists of four attributes: the sender and target mail addresses (*sender* and *target* attributes), the subject and the content of the mail (*subject* and *content* attributes). All getter and setter accessors for these attributes are also implemented. Accessors for the *sender* attribute are shown in Figure 8.

```
Object subclass: #St11Mail
  instanceVariableNames: 'sender target subject content'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'St11MailBox'
```

Fig. 7. First version of the St11Mail class

So far, we can generate the corresponding Java component to implement the target system. But, it might be time

```
St11Mail>>sender
  ^ sender
St11Mail>>sender: aString
  sender := aString
```

Fig. 8. The set and get accessors for the *sender* attribute

consuming to generate code, to implement the email sending function, to set up a test and to run it.

The basic function of our mail client is sending mail. Thus before the Java package generation and the target system implementation, we would ensure that sending mail is running as expected with our meta-model.

Using Pharo, we can implement the simple script shown in Figure 9. This validation script builds the *mails* array with several *St11Mail* instances. Then, with each *St11Mail*, a *MailMessage* is instantiated and really sent with the help of the Pharo *MailSender* class.

```
| mails |
mails := {
  (St11Mail new sender: 'apl@univ-brest.fr';
    target: 'fsi@univ-brest.fr';
    subject: 'Smalltalk 2011';
    content: 'Are you ready for Bueno Aires ?';
    yourself).
  St11Mail new ...}.
mails
  do: [:m | | mes mime |
    mes := MailMessage empty.
    mes setField: 'from' toString: m sender.
    mes setField: 'to' toString: m target.
    mes setField: 'subject' toString: m subject.
    mime := MIMEDocument
      contentType: 'text/plain'
      content: m content.
    mes body: mime.
    MailSender sendMessage: mes].
```

Fig. 9. A script to early evaluate the mail system

Running this script, we are noticing that sending mail is possible. Thus we can decide with some degree of confidence, to generate the Java code and to build a first version of our target system.

This validation script exhibits an important gain using Smalltalk as a meta-workbench. Indeed, the designer benefits from a very rich system allowing meaningful validations to be implemented. For example the *MailMessage*, *MailSender* and *MimeDocument* classes as well as a network layer are supplied by the Pharo system. Validation scripts and test cases can be implemented and run at each time a meta-model is updated.

4.3. *Declaring and manipulating types*

To generate the Java code, we need first to implement a Smalltalk to Java code generator. To implement this generator, we have to declare the static types of *StMail* attributes and to browse the types graph of the meta-model. With Platypus, this is achieved with a separate EXPRESS schema. The first version of the EXPRESS schema for our

```
SCHEMA St11MailBox;

  ENTITY St11Mail;
    sender : STRING;
    target : STRING;
    subject : STRING;
    content : STRING;
  END_ENTITY;

END_SCHEMA;
```

Fig. 10. First EXPRESS schema for the mail sending application

sending mail system is shown in Figure 10. It consists in a single *St11Mail* entity. The correspondence between the *St11Mail* entity and the related Smalltalk class is straightforward. When the EXPRESS schema is compiled, Platypus automatically reifies the EXPRESS entity *St11mail* in a Smalltalk representation that is directly accessible from the *St11Mail* class itself. Thus, we can browse the EXPRESS declarations. For instance, the script of Figure 11 prints the type of the *sender* attribute.

```
| attr |
attr := St11Mail platypusMetaData explicitAttributes
  detect: [:a | a name = 'sender'].
Transcript show: attr domain asClearText.
```

Fig. 11. Accessing an EXPRESS attribute description with Smalltalk

4.4. *Generating the code*

To generate parts of the target system, we must implement a code generator. The code generator can be built wile browsing the EXPRESS declarations as explained in the previous Section.

In Section 3.3, we explained that the target system and the meta-workbench can exchange models for validation purpose. Platypus comes with an EXPRESS to Java code generator that produces the java domain meta-model and a STEP file exchange component [1]. Using the code generator, the generated code is a package that contains:

- a Java domain meta-model with each entity translated as a Java class (e.g. *St11Mail* Java class);
- a STEP file exchange component with a dedicated repository class (e.g. *St11MailBoxStepRepository* class).

The Java sending mail system is implemented using the generated package. While the package runs, it builds *St11Mail* instances. The instances can be registered into a *St11MailBoxStepRepository*. A *St11MailBoxStepRepository* can produce a STEP file through a serialization of its contents. The file contains a *St11MaibBox* conforming model. It can be read from our

[1] Currently, an EXPRESS to c++ code generator is also available

meta-workbench and used to further validate the corresponding $St11MailBox$ meta-models.

4.5. *Using conforming models produced from the target system*

Let us suppose that we generated the Java component from the $St11MailBox$ EXPRESS schema and that the target system has been built with the generated component. The running target system has to manage its own $St11Mail$ instances. A STEP exchange file can be produced from the target system and materialized in Pharo. The script given in Figure 12 shows how to read an exchange file and how to use the materialized instances.

```
| repo |
" Creation of the repository "
repo := St11MailBoxRepository new.
" materialization from the 'mails.step' file "
repo stepFileIn: 'mails.step'.
repo
  allInstancesOf: St11Mail
  do: [:m | | mes mime |
    mes := MailMessage empty.
    mes setField: 'from' toString: m sender.
    mes setField: 'to' toString: m target.
    mes setField: 'subject' toString: m subject.
    mime := MIMEDocument
      contentType: 'text/plain'
      content: m content.
    mes body: mime.
    MailSender sendMessage: mes].
```

Fig. 12. Evaluating the mail system with instances produced from the target system

4.6. *Improving the meta-model*

While a system is under implementation, the meta-model is continuously evolving. Producing a new version of the meta-model implies that the Smalltalk implementation and the corresponding EXPRESS specification are updated.

For a basic evolution, the EXPRESS specification can be directly changed. Then, the Smalltalk code is automatically updated accordingly.

A problem may arise while using a conforming model produced from the target system. Regarding to the sending mail system example, some instances of $St11Mail$ might have been created with a misformed *sender* or *target* mail address. A possible problem fix would be to better specify the concept of mail adress. Figure 13 shows a new version of the meta-model using the $St11MailAddress$ concept.

```
SCHEMA St11MailBox;

  ENTITY St11Mail;
    senderAdr : St11MailAddress;
    targetAdr : St11MailAddress;
    subject : OPTIONAL STRING;
    content : STRING;
  DERIVE
    sender : STRING := senderAdr.address;
    target : STRING := targetAdr.address;
  END_ENTITY;

  ENTITY St11MailAddress;
    localPart : STRING;
    domain : STRING;
  DERIVE
    address : STRING := localPart + '@' + domain;
  WHERE
    localPartRule : localPart > '';
    domainRule : domain > '';
  END_ENTITY;

END_SCHEMA;
```

Fig. 13. A new version of the St11MailBox meta-model

From the meta-workbench point of view, the immediate benefit is that validations can be enforced with further checking. Indeed, rules and derived attributes can be used by validation script.

```
| repo |
" Creation of the repository "
repo := St11MailBoxRepository new.
" materialization from the 'mails.step' file "
repo stepFileIn: 'mails.step'.
repo
  allInstancesOf: St11MailAddress
  do: [:m |
  (m platypusMetaData whereRules
    collect: [:wr | wr label asSymbol])
    do: [:r | self assert: (m perform: r)]].
```

Fig. 14. Checking meta-model rules in Smalltalk

For instance, the script given in Figure 14 implements a checking of all local rules specified into the $St11MailAddress$ entity. This validation script works as follow:

– first, a STEP file is read by a repository, as a result, the repository contains $St11Mail$ and $St11MailAddress$ instances;
– for each $St11MailAddress$, the names of the local rules declared in the $St11MailAddress$ entity are collected; each of this name is also the name of a $St11MailAddress$ instance method which has been automatically generated by Platypus;
– for each collected rule name, send the corresponding selector to the current $St11MailAddress$; if the result of the method execution is false, then raise and exception.

From the target system implementation point of view, the designer can use the new Java functions which are generated according to the derive attributes and the rules. As an example, the $St11MailAdress$ rules can be invoked from the user interface to validate the user input [2].

5. An implementation with Pharo and Platypus

Platypus consists in the implementation of a STEP/EXPRESS environment in Smalltalk. In this Section we explain how it is implemented and especially how static types are internally managed.

5.1. *Two modeling levels*

Domain modeling is achieved with Pharo. During an iteration, we are handling domain concepts and elaborating (or re-elaborating) the domain model according to the functional requirements. We call *prototyping level* the level of abstraction whereas this kind of modeling is performed.

When a domain model reaches a mature state according to the iteration goals, then the domain types are declared separately with the help of a set of EXPRESS schemas. Only types of the meta-model that are shared with the target system are declared.

In Pharo, additional checkers can use declared types and constraints to enforce the validation of the domain meta-models. This additional work is done at a lower level of abstraction that we call the *typing level* of abstraction. Figure 15 depicts these two modeling levels.

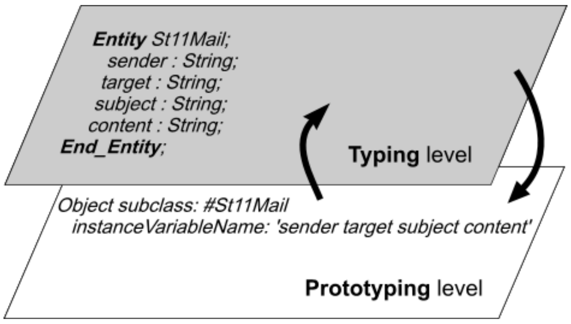

Fig. 15. Prototyping and typing levels

When an EXPRESS schema is integrated, then the corresponding Smalltalk representation is built or updated. Each first-class EXPRESS element is represented as a corresponding Smalltalk class. Thus, these two levels of abstraction are causally connected.

Any change in an EXPRESS schema is automatically reported on the Smalltalk representation. Specific behaviors and specific states that can are added in the Smalltalk representation for test or prototyping purpose are preserved when the Smalltalk representation is updated [3].

A designer is not constrained to start from the prototyping level. In fact, we noticed that it is often very comfortable to start a design from the typing level by directly integrating an EXPRESS schema.

Let us take again the example of the entity *St11Mail* of Figure 15. When the EXPRESS schema is compiled, a *St11Mail* class is created or updated. This class is the Smalltalk representation of the *St11Mail* EXPRESS entity.

5.2. *Internal representation of types*

As depicted in Figure 16, the two modeling abstraction levels are also implemented at the meta-meta-model architecture level. Indeed, the meta-model of Platypus is specified itself as an EXPRESS schema and also managed by Platypus. The Platypus meta-model describes all EXPRESS elements and also all concepts for the handling of STEP instances. Thus, it provides a description of the abstract syntax tree (AST) which is built when an EXPRESS schema is compiled. A part of the Platypus implementation is generated thanks to this meta-model.

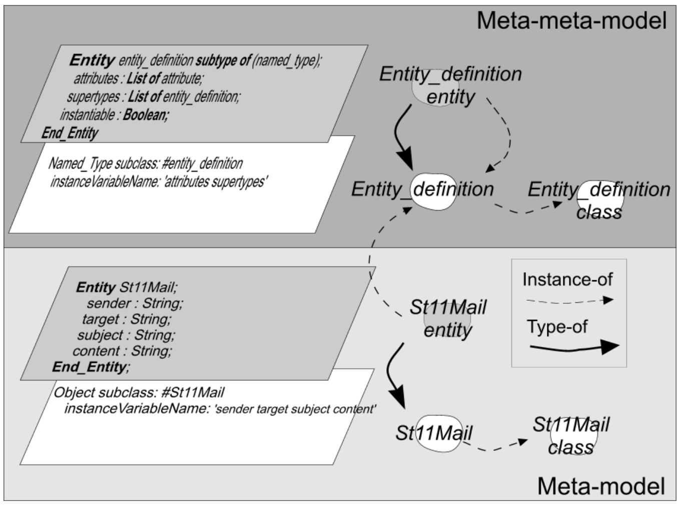
Meta-meta-model
Entity entity_definition subtype of (named_type);
attributes : List of attribute;
supertypes : List of entity_definition;
instantiable : Boolean;
End_Entity
Named_Type subclass: #entity_definition
instanceVariableName: 'attributes supertypes'
Entity_definition entity
Entity_definition
Entity_definition class
Entity St11Mail;
sender : String;
target : String;
subject : String;
content : String;
End_Entity;
Object subclass: #St11Mail
instanceVariableName: 'sender target subject content'
St11Mail entity
St11Mail
St11Mail class
Instance-of
Type-of
Meta-model

Fig. 16. Platypus modeling levels

The compilation of an EXPRESS meta-model produces the related Smalltalk code (Prototyping level) and also a set of instances of the Platypus meta-meta-model. All the types declarations of the Typing level can be queried from the Prototyping level (introspection).

As explained in Section 4.3, the #platypusMetaData message can be sent to a class that has been automatically generated by Platypus. In fact, the sending of #platypusMetaData to a receiver returns a Platypus meta-meta-model instance which owns the description of the receiver.

Regarding the $St11Mail$ example, sending #platypusMetaData to $St11Mail$ returns an instance of $PltEntityDefinition$. $PltEntityDefinition$ is the Smalltalk class which is generated from the $entity\_definition$ entity specified in the Platypus meta-meta-model. $entity\_definition$ describes what an entity is. For instance, it contains the entity attributes and the local rules descriptions.

[2] Our Smalltalk code generator actually generates code for derived attributes and for rules but currently, our Java code generator do not and the generated has to be manually updated

[3] We are currently working on the inverse causal connection, from the Smalltalk representation to the EXPRESS schemas

5.3. *Specialization of Platypus*

We may implement a specific behavior in the Platypus meta-meta-model class. As an example, for implementing a particular code generator, one can add specific methods to the Platypus meta-meta-model classes. It is also possible to implement a subclass of a Platypus meta-meta-model class. This provides a system designer with the ability to specialize the Platypus environement itself.

6. Related work

A lot of meta-modeling tools have been proposed to facilitate the specification and the verification of meta-models. Dome [35], MetaEdit+ [28] and MetaGme [16] have been developed to design domain specific meta-models. Dome is implemented in Smalltalk and was initially dedicated to the automatic building of Smalltalk components and of documentation. Meta-models are designed graphically and a dedicated language (a Scheme like language) is available for constraint specification and evaluation and also for code generating. MetaEdit+ is also a graphical tool for the specification of domain specific graphical modeling language and for code generating. As far as we know, it does not provide any way to express and evaluate domain constraints. MetaGme is a more recent generic tool for the specification of domain specific languages. MetaGme is based on an UML notation and allows a designer to specify and evaluate constraints.

These tools are suitable for domain specific modeling and meta-models verification. However, tests and simulations can not really be implemented because they do not provide enough programming capabilities.

EMF/Ecore [15] and Kermeta [9] are two meta-modeling frameworks based on Ecore. They are implemented within Eclipse. EMF is made to efficiently design the structure of meta-models. Many code generators are available and a lot of modeling tools are build upon EMF. Kermeta is a general purpose meta-modeling tools. The Kermeta meta-modeling language is very similar to the EXPRESS language. The behavior can be implemented with the help of *aspects* which are reusable domain functions written with a Java-like syntax. Kermeta also relies on code generation for meta-models implementation and running. All these environments are using static typing for the specification of meta-models.

As for our work, the motivation of Riehle and al. for an UML virtual machine [12] was to bring early validation for UML models. The goal was to build an environment based on a Meta Object Protocol which benefits from a causally connected modeling architecture. However, as far as we know, the possibilities of this interpreter regarding testing and prototyping were not explained.

The Moose [36] is a very powerful language-independent environment for reverse-and re-engineering complex software systems. Moose provides a set of services including a common meta-model, metrics evaluation and visualization, a model repository, and generic GUI support for querying, browsing and grouping [36]. As our work, Moose is fully integrated in Smalltalk. Moose can also be used for meta-models verification and validation. The implementation of Moose relies also on a four level modeling architecture and our experiments showed us that Moose could be used for our approach. The main difference concerns the ways meta-models are integrated in a Smalltalk system. We are using the EXPRESS data modeling language whereas Moose meta-models are declared with the help of method annotations or via the analysis of Java or C++ source code.

7. Conclusion

In this paper, we have presented our motivations for the use of Smalltalk as an MDE workbench. We have also showed how an iterative software process can be adapted to use Smalltalk together with an additional static typing capability for the very early validation of meta-models. Static typing is available through the use of the modeling language EXPRESS and a dedicated model compiler.

References

[1] D. C. Schmidt, Model-driven engineering, IEEE Computer 39 (2).
URL http://www.truststc.org/pubs/30.html

[2] P. Mohagheghi, V. Dehlen, Where is the proof? - a review of experiences from applying mde in industry, in: Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications, ECMDA-FA '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 432–443.
URL http://dx.doi.org/10.1007/978-3-540-69100-6_31

[3] S. J. Mellor, Models. models. models. so what?, in: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, MODELS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 1–1.
URL http://dx.doi.org/10.1007/978-3-642-04425-0_1

[4] L. Osterweil, Strategic directions in software quality, ACM Comput. Surv. 28 (1996) 738–750.
URL http://doi.acm.org/10.1145/242223.242288

[5] Research Triangle Institute, The Economic Impacts of Inadequate Infrastructure for Software Testing, Sponsored by the Department of Commerce's National Institute of Standards and Technology (2002).

[6] N. Mellegård, M. Staron, Characterizing model usage in embedded software engineering: a case study, in: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ECSA

'10, ACM, New York, NY, USA, 2010, pp. 245–252.
URL http://doi.acm.org/10.1145/1842752.1842800

[7] K. Beck, Embracing change with extreme programming, Computer 32 (1999) 70–77.

[8] S. Ducasse, T. Girba, Using smalltalk as a reflective executable meta-language, in: International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2006). volume 4199 of LNCS, Springer-Verlag, 2006, pp. 604–618.

[9] P. Muller, F. Fleurey, J. Jézéquel, Weaving executability into object-oriented meta-languages, in: in: International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005, Springer, 2005, pp. 264–278.

[10] R. F. Paige, Specification-driven development of an executable metamodel, in: In Proc. Workshop in Software Model Engineering 2004, co-located with UML, 2004.

[11] R. Paige, P. Brooke, J. Ostroff, Agile development of a metamodel in eiffel, in: Proc. Fifteenth IEEE International Symposium on Software Reliability Engineering 2004, 2004.

[12] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe, The architecture of a uml virtual machine, SIGPLAN Not. 36 (2001) 327–341.
URL http://doi.acm.org/10.1145/504311.504306

[13] O. Nierstrasz, M. Denker, L. Renggli, Model-Centric, Context-Aware Software Adaptation., in: B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems, Vol. 5525 of LNCS, Springer, 2009, pp. 128–145.
URL http://hal.inria.fr/inria-00532825/en/

[14] S. Sendall, W. Kozaczynski, Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software 20 (5) (2003) 42–45.

[15] F. Budinsky, S. A. Brodsky, E. Merks, Eclipse Modeling Framework, Pearson Education, 2003.

[16] J. Davis, Gme: the generic modeling environment, in: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 82–83.
URL http://doi.acm.org/10.1145/949344.949360

[17] F. Jouault, J. Bézivin, KM3: a DSL for Metamodel Specification, in: FMOODS'06, Vol. 4037, 2006, pp. 171–185.

[18] Object Management Group (OMG), Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04 (2003).
URL http://www.omg.org/docs/ptc/03-10-04.pdf

[19] Eclipse project, http://www.eclipse.org/.

[20] B. Foote, Objects, Reflection, and Open Languages, Workshop on Object-Oriented Reflection and Metalevel Architecture - ECOOP'92.
URL http://www.laputan.org

[21] G. Kiczales, J. D. Rivieres, The Art of the Metaobject Protocol, MIT Press, Cambridge, MA, USA, 1991.

[22] Pharo.
URL http://www.pharo-project.org

[23] A. Plantec, V. Ribaud, PLATYPUS : A STEP-based Integration Framework, in: 14th Interdisciplinary Information Management Talks (IDIMT-2006), République Tchèque, 2006, pp. 261–274.
URL http://hal.univ-brest.fr/hal-00504325/en/

[24] Platypus (2004).
URL http://cassoulet.univ-brest.fr/mme

[25] J. Bézivin, O. Gerbé, Towards a Precise Definition of the OMG/MDA Framework, in: Automated Software Engineering (ASE 2001), IEEE Computer Society, Los Alamitos CA, 2001, pp. 273–282.

[26] K. Thomas, Matters of (Meta-)Modeling, Software and Systems Modeling 5 (4) (2006) 369–385.
URL http://www-adele.imag.fr/users/German.Vega/idm/seance4/MattersOfMetaModeling.pdf

[27] OMG, Model Driven Architecture (2003).
URL http://www.omg.org/mda

[28] MetaEdit+ Technical Summary.
URL http://www.metacase.com

[29] OMG, OMG Model Driven Architecture.
URL http://www.omg.org/mda/

[30] ISO 10303-1, Part 1: Overview and fundamental principles (1994).

[31] ISO TC184/SC4/WG11 N041 WD, EXPRESS Language Reference Manual (1997).

[32] I. Jacobson, G. Booch, J. Rumbaugh, The unified software development process, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[33] L. Tratt, R. Wuyts, Guest editors' introduction: Dynamically typed languages, IEEE Software 24 (5) (2007) 28–30.

[34] ISO 10303-21, Part 21: Clear Text Encoding of the Exchange Structure (1994).

[35] E. Engstrom, J. Krueger, Building and rapidly evolving domain-specific tools with dome, in: Proceedings of IEEE International Symposium on Computer-Aided Control System Design (CACSD, 2000, pp. 83–88.

[36] O. Nierstrasz, S. Ducasse, T. Gǐrba, The story of moose: an agile reengineering environment, SIGSOFT Softw. Eng. Notes 30 (2005) 1–10.
URL http://doi.acm.org/10.1145/1095430.1081707