Softw Syst Model (2008) 7:329–343
DOI 10.1007/s10270-007-0058-8

REGULAR PAPER

MDA Tool Components: a proposal for packaging know-how in model driven development

Reda Bendraou · Philippe Desfray · Marie-Pierre Gervais · Alexis Muller

Received: 21 April 2006 / Revised: 10 March 2007 / Accepted: 12 April 2007 / Published online: 23 May 2007

Abstract As the Model Driven Development (MDD) and Product Line Engineering (PLE) appear as major trends for reducing software development complexity and costs, an important missing stone becomes more visible: there is no standard and reusable assets for packaging the know-how and artifacts required when applying these approaches. To overcome this limit, we introduce in this paper the notion of MDA Tool Component, i.e., a packaging unit for encapsulating business know-how and required resources in order to support specific modeling activities on a certain kind of model. The aim of this work is to provide a standard way for representing this know-how packaging unit. This is done by introducing a two-layer MOF-compliant metamodel. Whilst the first layer focuses on the definition of the structure and contents of the MDA Tool Component, the second layer introduces a language independent way for describing its behavior. An OMG RFP (Request For Proposal) has been issued in order to standardize this approach.

Keywords MDD · Packaging of know-how · MDA Tool Component · Reusability

Communicated by Prof. Miguel de Miguel.

This work is supported in part by the IST European project "MODELWARE" (contract no 511731) and extends the work presented in the paper entitled *"MDA Components: A Flexible Way for Implementing the MDA Approach"* edited in proceedings of the ECMDA-FA'05 conference.

R. Bendraou (✉) · M.-P. Gervais · A. Muller
Laboratoire d'Informatique de Paris 6,
8 rue du Capitaine Scott, 75015 Paris, France
e-mail: Reda.Bendraou@lip6.fr

A. Muller
e-mail: Alexis.Muller@lip6.fr

P. Desfray
Softeam, 144 Av. des Champs Elysées, 75008 Paris, France
e-mail: Philippe.Desfray@softeam.fr

M.-P. Gervais
Université Paris X, 144 Av. des Champs Elysées,
75008 Paris, France
e-mail: Marie-Pierre.Gervais@lip6.fr

1 Introduction

In software industry, growing expectations for reliable software in a short time to market makes development processes increasingly complex. The continuing evolution of technologies and the need for sophisticated information systems will not improve this situation. Then, it becomes more and more difficult for companies to respect deadlines and to provide software with the expected functionalities. According to the Standish Group CHAOS study report, in 2004, when 29% of projects succeeded (time, budget and required functions), among 53% are challenged (late, over budget and/or with less than the required features and functions); and 18% have failed (cancelled prior to completion or delivered and never used) [19]. Moreover, if we look closer at the development processes, we can notice that a convergence exists in the practice of software production. Most of the time, developers and designers apply the same steps, handle similar tools and apply the identical tests. Unfortunately, this know-how is generally scattered and is not capitalized. To face these challenges, model based approaches have emerged. Nowadays, models can be explicitly manipulated through meta-modeling techniques, dedicated tools and processes or model transformation chains. This set of activities is known as: "Model Driven Development".

The main motivation behind Model Driven Development (MDD) is the reduction of delays and costs by the capitalization of design efforts (models) at each stage, and the automation, as far as possible, of transitions between these stages. Then, it would be possible to separate high level business oriented models from low level architectural and technological ones and to reuse them from one application to another. A primary example of MDD is the OMG's (Object Management Group) Model Driven Architecture (MDA) initiative [10]. The MDA advocates the distinction between models designed independently of any technical considerations of the underlying platform i.e. PIM (Platform Independent Model) and models that include such considerations i.e. PSM (Platform Specific Model). In order to facilitate a model driven approach for software development, a growing family of standards for representing variety of domain specific models emerged. Examples of such standards are UML (Unified Modeling language) [20,21], MOF (Meta Object Facility) [12], SPEM (Software Process Engineering Metamodel) [17], EDOC (Enterprise Distributed Object Computing) [5], etc. Besides, panoply of tools claiming compliance with MDA is shown on the OMG's website [9]. Nevertheless, the expert's know-how, domain specific features i.e. helpers, configuration parameters, libraries etc, and customized actions and knowledge applied on these models are provided neither by MDA standards nor by existing MDA-compliant tools. Indeed, model driven approaches enable one to capitalize business or technical models but it does not tell anything about how to define these models or how to build systems with these models. It becomes obvious that currently, MDD lacks of reusable and standard entities that can capitalize and encapsulate the knowledge and required resources when promoting these approaches, a key condition for a more cost-effective software production.

To overcome this limit, we introduce in this paper the notion of MDA Tool Component (MDATC), i.e., a packaging unit which encapsulates business know-how and required resources in order to support specific modeling activities on a certain kind of models. One objective of MDATCs is to provide industrial enterprises with a tool and platform independent formalism for representing their know-how. Thus, by using MDATC as means to describe know how, software processes, design methodologies and modeling activities can then be represented and exchanged in a standard format. Another objective of MDATCs is that tools can be customized according to the modeling activities supported by the MDATC. They will be able to support new functionalities, access design artifacts and all the required material simply by integrating MDATCs to their environment. Thus, tool-vendors would have to integrate one MDATC related to every new context or domain rather than redesigning their tools in order to take this context into account. Indeed, in order to support modeling features needed for a specific domain, tool venders customize their tools. Nevertheless, most often this is done in a proprietary way and this know-how can not be exported/shared with other tools. This lead them to adapt their tools every time a new profile, a new standard, operation on models or a new platform appears. This cannot be a long term solution. MDATCs are autonomous, platform-independent and promote MDA standards. MDATCs are not business components; they are not assembled to build a system. They aim to be used by tool providers and method engineers to customize existing tools for a specific domain. An RFP has been issued at the OMG in order to promote this approach and five companies have already manifested their intention to participate in the standardization process [15].

An MDATC comes in form of two-layers MOF-compliant metamodel. The first layer, called MDATC Infrastructure extends main UML2.0 Infrastructure classes in order to define the structure and contents of MDATCs. The second layer is based on the UML2.0 Superstructure. It extends key constructs of the UML2.0 Superstructure required for the definition of the modeling activities supported by the MDATC and which apply on its contents. We also discuss primary requirements, basis and challenges for the deployment and execution of such know-how packaging unit and we introduce a framework for MDATC coordination.Previous efforts were done in an ITEA (Information Technology European Advancement) research project called Families [6], from which several publications emerged: Bézivin et al. [1] and [4] in introducing the notion of MDATC.

This paper is organized as follows; in order to avoid any confusion with software components, Sect. 2 presents how MDATCs relate to them. Section 3 gives an overview of the MDATC architecture and sets some of the requirements that should be satisfied in order to support their deployment. In Sect. 4, we present our MOF-compliant metamodel for MDATC. It is composed of two parts; the former describes MDATC structure and contents and the latter specifies the concepts used to describe its behavior and the activities it supports. To this end, the latter makes use of UML2.0 Activity and Action constructs. A Framework for MDATC coordination is introduced in Sect. 5 and Sect. 6 presents related work. Finally, Sect. 7 highlights some perspectives.

2 MDA Tool Components versus software component

Because the term of Component is widely used with different meanings in the software area, we find it necessary to start by clarifying how MDA Tool Components relate to it. Grady Booch et al., define a component as *"a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Typically, it represents the physical packaging of otherwise logical*

elements, such as classes, interfaces, and collaborations" [3]. One property that MDA Tool Components share with software components is their ability to package the logical units necessary for an activity achievement. However, unlike software components, first class citizens of MDATC are models. We mean here by model any instance—at first or second level—of the MOF metamodel e.g. the UML metamodel, its model instances or Profiles. Likewise, an MDATC packages all required artifacts for its deployment and its execution. A non exhaustive list of such artifacts could be libraries (e.g. Java, C++, etc.), guidelines, model transformation rules, configuration parameters, consistency rules in form of OCL code attached to models, icons, etc. While the principal goal of the software component discipline is to enable practical reuse of software parts and amortization of investments over multiple applications, the MDATC vision aims at enabling reuse of a certain kind of know-how applied on models. For example, in the context of MDA, we can imagine an MDATC—one or a collaboration of MDATCs—for specifying the PIM of an application domain, one for transforming that PIM to a PSM and another one for generating code from that latter. Sequencing these three MDATCs then constitutes an MDATC-based production line. Of course, this would be one possible way but not the unique one of sequencing MDATCs in order to realize an application with respect to MDA recommendations.

To complete the comparison with software components, we address the notion of services required or provided by MDATCs. During its execution, an MDATC provides services and may require some. Services offered by MDATCs represent operations to be applied on models e.g. model editing, model consistency checking, model transformation, model comparisons and so on. The catalogue of services provided or required by MDATC is equivalent to software component interfaces. They represent the behavior offered by the MDATC. The interface of a software component is realized i.e. implemented by a set of operations in a specific programming language. As for MDATC, services provided might be defined either independently of any technological considerations or by using a specific language e.g. C++, Java, etc. One advantage of doing this is to keep service specifications platform independent and less vulnerable to the continuing evolution of technologies. Then, service specifications can evolve, are easily maintainable and analyzable.

3 MDA Tool Component

We give here an overview of the MDA Tool Component by providing the motivation of introducing such a notion, defining it in a formal way by means of a metamodel and describing some features that must be supported in order to deploy and to execute MDATCs

3.1 Rationale

In order to implement an MDA approach for a particular domain or context, one has to:

1. Define the appropriate modeling abstractions. Using the MDA related technologies, this means that specific metamodels or UML profiles are established, and that consistency checks for models are specified.
2. Implement production automation rules that will translate a given level of abstraction into another level of abstraction, or produce some development artifact.
3. Identify reusable model artifacts that represent software concepts applicable in different projects in the same domain or technology.
4. Define the steps and activities that have to be conducted and how they relate to the set of artifacts used or produced during the implementation of the MDA approach

In absence of a standardized method for implementing MDA, the above steps represent our view on how to implement an MDA approach. Actually, there may be more activities to conduct depending on the project's context and objectives. The MDA approach should be supported by all sorts of services in order to be easily adopted by software developers who will need to apply it. Services such as wizards, on line help, consistency checks and connection to platform development tools are necessary for the good usage and acceptance of an MDA approach. Each abstraction layer, represented by a model type, must be defined with a metamodel or profile and benefits from a complete modeling environment (GUI, checks, Help, process, generators, etc.). The MDATC is the proposed means to reach this objective. Methodological or process related rules can also be attached to the MDATC through specific additional functionalities: Guidelines, rules, descriptions of work products to be delivered, roles participating in the usage of the MDATC are examples of such process related aspects. The result is that many kinds of elements need to be packaged together, in order to provide a complete solution for implementing an MDA approach. That single unit of packaging is called "MDA Tool Component" (MDATC). An MDATC contains all necessary material to customize an existing modeling environment in order to help engineers to apply an MDD process for a specific domain or context.

3.2 Definition

An MDA Tool Component is a deployable and packaging unit of know-how for the definition of a certain kind of models with dedicated tools, services and resources. The types of model can for example be a PIM or a PSM, defined in the form of a metamodel or a UML profile. The MDATC

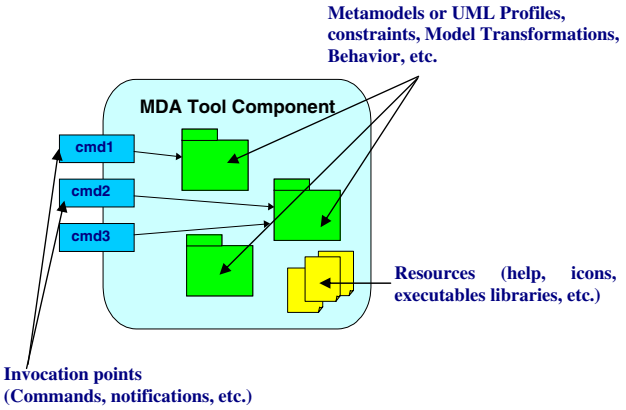


Fig. 1 An MDA Tool Component packages model definitions, attached services and resources

concept, in essence, aims at customizing an existing modeling tool, or group of tools. MDATCs are for tool providers and method engineers, not for those developing business systems. By buying or using an MDA Tool Component, an MDA tool owner can perform, for instance, a model editing, a model transformation or code generation according to the rules, scope and process specified by the component inside its MDA tool. An MDATC does not necessarily define a new kind of model. It can provide services for a model defined in another component or by modeling tools, like checking an UML diagram. It can also provide services not directly focused on models, like compiling source code. An MDATC needs to be connected to the customized toolset, and to be executed accordingly in order to extend the behavior and services provided by the host toolset. When deployed, the MDATC has to be executed in an MDA *Container*, which is a dedicated execution environment that can load the attached artifacts and descriptions, and that can execute the attached behaviors according to a predefined lifecycle. The MDA *Container* is embedded in a hosting tool (e.g., UML Case tool): it binds the hosting tool functionality to the loaded MDATC. MDATCs include the definition of "*invocation points*", which abstract the different mechanisms by which the MDATC can be solicited by the environment, such as provided services, events on which the MDATC reacts or menu entries provided to the end user (see Fig. 1.). These interaction points define the connection between the MDATC's external environment (e.g., end users, other tools or MDATCs) and the MDATC's implementation. In Sect. 5 we address in more details how MDATCs, MDA *Container* and the Hosting tool may interact and we give an architectural view of the ensemble.

3.3 Steps from the definition to the usage of MDATCs

The following steps and tools are necessary to support the definition, deployment and usage of MDATCs (see Fig. 2):

1. *Modeling and definition of an MDATC*: In this stage, a dedicated modeling tool called *MDATC Modeler* is used to define the profiles or metamodels supported by the MDATC. The modeling support includes the modeling of profiles or metamodel or the capacity to import these definitions using XMI, the modeling or definition of the invocation points of the MDATC, the modeling or definition of the behavior, and the modeling or definition of the packaging of the MDATC.
2. *Producing and packaging an MDATC*: The definition of the MDATC needs to be checked, compiled and produced into a packaged form. The MDATC *Modeler* tool also supports this step. The result is a file in a specific format, storing the implementation of the MDATC.
3. *Deploying an MDATC*: Using the MDATC *packaging unit*, the MDATC is made accessible to a modeling tool, and can be selected by users. The targeted tools must embed an MDA *Container* to be capable of loading and executing MDATCs.
4. *Applying an MDATC*: Once deployed, the users can select the MDATC in their modeling environment, and it will be executed. At this stage, the MDA *Container* loads and executes the MDATC that will customize the modeling environment and adds a new functionality. The profiles packaged by the MDATC will be applied to the model in the user's current work context, or the metamodels packaged by the MDATC will be interpreted to type the models created under the hosting tool.
5. *Maintaining existing MDATCs*: to make the MDATCs reusable, an organization should maintain libraries of MDATCs that can be applied in projects of this organization. These MDATCs can evolve to support new assets, consistency checks and modeling extensions.

In practice, these steps must be iterated many times before producing valuable components. Indeed, building an MDA tool chain for a specific domain is a long and hard task, needing both domain and modeling experts. But once built, such tool chains can provides a high degree of automation as demonstrated in [16]. The goal of MDATCs is to capture this know how in order to reuse it for new projects in the same domain.

3.4 Constituents of MDATCs

Typically, an MDA Tool Component can be used to define, partially or entirely, one or several specific CIMs, PIMs or PSMs. Each MDATC must provide some technical features used by the hosting tool to load and manage it:

- A predefined *lifecycle interface implementation* that allows the MDA *Container* to manage the MDATC;

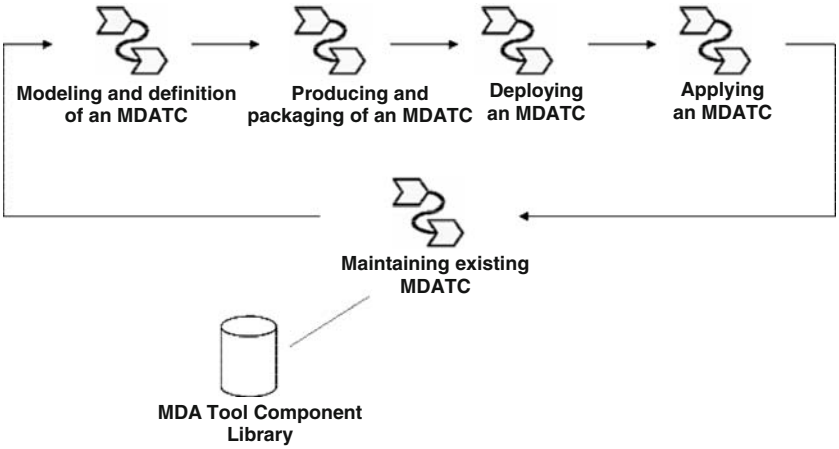


Fig. 2 MDA Tool Component use case process

- *Provided services* that connect end user interactions as received by the MDA *Container* environment to the behavior packaged within the MDATC.
- *Required services* that specify the services that are requested from another MDATC, or from the external environment.
- *Deployment parameters* that provide a means of customizing an MDATC to its deployment environment or to its global end user options;
- *Resources* necessary for the execution of MDATC. These can be libraries, icons, on-line help, user manuals, etc.;
- Elements related to the distribution policy, such as the licensing scheme, or the protection of MDATC sources.

Each MDATC also provides some valuable artifacts and features for its specific domain:

- *Definitions of models*, in the form of a metamodel or a profile;
- *Consistency rules* attached to the model definitions. They can be implemented by executable code (e.g. Java) or OCL code;
- *Model transformations* that can be applied to the packaged model definitions. They can be implemented by executable code (e.g. Java or 'J' [8]) or QVT [13];
- *Functionalities or behavior* attached to the model definitions that can be expressed in any executable language, and that can typically provide such additional services as a dedicated GUI or the usage of external libraries;
- Other elements that can be attached to an MDATC, and that may depend on a development methodology, such as design documentation, models, tests, etc.;

Of course, MDATC constituents are not restricted to those defined above and vary depending on the purpose and domain the MDATC is used for.

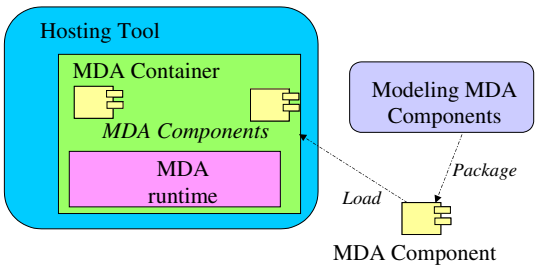


Fig. 3 Running MDATCs into MDA Containers

3.5 Runtime support of MDATCs: the MDA Container

The MDATC execution has to be supported by a layer independent of the tool that executes it. The objective is to provide an infrastructure that is capable to be embedded within model-based tools (such as Case Tools, Meta Case Tools, or any other tool providing model oriented services) and which provides in a uniform—and hopefully standardized—way the services provided by the hosting tool. This kind of architecture is well supported by the "container pattern" widely used by component based architectures such as CCM or EJB. Figure 3 below shows an MDATC embedded within an MDA *Container* which isolates the MDATC from the hosting tool, allowing thus to run the same MDATC on several kinds of hosting tools.

Invocation points defined by the MDATCs are the mechanisms by which the MDA *Container* will run the MDATC, and let the MDATC cooperate with the hosting tool. User interactions, events, required services can be transmitted from the hosting tool to the MDATC through the MDA *Container*. Provided and required services can also be used to establish a dialog between several loaded MDATCs. There are predefined services requested by containers, that every MDATC shall implement, in order to let containers manage the lifecycle of the loaded MDATCs. It is not expected that any hosting tool is capable of providing every requested environment and services to any MDATC. MDATCs can therefore

express requirements on the capacities of the hosting tools. For example, some tools are capable to interpret metamodel definitions, other support profiles for a certain version of UML, some have a strong GUI capacity, etc. It is up to the MDA *Container* to check that the hosting tool fits to the requirements expressed by the MDATC to be loaded. Concerns like distribution or versioning are not directly taken into account by MDATCs but can be supported by the hosting tool [4b].

In order to be used together and/or in different tools, MDATCs must be defined with a common and standard language. MDATCs are defined using the MOF standard. In the next section, we provide a more formal definition of MDATC in form of a metamodel.

4 The MDATC metamodel

The definition of the MDATC we propose comes in form of two-layers metamodel, namely the *MDATC Infrastructure* layer and the *MDATC Behavior* layer. The *MDATC Infrastructure* layer is a MOF-compliant metamodel. It extends the UML2.0 Infrastructure concepts in order to specify the structure of the MDATC, whilst the Behavior layer takes advantage of the recently adopted UML2.0 Superstructure constructs in order to provide a platform and language independent way for specifying the behavior of MDATCs. Below we introduce both layers in more details.

4.1 The MDATC Infrastructure layer

Figure 4 shows the part of the MDATC metamodel corresponding to the *Infrastructure* layer. A specific effort has been conducted to align this metamodel to the UML2.0 Infrastructure standard. This metamodel contains the definition of an MDATC and its related classes.

MDATCs are defined in terms of related model definitions, expressed as packages, dependencies on other MDATCs, packaged physical units, and provided services. MDATCs are a kind of NameSpace. They act as a namespace for each aggregation where the opposite role subsets "ownedmember". The model definition is a kind of package, corresponding to the OMG metamodel definitions. These packages can be profiles, or packages representing metamodels. When the MDATC refers to standards, the model definition packages are referenced through an uri, using the OMG identification mechanism as defined in the MOF standard. MDATCs are represented as an aggregation of the elements to be packaged. A Service is the formal declaration of an exposed functionality that a tool or a MDATC can provide, and that a tool or an MDATC can require. MCServices can be called internally through commands, internal calls, or event reception. Artifacts represent in general the resources or artifacts related to the MDATC. They can for example be libraries, documents, test models, user guides, icons, etc. Their exact kinds shall be defined by creating subclasses of Artifact in more concrete subPackages. These configurations are very specific to the methodology recommended for building MDATC and can vary widely. The intent behind MDATC Artifact is close to the semantics of UML2.0::SuperStructure::Artifact. Therefore, in order to match both semantics (using package merge), the name has been deliberately chosen to be the same. InvocationPoint describes how an MDATC can be activated and which MDATC service is activated. It is then up to the execution container to provide the adequate mechanism to support the specified extension point and activate the service when required. An MDATC can assemble other MDATCs. The assembly can be specified as a reference to another MDATC. In that case, the assembled MDATC is requested for the execution of the assembly. The assembly can be specified as "Embedded". In that case, the assembled MDATC is packaged within the assembly one, and deployed before the assembly in the local execution space.

It is expected that each language used for expressing the behavior of an MDATC has an OO structure: It can be structured by packages that own classes having operations in which the language instructions are expressed as implementation. This is the case for QVT [13], Java, and "J"[8], but also for a vast majority of languages. As we will see below, we also propose a language independent way to specify the behavior of an MDATC, but still we leave the possibility open to use a specific language, at the cost of reducing the hosting tool independence level.

4.2 The MDATC Behavior layer

As introduced above, MDATCs have a behavior that might be specified independently of any specific language. Developers will have the possibility to define the MDATC behavior either in a specific language or in an independent way. By choosing the latter solution, they will overcome the problem of the continuing technology changes and business evolutions. Indeed, when having a platform-independent behavior description, then it becomes easier to maintain, to evolve and to generate the behavior into a specific platform or language. To this end, we propose the *MDATC Behavior* layer that is a UML2.0 Superstructure-based metamodel which extends UML2.0 Activity and Action constructs by adding the properties and semantics required for the description of MDATC behaviors. Then, the "packageMerge" facility offered by the UML 2.0 standard can be used in order to merge the two layers, i.e., *MDATC Infrastructure* and *MDATC Behavior* (see Fig. 5). As a result, we obtain a full specification that allows the definition of both structural and behavioral aspects of MDATCs.

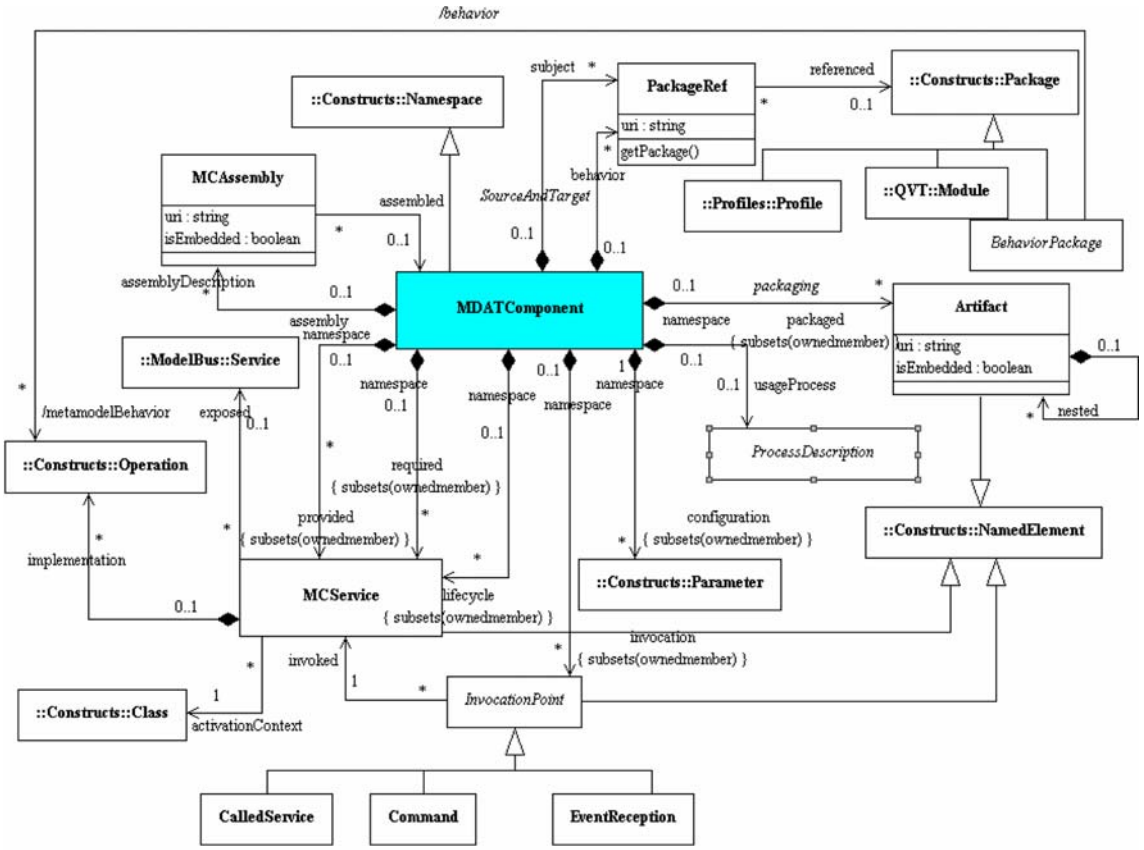


Fig. 4 A global overview of the MDATC infrastructure layer

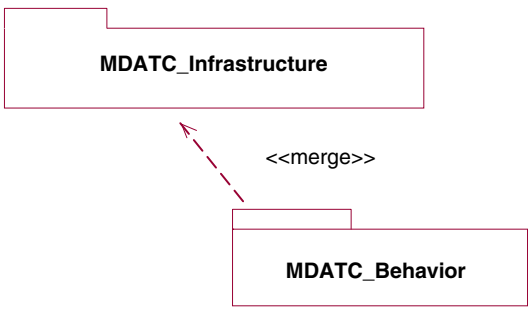


Fig. 5 The MDATC metamodel: package merge of MDATC layers

The metamodel representing the *MDAC Behavior* layer comes in form of package hierarchies. The outermost level contains two packages: the Behavior_Foundation package and the Behavior_Extensions package (see Fig. 6).

The Behavior_Foundation package contains all UML2.0 packages required as a basis for MDATC behavior descriptions. Main ones are those related to Activities, Actions, and the Kernel package, the core of UML2.0. The Behavior_Extensions package holds classes that extend UML2.0 constructs introduced in the Behavior_Foundation package. Figures 7, 8, 9 point out how concepts of both packages are interconnected. They represent a global overview of the *MDATC Behavior* layer. Lighted boxes in figures represent UML2.0 classes. Shaded boxes represent those we specified and that inherit UML2.0 classes.

The main class of the Behavior_Extensions package is the MDATComponent class. An MDATComponent inherits the UML2.0::BehavioredClassifier class (Fig. 7). A BehavioredClassifier is a Classifier that has behavior specifications defined in its namespace. One of these may specify the classifier's behavior itself which will be invoked when an instance of the BehavioredClassifier is created. One advantage is that the MDATC's behavior can be represented by state machines; this adds more control on the MDATC lifecycle. Another advantage is, being a Classifier, an MDATC can encapsulate, i.e., own other classifiers such as Artifacts as well as ActivityPerformer on these artifacts. An Artifact is the specification of a physical piece of information that is produced, consumed, or modified by an MDATC.

An MDATC provide Services and may require some. Services offered by the MDATC are realized by Operations (Fig. 7). Operations are described in terms of Activities which contain a set of Actions with an executable semantics. An Action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. Input to, respectively, output from, an action is a typed element. It

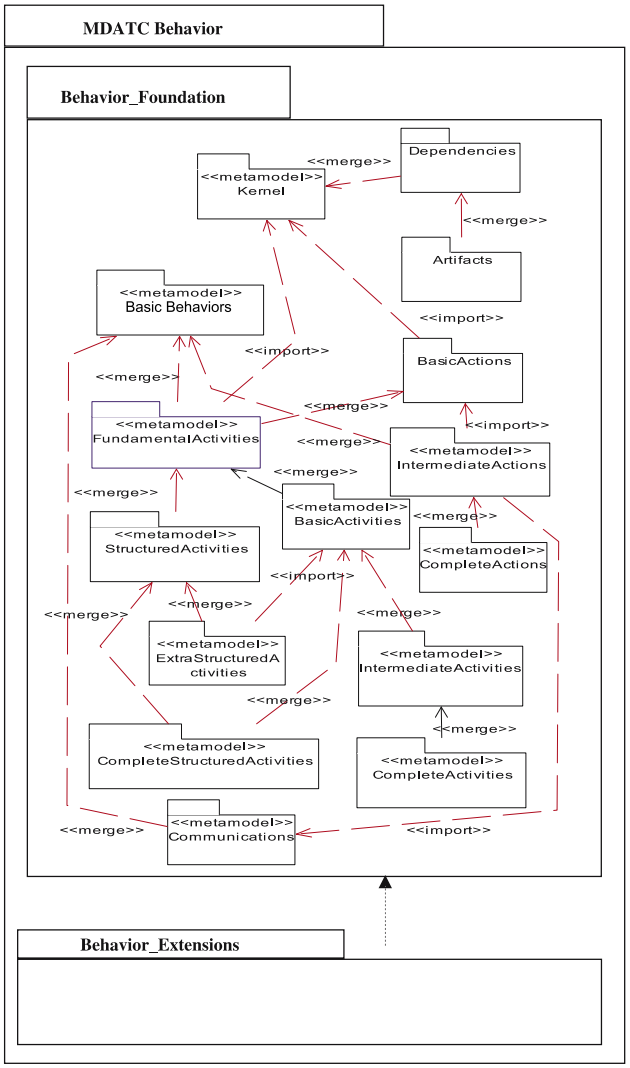


Fig. 6 MDATC Behavior layer: package hierarchies

represents the Pin of the action. A Pin is typed by a Classifier. Actions consume and produce artifacts. The relation between an action and artifacts it handles is made through the fact that artifacts are Classifiers and Inputs and Outputs of an action have a type which is specified by a Classifier too (Figs. 7 and 8). This would allow actions to manipulate artifacts as easily as calling a method while passing it parameters in usual OO programming languages.

Activities have one or more ActivityPerformer who are in charge of the activity and more particularly of actions owned by it. An ActivityPerformer can be a ResponsibleRole or a SoftwareTool (e.g. compilers, model transformation engines...) (Fig. 8). A ResponsibleRole describes rights and responsibilities of the Human who will interact with the MDATC during its execution. A Human may be an agent or a team; it has a name, a (set of) skill(s) and an authority.

Finally, having in mind that an MDATC may need some tool facilities during execution-time, we decide to extend the Actions model. The CallToolServiceAction is a CallAction (see Fig. 9). It has InputPins which represent the arguments of the call and OutputPins as call results. We make the assumption that a ToolService has a name and a set of typed parameters. One constraint on the CallToolServiceAction, would be that CallToolServiceAction arguments fit to ToolService parameters (in number and type). The model of the tool (list of services, parameters of services, binding mode...) is outside the scope of this work, you will find more details in [2,18].

5 MDA Tool Component framework

After having defined MDATC, in this section we address the concern of their coordination. MDA Tool Components shall have a predefined protocol in order to behave in a manageable way, and to be able to cooperate together and with their environments. This protocol will in particular be used by the MDA *Container* that will manage accordingly each MDA Tool Component. We will see in this section that the protocol needs indeed to support more complex situations, where external participants or other MDATCs can intervene, and need to be coordinated. Once the protocol is formalized, an API which supports this protocol can be defined, which will allow the MDA *Container* to interact with the MDA Tool Components. In the following, we focus on a Java implementation, and provide a Java based API to manage the MDA Tool Components, and the MDAC *Container* but first; we start by giving the different use cases of MDATCs coordination.

5.1 Use cases for MDA Tool Component coordination

An MDA Tool Component interoperates with the MDA *Container*, with the MDA *Container's* environment, and with other MDA Tool Components.

- **The basic coordination** use case: is when an MDATC is loaded, one command is activated, and then the MDATC is unloaded. This use case contains the major protocol aspects, and the fundamental coordination required for an MDATC.
- **MDATC/external world cooperation** use case: in this use case, the external world (e.g. an end user or third party tools) will interoperate with the MDATC, through several requests, and the MDATC may have to react to these requests that may be parallel or interleaved.
- **MDATC interaction use case**: several MDATCs interact, and send requests to each others. This use case is indeed very close to use case 2. There is no fundamental difference between an MDAC sending a request to another MDAC, or an external participant sending a request to an

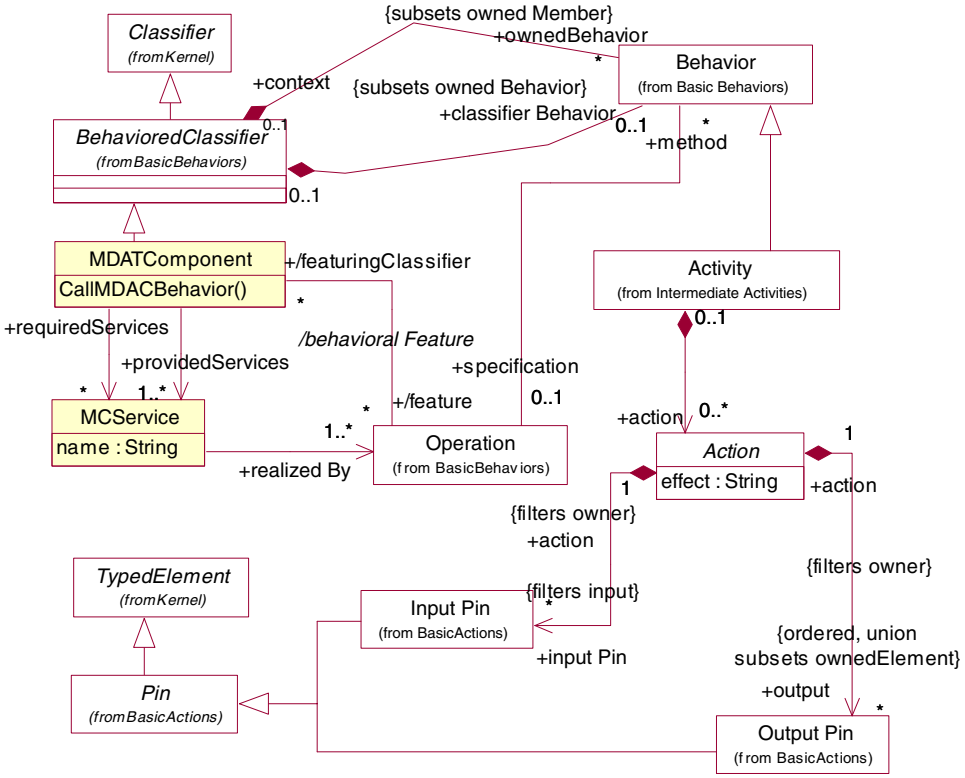


Fig. 7 MDATComponent inherits UML2.0 BehavioredClassifier

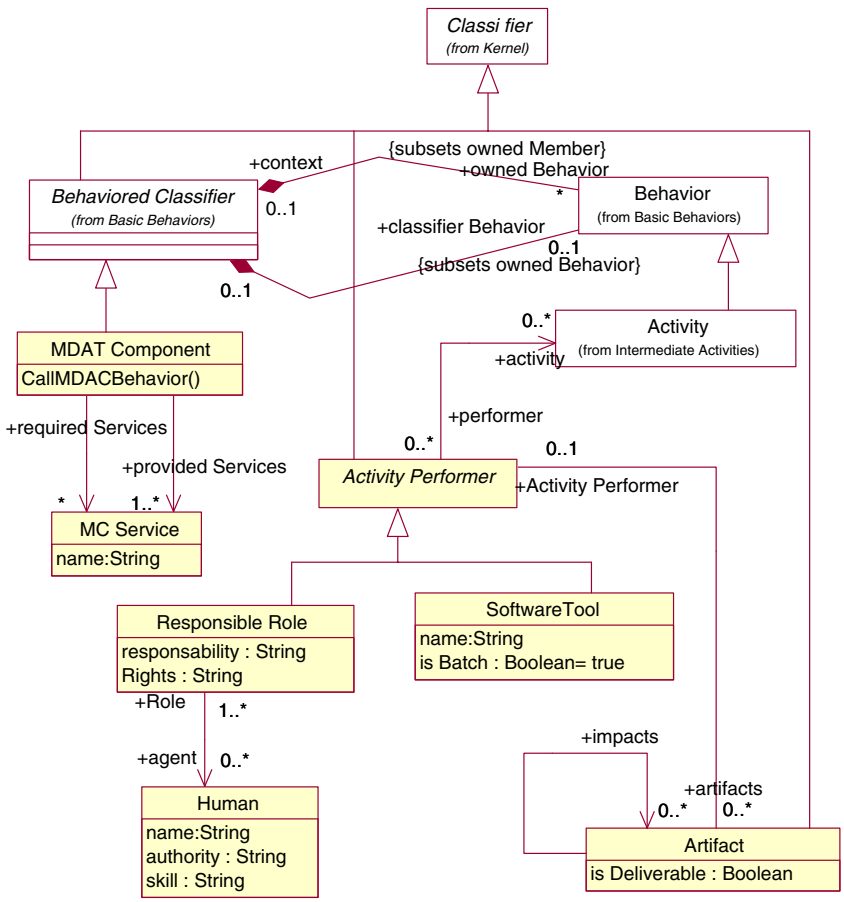


Fig. 8 An MDATComponent encapsulates Artifacts and Activity Performer specifications (Roles, Tools)

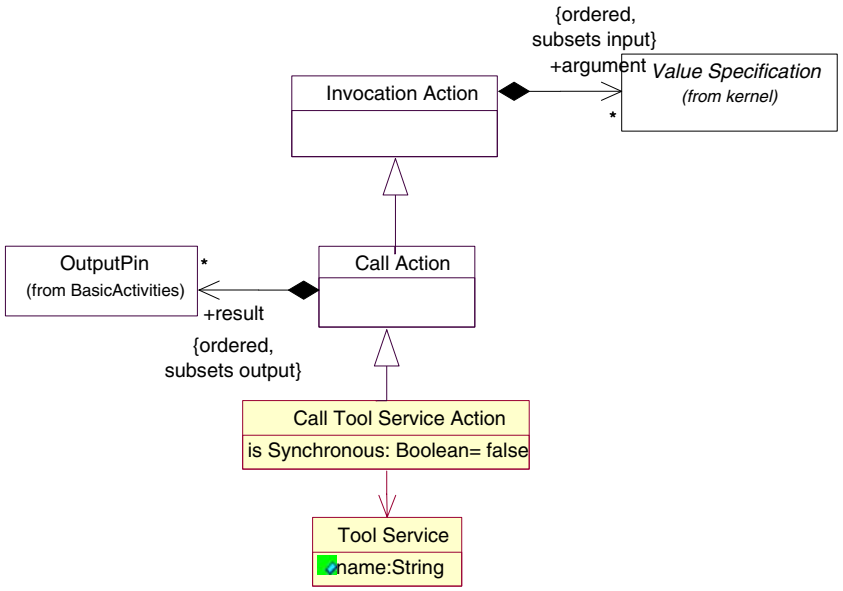


Fig. 9 The CallToolServiceAction

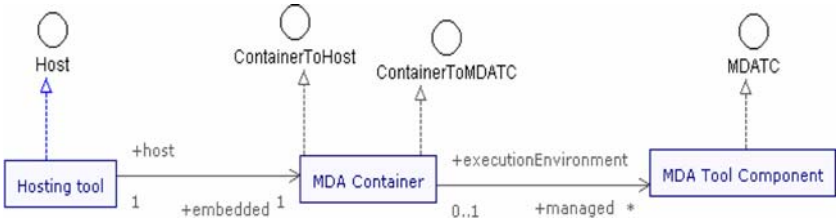


Fig. 10 The main constituents of an MDATC environment

MDAC. Indeed, the architecture shall be defined in such a way that these two use cases are similar. In addition, these two use cases can be mixed, so that an MDATC receives requests simultaneously from third party tools and from other MDATCs. In the case of MDATC, the usage between MDATCs can and shall be previously modeled. This allows the MDATC *Container* to make sure that the required MDATCs are pre-loaded, when the requester MDATC is loaded. That requirement cannot be set to third party tools.

5.2 Architectural view on MDA Tool Components coordination

In Fig. 10 we can see the main constituents of an MDATC environment. We notice that an MDA *Container* can manage several MDA Tool Components. The MDA *Container* is hosted by the hosting tool. Each of these constituents implements a specific set of interfaces that standardize their cooperation. The MDA *Container* exposes two interfaces, one devoted to its interaction with the MDA Tool Component, and the other one devoted to its interaction with the hosting tool. The "Host" interface specifies the services that a hosting tool shall provide to an MDA *Container*. A hosting tool embeds at most one MDA *Container*.

Figure 11 shows the assembly of the MDATC constituents. It represents the connection between required and provided interfaces of each component. We see here the independence provided by the respective interfaces defined by the architecture: there is no direct awareness of the concrete hosting tool for the MDA *Container* (at least in its standard part), and this applies for each of these elements. In addition, there is no direct connection between the hosting tool and the MDATC, thus guaranteeing their total independence.

5.3 The MDATC lifecycle

After it has been packaged, an MDA Tool Component is deployed. The first deployment step simply consists of making the packaged MDAC available, in some convenient place accessible from the network or the local disk. It can then be loaded and deployed in the runtime environment of the MDA Container. After the start of the MDATC, it goes into the "idle" state. It is considered that the "start" transition cannot fail. Services are provided to the MDATC that can implement a specific behavior during the start transition. Whether it works or not, does not change the resulting state "Idle". During the start transition, the MDATC has the opportunity to set up its runtime initial configuration, such as GUI appearance or specific global initialization procedures. When it is invoked from the idle state, the MDATC becomes active (see

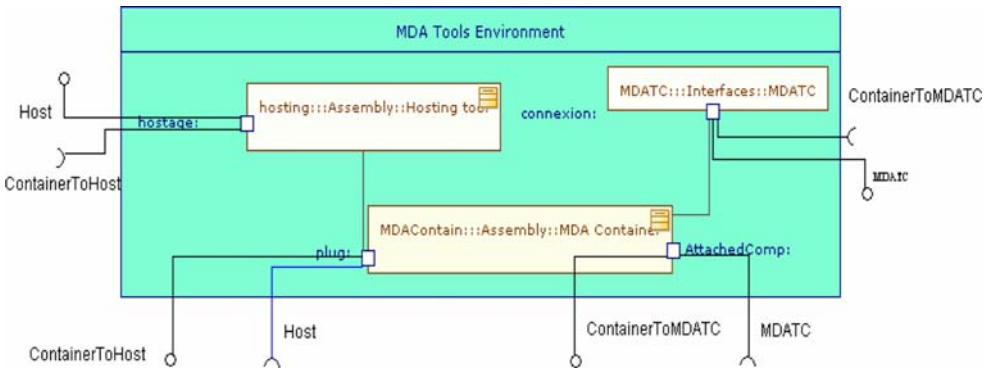


Fig. 11 Assembly of the MDATC constituents

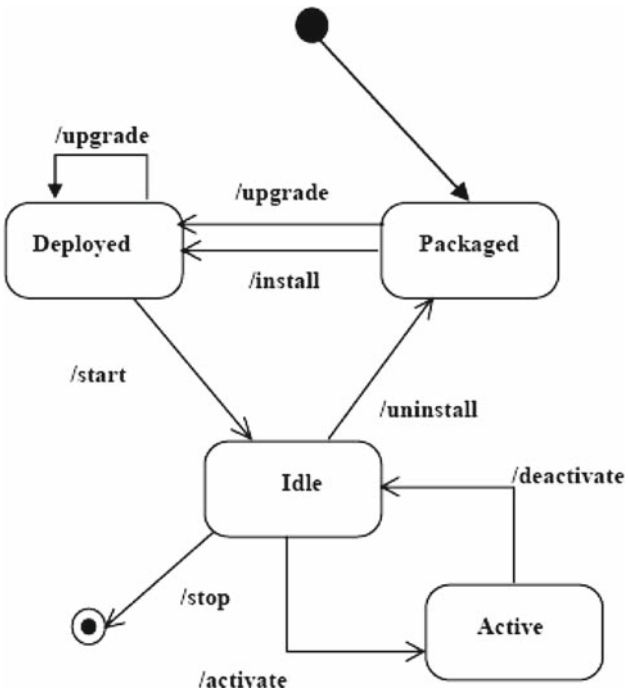


Fig. 12 MDATC lifecycle

Fig. 12). From the idle state, an MDA Tool Component can be uninstalled, which will bring it back to the packaged state. From the idle state, an MDATC can finally be stopped. The MDATC is aware of these state changes, and can process some code for each one of them.

The MDA *Container* provides the mechanism to handle the lifecycle by calling the following set of predefined operations on the MDATC metaclass. MDATCs can then provide specific behavior for each of these predefined operations. The following transitions are triggering a call to services provided by the MDA Tool Components of the same name:

- Install: the MDA Tool Component is added to a current model environment. The install service provided by the MDATC will do the necessary processing to initialize the environment required for its execution. Typically, elements related to file setting, register initialization and other global settings are initialized there.
- Uninstall: the MDA Tool Component is removed from the current model environment. Elements created or changed in the global setting by the install service are typically cancelled.
- Start: the hosting tool is starting, the start service of the deployed MDA Tool Components are called. Typically, dynamic initialization is realized there, such as adding GUI elements to the hosting tool.
- Stop: The hosting tool is stopping, and all associated MDA Tool Components will be stopped through their "stop" service.
- Upgrade: a new version of the MDA Tool Component is being installed. The "upgrade" service of MDATCs is called. This service typically does the same kind of job as "install", but in an updating perspective. Issues such as backward compatibility, upgrading tools and MDA Tool Components are managed through this transition.

The MDA *Container* will manage the MDA Tool lifecycle by invoking these operations, according to the lifecycle defined in Fig. 12. The "activate" and "deactivate" transitions do not have a single service match. The "activate" transition corresponds to an invocation point triggering, that the MDA *Container* will transfer to the MDATC, by calling a specific service. The end of the service processing will correspond to the "deactivate" transition.

5.4 APIs for the MDATC Framework

As shown in the architectural view on MDA Tool Components coordination (cf. 5.2), four interfaces have been defined, that shall support the services necessary for an MDATC to interact with the host tool, and that shall support the MDATC/MDATC *Container* cooperation, according to the

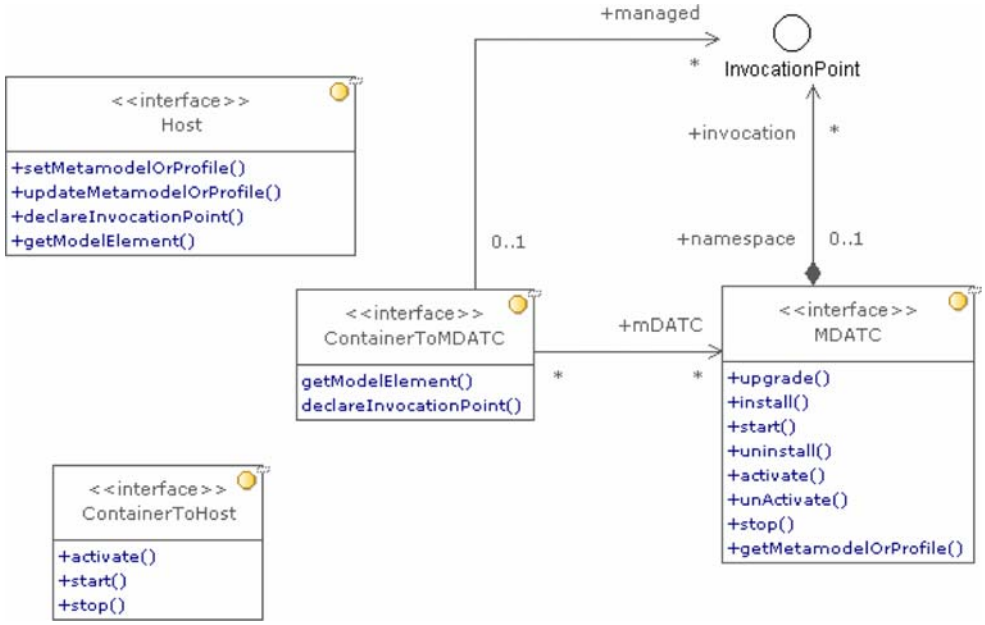


Fig. 13 Interfaces and their API

MDATC lifecycle. In the following, we detail each of them (see Fig. 13):

The **Host** interface defines the services provided by the hosting tool.

- SetMetamodelOrProfile: the metamodel or profile provided by a loaded MDA Tool Component must be applied to the hosting tool. A service is provided for that purpose. This service may not be capable of setting several metamodels. In that case an error is returned to the caller.
- UpdateMetamodelOrProfile: When a new version of an MDA Tool Component is loaded, then the hosting tool shall update its metamodel or profile. That is a high level capacity. The new metamodel must be backward compatible or transformation rules must be provided to update existing instances. A tool may send an error on this, if this service is not supported by its infrastructure.
- DeclareInvocationPoint: Invocation points defined for the MDA Tool Component are declared to the hosting tool, which may have very specific ways of attaching them to its GUI. The invocation point instance, which will be used by the MDA Tool Component later as an identification of the invocation to handle, is passed as a parameter. Some GUI preferences are provided: they can express if the invocation point is to be managed as an event, as a menu entry, or as another kind of interaction, and GUI preferences that are very specific to the hosting tool.
- GetModelElement: the Hosting tool shall be capable of providing access to handle model elements. This service is very under specified, but shows that a service set is to be provided there.

The **"ContainerToHost"** interface defines the services that the container provides to the hosting tool.

- Activate: activates an invocation point. The hosting tool has received an invocation through its GUI or API that has been declared as related to an invocation point. It then asks the MDA *Container* to manage the proper MDA Tool Component, and to execute the related behavior.
- Start: start of the MDA *Container*. This happens at the hosting tool launch time. The MDA *Container* shall initiate all the declared MDA Tool Components, and run its behavior.
- Stop: The MDA Container is requested to stop when the hosting tool stops. It is asked to properly stop all the active MDA Tool Components.

The **ContainerToMDATC** interface provides all the necessary services for the MDA Tool Component to run.

- GetModelElement: the *Container* shall be capable of providing access to handle model elements. This service is very under specified, but shows that a service set is to be provided there.
- DeclareInvocationPoint: Invocation points defined for the MDA Tool Component are declared to the *Container*, some aspects that must be generically managed are specific to the hosting tool. The invocation point instance, which will be used by the MDA Tool Component later as an identification of the invocation to handle, is passed as a parameter.

Finally, the **MDATC** interface defines the services that an MDATC shall implement to be managed by the MDA *Container*. Most of these services are implementing the MDATC lifecycle as described before. Its services are already explained in the MDATC lifecycle. If the MDATC has to be installed or upgraded, then the service "getMetamodelOrProfile" is called, in order to set accordingly the container.

6 Related works

At the technical level, MDATCs can be compared to EMF (Eclipse Modeling Framework) based plug-ins or Objecteering modules. They aim to improve tool functionalities. But, EMF plug-ins and Objecteering modules are tool specific and their requirements are defined in term of tool and library versions. On the contrary MDATCs requirements are at the conceptual level. MDATCs services are defined related to a metamodel that can be provided by the hosting tool or by another MDATC. Furthermore MDATCs themselves are defined using a model driven approach, Eclipse plug-ins or Objecteering modules can be considered as targeting platforms. Like MDA proposes platform independent business models we propose, with MDATCs, tool independent metamodel and service definitions. MDATCs aim to capitalize know-how in a packaged form independent of tool evolutions.

One of the main contributions in the literature that relates to the MDA Tool Component is the Microsoft's Software Factories vision. A Software factory (SF) is a product line that configures extensible development tools e.g. Microsoft Visual Studio Team System (VSTS) with packaged content and guidance, carefully designed for building specific kinds of applications [7]. Main concepts in SF are the software factory schema and the software factory template. A SF schema lists artifacts like source code, SQL files, etc. and how they should be combined to create a product. It specifies which Domain Specific Languages (DSLs) should be used and defines the product line architecture. As for SF template, it packages all the artifacts described in the SF schema. It provides patterns, guidance, templates, DSL editing tools, etc. used to build the product. Then, an extensible development environment will be configured by the SF template in order to become a SF for a product family.

Looking at these definitions of SF, one can deduce that Microsoft's Software Factories and MDA Tool Components address the same problem. Indeed, SF and MDATC promote the same vision of reusing software skills for the sake of a more cost-effective software production and short time-to-market. However, when MDATC first class entities are MOF instance models, SF promotes the use of DSLs. This is, in our opinion, a delicate issue. In [7], authors argued that in some cases, UML profiles or MOF are not well suited for modeling some business concepts which are more naturally expressed in a specialized syntax. They also support that DSLs are easy to create, to evolve and can be used to implement solutions based on these DSL. Our vision is different. In our case, the primary reason why we chose a standard way (i.e. MOF instances such like UML) for representing models is to be independent of any platform or language. Using a DSL for making models more expressive is not a problem in itself. The problem is that it requires that the DSL semantics be understandable by tools and project stakeholders. Then, even if tools are compatible within the same SF template (package), how do we deal when these tools need to exchange models with other tools using different DSLs? Moreover, if the behavior of a software factory is defined through thousands of code source lines, how the SF customer could maintain or customize the SF? We believe that mechanisms like stereotypes and tagged values offered by UML profiles as well as OCL (Object Constraint Language) [22] are expressive enough to capture the characteristics of a specific domain context. The OMG already provides Profiles for Software Process Engineering (SPEM), System engineering, test modeling, QOS modeling, and the already long existing list of profiles will proliferate within the coming years [14]. For those who may find limitations of using UML profile, MOF can be used as formalism for defining domain specific language metamodels. This allows leveraging UML standard tools and training. Thus even if SF and MDATCs share the same vision of promoting software production lines, they do not follow the same approach. However, MDA Tool Components can be used by Software Factories as a standard mean for representing certain knowledge and for packaging required artifacts and tools for applying it.

7 Conclusion

In this paper, we introduced the notion of MDA Tool Component, i.e., a packaging unit for encapsulating business know-how and required resources in order to support specific operations on a certain kind of model. A standard formalism for representing MDA Tool Components was proposed in form of MOF-compliant metamodel, requirements for their deployment and execution was established and a framework for their coordination is proposed. Most of the work which has contributed to this paper has been done within the Modelware IST project [11]. Tooling and Use Cases are currently underway and a first implementation is in finalization phase. It relates to an MDA Tool Component providing and packaging all the essential know-how and material required for the design of a CCM (CORBA Component Model) application. That latter makes use of UML according to the UML profile for the CCM standard. In parallel, there is an OMG effort to standardize the notion of MDA Tool Component.

Currently the writing of an RFP is underway. We believe that the MDA Tool Component notion, once standardized and tooled will provide its full power to the MDA technique, in order to capitalize, share, reuse, improve and automatically apply software development know-how. Still there is work to do in MDATC. It mainly relates to the packaging issue and the possibility to use the Reusable Asset Specification [14] at this aim is under investigation.

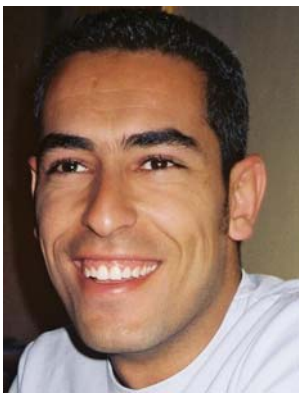
The goal of MDATCs is not to build a full automated tool chain but helping developer to add to his usual modeling tool all necessary assets and knowledge (metamodels, guidelines, model-checker, transformations,...) for a specific domain.

For now, as we mentioned it in Sect. 3.5, technical concerns of modeling process (model serialization, model concurrency editing, versioning, etc.) are not managed by MDATCs but by the hosting tool. It will be interesting to study integrating such concerns into MDATCs, maybe by managing each technical concern by an MDATC, and thus applying MDATCs idea to the modeling domain itself.

References

1. Bézivin, J., Gérard, S., Muller, P.-A., Rioux, L.: MDA components: challenges and opportunities. In: Metamodelling for MDA, York, England (2003)
2. Blanc, X., Gervais, M.P., Sriplakich, P.: Model bus: towards the interoperability of modelling tools. In: Proceedings of the MDAFA'04. Linköping University, Sweden (2004)
3. Booch, G., Rumbaugh, J., Jacobson, I.: The unified modeling language user guide. Addison-Wesley Professional; 2 edn, (2004)
4. Desfray, P.: Techniques for the early definition of MDA artifacts in a UML based development. Enterprise UML & MDA, London May 12 and 13 at: http://www.enterpriseconferences.co.uk/programme.pdf
5. EDOC: UML profile for enterprise distributed object computing, OMG Document ptc/02-02-05, 2002 http://www.omg.org
6. Families ITEA Project at: http://www.esi.es/en/Projects/Families/
7. Greenfield, J., Short, K.: Software factories:assembling applications with patterns, models, frameworks and tools. In: Proceedings of the 18th conference on object oriented programming systems languages and applications (OOPSLA), Anheim, CA, USA, ACM press, New York (2003)
8. J language, at: http://www.objecteering.com/pdf/whitepapers/us/uml_profiles.pdf
9. MDA Development tools, at: http://www.omg.org/mda/committed-products.htm
10. MDA Guide: Model driven architecture (MDA), OMG TC document ormsc/2001-07-01, July 2001, at http://www.omg.org
11. MODELWARE Project, at http://www.modelware-ist.org
12. MOF 1.4.: Meta-Object Facility, OMG document formal/2002-04-03, April 2002, at http://www.omg.org
13. OMG, MOF2.0 QVT (Query /Views/Transformations), Final Adapted Specification, OMG document ptc/05-11-01, November 2005, at http://www.omg.org
14. OMG specifications at: http://www.omg.org/technology/documents/modeling_spec_catalog.htm
15. OMG TC Work in Progress http://www.omg.org/schedule/
16. Presso, J.M., Belaunde, M.: Applying MDA to voice applications: an experience in building an mda tool chain. In: Proceedings of the model driven architecture - foundations and applications (ECMDA-FA), LNCS 3748 (2005)
17. SPEM1.1: Software process engineering metamodel. OMG document formal/02-11/14, November 2002, at http://www.omg.org.
18. Sriplakich, P., Blanc, X., Gervais, M.-P.: Supporting collaborative development in an open MDA environnement, IEEE Int'l conference on software maintenance (ICSM) (2006)
19. Standish Group: 2004 Third quarter research report. at: http://www.standishgroup.com. Page last visit: June 13 (2005)
20. UML2.0 Infrastructure: Unified modelling language, Final Adopted Specification, OMG document ptc/03-09-15, December 2003, at http://www.omg.org
21. UML2.0 Superstructure: Unified modelling language, Available Specification, OMG document ptc/04-10-02, October 2004, at http://www.omg.org
22. UML2.0 OCL Specification: Unified modelling language 2.0 object constraint language. Adopted Specification, OMG document formal/03-10-14, October 2003, at http://www.omg.org

Author's Biography



Reda Bendraou is a PhD candidate at the LIP6 computer science laboratory of the Pierre & Marie Curie University, Paris. His research interests focus on Model Driven Development approaches applied to software process modeling and execution, OO methodologies and UML model executability and semantics. He is also an OMG member. He participates at the standardisation efforts of SPEM2.0, the OMG standard dedicated to software process modeling.

Philippe Desfray, is an expert in object oriented method, and VP for R&D in the SOFTEAM company. He has created an object oriented method in 1990, published three books, and has conducted the development of the Objecteering Case tool. In 1994, he has introduced a technique called ≪Hypergenericity≫ close to the UML profile technique, supporting model transformation. His continuous work on Model driven engineering has conducted him to heavily influence the ≪UML Profile standard≫, and to drive the development of MDA based evolutions of the Objecteering case tool. Since 1994, Philippe Desfray represents SOFTEAM as a Contributing Member at the OMG, and actively participates to the UML definition. In particular, Philippe has been leading the definition of the UML Profile mechanism for UML1.4, and UML2.0.

Marie-Pierre Gervais is a Full Professor of computer science at the University of Paris X (UFR SEGMI). She doing her research work in the "Modeling & Verification" team at the Laboratoire d'Informatique de Paris 6 (LIP6). She is leading the ODAC project, a project which main concerns deal with the modeling of open and complex applications in a distributed environment. Her research interests focus on the construction of open and distributed applications, Model Driven Development and Model Composition. She is also an OMG member and participated in the RM-ODP working group.

Alexis Muller received his PhD degree in computer science from Lille University, France in 2006. He is currently occupying a post-doc position at LIP6, University of Pierre & Marie Curie. His research interests focus on model driven development, model composition and formalization.