

<b>OPEN CANARIAS, S.L. [ATC USER GUIDE]</b>	
<b>Title:</b>	ATC USER GUIDE
<b>Author:</b>	Victor Sánchez - vsanchez ( at ) opencanarias (dot) com
<b>Abstract:</b>	
<b>Release to:</b>	Public
<b>Copy to:</b>	

<b>Version History</b>			
<b>Version</b>	<b>Date</b>	<b>Author</b>	<b>Description</b>
0.8	23/JUN/2008	Victor Sánchez	Initial Version. Limited Content.

## 1. Preface

### ***Objectives***

The goal of this guide is to fully describe the ATC model transformation language, its structure, manifestations, common usage form and insights. It takes a tour on discusses the ATC metamodel's contents. It also addresses the intricacies and subtleties of the language and how to assemble different pieces to express meaningful things, so readers become acquainted with them and become ready to fully exploit its power if confronted with the need of creating ATC instances. Along with this document, examples on how to make the different parts of ATC collaborate together to define model transformation actions are also being shown. Currently only an overview of ATC and a thorough explanation of a simple example are covered. The underlying virtual machine, with its different execution mechanisms and particular implementation alternatives, will also be discussed in future versions of this document.

### ***Scope***

This guide is meant to represent a user guide about every ATC topic. And at the same time, it resembles but falls short of becoming a complete reference guide on its own. To get detailed specifications about elements of the ATC metamodel, you should try the ATC's javadoc documentation, which is expected to be available sometime soon. Source code for each instruction is also a practical valuable documentation source. On the other hand, ATC, although already quite mature, is under steady development, so this document could be subject to revisions to reflect those changes and to remain as up to date with the implementation as possible. In the meantime, we are sorry for any inconveniences potential delays in updating these changes or the changes themselves may cause to you.

### ***Who this guide is intended for***

This guide is targeted at those who are interested in trying a low level model transformation engine, academically or professionally. Or in developing a translation from another, higher-level model transformation language, which, in the end, is the primary goal of the ATC language. The consequences of providing such translations is that high-level languages become compatible within the ATC transformation engine environment and can be executed by it.

On the other hand, you know that ATC is of no interest for you if you already have a transformation language of choice with an executing implementation available with which you feel comfortable enough. Interest in a centralized model transformation engine with support to several different languages may manifest to users who make use of one or several of the available languages. Even in such cases, if nothing else is at stake, there is no real need to make efforts towards knowing the particular details of the ATC language

because the underlying translation will become ideally completely transparent.

For those who are really interested, don't let the supposedly low-level nature of ATC make you feel uncomfortable with the language. User experience has been very positive so far. It is worth mentioning that due to this low-level nature, a complementary technology named GATC has been developed around it which leverages much of ATC's implicit power and enhances user productivity. A GATC user guide will also be published shortly.

It is assumed that readers have grounds on at least one OO programming language, preferably Java, and know something about Model-Driven Engineering, and in particular Model-Driven Architecture (MDA), concretely, some MOF meta-modeling notions. And also about the Eclipse platform and the Eclipse Modeling Framework (EMF) plugin. Eventhough perhaps this would not be really necessary, it might be possible not to be able to completely follow the reasoning and the goals presented here otherwise.

### ***Model transformations: why should I care***

How model transformations fit in the big scheme of modeling and metamodeling is something to have sorted out in order to assess if such a discipline may be useful at all for our interests. Models are usually defined via modeling languages. The definition of these is assisted by metamodels. These in turn are themselves defined by using a common language devised for this sole purpose. This language is said to be a meta-metamodel. This way, any official or custom made metamodels are constructed by relying on the same underlying material. Also, it turns out that this language can also auto-define itself. It is a requirement for its own suitability as a meta-metamodel. This means that the construction of a particular meta-meta-metamodel that would allow us to define our meta-metamodel is not necessary anymore, because the latter can achieve this goal by itself. As a consequence the language used to define each metamodel in our computational system is a single one. Moreover, via this language, metamodels can be uniformly defined, and therefore analyzed (and in general, managed). This meta-metamodel is called the MOF in MDA.

Models are related with metamodels by the principle of conformance. It is said that a model conforms to a metamodel when each and every model element is in turn an instance of any given metaelement/metaclass that belongs to (is defined in) such metamodel. It is also said that the model has a type and that type is the same metamodel the model conforms to. A lot of activities that share a common defining language can be applied uniformly upon models conforming to those metamodels that are defined via the same meta-metamodel. One such an important activity is model transformation. Usually, a model transformation produces an output model conforming to a certain metamodel out from a source model conforming to its own respective metamodel. Of course, this is a simplistic view of the possibilities of model transformations. The important point here is that thanks to the uniformity in the definition of metamodels via a unique meta-metamodel language (MOF in the case of MDA), languages can be created that, for instance, allow us define relationships or mappings or rules that are meant to hold between metamodels. At a finer-grain level, this implies, among others, to describe under which conditions a model element of a certain type must be created or

eliminated or which state it must take. All this is possible regardless of which are the particular metamodels involved in a transformation.

Model transformations are like programs. They describe declarative expressions or imperative statements many of which refer to the contents of the appropriate metamodels. That is why the uniform definition of the metamodels via a unique language is crucial. If that were not so, model transformations would be metamodel-dependent ad-hoc formalisms.

It is also obvious that the definitions of model transformation instances cannot directly refer to the contents of any particular model. They would get stuck only being executable only upon these, thus eradicating generality and applicability over the entire set of all models conforming to the same metamodel. For that reason, model transformations make citations only to metamodels. This is analogous to programs defining types and variables, and treating them symbolically. Instances and actual values are unknown until runtime, after programs have been written and are already under execution.

### ***What you will learn***

After you finish reading this document, you should be able to understand most ATC model instances rather easily. In fact, you should be capable of know the best way to combine the different ATC components to generate instances that represent model transformation pieces carrying the desired specific semantics. Or at least have an idea about where to find that information to fulfill your needs.

You should also be capable of editing meaningful ATC transformations in the EMF's Sample Ecore Model Editor, once you gather the working mechanical essence of ATC. Although this editing task is not the means by which instances of the language were originally meant to be created, it has been done with very complex model transformations such as the translation of some QVT languages into the same ATC. So it is not too much complicated to construct ATC instances that work the intended way. In any case, ATC as a transformation language has been devised with the primary purpose of serving as a middle-layer with which support to higher-level languages can be achieved through translation (traditional parsing + model generation, and then model transformation, etc.). This means that user-friendliness has not always been the primary concern driving its design.

This guide then provides a deep knowledge about ATC, enough to be able to take the right decisions during programming, so efficient ATC transformation instances that represent the intended semantics can be hand-made or generated.

### ***About model transformation language translations***

Given a high-level transformation language *A*, the user first programs and creates transformation instances whose notation complies with the concrete syntax of *A*. Then a translation process converts them to their equivalent ATC code.

As an example of such languages, MDA is promoting a model transformation standard named **MOF 2.0 Queries, Views, Transformations (QVT for short)**. It is a suite composed by three languages, each with its own distinctive properties. This specification is approaching version 1.0. In the meantime several attempts to supporting the QVT languages with more or less success exist. One of our goals with ATC has been to isolate from standard changes during the consolidation and evolution/maintenance phases. Current support for these QVT languages via ATC is based on the UMLX editors available at the MDT project in Eclipse. Such editors produce a QVT model transformation model. This is the model that is translated into ATC via some specific ATC model transformations already developed.

## 2.Introduction

### Context

Open Canarias is currently developing an MDA-based Model-Driven IDE. The name for this tool is **Models for Software Engineering Technologies, ModelSET™** (or MSET) for short [[www.modelset.es](http://www.modelset.es)]. **ModelSET™** comes with a transformation engine called **VTE** (Virtual Transformation Engine), integrated in this environment. This engine is special in the sense that it is potentially capable of executing a diversity of model transformation languages indirectly via translation into the low-level model transformation language named **ATC** (Atomic Transformation Code).

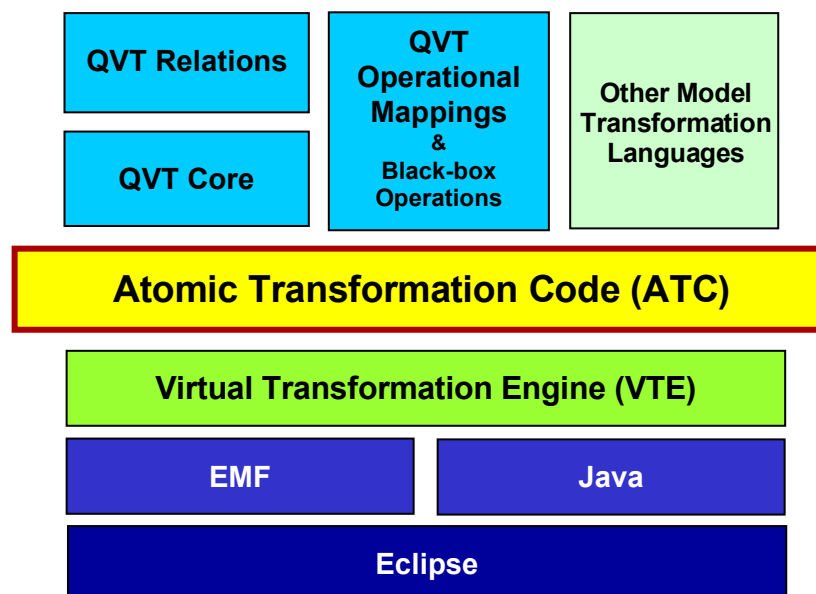


Figure 2.a: Layered Architecture of the ATC & VTE implementation infrastructure

The purpose of VTE is to interpret and execute the ATC language's model transformation instances, so it doesn't directly understand any other transformation language syntax, as implied in Figure 2.a. The user, however, should ideally be able to launch transformations on any model transformation language registered in Eclipse for which ATC translation support has been provided, and the environment should take responsibility of the details transparently.

### **Notes on consensual terminology**

In this document we interpret the terms *Transformation Definition*, *Transformation Specification* or *Transformation Instance*, and even sometimes merely *Transformation*, to refer to the same concept, which is, a transformation that is executable (regardless of whether engines to perform such execution exist or not; nevertheless, we will assume they can exist and they do). The word *model* can precede any of these terms without changing their meaning throughout this whole document, except, if ever necessary, where otherwise noted. In fact, the pair of words *model transformation* can sometimes be used with slightly different meanings, for instance, to refer to the whole discipline related with transforming models. Fortunately, the context in which this pair of words is used helps disambiguate any chance of confusion.

On its part, we will say that the concept *model element* applies to a node of the graph a model (in particular, an ATC model transformation) represents. This is to say that a model element is an instance of a specific metaclass. This distinguishes model elements from mere data type instances. The latter can be seen as values (in a broad sense, not just numbers or character arrays) that can contribute to the state of the former. We also can consistently get rid of the word *model* before *element* without changing its meaning, that is, to refer to the same metaclass instance concept unambiguously.

## **3.The ATC Language**

### **ATC description**

As stated above, ATC stands for **Atomic Transformation Code**. Placed upon VTE, it is an imperative low-level language that lets us define model transformations the engine will execute in a very efficient way. It is essentially a dynamically typed language that deals with models, their interrelations, their creation, deletion, modifications, etc.

In particular, what concerns with dealing with types found in the involved metamodels are treated dynamically, except when we want to reinterpret them as usual fundamental types such as integers or strings. In that case the ATC type system comes into play. Although it is not being enforced for the time being, the language has means to accurately specify the types of formal parameters in operations, and it is recommended to do so, since experience tells that almost all runtime type incompatibility errors identified in practice originate at

parameter passing, and logs of transformation executions help quickly identify such inconsistencies.

So far, no model-to-text conversion capabilities are present in the ATC language. This means that ATC only handles models. And it does so through a metamodeling infrastructure. So any model suitable to feed an ATC transformation must always be conformant to a metamodel. This metamodel is required in turn to be registered in the metamodeling environment used, which in our case is the *Eclipse Modeling Framework* (EMF).

ATC's low level gives it the status of a *byte-code* language, its instructions being just atomic building blocks to define model transformations which come from two sources: a turing-complete set of instructions to mimick traditional programming languages and a small set of model-related instructions to define model queries and modifications:

- It offers the fundamental **imperative constructs** that make up an imperative language, such as conditional branchings, loops or arithmetic operations. Their implementation relies on the same constructs found in the Java language. Among these instructions are also included fairly complex type manipulations such as a non comprehensive support to the `java.util.List`, `java.lang.String`, or even `java.util.Map` API.

- It offers building blocks specifically aimed towards supporting model queries and transformations. It would be convenient to define what the action of model modification entails at the implementation level. Some say that it includes creation, modification and deletion. This is a fairly high-level view which can be enough for our discussion. The **model-related ATC commands** are in fact more granular. The implementation of these commands is based upon the EMF API.

### **Abstract Syntax**

ATC has itself been built around the metamodeling concept, which is its internal, abstract representation. This means that any ATC transformation can be seen as a model on its own. The ATC metamodel is currently based on the EMF's **Ecore** meta-metamodel. Ecore is equivalent to OMG's *MetaObject Facility* (MOF), more concisely it presently covers the *Essential MOF* (EMOF) specification, which is a subset of the *Complete MOF* (CMOF). EMOF is replaced by Ecore in our infrastructure. This infrastructure depends on the Eclipse platform, which is a multipurpose plugin-based Java IDE, so for the moment, and given the fact that we want to integrate ModelSET in the Eclipse environment, our engine implementation is currently coded as a set of Eclipse plug-ins written in Java, as previously mentioned. All this layered architecture is summarized in Figure 2.a.

Special care has been taken in trying to avoid any kind of coupling with Java or EMF from the language instruction set design standpoint. This assumption has not been tried yet in practice, i.e., the ATC infrastructure has not been ported to other metamodeling infrastructures, such as MDR or other programming languages instead of Java. Therefore, the true neutrality of the ATC approach remains untested and cannot be guaranteed. Nevertheless, it could be argued that EMF is a de-facto metamodeling infrastructure

standard, and Java is currently well established as one of the most used programming languages.

Currently, the ATC metamodel consists of some packages each with several class hierarchies defined inside. Some of them transcend the intrapackage containment. As expected, the ATC metamodel reflects quite well the activities encoded in the language in terms of commands or instructions. Usually one metaclass is defined per command, although there are some command-related metaclasses that embody several modes of operation.

An ATC Transformation definition is expressed as a graph of objects related together in a hierarchical way, where each object is an instance of an ATC meta-class. That is, this transformation definition constitutes in fact a model that conforms to the ATC metamodel. So transformations in ATC can be defined by using only static constructs found in its abstract syntax. ATC is an imperative language for describing behaviour, what in UML can be represented by (a combination of) diagrams that hold dynamic information, such as the Sequence Diagram. The metamodel contains and describes static concepts (and their interrelations as well) whose precise attached behavioural semantics is currently defined separately.

In particular for ATC, a *transformation definition* comprises a root element whose type is the ATC metaclass **AtcTransformation**. This root element will in turn contain a series of model elements in a graph fashion. Although it is allowed by EMF it is not advisable to define more than one single root element on an ATC transformation specification file and we currently don't guarantee execution support for such multiple definitions.

### **Concrete Syntax**

The object oriented scheme chosen for the ATC implementation has two direct consequences regarding its concrete syntax. As ATC instances manifest themselves as models, they can be modeled or inspected with traditional modeling tools (in the event that this activity would be required). We have already a **graphical concrete syntax** at our disposal for these kinds of situations, namely the UML class diagram. We can choose the basic EMF editing facilities (the **Sample Ecore Model Editor**) as a modeling tool to handle, inspect and modify an ATC file. Although some may argue that it is a little bit rough, it comes at a zero cost entry point. Text output attached with each node has been reprogrammed to show understandable and meaningful information, which is very useful to understand transformations and also an unvaluable guide during the creation or modification of any piece of ATC code. It could also be possible to edit ATC models in a more graphical approach, by means of the Eclipse's **GMF** (*Graphical Modeling Framework*), but support for this has not yet been explored.

## ATC First Contact: a Model Transformation Example

As an example of how an ATC transformation looks like in the Sample Ecore Model Editor you can check figure 3.a. This example reveals many things at once. As already stated, an ATC transformation is a singly rooted graph whose root is an instance of the **AtcTransformation** metaclass. This is a named element. Its name in this particular example is `TransformationExample`.

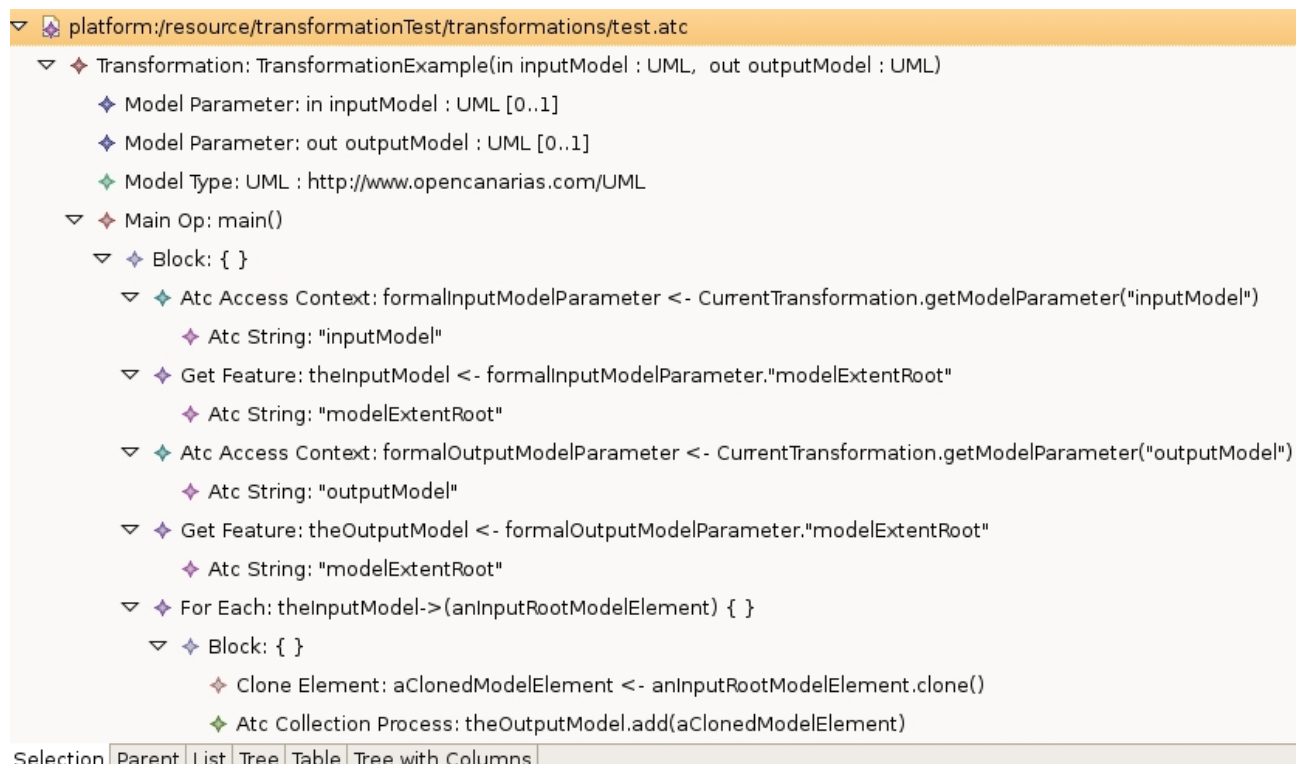


Figure 3.a: Example of an ATC model transformation

An ATC model transformation basically contains three different types of elements:

- **model types**, which represent the metamodels to which involved models must conform to
- **model parameters**, which represent the formal specification of the models that will participate in each execution of the transformation. Their type must be specified along with their definition. Such a type will be one of the model types defined in the transformation.
- **operations**, these are the entities where the main semantics of the transformation is encoded. There are basically two different types of operations in ATC: **AtcMainOp** and **AtcFunction**. *AtcMainOp* is meant to become the entry point of the transformation execution. Once created and available for the transformation, it must be assigned in the

AtcTransformation `main` reference<sup>1</sup> to really be installed as the actual entry point. Several `AtcMainOps` can be defined in the same transformation. *AtcFunction* differs from *AtcMainOp* because it cannot be set as the transformation execution's entry point, and because it may define input or output parameters, similar to any arbitrary method in an OOP language. Such parameters can refer to instances of Data Types (which includes Enumerated types) or Classes, and they can be mono- or multi-valued. Naturally, these also include the ATC built-in type system, since it is defined also as classes and datatypes in its metamodel.

This example only contains a single main operation and no functions. The main operation's body is an **AtcBlock**, a concept meant to contain an ordered collection of instructions to be executed sequentially.

**Model parameters and model types:** An input model is defined as an **AtcModelParameter** that conforms to a certain **AtcModelType** (in the sense of metamodel) named UML with name `inputModel`. This model type does not need to be the well known Unified Modeling Language. It can be a reduced subset or does not even have to be related with it at all.

Another model is involved in the transformation, defined by the *AtcModelParameter* named `outputModel`. Incidentally this model parameter shares the same model type as the input model. This fact is circumstantial. It has been programmed like this just for simplicity. Usually each model type will be used by its own model parameter in a given transformation. For instance, in transformations that map two different domains, such as the often mentioned UML2RDBMS model transformation, which, among others, relates UML classes and packages with Relational Database Management Systems' tables and schemas.

ATC supports the definition of any number of model parameters. This means that not only 1-to-1 model transformations are definable in ATC. M-to-N model transformations are also supported. An undefined number of transformations involve is possible but it is currently under study. Each model parameter can be assigned its own *direction kind* in ATC, which indicates if they are meant to be used in read-only mode, read-write or just write, respectively, in analogy with the handling of traditional files and buffers. This is borrowed from the QVT Operational Mappings language.

**ATC Atom Types:** Now it seems a good time to comment on the ATC instruction set definition at the metamodeling level. Several class hierarchies are identified in the ATC metamodel class diagram, as stated in Section 4. We will take special attention to a particular tree of such metaclasses, the one that subclasses directly or indirectly from **AtcAtomType**. Subclasses of *AtomType* are considered the ATC metaelements that are related with the language instructions. These are called the ATC **atom types**. Their

---

<sup>1</sup> In this document, and in contexts such as this, *reference* is a term used as synonymous of the `EReference` concept found in EMF's `Ecore`. It is similar to UML's `Property`, but it only applies to properties whose type must be an `EClass`, that is, a metaclass. The complementary concept is `EAttribute`, whose type can only be an `EDataType`. We will refer to that with the term *attribute*.

instances are thus called ATC **atoms**. That is, they are runtime objects representing executable semantic units or building blocks. Atom types can be considered the counterparts of programming language instructions. In this sense, they determine the semantic granularity of the ATC language. *AtcBlock* also belongs to the *AtcAtomType* hierarchy, and is the usual choice for a functional operation body.

We say that ATC model elements whose type inherits from **AtomType** are ATC **atoms** in the sense that they are bound to a given minimalistic, byte-code-like, atomic semantical functionality attached to the type to which they belong. So a concrete ATC atom, and in particular, its associated metaclass, possesses a transformation semantic information that distinguishes itself from other types.

Back to the example in Figure 3.a, it is rather easy to follow the meaning of the ATC atoms that belong to *AtcBlock* and make up the body of the *AtcMainOp*. The semantics of this transformation example is simply explained as follows:

*traverse the source (input) model and copy it into the target (output) model.*

There are several things involved in this example worth discussing. For instance, often ATC atoms compose other atoms that provide them with certain information required for proper execution. Such information may be static or dynamic. Dynamic information provides a lot of flexibility in the definition of common operations programmed in ATC. In the example we see several **AtcString** atoms that provide their respective parents with a String literal value. These, of course, are examples of static information being fed to some atoms via subordinates.

Another very important aspect of the ATC language reflected in the example is that models are seen as lists of model elements, as can be deduced from the iteration of `theInputModel` carried out by an **AtcForEach** atom. These model elements act as the set of model roots. Usually there is only one root element contained in the list. This has the consequence that whichever list containing model elements is considered (which could be a branch in a model, or even a single model element by itself, can be considered as nearly models in their own rights). Moreover, models involved in a transformation execution may be obtained as parts of bigger models. That is why ATC does not say that a model is a model. Instead it considers it a *model extent*. This model extent is defined by the list of root model elements it is composed by, and in an ATC transformation it is stored inside a formal parameter, the corresponding *AtcModelParameter*, concretely, in the `modelExtentRoot` reference.

In order to make a model (the list of root model elements) in the pool of local variables of a functional operation such as an *AtcMainOp* we need to first access its associated model parameter. This is achieved by means of **AtcAccessContext**. Once the model parameter is assigned to a local variable (e.g.: `formalInputModelParameter`), it can be queried for the value of the feature named `modelExtentRoot` via **AtcGetFeature** to obtain the model

extent root in another local variable (e.g.: `theInputModel`). Unless a special atom type is used to indicate otherwise, all variables are local to the functional operation in which they are (dynamically) defined and used.

The variable `theOutputModel` is being updated inside the **AtcForEach** loop by means of **AtcCollectionProcess** programmed with the **add** mode. Since that variable is (a Java reference to) the output model, this atom is effectively altering the contents of the output model. Such contents will be conveniently persisted at the end of the transformation execution, thus effectively changing the output model contents in the system. In fact, it being an output model, it is supposed to be empty or nonexistent before execution, and it is expected that it may potentially earn some contents after execution (depending on whether the source model was or wasn't empty). **AtcCloneElement** performs a deep copy of the source model element (*`anInputRootModelElement`*) into the *`aClonedModelElement`* variable.

### **More on the ATC's Concrete Syntax**

Models and in particular class diagrams, can be described in files with a specific textual format. The contents in any of these files would reflect all the information about positions of diagram elements, etc., along with the really important information concerning the model contents. In our case, ATC instances are filled with ATC object states and relationships. Of course, text for this representation can adopt multiple syntaxes. XMI is an interesting one. It is an OMG standard for inter-tool data interchange, and is the format adopted by EMF as the canonical default storage for models.

Of course, XMI is capable of capturing all the information that needs to be persisted, so again by using EMF we have the prize of obtaining direct textual persistence support for free. Given the fact that ATC is not intended for human use, we don't currently plan to create a particular suitable, user friendly, textual concrete syntax for describing ATC instances/models in a more compact and readable fashion, so we rely on the EMF's XMI notation. Thus, presently, the ATC **textual concrete syntax** is provided by the EMF's **XMI** implementation, which currently supports version 2.1. Another way of approaching this subject would be to consider the ATC's concrete syntax that of the text output shown per node in the EMF Editor. The relationship between the contents of XMI model instances and the ATC objects they describe are straightforward, given the XMI nature as a descriptor for arbitrarily structured data.

The XMI version of the *TransformationExample* ATC model transformation example can be found on Appendix A. It is important to know that default values are optimized and left out of the text. The XMI specification allows this. Thus, when some information, basically attribute values seem to be missing from the XMI text, their default value upon model load will be that defined in the corresponding metamodel.