Reusable Modelling Tool Assets: Deployment of MDA Artefacts

MIGUEL A. DE MIGUEL, EMILIO SALAZAR, JUAN P. SILVA, JAVIER FERNANDEZ-BRIONES

Technical University of Madrid, E.T.S.I. Telecomunicación

Ciudad Universitaria, 28040 Madrid Spain

{mmiguel,esalazar,psilva,jfbriones}@dit.upm.es
+34 913 367 366 3047

Abstract. Model driven development attempts to resolve some common problems of current software architectures in order to reduce the complexity of software development: i*) how to increase the level of abstraction by centring on software models;* ii) *how to automate the software development process through the use of transformations and generators*; and iii) *how to separate domain, technology and technological concerns so as to avoid confusion arising from the combination of different types of concepts.* Model driven development uses two basic solutions to resolve these problems: i) description of specialised modelling languages and ii) model transformations and mappings. For each domain and technology, MDSD (Model-Driven Software Development) requires specific MDA (Model Driven Architecture) artefacts for the definition of specialised languages and transformations that address specific modelling languages and platforms. The application of MDSD in a specific domain and technology combines multiple interdependent MDA technologies (e.g. MOF (Meta-Object Facilities), QVT (Query-View and Transformation), MOF2Text, UML (Unified Modelling Language) extensions, OCL (Object Constraint Language)); MDSD combines these technologies to construct and improve tools that support the model driven development process adapted to specific domains, technologies and platforms (e.g. e-commerce, safety-critical software systems, and SOA (Service Oriented Architecture)).

The maintenance and evolution of software models require solutions in order to integrate all these MDA technologies and avoid dependency on their tools. The various kinds of MDA artefacts have interdependencies (e.g. model transformation and modelling language extensions), which complicate their reuse and their adaptation to new development environments. This chapter proposes solutions for the integration of all MDA technologies based on reusable modelling tool assets, and solutions for deploying artefacts so as to provide modelling tool independence. MDSD must address these problems because, in the near future, the migration of these developments to new development platforms will be as complex as current migrations from one specific run-time platform to another.

Keywords: MDA artefacts, Reusable Modelling Assets, Eclipse Modelling Tools, Modelling Tool Environments

1 Introduction

A basic objective of model-driven software development is to place emphasis on the model when developing software. This is a change from the current situation, in that it shifts the role of models from contemplative to productive. The goal of model-driven engineering is to define a complete life-cycle method based on the use of various models automating a seamless process from analysis to code generation [10]. This discipline puts all the software artefacts in the right place (e.g. business models, architectural models and design patterns) and actively uses them in order to produce and deploy applications.

Models provide solutions for different types of problems: i) description of problems and their concepts, ii) validation of descriptions and concepts represented through checking and analysis techniques, iii) model transformation and generation of code, configurations, and documentation.

Separation of concerns avoids the confusion generated by combining different types of concepts. Model-driven approaches introduce solutions for specialising models for specific concerns and for interconnecting concerns based on model transformations. This approach reduces the complexity of models through specialised modelling activities that are separated. It improves communications between stakeholders by using models to support the exchange of information. However, separation of concerns often requires specialised modelling languages to describe specific concerns, and the interoperability of specialised languages requires tools integration.

MDA proposes a set of languages and technologies [7] to construct of modelling tools that adapt MDSD to specific platforms (e.g. EJB (Enterprise Java Beans), RTSJ (Real-Time Java Specification)) and technologies (e.g. transactions, security). Standards defining such languages are: MOF [18], QVT [19], MOF2Text [21], OCL [17], UML [14], UML profiles and RAS (Reusable Asset Specification) [20]. MDSD combines these languages to create artefact infrastructures, applicable in modelling tools, to then construct MDSD environments. However the artefact's dependence on tool's infrastructures makes the artefacts tool-dependent and therefore application models are also tool-dependent. The MDA philosophy to avoid platform dependency based on PIM (Platform Independent Model) and PSM (Platform Specific Models) is not reflected in the development of MDA artefacts. For example, UML modelling-tool facilities, such as profile registration and support of stereotype applications based on modelling framework tools (e.g. EMF (Eclipse Modelling Framework)), make the profiles and the models that reuse the profiles tool dependent. When exporting the models, the profiles must be exported too, and manual adaptations must be done within the model, because the profiles installed in the target tool cannot be reused. This process requires extensive experience working with models and is not feasible for complex models.

From a conceptual view-point, this chapter addresses four important challenges of MDSD methods:

1. MDA artefact development requires significant effort and has a large number of modelling tool dependencies. Adoption of these technologies must be flexible and should not impose specific development environments. The solutions proposed in this chapter improve the *adoption of MDA artefacts in different modelling frameworks* and this improves the adoption of MDA artefacts because it does not impose specific tools.
2. MDA proposes solutions to reduce dependencies on execution platforms, but it creates dependencies on modelling tools. The solutions proposed in this chapter increase the maintainability of model-driven applications because they increase the *portability of MDA artefacts* and this avoids dependencies on modelling tools.
3. OMG has created multiple standards and languages for the specification of transformation behaviours and MDA functionality. These languages include QVT operational, QVT relational, MOF2Text, OCL and Java. No standard solutions are defined for the interoperability of these languages (e.g. invoke QVT transformations from Java or from MOF2Text). This chapter proposes solutions for handling and simplifying this *MDA behaviour artefact interoperability*.
4. *The learning curve of MDA infrastructures* for specific domains and technologies (e.g. DSL, specific transformations and generators, specific UML extensions) should be as short as possible and their application should increase productivity as much as possible. The relationships between learning duration and application benefits must be positive, and only a well planned learning curve can make this relationship positive. The solutions introduced in this chapter pay special attention to the integration of MDA artefacts and to tool support for tutorials and documentation of domain specific MDA artefacts.

This chapter introduces the concept of Reusable Modelling Asset (RMA) and its application. RMA defines formats for interchanging assets, including MDA artefacts, to make them reusable in different modelling tools, and to support their tools' independence. This solution is based on the MDATC OMG Standard [24]. ERMA is an implementation of this framework in *Eclipse*. Implementations of Eclipse Reusable Modelling Assets (ERMA) are available[1]. The RAS standard [20] defines languages and infrastructures for the interchange of software assets. RMA defines an RAS profile that extends the default RAS profile to support the description of MDA artefacts that can be combined to create a specific modelling-tool asset, supporting the application of MDSD in specific domains and technologies. Figure 1 represents the general structure of an ERMA asset file. This structure is based on the RAS standard.

Figure 1

[1] http://138.4.11.45/~erma/w/index.php/CHESS:General

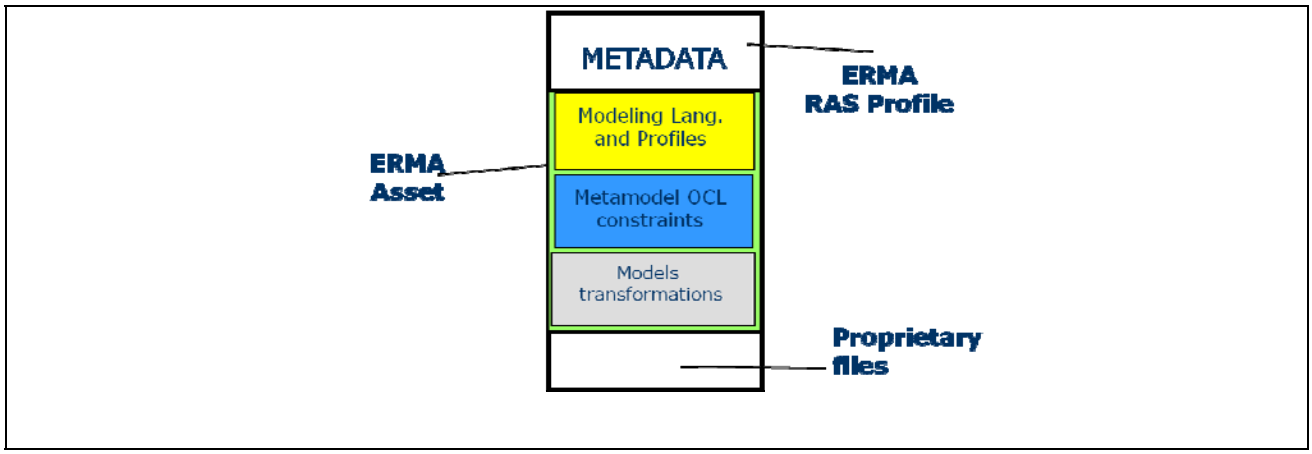

Figure 1. RMA File General Structure

RMA provides solutions for interchanging and combining MDA artefacts, concentrating most modelling tool dependencies in RMA implementations; RMA makes assets reusable in different modelling tools. RMA defines services for: i) editing RMA assets and their artefacts, ii) packaging artefacts and metadata into RAS format enabling the relocation of models (e.g. UML profiles, abstract and concrete modelling language specifications) and transformations (e.g. QVT operational, and MOF2Text modules), iii) delivery of assets to RAS repositories, iv) downloading of assets, validating and relocating asset dependencies (dependencies with other assets and with external artefacts) on the target platform, v) deployment of assets and customisation for the target platform, and vi) installation/de-installation of deployed artefacts.

Eclipse plug-ins and features [27], and OSGi Bundles [26] are examples of software infrastructures that could be used to support RMA objectives, but they have four significant features that RMA tries to avoid: i) *Eclipse* plug-ins and OSGi Bundles are designed to support Java applications, while RMA is designed to support modelling languages and behaviour specification including languages such as QVT and MOF2Text, which are not supported in plug-ins and bundles. ii) *Eclipse* plug-ins depends on the *Eclipse* tool, while RMA attempts to avoid tool dependency as much as possible. RMA is based on OMG standards, for which there are only two important exceptions: EMF and GMF. EMF is currently the de-facto standard in defining modelling languages and *GenModel* models provide support to avoid modifications to generated code. This language specification does not have an equivalent in EMOF. Diagram Definition (DD) [25] is the OMG standard that supports the same concepts supported in GMF, but this standard is in the process of being finalised and there is no *Eclipse* open source implementation available yet. iii) Plug-ins and bundles are not designed to be integrated into RAS frameworks (e.g. their integration in RAS repositories and management tools would require a wrapper asset that describes the asset, which should describe concepts such as plug-in dependencies in terms of asset dependencies). IBM has created some adaptations in asset management tools to support this approach for *Eclipse* features and plug-ins. iv) RMA is designed to be edited in RMA diagrams (a DSL (Domain Specific Language) and a diagram editor, an alternative to which could be a UML profile) that support the editing of RMA assets; XML schemas, text editors and specialised editors support

the editing of plug-ins and bundle manifest files. Some UML modelling tools (e.g. RSA[2], Modelio[3] and AndroMDA[4]) provide frameworks for editing and handling modules and cartridges that integrate different kinds of artefacts, but they frequently only support a limited number of artefacts and are sometimes tool specific; we cannot interchange these modules between different tools.

IBM-Rational has done some important developments of tools (e.g. Rational Asset Manager) and methods [1] for the application of asset-based software development. These solutions are well integrated with UML modelling tools.

The current implementation of ERMA is based on several *Eclipse* incubation projects and is therefore in a evolving state (e.g. *QVTo* has modified the type of URI (Uniform Resource Identifier) accepted in the *modeltypes* definition in recent *Eclipse* versions; incubation versions of QVT Relations support some QVT standard packages such as *QVTRelation* and *QVTBase* but some packages, such as *QVTOperational*, are supported in neither QVT declarative nor QVT operational, and because of this, QVT abstract syntax and QVT XMI (XML Metadata Interchange) interchange are only partially integrated).

ERMA has been used in the development of multiple assets. These are assets that allow model driven support for complex kinds of development such as Safety applications and MARTE based model development environments (see the annex to this chapter). ERMA RAS repositories provide multiple assets for these two types of development environments (MARTE based development and Safe-Aware modelling systems). These assets support several kinds of MDA artefacts, such as UML profiles and model libraries, QVTo, Acceleo and Java transformations, *ecore* and *genmodel* based DSL and OCL based commands. These DSL languages integrate some analysis tools into Eclipse, such as Item Toolkit for safety analysis and MAST for real-time analysis. These assets have been developed to demonstrate the applicability of ERMA in real applications.

In the remainder of this chapter, Section 2 introduces the RMA life cycle to apply RMA concepts, Section 3 introduces the RMA specification language, Section 4 describes the design and implementation of asset deployments, Section 5 introduces future work, and finally Section 6 includes some discussion and conclusions.

[2] http://www.ibm.com/developerworks/rational/library/06/1114_kelsey/index.html

[3] http://www.modeliosoft.com/en/modules/modelio-modules.html

[4] http://www.andromda.org/docs/andromda-cartridges/index.html

2 General Introduction to the RMA Life-Cycle

The development and application of RMAs are broken down into several phases. Two important kinds of modeller actors are: *RMA developer* and *applications modeller*. The *RMA developer* designs and implements modelling infrastructures that should be reusable in different projects developed for the same domain and technologies. The *application modeller* develops the application models and reuses the assets in the RMA that are available for the construction of application models and their implementation.

Figure 2 summarises the activity phases in RMA development and application, and Figure 3 represents an example of the deployment of tools to execute these activities. These phases and activities are:

1. **Development**: The *RMA developer* carries out development activities on MDA artefacts. Development places most of the effort in the RMA life-cycle and, in particular, in the development of MDA artefacts. All activities in this phase would be executed in the *Eclipse Source Modelling Tool* node in Figure 3. The results of this phase are RMA assets that can be released.
    a. *Development of MDA Artefacts*. MDA artefacts are: UML Profiles, UML model libraries, QVT, Java and MOF2Text transformations, DSL abstract (EMF) and concrete syntax (GMF), and additional elements such as OCL and GUI (Graphical User Interface) wizards. The *RMA developer* should use specific editors such as UML profile modellers, and QVT editors.
    b. *RMA Editing*. The RMA modeller is based on the RMA specification language (introduced in Section 3). This language supports the integration of MDA artefacts into a common asset; and the RMA model defines the contents of the RMA asset and its dependencies on external artefacts. MDA artefacts are developed with a common purpose (e.g. to support the application of MDSD in technologies such as SOA), and different kinds of artefacts can be combined (e.g. UML extensions, DSL for configuration languages, model transformations and code generators). The RMA models combine all these artefacts into a common asset.
    c. *RMA Assembly*: Frequently an RMA asset depends on other assets for several reasons: i) the input and output languages of behaviours are included in other assets, ii) the pipeline combines transformations and generators to generate target code or documents, and the generators and transformations are distributed over several assets; a pipeline transformation combines all these transformations. iii) A new language is defined that extends profiles or modelling languages included in other assets, and the extension reuses behaviours included in the extended models. iv) Some general purpose assets are applicable in multiple contexts and specialised assets reuse and customise these general assets (e.g. façade assets that provide support for handling several languages, for example *ecore* and UML, with a common meta-model façade). The RMA language includes assembly dependencies, usage relationships on behaviours included in other assets, and dependencies for describing

references to artefacts included in other assets or in the modelling tool (e.g. dependencies on standard languages meta-models such as OCL) .

Figure 2

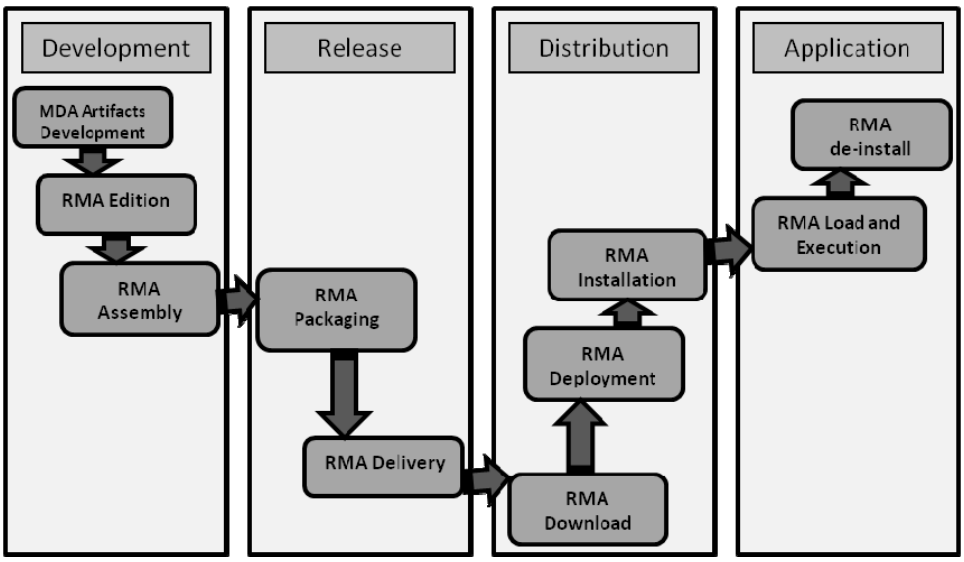

Figure 2. RMA Development and Application Phases and Activities

2. **Release**: The editing and assembly of an asset yields a specification of the contents and dependencies of the asset. This specification includes references to the models and files developed with MDA artefacts editors (e.g. MOF2Text modules, and Java jar files). This specification will be used to publish the asset in an RMA asset repository. To publish the asset, we must package the asset in an RAS file and deliver it to the repository:
    a. *RMA Packaging*: the packaging of assets takes into account all references from the RMA specification and packages all models and files into a single RAS [20] file (a zip file which includes a manifest file that provides a description of the asset contents and dependencies). The packaging resolves the relocation of models; RMA packaging supports *zip* and *ras* URI schemes to reference modelling elements embedded in other RAS files. As an asset can assemble other assets, contained in the global asset; the packaging will include container and containment assets in a common RAS file. The *Eclipse Source Modelling Tool* node in Figure 3 executes the RMA packaging.
    b. *RMA Delivery*: The RAS asset includes the manifest file, and we can deliver the asset in a RAS repository that supports the ERMA RAS profile. Modelling tools can connect to the repository, search for and download assets in the repository. Delivery takes into account asset identifiers and version numbers for the identification of assets in the repository. In the deployment scenario included in Figure 3, the delivered asset will be available in the *RAS*

Working Group Server repository; in the ERMA implementation, a WebSphere SOA server hosts this repository.

3. **Distribution**: An *application modeller* can reuse RMA assets for the construction of application models and for their transformation.
    a. *RMA Download*: *The application modeller* can search for assets and look through the documentation in the assets, selecting the assets to reuse it in its modelling tools and models. Downloading an asset will copy the asset from the repository and will create an asset project in the target tool for the deployment and installation of the asset. In the example depicted in Figure 3, the *ERMA Target Modelling Tool* is the target tool and it downloads the asset from the *RAS Working Group Server* while the documentation is available at the *ERMA MediaWiki* server.
    b. *RMA Deployment*: Section 4 includes a detailed description of deployment activity for the asset and its artefacts. The RMA implementation within the target modelling tool supports the deployment of assets and artefacts into the target modelling tool. The deployment reuses target tool infrastructures (e.g. plug-ins, projects, profile supports) for the integration of new artefacts into the target tool. In the example in Figure 3, the *ERMA Target Modelling Tool* executes the deployment; embedded documentation will be deployed at *Target MediaWiki*, and the remaining artefacts are deployed at the *ERMA Target Modelling Tool*.
    c. *RMA Installation*: Deployment generates artefacts for the target modelling tool, and these artefacts are installed in the target tool. This installation is tool-dependent. But in *Eclipse*-based modelling tools, installation will include plug-ins, projects and features that the deployment generates.

Figure 3

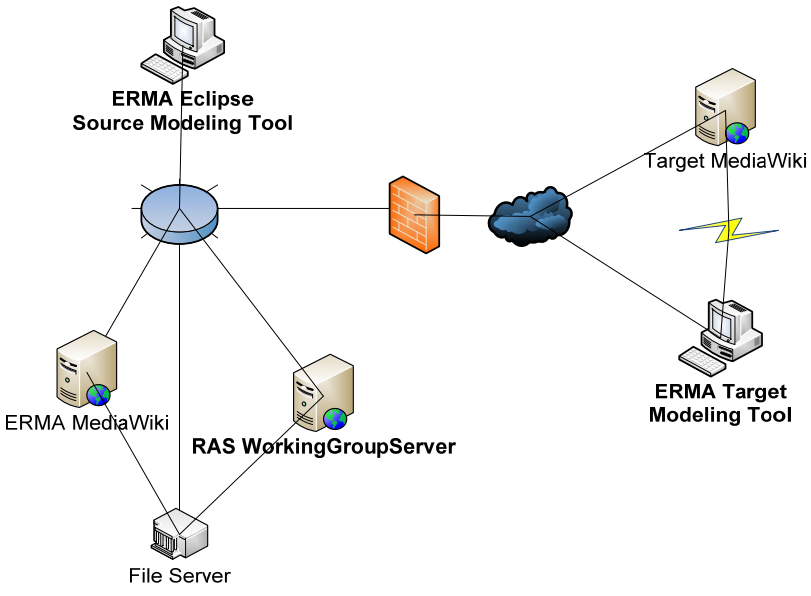

Figure 3. Example of distribution for RMA Servers and Modelling Tools

4. **Application**: *The application modeller* reuses asset behaviour and modelling languages and extensions to construct application models. In the example in Figure 3, the *ERMA Target Modelling Tool* executes the asset application.
    a. *RMA Loading and Execution*: The installed artefacts create new types of repositories to support new DSLs, installation registers, new UML profiles and model libraries, creating new commands that support the execution of transformations and generators. They also provide new commands to access documentation and for general asset configuration. All these new facilities support the application of MDSD techniques in the domain and technologies supported in the asset.
    b. *RMA De-install*: The target modelling tool can de-install the asset if the asset facilities are not going to be reused. De-installation removes the UML profiles and model libraries, the modelling languages supported in the asset and the commands that provide access to transformations in the target tool.

3 Introduction to the RMA Language Specification

The RMA language specification is based on the MDATC [24] standard. The RMA language provides support for the description of MDA artefacts that are combined for a common purpose. In *Eclipse Modelling*, MDA, languages and frameworks have associated editors and run-time support for the specific languages (e.g. *ecore* modeller, QVTo editor, MOF2Text language editor); but there is no common framework for the description of integrated solutions.

ERMA use the RMA language for two main purposes: editing RMA assets to define their contents and dependencies, and generating the RAS manifest file (*METADATA* in the *ras* file in Figure 1). The ERMA framework includes an EMF-based implementation of the RMA language, a GMF based RMA diagram editor, and a QVT transformation from the RMA language to the RAS ERMA profile. Figure 4 shows the integration of the ERMA diagram editor and QVT transformation into the RAS ERMA Profile. The editor integrates ERMA models and diagrams and other MDA editors, and it also handles ERMA models (supported in an EMF repository). ERMA packaging tools integrate the QVT transformation from an ERMA model to an ERMA RAS manifest file defining the *ras* file contents. Figure 7 includes an example of an RMA diagram; this diagram represents four assets, five *assembly* relationships and three *import* dependencies. Each asset in the diagram includes some artefacts (QVT, EMF, MOF2Text and Java jar). These artefacts define new modelling languages and their associated transformations. The *Global* asset reuses transformations included in other assets and provides a GUI command (*GlobalCommand*) to execute the combined transformation.

Figure 4

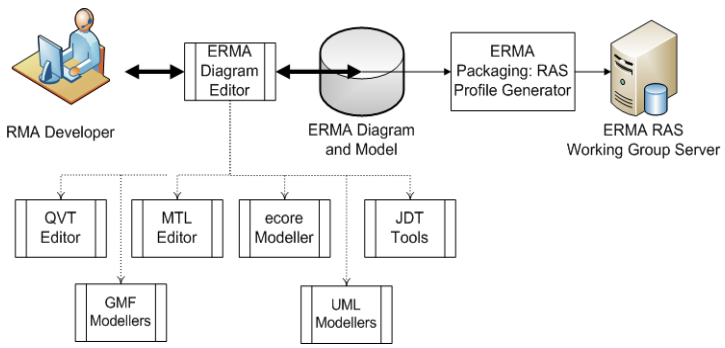

Figure 4. ERMA Modeller and RAS manifest file generator

The RMA Language specification is an extension of the *Constructs* package in *UML Infrastructure* [16]. The RMA language specification merges *Constructs* and *PrimitiveTypes* packages into *UML Infrastructure* and packages that include the RMA language specification. *Constructs* is the package that merges the *Kernel* package in *UML Superstructure* [14] (*Kernel* is the specification core for UML class concepts in the UML 2.x *Superstructure*) and provides a minimum language for the presentation of classes, packages and simple data types. The ERMA reuses and modifies the *UML Infrastructure* standard; but the ERMA complies with the standard and only modifies the body constraints of meta-class operations and the constraint annotations of derived associations to support *UML Infrastructure* behaviours in OCL and *ecore* (operation bodies and derived associations are implemented in OCL and the code of operations and derived associations are embedded in the ERMA *ecore* meta-models). The RMA extensions to *UML Infrastructure* include constructors for the representation of MDA artefacts (Sections 3.1 and 3.2), and *interaction points* that represent functionality provided in the asset (Section 3.3), and relationships for the description of assets and artefact dependencies (Section 3.4). RMA modelling elements include references to the following modelling language specifications: MOF2Text, QVT, GenModel, *ecore*, GMF gmfgraph, GMF gmfmap, GMF tooldef , UML *Superstructure* and OCL. The ERMA implementation of *erma* language, in *ecore,* includes references to the equivalent *ecore* meta-models.

3.1. RMA Artefacts

RMA assets are the core modelling elements in the RMA modelling language, and RMA artefacts are the main elements used to describe assets.

RMA supports three kinds of MDA artefacts: artefacts for defining modelling languages, software behaviour artefacts for handling and transforming models, and general artefacts. Table 1 depicts the different types of artefacts. Software behaviour artefacts describe transformations and other behaviours (e.g. model queries)

provided in the asset. Modelling language artefacts define modelling languages to extend UML modellers and to create new DSLs. Assets can reuse elements in the modelling languages as data types (e.g. parameters, results and invocation context) to describe the provided functionalities. The RMA can contain artefacts that will be embedded in the asset, and RMA assets can reference standard artefacts available in any modelling tool (e.g. OCL meta-model and UML profile *Standard*), as well as artefacts contained in other assets.

Types of RMA Artefacts	RMA Artefacts	
Software Behavior	• EMF Meta-model • MOF2Text o MOF2Text concrete syntax o MOF2Text XMI model • Java Jar	• QVT o QVT Operational ▪ QVTo concrete syntax ▪ QVTo XMI model o QVT Relational o QVTr concrete syntax o QVTr XMI model
Modelling Languages	• UML Profile • UML Model Library	• EMF Meta-model • GMF Diagram Definition
General	• Media Wiki documents • Icons	• Executables • Others

Table 1. Software behaviour, modelling language and general artefacts

The RMA language considers two modelling language specifications: UML modelling language extensions (based on UML profiles and UML model libraries) and DSLs (based on EMF and GMF). The RMA language specification includes four modelling elements for describing these kinds of artefacts, and each modelling element specialises in describing one artefact type.

In the RMA specification, artefact meta-classes extend the *Infrastructure::Core::Constructs::Package* meta-class and one attribute of the *UML infrastructure* packages is *visibility*. Allowed values for *visibility* are *private*, *package* and *public*. Only elements in the asset can reference *private* artefacts, other assets can reuse and extend *public* artefacts and only elements in the same RMA package[5] can include references to *package* artefacts. The relationships between artefacts depend on the languages (e.g. GMF include references to *ecore* and other GMF models, and UML profiles can include references to other UML profiles and UML model libraries), but all these references (or inheritances) must guarantee the *visibility* of the artefacts.

3.2. RMA Software Behaviour Languages

[5] RMA assets can include hierarchies of packages defining RMA data types, artefacts and other RMA modelling elements

The RMA asset specifications include model handlers to transform models and to use or edit the information they contain. The *application modeller* and other asserts can use these handlers (in the case of *public* artefacts), or they can be used for local purposes only (in the case of *private* artefacts).

Software behaviour artefacts support the implementation of functional specifications (e.g. transformations, generators, GUI wizards and edition support) included in the assets. MDA languages used to represent software behaviours are QVT, MOF2Text and Java. EMF *ecore* operations can also be used in the description of software behaviours (EMF artefacts can represent modelling languages or software behaviour or both). OCL and Java can describe implementations of *ecore* operations. RMA *interaction points* (Section 3.3) specify software behaviour operations that are handled in the asset, and they are also the core elements for the specification of commands, operation calls and other asset behaviours.

Software behaviour artefacts can include dependencies: one artefact can reuse another artefact in the same language. QVT, MOF2Text and *ecore* behaviours can invoke operations in Java jar artefacts (Java jar artefacts can support QVT *black-boxes* and Java implementations of *ecore* operations can reference Java libraries contained in other Java artefacts), and Java artefacts can invoke any kind of behaviour.

3.3. RMA Interaction Points

Many languages support MDA behaviours (in RMA they are QVTo, QVTr, MOF2Text, OCL, Java) and they are interoperable. RMA *interaction points* reduce the complexity of interaction and simplify the invocation of behaviours from one language to another.

RMA *interaction points* specify functionality that the RMA asset supports and they also specify the interfaces between behaviour artefacts. An *interaction point* has an associated set of *UML infrastructure* operations that specify the services supported at the *interaction point*. OCL body constraints or software behaviours (Java methods, *ecore* operations, QVT transformations and mappings, and MOF2Text modules) support the implementation of operations at *interaction points* (multiple artefacts can support the same *interaction point*, but an operation only has one associated artefact or one OCL value expression). The RMA language supports OCL constraints on the description of the body of operations (RMA reuses UML *Infrastructure* to describe operations). When the *interaction point* implementation is in another language, it references behaviours based on *interaction point implementation* relationships (this is a relationship between a software behaviour artefact and an *interaction point*). An *interaction point* specification can be associated with an execution context (OCL and EMF software behaviours require a context). The run-time execution of an *interaction point* operation must be performed within the context of a data type that defines the *interaction point* context. Frequently, the context references a meta-class that defines the type of modelling elements that can be used as run-time context.

The RMA asset specification includes *RMA call provided* and *RMA call required*. *RMA call required* represents language independent behaviour to be reused. Asset assembly must link *RMA call required* and *RMA call provided* to identify the asset in the target tool that will support the software behaviour. All *RMA call required* elements in an asset must be resolved before packaging. The *implementation of external interaction* modelling element represents the relationship between *call required* and *call provided* elements.

Interaction points are used for four purposes:

- *To define commands that the asset includes in GUI menus*: an asset specification can include the GUI commands provided in the asset. The command specification includes the symbolic name for the command, a reference to the icon, and a reference to the *interaction point* that specifies the behaviour. The deployment of assets generates tool-dependent code to support the command execution: the code takes the execution context into account, requests values for the operation parameters, invokes the implementation operation, and displays the execution results. *GlobalCommand* in Figure 7 is an example of a provided command; the Java implementation (*GlobalImpl*) consists of two transformations (*ModelX2Modely* and *Modely2CodeZ*).
- *To describe functionality that one asset provides to other assets*: the asset can provide services to other assets. The main purpose of *interaction points* is to reduce language dependencies among assets. When asset maintenance modifies the implementation language for an *interaction point*, and the operations have the same interface, the clients do not need to be modified. The deployment of *interaction points* that support *call provided* generates Java, and QVT and MOF2Text *black-boxes*. Software behaviour languages use generated code and the RMA run-time reflexion APIs (Application Programming Interface) to invoke operations in a language independent way. The modelling elements *interaction point*, *RMA call provided* and *implementation of external interaction* specify the provided services. *ModelX2ModelY*, *QueryInX*, and *ModelY2CodeZ* in Figure 7 are examples of *RMA call provided* modelling elements. The *implementation of external interaction* modelling element links these elements with their implementation specification in *interaction points*.
- *Local interoperations:* software behaviour assets can invoke the execution of *private* operations implemented in some other artefact (in the same language or a different language) defined in the same asset. For local interoperations, the asset includes the specification of *interaction points* but it does not include the specification of *RMA call provided*.
- *Behaviours to execute during the installation and de-installation of assets*: Asset specifications can include the specification of *interaction points* that should be executed during the installation and de-installation of assets. In general these are *private* behaviours.

3.4. RMA Relationships

The previous subsection introduced three relationships in RMA modelling language: *implementation of external interaction*, *implementation of interaction point* and *implementation of external interaction*. The RMA extends *UML Infrastructure* and the RMA specification reuses and extends relationships included in the *UML Infrastructure* meta-model (e.g. *DirectedRelationship*).

Two fundamental relationships in the RMA language are:

- *External artefact references*: frequently, software behaviour assets and modelling language artefacts extend or reuse standard behaviour and languages available in MDA modelling tools (e.g. UML meta-model, OCL model library, and Java standard libraries). The RMA *reference* relationship represents these dependencies; the dependent artefacts can extend and invoke the modelling elements and behaviours, and *interaction points* can reuse the meta-classes to define context and operation data types. The deployment of assets must take these dependencies into account.
- *Assembly of assets*: the *implementation of external interaction* and *assembly* relationships will represent these dependencies, when an asset reuses artefacts in another asset. The deployment and installation of assets will resolve these dependencies in the target platform. The RMA supports two types of *assembly* specification: embedded assembly and combined (not embedded) assembly. Embedded assembly represents the composition of assets to create a more complex asset, while combined assembly represents some aggregation dependencies (several assets can be combined into a common asset, but combiner assets do not package or deploy the combined asset). The release of an asset that embeds other assets will package, in the same asset file, the artefacts in the embedded asset and artefacts included in the container asset. The distribution of combined assets will resolve asset assembly, and the assembled asset must be available in the target tool for the distribution of assembler assets. Figure 7 includes some combined assembly relationships, where the *LanguageY* asset includes a modelling language specification (*Y*), and the *CodeZ* asset includes a generator from *Y* to *Z*. The generator can only be executed when the source language is supported in the target tool, and because of this assembly is needed. This assembly is combined and not embedded, because it would be possible to have a modelling tool that supports *Y* language and does not support the generator.

3.5. Related Work

RMA language and infrastructures depends on several OMG standards (*UML Superstructure* [14], QVT [19], MOF2Text [21], MOF [18] and OCL [17]) and three standards in particular: MDATC [24], *UML Infrastructure* [16] and RAS [20]. The implementation of these standards in *Eclipse Modelling* is the main reference for ERMA design.

Some UML modelling tools and MDA frameworks integrate modelling tool extensions solutions. These extension approaches are tool specific, and they integrate specific types of artefacts and languages to specify transformations. Some tool examples are:

- *Modelio: Modules*. *Modelio* is a UML modelling tool and its environment supports the installation of modules that customise the tool for specific technologies and domains. The two fundamental artefacts in modules are UML profiles and Java based transformations. Java transformations are based on a proprietary API to access repositories.
- *IBM Rational Software Architect: Pluglet*. Pluglets are Java applications that are integrated into the RSA/Eclipse modelling tool that can access the application model in the model space using EMF/UML2 interfaces. RSA still does not integrate *QVT Relational* or *Acceleo* projects.
- *AndroMDA: Cartridges*. Cartridges provide support for handling model elements, in particular, annotated elements with stereotypes or model elements that meet certain conditions. Cartridges process these elements using template files defined within the cartridge descriptor.

RMA language and services are based on the MDATC standard, which proposes solutions for the integration of MDA artefacts and RAS. *Bendraou* and others [4,5] encourage using the MDATC concept, but they do not provide solutions for integration into RAS standards or *Eclipse Modelling* tools.

Model-Bus modelling infrastructure [2,6] look for solutions to integrate different kinds of modelling tool services. Model-Bus is centred in the model service concept. Model-Bus provides support to communicate with different model services, but is not centred on the interchange of these services between modelling tools.

IBM and other companies have developed infrastructures and tools for the application of asset-based software development. These solutions, in general, are oriented to software applications development [11,12,13]. This chapter focuses on the construction of assets for development and application of MDA artefacts. Rational Asset Manager[6] is good example of a framework for the management of assets; it provides solutions for the construction, export and import of assets, and it is well integrated with RSA modelling tool. IBM has customized these tools for the development of MDA artefacts [1]. But IBM engineers paid special attention to RSA specific kinds of artefacts or *Eclipse* specific artefacts (e.g. JET, Java Emitter Templates, and plug-in features. See chapters 18, 19 and 20 in [1]) and there is no special attention to the customization of assets in alternative modelling tools (other than RSA). These solutions need improvements for the application of standards such as QVT and MOF2Text, and for the support of concrete syntaxes and DSL

[6] http://www.ibm.com/developerworks/rational/tutorials/r-helloram/resources.html, http://www.ibm.com/software/awdtools/ram/

graphical editors. Solutions presented in this chapter, such as the integration of UML profiles in meta-model frameworks, could improve these methods and tools.

4 Deployment of RMA Assets

Point 3.b (*Distribution – Deployment*) in Section 2 introduces RMA deployment activity and its integration within the RMA life-cycle. This section introduces the most important challenges and their solutions using *Eclipse Modelling Tools* for the deployment of RMA assets. The challenges are broken down into several subsections, and each subsection includes a subset of the solutions. The solutions included in this chapter (ERMA) depend on implementations integrated into *Eclipse Modeling Tools*[7], *Helios Release* SR2 (build id: 20110218-0911) and the *Eclipse* EMF, MDT, M2M, M2T, and GMF projects integrated in this release.

4.1 General Issues of MDA Artefact Deployment

The deployment of an RMA asset creates new mechanisms in the target tool to support MDA artefacts. Deployment distributes and relocates models, templates, java jar files and other MDA artefacts around these new mechanisms. Relocation and redistribution in target mechanisms must resolve several general problems:

Challenge 1.1 - Relocation of internal RMA references and zip and ras URI schemes. To maintain the consistency of models and other artefacts, references must be updated during relocations. RMA relocation solutions are based on XMI [22] interchanging models. The URI is the most common solution to support references in XMI. RMA is also based on RAS, and RAS packages artefacts in *ras* format (*zip* files with the RAS structure and manifest files). To make both approaches compatible, deployment in the target tool must support *zip* and *ras* schemes in URI handlers. The implementation of eclipse used does not support these schemes. Solutions for this challenge are on page 18.

Challenge 1.2 - Relocation of external RMA references. Models in *ras* files can include references to models and files embedded in other assets. During its life cycle an asset has multiple locations and, before application, the reused-assets models are relocated into target tool mechanisms (e.g. projects, configuration files and cartridges), where references must be updated to the new locations. Relocations must be considered when deploying and installing reused assets. Relocations update the following references (targets can be both internal artefacts and reused-assets artefacts): external references in UML profiles, model libraries, EMF models, GMF models, QVT and MOF2Text XMI files, URI in QVTo *model types* and input *typed models* in MOF2Text. RMA models includes references to MDA artefacts, URL references to general artefacts such as *MediaWiki* documents and icons, and references to Java jar files. Solutions for this challenge are on page 19.

[7] http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/heliossr1

Challenge 1.3 - Visibility of MDA artefacts. In general, the *application modeller* reuses the installed MDA artefacts (profiles, *ecore* models, transformations) embedded in an RMA asset. However, sometimes these artefacts are created for internal purposes: decomposition of transformations into several intermediate transformations, *ecore* models to represent intermediate transformations, Java jar files that support *black-boxes* for QVT and MOF2Text transformations or shape definitions for GMF canvas. *Application modeller* and other assets should only access *public* artefacts. Solutions for this challenge are on page 19.

Challenge 1.4 - Reuse of standard models and Software Behaviours. Frequently, RMA artefacts reuse and extend standard models and artefacts such as the UML meta-model and standard profiles, the OCL meta-model, and Java standard libraries. These artefacts are not embedded in RMA assets but RMA assets can reuse them (e.g. an RMA asset that packages code generators for UML models reuses the UML *ecore* model). The location of these models and libraries is tool-dependent. The asset must specify these dependencies and the deployment must provide support to resolve model and behaviour dependencies. Solutions for this challenge are on page 20.

4.1.1. Solutions in ERMA for MDA Artefacts Deployment

ERMA implements the RMA specification in *Eclipse Modelling* and the deployment creates *Eclipse* plug-ins, EMF projects, GMF projects, QVT projects and Acceleo projects. ERMA deployment generates one QVT project for each QVT artefact and one *Acceleo* project for each MOF2Text artefact. All other artefacts are deployed into a single ERMA plug-in project. This plug-in includes the models (EMF, GMF and UML models), Java jar files, and other artefacts, icons and files. The deployment of *genmodel* and *gmfgen* models can generate additional plug-ins. All projects and plug-ins include ERMA plug-in dependencies, and the ERMA plug-in project includes plug-in dependencies from other projects. The ERMA plug-in project exports non-private Java packages, and limits the visibility of Java packages when artefact *visibility* is *packaged*.

ERMA solutions for challenges included in this section are:

Solutions for Challenge 1.1 - Relocation of internal RMA references and zip and ras URI schemes (see page 17). ERMA packaging services use relative path URIs for references to models and files included in the same *ras* file. The packaging service creates one folder for each embedded asset and the folder includes the artefacts in the embedded asset. EMF includes one Java class and two interfaces (and their implementations) that are fundamental for handling URIs in EMF:

a. *URI:* The URI class includes data for representing and parsing URI. URI handles schemes, fragments and other URI data, and can also handle schemes such as *zip* and *ras*. The class only represents these kinds of URIs.

b. *URIHandler:* this Java type provides information about the URI and implementations can handle specific types of URI. Zip and ras URIs require special implementations of this interface for handling the URI.

c. *URIConverter: URIConverter* is used in the framework for two purposes: to normalise URIs (convert one URI to another) before using them to create input and output streams to access resource information (each resource is associated with the URI that locates the resource's contents).

ERMA implements *URIHandler* and *URIConverter* interfaces with special classes that support *zip* and *ras* schemes. They normalise the references between models included in the same asset file and between models located in different assets. The *URIConverter* implementation uses a mapping table to support this transformation. Zip files impose some restrictions on *URIConverter* implementation. The most important one is that two output streams for the same zip file (two zip entries) cannot be handled at the same time.

An alternative solution studied is based on navigation with URI proxies. EMF supports the loading of models without remote-reference resolution (remote references can be maintained as unresolved URIs); the resolution is done explicitly (and URI can be substituted). But some generated Java code does not support this. For example, when the Java code for derived associations navigates the reference, it indirectly resolves other associations, and current implementations do not take into account unresolved remote references. The UML meta-model includes many derived associations, and UML profiles and model libraries have this same problem.

Solutions for Challenge 1.2 - Relocation of external RMA references (see page 17). Model deployment considers two scenarios: referenced models that are only available in installed plug-ins (the assets were installed, but current deployments are not in the same workspace as the installed assets), referenced models are in deployed projects in the same workspace (these projects could not be installed yet). The deployment relocation must take registered assets into account to locate the first case. For the second, the deployment looks for ERMA projects in the same workspace. Depending on the situation, *URIConverter* implementations are used in different ways. References to standard models must consider the registered models to locate these models in *Eclipse* plug-ins.

Solutions for Challenge 1.3 - Visibility of MDA artefacts (see page 18). ERMA deploys MDA artefacts to *Eclipse* plug-ins. Three kinds of visibility facilities for Eclipse plug-ins are handled during deployment: registrations in *Eclipse* extensions-points (EMF and GMF registration points for modelling languages, UML2 registration points for profiles, QVT transformation registrations and ERMA registration points for assets), Java exported packages and package visibility, and Java project *classpath*. Deployment of Java artefacts must take the Java exported packages, package visibility and *classpaths* into account, although these attributes of Eclipse manifest files do not support MDA specific languages such as QVT and MOF2Text. For specific MDA language packaging support validation on behaviour visibility, QVT *private* artefacts do not register

transformations (*Acceleo* does not include registration facilities for MOF2Text). The Java code generated locates the transformations (*public* or *private*) in the QVT and *Acceleo* plug-ins deployed. The *Challenge 3.1 - Visibility of EMF modelling languages* discusses specific details for EMF artefacts.

Solutions for Challenge 1.4 - Reusing standard models and Software Behaviours (see page 18). Another basic topic for supporting model relocation is unique and universal names (*nsURI*) for EMF packages and UML Profiles. These names are used to make universal references to XMI representation of profiles and meta-models, and these names must be employed to locate meta-models and profiles in *Eclipse* packages, which can be references to standard packages or references to models included in assets. *Eclipse Modelling* supports additional solutions based on the *pathmap* URI scheme to reduce dependencies on profile and meta-model versions (the same *pathmap* URI can reference the specific version of meta-model or profile installed in the Eclipse tool). Frequently, *pathmap* is used to represent the prefix of namespace paths to meta-models and profiles. For example, UML2 projects, in general, use the URI *pathmap://UML_PROFILES/Standard.profile.uml* to locate the *Standard* profile, and the two OMG *nsURI* for the two versions of *Standard* profile in UML 2.4 [15] are: *http://www.omg.org/spec/UML/20100901/StandardProfileL2* and *http://www.omg.org/spec/UML/20100901/StandardProfileL3*. *Eclipse* uses some schemes and names (e.g. *platform, pathmap, plugin, resource*) that can create problems for model interchange and they should be avoided. Current implementations of the QVT compiler and run-time resolve *ecore* meta-models with the registered *nsURI*.

4.2 Deployment of UML Profiles and Model Libraries

The deployment of UML profiles and model libraries must consider the issues mentioned in Section 4.1. Two different UML tools can interchange Profile and model libraries in XMI format. But the deployment of XMI profiles faces the following issues:

Challenge 2.1 - Profile repositories of the target modelling tool. UML modelling tools have their own file or data base formats to represent installed profiles. The deployment must convert the XMI format to the target format. Solutions for this challenge are on page 23.

Challenge 2.2 - References to standard and installed profiles and to modelling libraries. Profiles and model libraries can include references to elements included in other profiles and libraries located in the same asset or in external profiles (e.g. references to UML *Standard* profile or to the *UMLPrimitiveTypes* model library); UML modelling tools have specific solution for locating profiles and libraries. The deployment of profiles must resolve these external references to XMI files inside the asset or to installed profiles and model libraries. Solutions for this challenge are on page 23.

Challenge 2.3 - OCL constraints of profiles. UML profiles can have associated OCL constraints, and each modelling tool has specific approaches to evaluate OCL constraints. Solutions for this challenge are on page 23.

Another important topic for the deployment of UML profiles is the technology used for applying profiles in UML modelling tools. Stereotype application requires some notation to represent these stereotype applications and the values of features defined in the stereotype. Two solutions are: i) to represent profiles with MOF models; MOF meta-classes map the stereotypes and their applications as instances of meta-classes in MOF models. The UML 2.2 *Superstructure* standard [14] and the revision task force for UML2.4 [15] (this is not normative yet) propose this solution in profile specification sections, but they are recommendations and not normative. The UML2 project in *Eclipse* uses this solution based on *ecore* models; ii) to represent stereotype applications as instances of a UML *InstanceSpecification* meta-class, or similar structures; the classifier associated with these instance specifications are the stereotypes in the profile, and MOF repositories (or MOF models) are needed to handle the profile in the tool. UML 2.3 and UML 2.4 have a mandatory interchange format for stereotype applications and these formats are oriented towards MOF-based schemas.

Challenge 2.4 - Generation of MOF models for UML profiles. In tools that support profiles with MOF models, the deployment must build the MOF models based on profile models, for the target modelling tool. Solutions for this challenge are on page 24.

Challenge 2.5 - Application of profiles in model libraries. Model Libraries (and models in general) that apply UML profiles must represent these applications in XMI files:

a. If the tool supports profiles with MOF, the runtime classifiers of stereotype applications are MOF elements (e.g. meta-classes), and these versions of MOF models should be consistent with the profile model. The XMI format of stereotype applications is based on schemes that define the MOF meta-classes for the profiles.
b. When the profile is modified, new versions of the MOF model must be created, since old applications maintain references to the old MOF models. As a result, it is common to maintain multiple versions of MOF mappings, to maintain consistent references to old versions. However in that case, the MOF models and profiles can be inconsistent.

Solutions for this challenge are in page 24.

Challenge 2.6 - Extension and references from profile to profile. For profiles supported by MOF models, when a profile references or extends another profile, the MOF model must reference or extend the appropriate MOF model. In Figure 5 a new profile extends the UML profile *Standard*. A new stereotype (*MyMetaclass*), in the new profile, extends the standard *Metaclass* stereotype in the *Standard* profile. The extender stereotype includes a reference (*related*) to the *Metaclass* stereotype in the *Standard* profile. We create a UML model that applies the *Standard* profile and the new profile (shown on the right side of Figure 5). Two UML classes are annotated with stereotypes *Metaclass* and *OneMyMetaclass*; it should be possible, during the edition of *related* property, to

reference both stereotype applications. The implementation of this example in *Eclipse* project UML2 (and many other UML modelling tools based on the UML2 project), only allows the *OneMyMetaclass* stereotype application to be assigned to the *related* property. The source of the problem is the reuse of the MOF (*ecore*) model of the *Standard* profile, from the new MOF (*ecore*) profile model generated. The new MOF (*ecore*) model does not include references to the *Standard* MOF (*ecore*) model. This is a common problem of profile deployment. Solutions for this challenge are on page 25.

Figure 5

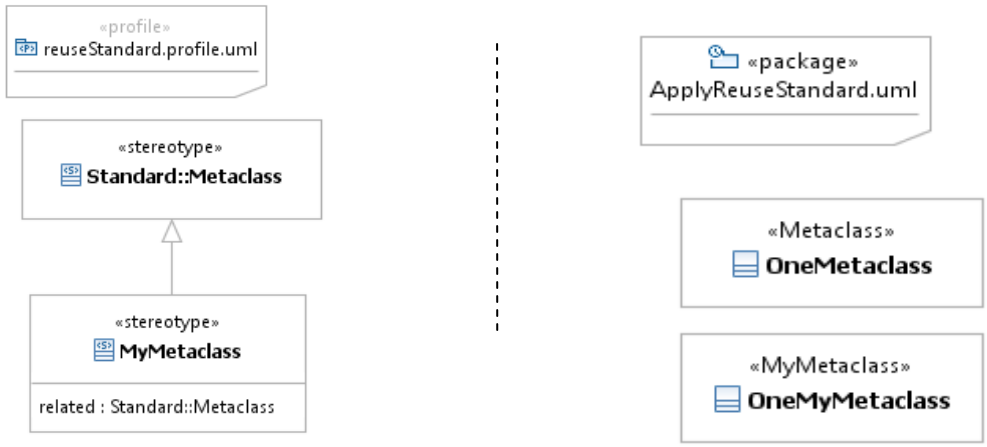

Figure 5. Problems arising from profile representation in MOF and profile extensions

Some additional problems in the deployment of profiles are:

Challenge 2.7 - Profile and model library registration. The location of Profile and Model Libraries must be registered during their application. UML modelling tools have proprietary methods for profile and model library registration (UML2 project and modelling tools that reuse this software do not use the same solutions). References to external profiles must employ these registration solutions. This is a common model interchange problem. The model and its profiles must be exported because the model interchanges that reuse registered profiles is a problem that has not yet been standardised yet. Solutions for this challenge are on page 25.

Challenge 2.8 - Registration of the profile MOF model. Registration of MOF models resolves some problems introduced in point 6. We return to the example in Figure 5; the UML2 project does not register the MOF (*ecore*) model associated with the *Standard* profile. As a result, the construction of the MOF model for the new profile cannot reference the *Standard* MOF (*ecore*) model. Solutions to this challenge are on page 25.

4.2.1. Solutions in ERMA for the Deployment of Profiles and Model Libraries

ERMA solutions for challenges included in this section are:

Solutions for Challenge 2.1 - Profile repositories on the target modelling tool (see page 20). The UML2 project supports UML profiles in *Eclipse Modelling*. RMA interchanges models with XMI format, and the UML2 format is based on EMF. The main difference between XMI and XML EMF formats is the annotations. Import/Export of XMI files in Eclipse Modelling must convert the EMF annotations into XMI format.

Solutions for Challenge 2.2 - References to standard and installed profiles and to modelling libraries (see page 20). *nsURI* for the standard models (profiles and model libraries) should be the fundamental reference to locate standard models. The UML Profile includes the attribute *nsURI* which is used to identify instances of the profile in XMI, as the *Profile* meta-class in UML2 does not adjust using the standard. We can represent this attribute with the stereotype *Ecore::ePackage*, but it is used to identify the supporting *ecore* model and not the profile. In ERMA, we use this *nsURI* for both purposes (the same name identifies the *ecore* that supports the profile and the UML profile). There is no *nsURI* to locate the UML model library. The *ModelLibrary* stereotype in *Standard* library does not include this attribute. *UML PrimitiveTypes* is included as normative in the UML *Superstructure* standard and the universal name is *http://www.omg.org/spec/UML/20100901/PrimitiveTypes.xmi*, although the model library does not include the URI. ERMA reuses UML2 registrations and proposes some additional solutions (see *Solutions for Challenge 2.7* -).

Solutions for Challenge 2.3 - OCL constraints in profiles (see page 21). UML2 supports profiles with *ecore* models (see *Challenge 2.4* -); UML2 maps UML profiles into *ecore* models that are attached to *Profile* element; stereotype applications are dynamic instances of the *ecore* class that represent the stereotype. The UML2 run-time provides traceability between the UML profile and the generated *EPackage*, which also works with OCL constraints attached to profiles. But OCL, UML2 and EMF use non-standard notations to generate OCL validations[8] (UML2 models used as input for the generation of *ecore* models with OCL constraints must include some *Ecore* annotations). Figure 6 includes an example of UML Profile (*pp*), where this profile includes a stereotypes (*S*) that extends the *Class* meta-class. The stereotype *S* includes an OCL constraint to ensure that the name of the extended class starts with *S* (*self.base_Class.name.substring(0,0) = 'S'*). The deployment of this profile must have non-standard *Ecore* annotations attached that are included in the example (*invocationDelegates*, *validationDelegates*, *settingDelegates*). The UML2 *Define* operation on the profile will automatically create the *ecore* model that supports the profile and this operation will attach the *ecore* model to the profile. Generation includes the OCL operations for validation of constraints. The model validation will validate these constraints, but the *ecore* model must be registered (see *Challenge 2.8* -and *Solutions for Challenge 2.8* -).

Figure 6

[8] http://wiki.eclipse.org/MDT/OCLinEcore

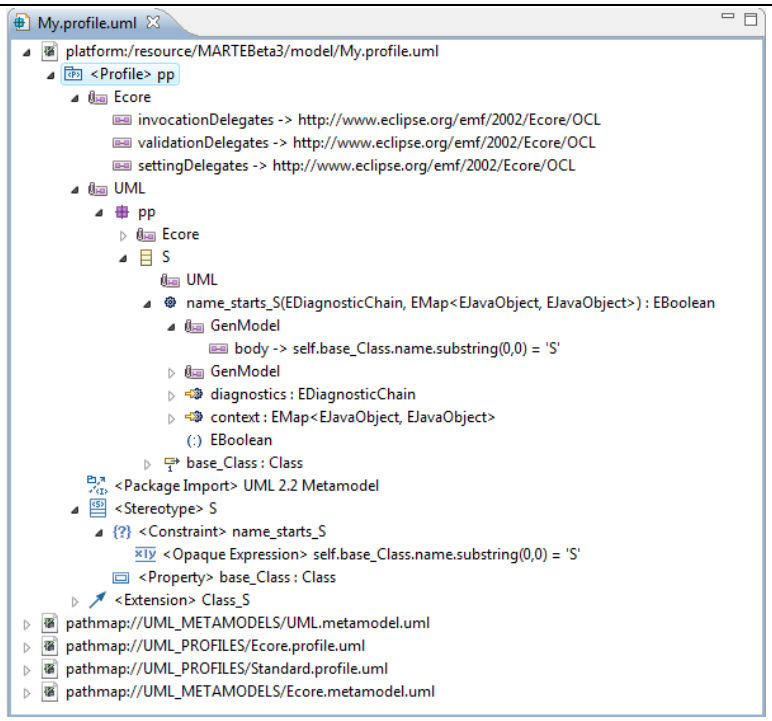

Figure 6. Problems because of the profiles representation in MOF and profile extensions

Solutions for Challenge 2.4 - Generation of MOF models for UML profiles (see page 21). As mentioned in the previous challenge, UML2 supports UML profiles with *ecore* models. Stereotype applications are dynamic instances to profile meta-models. *Ecore* models of profiles are not supported on Java generated run-time repositories; *DynamicEObject* is the Java interface that supports stereotype applications. ERMA deployment of UML profiles invokes the *define* operation for profiles in UML2. This operation generates the *ecore* model and attaches the *ecore* model to the UML profile as an *UML* annotation and the annotated profile is delivered. ERMA does not execute the *define* operation when the profile is annotated (this annotation is not standard), although some scenarios must *define* the profile during the editing of the asset. For example the MARTE standard profile includes profiles and a model library. Profiles use the model library and the model library applies some profiles. We must then define the profile during the editing of the model library.

Solutions for Challenge 2.5 - Application of profiles in model libraries (see page 21). The three main references from an XMI model to an applied profile are: i) identification of the *nsURI* that locates the XML schemas for the profile, ii) the *ProfileApplication* UML modelling element includes a reference to the profile and EMF annotations can reference the *ecore* profile model. These annotations must be included as XMI Extensions but this can create portability problems, iii) Stereotype applications are represented as XML data types that define the schemas associated with *ecore* models of profiles. These are not direct references like the others, however *ecore* models of profiles are needed to represent the elements. In ERMA, we assume that a UML profile does not have more than one associated *ecore* model so as to avoid inconsistencies between *ecore* and profile

models. This assumption avoids the problem of introducing a reference to the *ecore* model. ERMA proposes one solution for the registration of *Model Libraries* (*Solutions for Challenge 2.2* -). Models that apply this uniform name will avoid tool dependencies.

Solutions for Challenge 2.6 - Extension and references from profile to profile (see page 21). UML *Superstructure* proposes two solutions for handling profile extensions: an extension based on merging the extended profile; and extension based on inheritance. The UML2 profile modeller imports the extended profiles, and the extender introduces references and inheritances from the extended stereotypes (the extended profile should not be modified). But the profile *define* operation (the transformation from profile to *ecore*) has two model generation approaches; when the extended-profile *ecore* model is not registered, the generator merges extended and extender stereotypes and the new *ecore* Class is a merger of both. At run-time, extender applications are not specialisations of the extended stereotype. In the second generation approach generation of *ecore* models from profiles requires registration of extended profiles (with inheritance or reference), and in this case generated *ecore* models reuse the *ecore* models of extended profiles.

Solutions for Challenge 2.7 - Profile and model library registration (see page 22). UML2 includes the Eclipse extension-point *org.eclipse.uml2.uml.dynamic_package* but UML2 does not include registration mechanisms for model libraries. UML2 uses a URI *pathmap* to reference model libraries such us UML *PrimitiveTypes* (*pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml*), while other UML2 based tools use different names. *Eclipse* modelling tools that reuse UML2 have proprietary extension-points for both kinds of models (e.g. Papyrus and RSA reuse UML2 but define different extension points). ERMA registers model libraries with the *http://asset_id_LIBRARY/version/ModelLibraryName* URI, where *asset_id* is the symbolic name of the asset, and *ModelLibraryName* is the symbolic name of the *Model/Package* element, and it reuses UML2 for profile registration. These registrations include the symbolic name and the plug-in URI of the file, which include the profile or the model library. The UML standard proposes maintaining URI names for profiles: *http://<profileParentQualifiedName>/<version>/<profileName>.xmi* (these names should be used when editing *nsURI* of profiles). ERMA deployment generates the *pathmap://asset_id_PROFILES/* and *pathmap://asset_id_LIBRARY/* extensions of the EMF *org.eclipse.emf.ecore.uri_mapping* extension-point for URI mapping of profiles and model asset libraries.

Solutions for Challenge 2.8 - Registration of profile MOF model (see page 22). The *ecore* model of the UML profile must be registered for various purposes (evaluation of OCL constraints, profile extensions and reuse of profiles in transformation languages such as QVT and MOF2Text). The main problem with this registration is that the extension-point *org.eclipse.emf.ecore.generated_package* requires the URI that represent the symbolic name and the Java class that supports the interface *Descriptor* in *EPackage* (EMF includes the extension point *dynamic_package*, but some tools do not look for registered *EPackage* in that extension, probably in

the near future must of tools will take into account both kinds of registrations). UML profiles are dynamic and there is no Java code for their *ecore* model and, thus, a Java wrapper class is needed to implement the interface *EPackage.Descriptor*. ERMA run-time includes an abstract profile wrapper class that includes most of the functionality, and deployment generates a specialization of this Java class for each profile; the generated class includes the specific profile URI and the fragment for the location of the *EPackage* inside the profile. The generated wrappers and the ERMA run-time library support EMF registration for *ecore* models of profiles.

4.3 Deployment of Modelling Languages: EMF and GMF

EMF [28] is the most popular *Eclipse* implementation of EMOF (Essential MOF) [18]. EMF Core is a framework for constructing abstract syntaxes for modelling languages and for automatic generation of run-time support for handling and editing models. EMF uses two modelling languages to describe meta-models: the *ecore* language represents the meta-model and *EMF Codegen* (*genmodel*) supports the customisation of generated code (run-time support and editor). RMA assets reuse EMF for describing RMA artefacts that represent the abstract syntax of modelling languages. An RMA EMF artefact references the *ecore* and *genmodel* models that describe the modelling language embedded in the delivered asset, and RMA deployment reuses the target modelling tool mechanisms for the deployment of *ecore*/*genmodel* models. RMA uses EMF, and not EMOF, because the *genmodel* language is not part of the EMOF standard, and this language provides important facilities to deploy modelling languages. Some other important issues in EMF and GMF deployment are:

Challenge 3.1 - Visibility of EMF modelling languages. Challenge 1.3 - Visibility of MDA artefacts introduces general problems of artefacts visibility, but EMF artefact visibility poses some challenges. EMF generators are designed for constructing general modelling languages. The generated code publishes and registers the new languages and exports most of the generated Java packages. The privacy of these languages and their Java packages simplify the modelling tools and prevent software errors. An example scenario of *private ecore* models is RAS tools, which often implement the RAS profiles with *ecore* models while the manifest files are the XMI files of these *ecore*-based profiles. RAS repositories and packagers use *ecore* repositories for handling these models, but they are not used to create diagram and navigator editors, as RAS tools reuse these models internally. *Ecore* provides a simple method to construct XML schemas and handle the XML data supported by these schemas. These new types of XML files are another example of *private ecore* models. Solutions for this challenge are on page 27.

Challenge 3.2 - ecore inter-models dependencies. The *ecore* language supports the reuse of some other *ecore* languages (based on inheritance relationships and cross-references). The generators must resolve the dependencies between *ecore*/*genmodel* models. The *ecore* and *genmodel* models include cross-references to the packages and classifiers that are used. The deployment must resolve cross-

references between *ecore*/*genmodel* models, and to resolve these cross-references we must consider the implementation details of the *ecore* deployment to locate the target models (the location of the *ecore*/*genmodel* model in source asset models, in general, is different than the location in generated mechanisms). Cross dependencies include dependencies of standard *ecore* models such as UML, OCL, *ecore* and MOF2Text. Solutions for this challenge are on page 28.

Challenge 3.3 - Generation of genmodel and gmfgen models. In RMA, the reference from an EMF artefact to a *genmodel* model and from a GMF artefact to a *gmfgen* model is optional. If the artefact does not include this reference, the RMA deployment uses target tool mechanisms to automatically generate the *genmodel* or *gmfgen* models, but the *genmodel* and *gmfgen* models will have default values. This approach avoids some tool dependencies and any remaining *Eclipse* dependencies are because *ecore* is not a strict implementation of the EMOF language. However, migration to EMOF modelling frameworks would be simpler. The final task force of the DD standard [25] does not include an equivalent to *gmfgen* language. In the future, dependencies on *gmfgen* will create *Eclipse* dependencies when specifying concrete modelling language syntaxes. Solutions for this challenge are on page 28.

Challenge 3.4 - Generation of derived references and ecore operations. EMF generators produce most of the Java code for run-time support and navigator editors. Two exceptions are the implementation of *Operations* and some *derived* associations. EMF provides two solutions for these implementations: modifying the generated Java code and annotating the *operation* and the *derived structural feature* to generate implementation (in OCL or in Java). The second approach has important advantages, because it is generator-independent and executing generators is simpler. An additional problem to resolve is the dependency of Java classes included in Java code modification and generation. This problem was introduced in *Challenge 1.4 - Reuse of standard models and Software Behaviours*. Solutions for this challenge are on page 28.

Challenge 3.5 - Integration of Java code modifications. The annotations used for operation and derived references resolve the most common modifications of Java code generated in EMF. But some precise customisations require code modification. When the code in the source models is modified before packaging the asset, these modifications are lost during deployment and execution of generators. Solutions for this challenge are on page 28.

4.3.1. Solutions in ERMA for the Deployment of Modelling Languages

ERMA solutions for the challenges in this section are:

Solutions for Challenge 3.1 - Visibility of EMF modelling languages (see page 26). In ERMA, the deployment of *public* EMF descriptors executes the EMF generators and this will create three Eclipse plug-ins, which register the generated model and export the Java packages. We have studied two solutions for the deployment of *private* EMF artefacts: the execution of EMF generators modifies the plug-in id and reuses the asset plug-in; Java packages are not exported. The problem

with this approach is the plug-in properties of EMF editors (two EMF edit projects can include the same identifier in the plug-in properties); model code can only be generated for *private* EMF artefacts. The second solution is to modify the package visibility in exported packages and to modify the extension-point of generated EMF plug-ins.

Solutions for Challenge 3.2 - ecore inter-model dependencies (see page 26). The execution of generators for EMF must resolve the references to other EMF projects that support reused/extended languages. Two scenarios are considered to locate these models and their projects: the EMF project is deployed in the same workspace, but not installed (ERMA reuses the deployed project); the EMF models are deployed, installed and registered (ERMA reuses installed plug-ins). The other cases will cause deployment errors because assembly dependency cannot be guaranteed.

Solutions for Challenge 3.3 - Generation of genmodel and gmfgen models (see page 27). When EMF or GMF descriptors in *erma* models do not include a reference to *genmodel and gmfgen*, and no EMF or GMF descriptor in the same asset references the *genmodel and gmfgen* that support generation, ERMA automatically generates the generation model from *ecore* or from *gmfmap*. This is done to simplify the generation of EMF descriptors from EMOF models (or DD models in the future).

Solutions for Challenge 3.4 - Generation of derived references and ecore operations (see page 27). ERMA assumes that operation code and derived associations are integrated in *ecore* models. These implementations must be Java or OCL, and the *Ecore* annotations for OCL and Java. *Genmodel* models should include all customised code supported in this model. ERMA does not support the integration of JET templates in EMF descriptors.

Solutions for Challenge 3.5 - Integration of Java code modifications (see page 27). Current solutions to modify Java code generated for EMF and GMF descriptors are i) the *installation interaction points* implemented in Java artefacts (Java code executed during asset installation), and ii) modification of deployed Java code for EMF and GMF artefacts, before installation. Both are complex solutions and some tools could automate the first solution (by generating Java code that integrates the non-generated code and modifies the generated code in the target).

4.4 Deployment of Behaviour Specifications in QVT, MOF2Text and Java

In RMA we consider four languages to describe transformations or any kind of behaviour specification: OCL, Java, QVT and MOF2Text. Frequently, we can implement the same behaviour in three of these languages (e.g. we can implement the same query for a modelling language in OCL, Java or QVT). This section discusses the most important issues in deploying QVT, MOF2Text and Java. OCL expressions are attached to *ecore* models, UML profiles and *interaction points.* Challenge 1.1 - *Relocation of internal RMA references and zip and ras URI schemes* and Challenge 1.2 - *Relocation of external RMA references*

introduce the general problems with relocating references. We must take these issues into account for Challenge 4.1 - *References to ecore meta-models* and Challenge 4.3 - *Dependencies of Black-Boxes* in this section. The main challenges in deploying QVT, MOF2Text and Java software behaviours are:

Challenge 4.1 - References to ecore meta-models. QVT modules and transformations reference abstract syntax meta-models with *modeltype* sentences that identify the meta-models of the modelling languages to be used in the transformation. MOF2Text includes references to the meta-model as module arguments. In both cases, the reference is the universal name URI (*nsURI*) associated with the package that represents the modelling language. This *nsURI* must be registered to locate the *ecore* model that supports the language. This can create confusion when we need to update the *ecore* model (for example to introduce some additional operations or attributes) while the transformation is being built. Solutions for this challenge are on page 30.

Challenge 4.2 - Transformations for UML models with UML Profile applications. When we transform a UML model with profile applications, we need to reference stereotype applications. There are two solutions: i) to navigate to the stereotype applications using UML meta-model operations for handling stereotypes (most of these operations are included in the *Element* meta-class), and ii) to define the profile as a meta-model in the transformer, if the profile is run-time supported with MOF. The first solution avoids any tool dependency (the implementation of profiles with MOF is not mandatory in the OMG standard), and the second avoids many problems since UML operations use generic types (*EObject*) to resolve stereotype applications (reflexion and some additional tools are needed for handling these values). Solutions for this challenge are on page 30.

Challenge 4.3 - Dependencies of Black-Boxes. The concrete syntax of *Black-Boxes* in QVT and some MOF2Text implementations (this is not part of the OMG standard) can invoke Java class methods. These Java classes must be embedded in some target tool infrastructure (e.g. a Java library, Java project or plug-in), and QVT or MOF2Text run-time support of target tools must locate these Java classes. Solutions for this challenge are on page 31.

Challenge 4.4 - XMI vs Concrete Syntax Interoperability and Operational vs Relational. The MOF2Text and QVT standards propose two approaches for transformation interoperability: i) *syntax exportable models* are transformations in the concrete syntax that modelling tools use for interchange; ii) XMI exportable models are the XMI serialisations that make up the MOF meta-model (MOF2Text and QVT standards include the MOF meta-model). Both approaches are useful for different purposes and the RMA language supports both approaches. Transformation delivery and deployment should be in the concrete syntax to reuse modules and transformations in the target tool, for developing new transformations and modules. Solutions for this challenge are on page 32.

Challenge 4.5 - Reuse of Eclipse plug-ins in Java code. Software behaviour in Java often reuses standard Java libraries (e.g. for the construction of specific wizards). *Challenge 1.4 - Reuse of standard models and Software Behaviours* and its solutions introduce general approaches to reusing Java standard libraries. *Eclipse Modelling* [9] is currently used to construct several UML modelling tools

(e.g. IBM RSA[9], Magic Draw[10], Papyrus[11]) and modelling tools in general. *Eclipse* includes many reusable and extensible plug-ins. The reuse of plug-ins (e.g. general wizards and console plug-ins) would allow access to general *Eclipse* infrastructures, but their reuse limits the portability of assets to non Eclipse-based modelling tools. Solutions to this challenge are on page 32.

4.4.1. Solutions in ERMA for the Deployment of Behaviour Specifications

The ERMA solutions for the challenges in this section are:

Solutions for Challenge 4.1 - References to ecore meta-models (see page 29). *QVTo* and *Acceleo* in *Eclipse Modelling* need the *ecore* models that reference the transformations and modules to compile and execute. They dynamically locate the registered *nsURI* of the *EPackages*. They indirectly use the default *EPackage.Registration* implementation and the meta-model registry utilities in EMF. The problem is that the development of transformations would require the installation of EMF artefacts for compilation purposes. *QVTo* and *Acceleo* do not need the run-time projects of *ecore* languages, as only the *ecore* models are needed. Some additional GUI commands for *QVTo* and *Acceleo* editors can support dynamic registration of *EPackages*, and parallel development of transformations, *ecore* models (e.g. OCL implementations of *ecore* operations) and UML profiles are supported in the same workspace. *QVTo* and *Acceleo* use *nsURI* to locate the *ecore* models, which avoids dependencies arising from EMF deployment in the transformations but they are dependent on *ecore* registrations and *ecore* models. Because of this, deployment of EMF and Profile artefacts should support dynamic registration of *ecore* models to avoid compilations errors in *QVTo* and *Acceleo* deployments (this compilation is only needed to avoid installing transformations with compilation errors, although they are compiled again before execution and would run without errors if the *ecore EPackages* are registered). An alternative solution is to use the *helper operation* and *intermediate property* in QVT transformations, but these operations and properties are handled at the QVT module level and they are not integrated into the *ecore* model for all transformations.

Solutions for Challenge 4.2 - Transformations for UML models with UML Profile applications (see page 29). We can use *QVTo* and *Acceleo* in the transformation of UML models with profile applications. Two approaches are available: reuse *uml::Element* operations for stereotypes (e.g. *getStereotypeApplications*, *applyStereotype*), which are applicable to any UML element, or reuse the *ecore* profile model and handle with this stereotype application in the model. We use the same example for both approaches, a simplified example of *QVTo* transformation from UML+MARTE Profile to an analysis language (the *EPackage* of EMF language is *mast_mdl*). We only introduce the beginning of this transformation, as the remaining mapping can follow the same approach. Two

[9] http://en.wikipedia.org/wiki/IBM_Rational_Software_Architect

[10] http://en.wikipedia.org/wiki/MagicDraw_UML

[11] http://en.wikipedia.org/wiki/Papyrus_(software)

alternative sentences in the *main* operation stereotype are:

```
marte_model.rootObjects()[SaAnalysisContext]->map mapAnalysis();
marte_model.objects()[Package]->map mapAnalysis2();
```

SaAnalysisContext is a MARTE stereotype; the first sentence selects the application of this stereotype and executes the *mapAnalysis* mapping on this selection. The second sentence selects the UML *Package* elements and executes the *mapAnalysis2* mapping. The rest of the *QVTo* transformation is:

```
modeltype UML uses uml("http://www.eclipse.org/uml2/3.0.0/UML");
modeltype MARTE_SAM uses SAM("http://MARTE.MARTE_AnalysisModel/schemas/SAM/1");
modeltype MAST uses mast_mdl("http://mast.unican.es/xmlmast/mast_mdl");
transformation MARTE2MAST(in marte_model : UML, out mast_model : MAST);
main() {

}
mapping Package::mapAnalysis2() : MAST::MASTMODELType {
        if (self.getStereotypeApplications()->
                 select(oclIsKindOf(SaAnalysisContext))->size() > 0) then {
                modelName:=self.name;
        } endif;
}
mapping SaAnalysisContext::mapAnalysis() : MAST::MASTMODELType {
        if (not self.base_Package.oclIsUndefined()) then {
                modelName:=self.base_Package.name;
        } endif;
}
```

In this example *mapAnalysis2* (a transformation based on *UML::Element* operations) uses *SaAnalysisContext* to select *Packages* annotated with this stereotype. An alternative is not to include the profile model type in the transformation and to use *getAppliedSteretype("SAM::SaAnalysisContext")* operation call. *mapAnalysis* (a transformation based on the *ecore* model of the profile) navigates from the stereotype application to the annotated UML elements. The reuse of the *ecore* models of the profiles has important advantages (e.g. static checking of data types for stereotype navigations), but requires the registration of the *EPackage* of the profiles, and some mappings from the UML profile to *ecore* may not be intuitive (e.g. some properties of associations). The transformations are compiled and executed for the *ecore* model and not for the UML profile model (both should be consistent).

Solutions for Challenge 4.3 - Dependencies of Black-Boxes (see page 29). Black-Boxes are supported in Java jar libraries that support Java RMA artefacts. The deployment of QVT and MOF2Text models to *QVTo* and *Acceleo* projects includes the dependency of the asset plug-in that exports the Java jar files. The visibility of exported black-box packages in the plug-in asset is limited to *QVTo* and *Acceleo* projects. Currently, *QVTo* does not support black-boxes as the QVT standard proposes (in

QVTo, Java methods must be `static` and the execution context should be the first argument), which can create interoperability problems. The MOF2Text standard [21] does not provide standard specifications for black-boxes, but *Acceleo* provides a limited implementation of Java method invocation. An alternative solution to black-boxes would be to attach Java and OCL code to operations in *ecore* models and then invoke the operations from transformations, although this solution is not possible when we use standard languages such as UML and OCL.

Solutions for Challenge 4.4 - XMI vs Concrete Syntax Interoperability and Operational vs Relational (see page 29). In *Eclipse Modelling, QVTo* project supports *QVT Operational* and *QVTr* supports QVT Relations. The ERMA language supports references to QVT transformations and modules in abstract and concrete syntax, and it includes an implementation of the QVT standard in *ecore*. *QVTo* only includes an internal and proprietary version of the *ecore* model of QVT Operational, and there are not public classes for handling these models (e.g. a parser from the concrete syntax to QVT meta-model instances). Thus, it is not possible to create QVT artefacts that support QVT Operational transformations in XMI based on *QVTo* (a parser would be needed from QVT Operational concrete syntax to QVT Operational in *ecore*). *QVTr* supports QVT Relations and QVT Core in *ecore* and provides EMF editors for these models. *QVTr* is at an early stage, and ERMA requires a basic API to validate models and compile concrete syntax, to execute transformations and to analyse transformations and modules to identify specific properties of transformations and modules, such as the parameter types of transformations and the model types used. This means that integration of *QVTr* is also still at an early stage and *QVTo* supports this for *QVT Operational* for concrete syntax. Some functions that ERMA uses are in internal *QVTo* packages because the *QVTo* public API is very limited. For example, the public API in *QVTo* provides support for the execution of transformations[12], but not for the introspection of the data type of transformation parameters.

Solutions for Challenge 4.5 - Reuse of Eclipse plug-ins in Java code (see page 29). An ERMA asset can include *reference* dependencies to external Java artefacts. The Java artefact includes the identifier for the external jar file. ERMA deployment starts looking for the jar file identifier in Java *classpath* and in Standard Java libraries (e.g. *rt.jar* and *resources.jar*). If the Java jar file is not found, ERMA deployment looks for an installed *Eclipse* plug-in with that identifier and, if the plug-in is found the deployment creates the dependency for that plug-in. This solution creates dependencies for *Eclipse* solutions, but it could be useful for RMA assets designed for *Eclipse*-based tools that reuse basic plug-ins, such as *org.eclipse.ui.console*, to access *Eclipse* console.

[12] http://wiki.eclipse.org/QVTOML/Examples/InvokeInJava

4.5 Deployment of Interaction Points

MDA has multiple languages to describe associated software behaviours, such as transformations, queries, and other functionality. RMA software behaviour languages support functionality that assets provide to *application modeller*s and to some other assets. *Application modellers* can invoke this functionality, and some assets can request execution of software behaviours supported in other assets. As a result of these relationships, there are a large number of inter-dependencies between languages and it should be possible to invoke, from any language, functionality supported in the other languages. Figure 7 includes an example where the application model command *GlobalCommand* (RMA asset *Global*) is implemented in Java and it reuses the transformations *ModelX2ModelY* in QVTO (RMA asset *LanguageY*) and *ModelYToCodeZ* in MOF2Text (RMA asset *CodeZ*), while *ModelX2ModelY* reuses the OCL query *QueryInX* (RMA asset *LanguageX*). The specifications of the *integration points* in this diagram (*QueryInXIP*, *GlobalIP*, *X2YIP*, and *Y2ZIP*) define the behaviour artefacts that support the operations included in the specification of each *interaction point* and provide support for the implementation of *external commands* and *ERMA calls provided*. The *import* relationships define the link from the *ERMA call required* to *ERMA call provided*, and the *assembly* dependencies define the assets required for the deployment and installation of each asset.

Figure 7

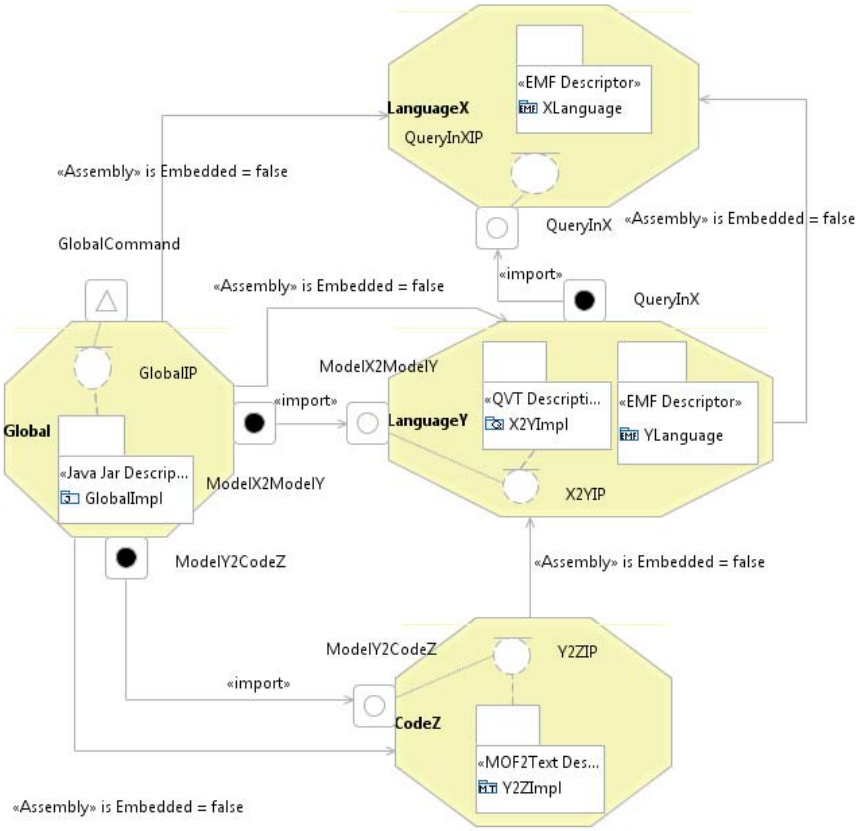

Figure 7. *Interaction Points* Example

The main objective of RMA *interaction points* is to reduce the complexity of inter-asset dependencies and to simplify the *application modeller* interface with software behaviour to unify its invocation. Section 3.3 introduces the concepts that support the definition of *interaction points*. The main challenges for the deployment and run-time support of interaction points are:

Challenge 5.1 - Links from RMA Interaction Points to MDA Languages. An *interaction point* specifies a port of interaction between assets or commands provided in the GUI of the application modelling tool. An *interaction point* specification is the same for all MDA implementation languages, but the *interaction points* require particular specifications to link their operations with the functional structures in each language. Solutions for this challenge are on page 34.

Challenge 5.2 - Inter-Language Invocation. The run-time support for *interaction points* consists of proxies for the functionality supported in the software behaviour specifications. For each language, the proxy uses special invocation methods to delegate the invocation. Solutions for this challenge are on page 35.

Challenge 5.3 - General GUI for Command Invocation. RMA run-time support provides a uniform solution to simplify the invocation of software behaviours for all MDA languages. This simplifies the usage of RMA assets and is the common reference for defining GUI commands that invoke software behaviours. Solutions for this challenge are on page 35.

Challenge 5.4 - Configuration of RMA Assets. Frequently, an RMA can include configuration parameters to customise the RMA asset in the concrete target tool. Examples of configuration parameters are parameters to represent input/output directories, usernames and logins to access data bases and repositories. Solutions for this challenge are on page 36.

4.5.1. Solutions in ERMA for the Deployment of Interaction Points

ERMA solutions for the challenges in this section are:

Solutions for Challenge 5.1 - Links from RMA Interaction Points to MDA Languages (see page 34). ERMA supports two types of implementation of *interaction point* operations: OCL and *software behaviour artefacts* (EMF, QVT, MOF2Text and Java). OCL constraints are integrated into the RMA language, since RMA extends *UML Infrastructure* and ERMA reuses OCL *Eclipse* project for compilation and execution of OCL constraints. RMA *interaction point implementation* relationships model the link from *interaction points* to *software behaviour artefacts*. The meta-class that supports relationships includes a key attribute that locates the operation implementation in the *software behaviour artefact*. For Java, the key represents the key binding to the Java methods; for QVTo and MOF2Text the key represents the qualified name for the transformation or module. One *interaction point* can have multiple associated *interaction point implementations*, because several *software behaviour artefacts* can support the same *interaction point*, but one operation must be associated with a single behaviour. The ERMA diagram editor includes a wizard for the edition of *interaction*

point implementation relationships, which navigates into Java jar files, QVTo transformations and modules and MOF2Text modules for selecting the methods, transformation and mappings that implement the operation. ERMA deployment reuses this information for code generation.

Solutions for Challenge 5.2 - Inter-Language Invocation (see page 34). The inter-language interaction of ERMA is based on two concepts: a run-time ERMA Java API that supports reflexion to locate and invoke *interaction points* and generate black-boxes in the deployment to be reused from MOF2Text and QVT. The generated black-boxes reuse the run-time API to invoke reused *interaction points* in Java. Figure 7 includes a Java artefact (*GlobalImpl*) that reuses *ERMA call provided* in other assets and supports the reused assets. In that example, one *ERMA call provided* (*ModelX2ModelY*) is supported in QVT and the other in MOF2Text (*ModelY2CodeZ*), but from the Java artefact the implementation language is opaque. The Java implementation would be based on the ERMA run-time reflexion API, and this implementation is (both *interaction points* are invoked with the same calls):

```
IContainer myContainer=null;
try {
	myContainer = IContainerLocator.eINSTANCE.getContainerOfModellingAsset("Global");
	IImportedERMACall x2yCall = myContainer.getImportedERMACall("ModelX2ModelY");
	IOperationClient x2yTransformation = x2yCall.getOperationClient("x2y");
	IImportedERMACall y2zCall = myContainer.getImportedERMACall("ModelY2CodeZ");
	IOperationClient y2zGeneration = y2zCall.getOperationClient("y2z");

	List<EObject> in=new ArrayList<EObject>();
	in.add(model);
	Object[] result=x2yTransformation.invoke(null,in);

	in=new ArrayList<EObject>();
	in.addAll((List<EObject>) result[0]);
	result=y2zGeneration.invoke(null,in);
	// result[0] contains the result y2zGeneration
} catch (ERMAContainerException e) {
	// ERMA exception handler
}
```

`IContainer` provides run-time support for the asset, and includes basic operations for introspection of *ERMA Calls* included in the asset. Other interfaces include operation to invoke *interaction point* operations. The generated ERMA run-time and code resolve execution depending on the *software behaviour language* that supports the operation. `IContainer` provides methods to get/set values of assets configuration parameters, and to access containers of assembled assets.

Solutions for Challenge 5.3 - General GUI for Command Invocation (see page 34). RMA assets can include *External Command* specifications. ERMA deployment generates Java classes for handling the commands that delegate the operation based on the ERMA run-time API. The generated code extends *Eclipse* extension-point *org.eclipse.ui.actionSets* and creates new entries in the *Eclipse Run*

Menu, creating one submenu for each asset. The submenu includes entries for executing commands, configuring asset parameters and downloading documentation artefacts. The Java code for commands introspects the operation that implement the command, requests values for parameters taking into account the modelling elements available in the active resource set, invokes the implementation of the operation and displays the results.

Solutions for Challenge 5.4 - Configuration of RMA Assets (see page 34). RMA assts can have associated configuration parameters. The *application modeller* can get and set the parameters with menu entries, and *software behaviour artefacts* can get and set their values with ERMA run-time interfaces.

4.6 Deployment of Documentation Artefacts

MDSD provides innovative solutions to increase the abstraction level in software development, but MDA based tools must consider the learning process of the infrastructure supporting the MDSD process. Especially important are the DSLs. To make a new RMA asset applicable, we should consider the percentage of time required to learn a new DSL or UML profile in the application project. For example, expert UML modellers may spend several months learning a new UML profile such as MARTE [23]. If engineers should be spending no more than one year modelling the project, the learning effort is too expensive for project development. To make the RMA asset applicable, the *RMA developer* must provide support to reduce the time for learning new languages and applying new asset commands and transformations. We must take the learning curve into account when designing a new modelling language and asset. Learning support should be integrated in the RMA asset and should reduce learning time:

Challenge 6.1 - Packaging and deployment of learning artefacts. Section 3.1 introduces artefacts for representing RMA learning documents. Two different approaches for the deployment of learning documents are: i) to maintain a single documentation server to be used from target tools; ii) to deploy the documentation to a target tool documentation server. The advantage of the first solution is centralised documentation maintenance while the advantage of the second is the independence of the target modelling environment. Solutions for this challenge are on page 36.

4.6.1. Solutions in ERMA for the Deployment of Documentation Artefacts

ERMA solutions for the challenges in this section are:

Solutions for Challenge 6.1 - Packaging and deployment of learning artefacts (see page 36). The use of *MediaWiki* [3] XML files is the most common approach to support documentation artefacts in ERMA. Other types of document are supported as RAS artefact, but ERMA deployment only deploys them as files into an artefact folder. In *erma* models, *MediaWiki* artefacts are hierarchies of artefacts that can include a URI reference to a *MediaWiki* page. ERMA considers two alternative

solutions for packaging *MediaWiki* artefacts: the artefacts can be embedded in the asset, or the artefact can be non-embedded. In the first case, the *MediaWiki* pages are packaged in the *ras* file, and the *ras* file includes pictures that reference the *MediaWiki* page. If other referenced *MediaWiki* pages are not embedded, the pages will maintain references to the source pages. The ERMA deployment delivers the *MediaWiki* pages to a *MediaWiki* server that can be on the same machine as the modelling target tool or on a different one (ERMA supports the configuration of a target *MediaWiki* server). If *MediaWiki* pages are not embedded, documentation menus and other code will reference the source pages. The second solution is better for sharing documentation pages and keeping all documentation updated, but it relies on Internet access. ERMA documentation commands extend the *org.eclipse.ui.actionSets Eclipse* extension-point and reuse web browsers included in the *org.eclipse.ui.browser Eclipse* plug-in to display *MediaWiki* pages. *MediaWiki* should be used to edit and maintain documentation pages, which should be independent of *MediaWiki* extensions. Packaging and deployment ERMA services are based on *import*/*export* functions in *MediaWiki*; *import* requires *sysop* privileges and ERMA *MediaWiki* configurations can configure user and password to be used for *MediaWiki* artefact deployments.

5 Future Research Directions

ERMA provides a framework to interchange modelling tools assets. A basic tool for asset interchange is RAS repositories. These repositories could be specialised for specific technologies and domains. The *ERMA RAS working group repository*[13] includes assets for the development of high integrity applications, including assets for MARTE profiles (MARTE profile modelling tools, MAST analysis tools, Ada2005 Ravenscar and Java RTSJ profile code generators), and some additional assets that support safety-aware model development: Safety & Dependability UML extensions, (FTA) Fault Tree and (FMECA) Failure mode, effects, and criticality analysis modelling languages, transformations from UML+Safety & Dependability to FTA and FMECA, and bridges from ERMA safety to Item Toolkit) [8]. OMG provides XMI models for some standards such as meta-models and profiles. Currently integration of these models into modelling tools requires experience in XMI and standard formats. Repositories that reduce customisation of these models are needed, because at present tool vendors customise these models for their tools, but do not provide open solutions, and clients depend on vendor supplied tools.

RMA supports run-time inter-asset dependency based on the *ERMA Call Provided – ERMA Call Required* relationship. Current implementations assume the execution of client and server assets in the same *Eclipse* tool. Elimination of this restriction has important advantages: to use transformation assets as remote services, to support working group modelling repositories, to learn and test assets based on remote services not

[13] http://138.4.11.45:9080/ras_erma/services/WorkGroupRepositoryServer

installed in the working tool. The main problem in eliminating this restriction and executing client and server in different tools and on different machines would be the serialisation of parameters, context and results, and the references that the parameters can include. EMF and XMI provide serialization support for modelling objects, but the limitations are the references that these objects can include. Some alternative solutions could be based on CDO (Connected Data Objects) EMF subproject, which is a distributed shared model of EMF. In CDO, multiple modelling tools share common repositories supported in distributed data bases, and CDO run-time generated code provides transparent access to the distributed data base. CDO supports the distributed notification of model access and modifications and transactions for remote models, but there is no CDO (and Net4J integrated in CDO) solution to the problem of remote transformation execution, which would integrate CDO and transformation languages. CDO solutions are not integrated in OMG standards, and it is not currently possible for different tool vendor modelling tools to share a common modelling repository. *ModelBus* [2] was another framework for the remote execution of modelling services. *ModelBus* is open source, but its problem is also standardisation.

Our current implementations of *ERMA RAS Working Group server* are based on a web server developed in *IBM WebSphere*. This web server implements a WSDL (Web Services Description Language) interface. Currently there is no RAS open source implementation for *Eclipse*. A run-time platform for the execution of RAS repositories could be *Eclipse WTP* (Web Tools Platform)[14].

6 Conclusions

Currently, complex software development based on model driven approaches are moving platform dependencies from run-time platforms and programming languages to model driven development frameworks. In the near future, similar efforts will be required to migrate current projects to new development tools. If model-driven software developments do not work on reducing the dependencies on model-driven development environments, the maintenance of these projects will require considerable future effort. RMA and ERMA represent a practical solution to avoiding modelling tools' dependencies on MDSD. The modelling tools' and MDA development environment dependencies create problems as complex as the problems of platform dependencies addressed in MDA technologies and bibliography (e.g. PIM (Platform Independent Model) to PSM (Platform Specific Model) development approaches [10]). MDSD should be as development environment independent as possible to reduce maintenance and development costs. However development independence require some efforts and limitations for development tools (e.g. we should avoid using proprietary development tools not based on standards, and reusing open-source and portable development tools reduces development environment dependencies).

[14] http://www.eclipse.org/webtools/

MDSD reuses basic modelling tools such as generators, transformations and modelling languages. But these basic tools are developed in multiple languages, and combining and interoperability are complex. Learning these languages and their combined application requires extensive learning for the engineers who apply these tools. A fundamental objective of RMA is to reduce this learning period. These solutions are based on two fundamental concepts: to support the interoperability of MDA languages, and to integrate all kinds of MDA artefacts into a common framework. ERMA proposes a solution to integrate the different MDA frameworks included in *Eclipse Modelling* and to simplify their interoperability.

The interoperability of MDA models requires some improvements to standards (OMG standards in particular). UML model libraries do not include notations to represent *nsURI* in the model (e.g. *MARTE Model Library nsURI* should be included in the model), as it can only be introduced in the standard document (the MARTE standard does not include the *nsURI* of *MARTE Model Library*). The UML *Profile* meta-class includes one *nsURI* to locate the model but, in general cases, two different *nsURI* are needed: the *nsURI* for the location of the profile model (the URI used in *ProfileApplication* relationship, where the target is a *Profile* modelling element, or the URI to be used in *PackageImport* in new profiles that extend other profiles), and the *nsURI* that locates the MOF model that supports the XMI for the profile (the URI that includes the XMI file for location of XML schemas and the URI to be used in QVT and MOF2Text transformations to reference stereotype applications). If the MOF model is embedded in the UML profile model (currently, this is not a standard solution) and there is only one MOF model attached to the profile, a single *nsURI* could be enough. But, in general cases, two different models (profile and MOF model) with two different registrations and a single universal name will require multiple registrations, and the same name will target different models in different registrations. UML profile standards do not generally include the specification of the *nsURI* for the profile (this attribute has been included in the *Profile* meta-class in recent versions of UML2) and the references to standard profiles are tool-dependent.

The learning curve is a fundamental concept in the application of MDSD technologies. For software engineers, to migrate from traditional development methods to MDA methods will require more effort than the migration to object-oriented languages and technologies some years ago. The migration from traditional development methods to MDA methods; the application of new modelling languages and transformations requires important learning effort and engineers and companies must invest time and energy. If the assets that support the new methods do not integrate education or tutorial support, it does not make sense financially to develop them. RMA includes some support for the integration of learning documents, but the development of these artefacts requires following some educational methods.

RMA design takes several steps to improve tool independency: RMA is based on MDA standards that reduce tool dependencies and software behaviour specifications in particular; ERMA implementation is based on *Eclipse* open-source projects, which simplify the interoperability of *Eclipse*-based modelling tools (ERMA

adaptation to *Eclipse*-based modelling tools does not require too much effort, and most efforts will involve integrating the required projects such as *Acceleo*, *QVTo* and *OCL*); RMA defines a framework to customise a modelling tool for a specific technology and domain, so that the same basic modelling assets could customise several modelling tools, where all these tools would provide similar modelling environments.

References

1. L. Ackerman, P. Elder, C. Busch, A. Lopez-Mancisidor, J. Kimura, R. S. Balaji (2008), *Strategic Reuse with Asset-Based Development*. IBM RedBooks.
2. A. Aldazabal, T. Baily, F. Nanclares, A. Sadovykh, C. Hein, M. Esser, T. Ritter (2008), Automated Model Driven Development Processes. In T. Ritter (Ed.) *Proceedings of the ECMDA workshop on Model Driven Tool and Process Integration*, Fraunhofer IRB Verlag, Stuttgart.
3. D. J. Barrett (2008), *MediaWiki: Wikipedia and Beyond*, O'Riley.
4. R. Bendraou, P. Desfray , M.P. Gervais and A. Muller (2008), MDA Tool Components: A Proposal for Packaging Know-how in Model Driven Development. *Journal on Software & System Modeling.* Springer, 7(3), 329-343
5. R. Bendraou, P. Desfray, M.P. Gervais (2005), MDA Components: A Flexible Way for Implementing the MDA Approach. In (Ed), *Model driven architecture: foundations and applications. First European conference, ECMDA-FA* 2005, Nuremberg, Germany, November 7-10.
6. X. Blanc, M.P. Gervais, P. Sriplakich (2005), Model Bus: Towards the Interoperability of Modelling Tools. In A. Hartman (Ed.), *Model Driven Architecture*, Lecture Notes in Computer Science, Volume 3599
7. M. A. de Miguel, J. Jourdan, and S. Salicki (2002), Practical Experiences in the Application of MDA, In J. Jezequel (Ed.), *Proceedings of Fifth International Conference on The Unified Modeling Language: UML 2002*, Springer Verlag
8. M. A. de Miguel, J. Fernández Briones, J. P. Silva, A. Alonso (2008), Integration of Safety Analysis in Model-Driven Software Development, *IET Software*, 2(3), 260 – 280.
9. Eclipse Foundation (2011). *Eclipse Modeling Project*, from http://www.eclipse.org/modeling/
10. D. Frankel (2003), *Model Driven Architecture*. Wiley & Sons
11. R. Goodwin; A. Ivan; S. Goh; R. Mohan; B. Srivastava; P. Mazzoleni; I. Naumov; R. Chopra; T. Bandyopadhyay; T. Rosinski, (2009) Improving SAP Projects with Model-Driven Technologies for Global Delivery, *IBM Research, Technical Paper (RC24879).*
12. V. K. Gurbani, A. Garvert, and J. D. Herbsleb (2010) Managing a Corporate Open Source Software Asset. ACM Communications, 52(2), 155-159
13. G. Larsen (2006). Model-driven development: Assets and reuse. *IBM Systems Journal*, 45(3), 541-553
14. Object Management Group, (February 2009). *OMG Unified Modeling Language, Superstructure, Version 2.2*, OMG document number formal/2009-02-02
15. Object Management Group, (November 2010). *OMG Unified Modeling Language, Superstructure, Version 2.4, Revision Task Force*, OMG document number ptc/2010-11-13
16. Object Management Group, (February 2009). *OMG Unified Modeling Language Infrastructure, Version 2.2*, OMG document number formal/2009-02-04
17. Object Management Group, (February 2010). *Object Constraint Language, Version 2.2*, OMG document number formal/2010-02-01
18. Object Management Group, (January 2006). *Meta Object Facility (MOF) Core Specification, Version 2.0*, OMG document number formal/06-01-01
19. Object Management Group, (January 2011). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, OMG document number formal/2011-01-01

20. Object Management Group, (November 2005) *OMG Reusable Asset Specification, Version 2.2,* OMG document number formal/05-11-02
21. Object Management Group, MOF Model to Text Transformation Language, Version 1.0, OMG document number formal/2008-01-16
22. Object Management Group, (January 2008). *MOF MOF 2.0/XMI Mapping, Version 2.1.1,* OMG document number formal/2007-12-01 (December 2007).
23. Object Management Group, (November 2009). *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.0,* OMG document number formal/2009-11-02
24. Object Management Group, (November 2009). *MDA Tool Component, Version 1.2, Reviewed Submission,* OMG document number ad/2009-11-03
25. Object Management Group, (December 2010). *Diagram Definition, Version 1.0, Final Task Force,* OMG document number ptc/2010-12-18
26. OSGi Service Platform (2009), *Core Specification, Release 4, Version 4.2,* OSGi Alliance.
27. S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, P. McCarthy (2003). *The Java Developer's Guide to Eclipse,* Addison Wesley.
28. D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. (2009). EMF*: Eclipse Modeling Framework (2nd Edition),* Addison Wesley.

APPENDIX A. ERMA Examples: ERMA Assets for MARTE Profile Application

UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (MARTE) [23] is an OMG standard for modelling real-time systems in UML 2.x. MARTE extensions enhance UML notations for the representation of real-time specific concepts (e.g. more precise specifications of time, hardware and real-time design patterns). MARTE was designed assuming it would be integrated with analysis tools (performance and scheduling analysis for some specific) and run-time platforms. This appendix introduces five ERMA assets for the application of MARTE in MDSD tools. These assets use MARTE for two main purposes: generation of scheduling analysis models for UML MARTE models and code generation for Ada 2005 and Java. Some additional assets could support performance analysis and code generations for other real-time programming languages.

Some real-time organizations have designed profiles of platforms and programming languages for use in development of real-time applications; Ada 2005 Ravenscar profile [1] and Java RTSJ [2] are two examples. These profiles define libraries and run-time environments to make Ada 2005 and Java applications time predictable. These two profiles assume the application of scheduling analysis methods to make the applications time-predictable. If we use these two platforms form the execution of MARTE applications, we must take into account their specific properties and the specific details of their code generators to make time-consistent scheduling analysis results and run times. Consistency of scheduling analysis models and code generation are fundamental problems to be addressed in the assets that we introduce in this section.

This section introduces five ERMA assets examples (in Figure 8) for the support and application of MARTE:

- *MARTEBeta3*: this is an asset for the integration of MARTE into UML modelling tools. This asset includes MARTE profiles, a model library and tutorials for their application. Some additional artefacts in this asset could be used to support the VSL (Value Specification Language) and other expressions handled in MARTE.
- *MAST_RMA*: this asset integrates MAST (Modeling and Analysis Suite for Real Time Applications)[15] into general modelling tools. MAST is an open source set of tools that enables the modelling of real-time applications and the performing of timing analysis on those applications. *MAST_RMA* supports the languages in *ecore* meta-models and includes artefacts for the execution of analysis. *MAST_RMA* allows MDA languages to be used for generating and editing MAST models and for constructing additional tools, such as diagrams editors. *MAST_RMA* is independent of MARTE; MAST_RMA is an example of an asset for the support of a DSL (MAST).

[15] http://mast.unican.es/

- *UML2AdaRavenscar*, *UML2RTSJ*: these two assets generate Java and Ada code for UML models that use RTSJ and Ada Ravenscar libraries and run-time environments. They are adaptations of Java and Ada code generators in MOF2Text, which take into account the specific limitations of these programming language profiles. These assets are independent of *MAST_RMA* and *MARTE*, and they could be used as we use some others code generators, but they are adapted for Ada Ravenscar and RTSJ profiles.
- *MARTE_SA_Gen*: this asset integrates transformations of MARTE models into MAST models and UML models that use RTSJ and Ada Ravenscar libraries and run-times. The transformation generates RTSJ or Ada Ravenscar patterns for MARTE modelling structures, and generates a MAST model taking into account MARTE::SAM profile annotations (SAM is a MARTE sub profile for the representation of scheduling analysis in UML 2.x).

Figure 8

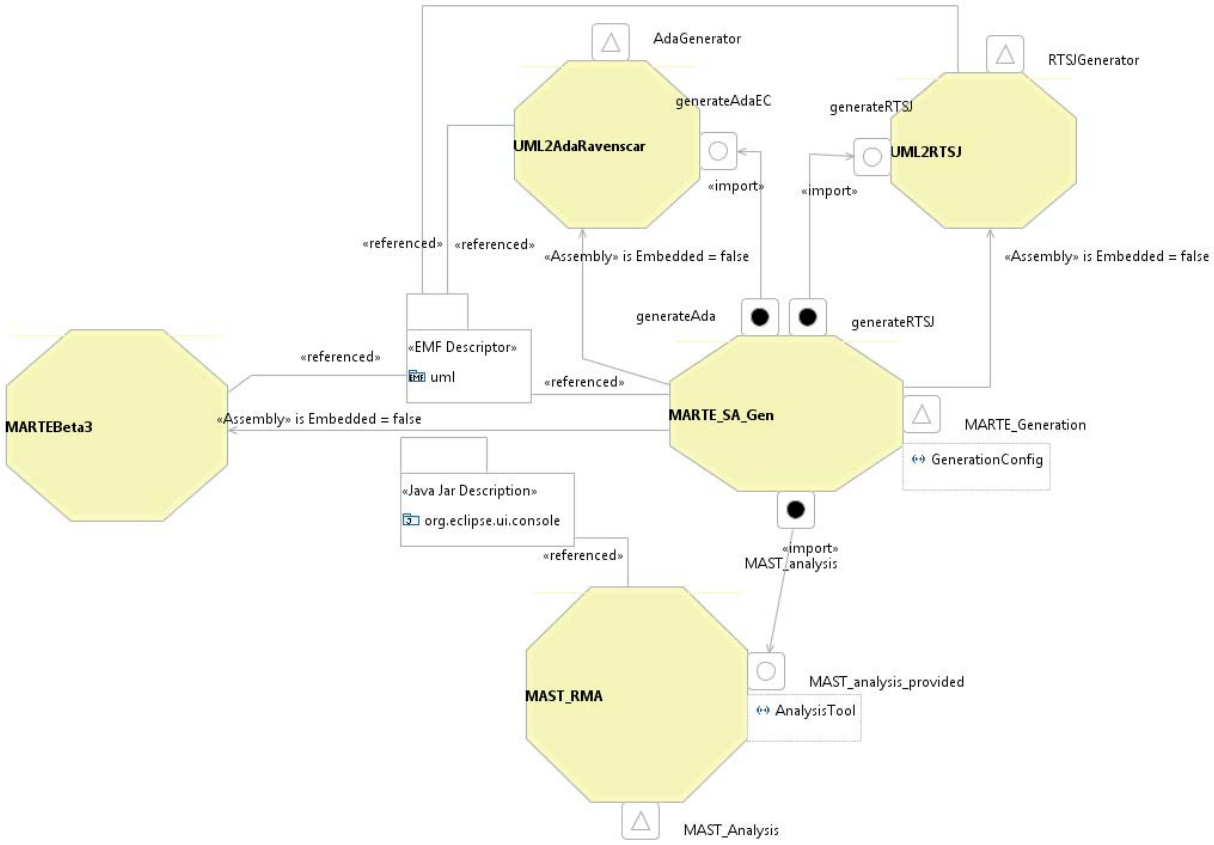

Figure 8. *General diagram of MARTE assets*

Figure 8 includes five assets and their dependencies. *MARTEBeta3* supports the MARTE standard and only depends on the UML meta-model. *MAST_RMA* is the implementation of the MAST language in *Eclipse Modelling* and reuses *org.eclipse.ui.console* Eclipse plug-in for the generation of log and error messages. *MAST_RMA* includes a configuration parameter for the selection of analysis algorithm. It exports a service for the invocation of analysis from other assets, and a GUI command for the execution of analysis from modelling tool menus. *UML2AdaRavenscar* and *UML2RTSJ* depend on a UML meta-model and they provide

a service for the execution of generators from other assets and from GUI commands. *MARTE_SA_Gen* depends on the other four assets (it assemblies the other assets, but they are not embedded because the others could be installed without *MARTE_SA_Gen*), and reuses the services provided in the others assets for the combined execution of analysis and code generators. Figure 9 includes a detailed ERMA diagram which includes the artefacts and interaction points for each asset.

The Beta3 version of the MARTE standard and the new version for MARTE 1.1 include two XMI models[16]: *MARTE UML profile* and *MARTE model library*. MARTE profiles use the model library and the model library applies some sub-profiles. The profile model and model library are in XMI format. The *ecore* models are needed to represent the stereotype applications in the model library XMI file. These interdependencies create important problems in the XMI delivery of MARTE models. *The MARTEBeta3* asset includes the MARTE profile and the model library (the artefacts are described in Figure 9), but in this scenario we cannot delegate to ERMA the generation of MOF models for profiles, because they are needed for editing the model library, and the model library is needed for editing profiles. Similar problems occur in the *HwCommunication/HwGeneral* and *HwStorageManager/HwComputing* sub-profiles because they depend on each other (they have cross references in stereotypes in both directions) and MOF model generation requires special attention and specific solutions (*define* operations for profiles in UML2 project do not support the MOF models generations for these profiles). An alternative solution would be to merge inter-dependent profiles into a single profile, but our solutions do not modify the standards.

MARTE does not include nsURI (universal names) for the identification of profiles. We included these names in profiles with the format: http://MARTE.<MARTE Packages> /schemas/<Profile Name>/1. *<MARTE Packages>* is the namespace for the profile (e.g. *MARTE.MARTE_DesignModel.HRM.HwLogical*). *<Profile Name>* is the name of the profile. Number 1 is the version number for the profile.

MOF2Text transformations *AdaGenerator* and *RTSJGenerator* reference several MOF2Text modules and a main module (each one is described in a single MOF2Text descriptor in Figure 9). There is a module for each design pattern and some additional modules for general Ada and Java programming structures [3]. The most important patterns are: *periodic*, *sporadic*, *simple*, *protected objects* and *passive*. They use general templates that support generation of exceptions handlers, operations and methods and guards and synchronisers. The MOF2Text modules implement the patterns in Java or Ada (e.g. instance of generics and generic classes, exception handlers and timing error handler, general execution cycles).

[16] http://www.omg.org/spec/MARTE/20090501, http://www.omg.org/spec/MARTE/20090502

The *MARTE_SA_Gen* asset includes a QVTo transformation: *SA2MAST_Ravenscar_RTSJ* (Figure 9 includes the QVT artefact in the *MARTE_SA_Gen* asset). This input model for this transformation is a UML+MARTE model and it generates two output models: one MAST model and one UML model. The configuration of this asset defines what kind of UML is generated (Ada Ravenscar or RTSJ) and the generated UML models import the model library included in the *UML2AdaRavenscar* or *UML2RTSJ* assets. The output of this transformation is based on real-time design patterns (e.g. *sporadic server*, and *protected object*) and these patterns are represented with UML classes that use Ada Ravenscar libraries and run-time, and UML classes that use RTSJ libraries, and their equivalent in MAST elements. Each mapping in the transformation is applied in the same MARTE element and the mapping provides consistent outputs.

Figure 9

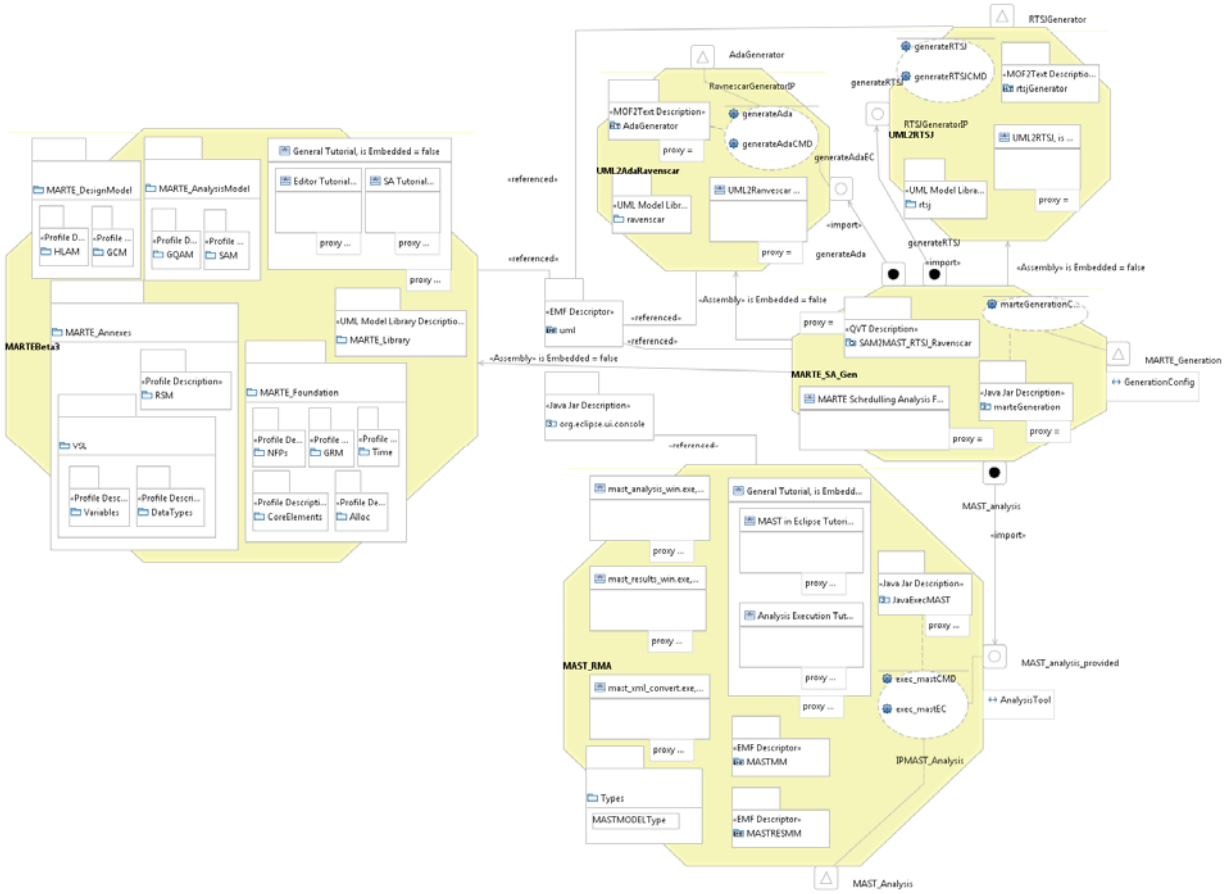

Figure 9. *Detailed diagram of MARTE assets*

References

1. International Organization for Standardization (June 2010). *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. ISO document number: ISO/IEC TR 24718:2005
2. Java Community Process (May 2009). *JSR-000282 Real-Time Specification for Java 1.1*. Java Specification document number: JSR 282: RTSJ version 1.1.

3. A. Burns,d A. Wellings (April 2009). *Real-Time Systems and Programming Languages (Fourth Edition) Ada 2005, Real-Time Java and C/Real-Time POSIX*. Addison Wesley Longmain.