SCOS-2000
Technical Note

MDA Study Prototyping
Technical Note

Document Status: Issue 1.0
Prepared By: Eugenio Zanatta
Date: 2004-07-26
<table>
<thead>
<tr>
<th>Action</th>
<th>Name</th>
<th>Date</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepared by</td>
<td>Eugenio Zanatta</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verified by</td>
<td>Gianluca Montroni (TOS-GIC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Eduardo Gómez (TOS-GIC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approved by</td>
<td>Eduardo Gómez (TOS-GIC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authorised by</td>
<td>Mauro Pecchioli (H/ TOS-GIC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Néstor Peccia (H/TOS-GI)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Distribution list**

<table>
<thead>
<tr>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESA/TOS-GIC</td>
</tr>
<tr>
<td>SCOS-2000 Industry Team</td>
</tr>
<tr>
<td>SCOS-2000 Document Library</td>
</tr>
</tbody>
</table>

© COPYRIGHT EUROPEAN SPACE AGENCY, 2004
The copyright of this document is vested in European Space Agency. This document may only be reproduced in whole or in part, stored in a retrieval system, transmitted in any form, or by any means electronic, mechanical, photocopying, or otherwise, with the prior permission of the owners.
## DOCUMENT CHANGE LOG

<table>
<thead>
<tr>
<th>Issue/Revision</th>
<th>Date</th>
<th>Modification Nr</th>
<th>Modified pages</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>26/07/2004</td>
<td>All</td>
<td></td>
<td>First issue</td>
</tr>
</tbody>
</table>

MDA Study Prototyping
PAGE ISSUE RECORD
Issue of this document comprises the following pages at the issue shown

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table of Contents

1 INTRODUCTION ................................................................................................................... 6
   1.1 SCOPE........................................................................................................................................ 6
   1.2 PURPOSE....................................................................................................................................... 6

2 MODEL DRIVEN ARCHITECTURE .................................................................................. 6
   2.1 MDA AND EXISTING SOFTWARE .......................................................................................... 6

3 MDA TOOLS............................................................................................................................... 7
   3.1 TOOLS CHARACTERISTICS ........................................................................................................ 7
      3.1.1 Kennedy Carter iUML............................................................................................................. 7
      3.1.2 Sodifrance Model In Action ..................................................................................................... 8
      3.1.3 Telelogic Tau Generation 2 ..................................................................................................... 8
   3.2 TOOLS INTERPRETATIONS OF MDA .................................................................................... 8

4 TELELOGIC TAU G2 IN THE PROTOTYPING PHASE ................................................. 9
   4.1 MODELLING PHASE .................................................................................................................... 9
      4.1.1 Use–Case Diagram .................................................................................................................... 9
      4.1.2 Class Diagram .......................................................................................................................... 10
      4.1.3 Sequence Diagram ................................................................................................................... 12
      4.1.4 Statechart Diagram .................................................................................................................. 13
      4.1.5 Architecture Diagram .............................................................................................................. 17
   4.2 GENERATION PHASE ................................................................................................................... 17
   4.3 COMPILATION PHASE .................................................................................................................. 18
      4.3.1 Problems during compilation ................................................................................................... 19
   4.4 SIMULATION AND DEBUGGING PHASE .................................................................................. 19

5 PROTOTYPE APPLICATION ...................................................................................................... 21
   5.1 PURPOSE OF THE PROTOTYPE ............................................................................................... 21

6 CONCLUSIONS ............................................................................................................................... 22
   6.1 MMI.............................................................................................................................................. 22
   6.2 LEARNING ..................................................................................................................................... 23
   6.3 APPLICATION SIZE .................................................................................................................... 23
1 INTRODUCTION

1.1 SCOPE
This document constitutes a Technical Note on the activities performed in the prototyping phase of the MDA and UML2 Study work package.

1.2 PURPOSE
The purpose of this document is to provide a technical report on the work done to build the prototype and related issues in the experimentation of the MDA methodology and UML standards, with the particular focus on the integration with the existing SCOS-2000 system. To perform this task the Telelogic Tau G2 tool was used.

2 MODEL DRIVEN ARCHITECTURE
MDA is a set of OMG standards that provide a comprehensive way to specify, develop, maintain and deploy software using formal, traceable models and model transformations. MDA technology is based on the creation of models. These models contain all the information the system needs to perform the task it is created for. MDA establishes that the models can be represented exhaustively by the UML diagrams. MDA establishes also a process to follow. The development process is divided in two main parts, still represented as models:

1. the Platform Independent Model (PIM);
2. the Platform Specific Model (PSM).

The “Model” of the application is created from the most abstract and essential definition in the PIM as a set of UML diagrams. The UML diagrams are also used to add information to the model. When all the functional information is in the model, a transformation to a PSM has to be defined. The transformation to the PSM only deals with how to map the model to a source code. This transformation is automatically made from an MDA tool, which can convert the model into a source code of a selected programming language. The transformation of this code to a running code is made through a mapping of the actions, to platform specific APIs.

2.1 MDA AND EXISTING SOFTWARE
What is just described is the whole process to use when a development has to be done from scratch. The MDA can also be applied to further develop already existing software. To achieve this result, first an analysis of the existing software has to be done. In fact, to get a real benefit from MDA, it is important to model the functionality of the system in the PIM. The method is of course based on the logical decomposition of the system in components. Through the MDA practice the interfaces among
the components can be modelled and the method can be eventually iterated to the components, while it is already available an overall behaviour of the system in the abstract model.

This approach can also be followed when the knowledge of the system functions is not complete. The decomposition of the system leads to the complete understanding of its logical model. The model is abstract, in the sense that it has to capture only the functional information of the system. During the decomposition the designer adds some information to the model: this information cannot be retrieved from inside the existing code, so the tool does not provide this functionality.

The OMG is aware of the limitations of the existing MDA-related standards regarding reverse engineering and modernization of legacy software. A new working group for Architecture-Driven modernization has been established with the objective of “Create specifications and promote industry consensus on modernization of existing applications” (http://adm.omg.org/, last accessed 26-July-2004).

3 MDA TOOLS

The MDA process is based on the existence of an MDA tool. To be an MDA tool, a tool has to be able to handle the PIM of the application completely, being able to make it run, even though in a simulated environment. This is a guarantee that all the business logic is inside the PIM.

The MDA realizes a further step in the abstraction of the programming from the machine. In MDA the UML diagrams can be considered the real source code of the application. A tool has to be able to generate traditional source code starting from a model of the system. The transformation to a code in a traditional programming language is kept, to take advantage from the already existing compilers and libraries, which are able to create very efficient machine code and allow integration with other systems.

A very important characteristic of an MDA tool is that it must be technology independent (using the OMG standards UML and CORBA created right for this purpose); the PSM is the only type of model where the tool can use specific mechanisms or technologies.

The various MDA tools can add to the basic functionality a drawing environment for the designer, the capability to make automated checks over the model, and the integration of compilers and libraries.

3.1 TOOLS CHARACTERISTICS

A brief investigation of some tools has been performed, to better understand the commonalities on the MDA and how they put it in practice, identifying also the differences, specific of each tool.

3.1.1 KENNEDY CARTER iUML
The tool is compliant to the version 1.5 of UML. It presents an IDE from which the designer can draw the major UML diagrams:

- Use Case Diagram
- Component Diagram
- Sequence Diagram
- Class Diagram
- Collaboration Diagram
- Statechart Diagram

The details of the system’s behaviour can be defined through the combination of Sequence Diagrams and a sort of script language, the Action Semantics Language. The tool has also the ability to manage the developers as users and groups, and it can perform a concurrent access control.

It seems that the tool was born on a Unix platform, generating C++ code for the GNU compiler. The windows version needs an emulation environment to run the model inside the IDE.

3.1.2 SODIFRANCE MODEL IN ACTION
This tool does not provide any facility to draw UML diagrams. It is able to import the diagrams generated from other specific tools like Rational Rose or Together in the proprietary format or in XMI. The tool has a well customisable code generator, thanks to the use of the Java platform, but generates only Java code.

3.1.3 TELELOGIC TAU GENERATION 2
This tool has the major characteristics to be UML 2 compliant and to have a full IDE, within which the designer/developer can draw the diagrams of the model, execute the model and generate the run-time application. The language used for the generated code is C or C++. This tool is the one used to perform the prototype activity related to this study, so it will be analysed in detail later.

3.2 TOOLS INTERPRETATIONS OF MDA
The differences among the tools are due to the definition of MDA itself. Each vendor can implement very different tools, therefore different development processes, while adhering to the MDA standard. Kennedy Carter’s iUML is based on the representation of the sequence of actions, performed during the system's life, essentially as a Sequence Diagram. To capture a full comprehensible concept of the actions flow, conditional notation is used so that the translation to a high level abstract language, the ASL, is possible.

In Telelogic Tau the system's behaviour is mainly captured by Finite State Machines. These machines receive and send signals, that correspond to the actions the Kennedy Carter tool deals with. The focus of this tool is more on states and interactions that change these states, than on a procedural representation of the implemented process.
For the differences shown above, the tool choice plays a big role on the developer's activities and know-how.

4 TELELOGIC TAU G2 IN THE PROTOTYPING PHASE

4.1 MODELLING PHASE

UML can be used in many phases of the software production activity, like planning, design, implementation, documentation and so on. Before MDA came out, UML was used mainly for documentation purpose and as a standard way of representing the design. The UML graphic notation was created to allow all people, technical and non-technical, to understand the software documentation without ambiguities.

With MDA, the UML is used for “developing”. It is particularly important for the MDA to understand how to use the constructs of the UML during the design phase. For the various needs of the design it is important to know when to use a particular construct or a diagram.

The modelling in Telelogic Tau is based on the definition of the state machines.

4.1.1 USE–CASE DIAGRAM

The Use-Case Diagrams are created at the beginning of the modelling phase to define the functions the system has to perform. For a simple system like the one used for this study, only one Use-Case Diagram is enough. The diagram is shown in Figure 1.

![Use-Case Diagram](image)

Figure 1; Use-Case Diagram

For more complex systems, many diagrams can be produced to clarify the services provided to the different actors. Some complex Use-Cases can be decomposed in sets of related Use-Cases at a more detailed level. The Use-Case Diagrams have a great importance in the documentation production, in
fact they capture the concept from the perspective of the user. In a top-down development all the subsequent development depends on the correctness of these diagrams.

4.1.2 CLASS DIAGRAM

The Class Diagram provides the most important static view of the system. In Telelogic Tau other entities are defined as classes: interfaces and signals.

In Figure 2 the two interfaces used by the system with the external User are shown. The signals will be shown after.

![Figure 2: External Interfaces](image)

Telelogic Tau makes a distinction between active and passive classes. The active classes are those classes with functional behaviour’s responsibility. The passive classes have as main role the maintenance of the information in their attributes. Telelogic Tau tool uses the Class Diagram also to distinguish the “active classes”; they have the two added vertical lines at the lateral sides of the rectangles as shown in Figure 3.
At this level of abstraction, the Class Diagram does not make any distinction among classes belonging to the system and the others, like the “User” class. The role of the class User is better shown in the Use-Case Diagram shown in paragraph 4.1.1 and in the Sequence Diagrams shown in paragraph 4.1.3. The diagram also shows a relation between the User and the “ToUser” interface. This dependence is defined only for documentation purpose and with different notation (dashed line) because the User is not subject to any designed activity, here is only expected that the User implements some way this interface.

4.1.2.1 MISChandler
The MISChandler class represents the complete system. A functional decomposition of this system leads to consider it as an aggregate of two classes: the Controller and the MISCfunctionProvider. The MISChandler has a communication port which has a bi-directional interface with the User. The “ToUser” interface is required; this means that the external User” has to implement it. For example, if a User is a person and the “ToUser” interface is a computer’s monitor, the person should be able to receive the information looking at the monitor. From the system point of view, the “FromUser” interface is more important than the other one, because it is the system’s responsibility to implement it. This difference is shown by the different symbol, a filled circle instead of an open arc.

4.1.2.2 Controller
As shown in the Class Diagram of Figure 3, the controller owns all the interfaces of the system with the external User. Thus the Controller is the front-end of the system.
4.1.2.3 MISCfunctionProvider

The MISCfunctionProvider has the responsibility of the back-end computation. The computation is already performed by the existing system, so this class is a wrapper of the SCOS-2000 components, which actually perform the tasks.

4.1.3 SEQUENCE DIAGRAM

The Sequence Diagrams are normally used at a high level of modelling, after the definition of the use cases. Normally the Sequence Diagrams are created in conjunction with the Class Diagrams in an iterating process. While defining the behaviour of the system in the Sequence Diagrams, new classes are required and added to the Class Diagrams. Updating the Class Diagrams, new responsibilities are assigned to the classes and then used in the Sequence Diagrams. In an Object Oriented development, normally the beginning this process is started with the Class Diagram.

The Sequence Diagrams are used for the definitions of the interactions among the actors and the system. In the modelling phase, while dividing the system into components, the Sequence Diagrams are used to define the interactions among components. In further decompositions, the objects shown in a Sequence Diagram can be instances of some classes. In this case the decomposition to the class level can be made in only one step.

For each Use-Case one Sequence Diagram was created, but in general, more than one Sequence Diagram can be created for a single Use-Case.

![Sequence Diagram for Use Case "Change Variable"](image)

The sequence of operations made by the system to retrieve a variable was made more complex, to make it more interesting and representative of possible situations.
The Telelogic Tau tool uses the Sequence Diagram during the Simulation and Debugging phase, described at paragraph 4.4.

4.1.4 STATECHART DIAGRAM

The system is modelled through state machines. A state machine is described by the set of its interfaces, which is used in the static representation, and the Statechart Diagram, which captures the dynamic representation. In a Statechart Diagram of a class, the states, the actions performed for the messages (“Signals”) received or for the changes of the states are defined. A state machine can be recursively decomposed into other state machines, some of which having a subset of their interface being part of the aggregate system's interface; the remaining part of their interface is connected to the corresponding interfaces of the other state machines inside the system. In UML, and in Object Oriented Programming in general, the smallest state machine is the class, because if a smaller portion of it can be isolated and treated as a state machine, it would be better to separate it and eventually decompose the previous class in various smaller classes.

In the top-down approach, first the user point of view is considered, then the internal details of the system. So the system from outside is defined by the interfaces it presents to the user and the “Signals”
exchanged with it. In Figure 6 is shown a diagram where the “Signals”, accepted and sent by the system, are defined.

**Figure 6; Interface Signals**

These signals are the same accepted and sent by the Controller class, as already explained. To define the internal behaviour of the Controller class as a state machine, it is necessary to know the possible interactions this class has with the rest of the system, in this case the MISCfunctionProvider class.

**Figure 7; Internal Signals**
The internal signals shown in Figure 7 do not belong to any interface; the complete classification of the internal signals is done in the Architecture Diagram explained in paragraph 4.1.5. The behaviour of the class Controller is defined in the Statechart Diagram of Figure 8.
Figure 8; Statechart Diagram for Controller
The diagram shows all the possible states the Controller class can have, and all the possibilities to move from one state to the other possible states. The tool provides some aid during the creation of the diagram, performing a formal check on it.

Analysing the diagram one can notice that there is no possible signal able to change the Controller state from “Idle” to “Idle” again. This kind of analysis can be useful while validating the model. The right analysis of the possible states for an active class is very important for the correctness of the system behaviour. If an error is introduced in such a diagram, the problem determination and correction can be very fast.

4.1.5 ARCHITECTURE DIAGRAM

The Telelogic Tau tool has the possibility to capture the internal communication of the classes through an Architecture Diagram. With this diagram, the designer can explicitly establish the signals that pass from a class to another. In the current system the only possible communication is between the classes Controller and MISCfunctionProvider. The diagram is shown in Figure 9.

![Figure 9; Architecture Diagram](image)

The tool allows the designer to assign “ports” to the active classes, specifying the signals each port can receive and send. Through this diagram the tool is able to connect the classes to each other, and the state machines, to produce the code, which implements the model.

4.2 GENERATION PHASE
The generation phase produces the source C or C++ code corresponding to the created model. This phase requires tuning for the target environment.

The model of the state machines is rich enough in information to generate the PSM. It is possible to set an active class as multithreaded or not, and it is also possible to define timers.

The integration with legacy system is made through well defined interfaces. So the definition of the transmission protocol layer, like CORBA, or the use of a particular library is driven through interfaces of the core of the application, handling the business process, and its boundary. The code that establishes how to handle the external libraries, has to be made manually, but the steps of the related transformation, from the model to the code using this predefined library is performed automatically.

The environment is modelled as what is outside the border of the application, and it is also what is not under control of the system. In the simulation or debugging phase the environment is automatically generated by the tool, which provides the developer to send the messages to the system. When the system will be in the real environment, a mapping between each call coming from outside the system and the messages the system can receive has to be provided manually.

In details, a unique file, env.C, contains four main functions:

1) init;
2) close;
3) inEnv;
4) outEnv.

These functions define how the environment behaves at the start-up and shutdown of the application, and how it behaves when a message passes from the environment to the system (inEnv) and vice versa (outEnv).

4.3 COMPILATION PHASE

In this phase the tool uses the selected C or C++ compiler to create an “artifact”. The artifact term is used in Telelogic Tau to identify a code generation. The same model can produce different codes using different artifacts. The main difference of the various artifacts is the “code generator” used. It is also possible to specify different main classes, thus letting the developer to execute only subsets of the whole system. The artifact for the simulation environment was created. In this environment the model can be executed and debugged.

To have the system run outside this environment, i.e. in the real environment, some code has to be written. A code generator, to provide the interaction of the outside world with the system, will use the hand made code as a plug-in. All the proprietary APIs, utility routines and connections with external software, either through CORBA or any other technique, has to be put at this phase.

In the simulation environment the interaction with the outside world is automatically created, because the outside is simulated by the tool and displayed as UML diagrams.
4.3.1 PROBLEMS DURING COMPILATION

The version of the Telelogic Tau G2 received by ESOC is 2.2.01 for Solaris 8. This version of the tool has a problem with the compiler. The compiler used in this study is the Forte6 compiler called Sun Workshop 6 Update 2, which is used for the current versions of SCOS-2000. Telelogic declares to support this compiler on Solaris 8. Some time was spent investigating the problem with the help of Telelogic personnel, without success.

The updated version 2.2.51 seems to work on the same machine, with neither any change in the Operating System nor in the Compiler or configuration.

4.4 SIMULATION AND DEBUGGING PHASE

The simulation and debugging phase allows the user of the Telelogic Tau tool to run the model in the same graphical environment where it was developed. The tool allows the tester to put breakpoints on the states and signals of a Statechart Diagram. To execute the generated code, the tester has to specify the signals sequence inside an ordered list. Telelogic Tau sends the signals to the system in chronological order.

In this prototype the signals belonging to the “FromUser” interface are used. The Telelogic Tau tool uses the Sequence Diagrams also during the simulation and debugging phase, so the tester is able to follow the operations flow either inside each active class, in a Statechart Diagram, or among the classes, in a Sequence Diagram.

The diagrams in Figure 10 and Figure 11 show the generated Sequence Diagram for the tested signals list.
Figure 10; Sequence Diagram of the Execution Flow, part 1

The tool generates the Sequence Diagram step by step in dependence of the signals specified in the list. In the diagram shown in Figure 10 and Figure 11 the test signals list is:

1. GetVariable
2. ChangeVariable
It is important to notice that in this case the signals, even though in a certain order, are sent to the system in unknown times. Only when the system is in a state where it is able to accept the next signal, it executes the appropriate actions. The only state allowed to accept signals from the “From User” interface is “Idle” in accordance with the Statechart Diagrams. In the generated Sequence Diagrams it is possible to recognize the same sequence of actions defined in the diagrams in Figure 5 and Figure 4.

5 Prototype application

5.1 Purpose of the prototype
The purpose of the prototype is to use MDA to test a development inside the S2K framework environment. The results of this study are:

- impact analysis of the adoption of MDA development process;
- reusability analysis of the existing S2K libraries with the MDA approach.

The S2K kernel has complex user interfaces and widespread internal client/server interfaces which make use of CORBA and other protocols. It has also some low-level libraries of utilities that implement some standard routines and access functions to the software's resources.

A good evaluation of the MDA should focus on the mapping of the typical utilization of these libraries.

The proposed application is a client application of the S2K MISC subsystem, so that it will interface some existing S2K libraries, using its set of API as well as CORBA.

From this application the user will be able to interact with the S2K system through the prototype's MMI, monitoring the system and even performing configuration changes on it while the system is up and running.

The main functions to be modelled in this application are:

1. reading and reporting queried data;
2. modifying existing data;
3. adding new data;
4. removing data.

The main flows of the functions shown above and some alternative flows have to be modelled, like user input checks, internal system checks and related error management. Beyond the modelling, some attention has to be paid while realizing the tool customisation. This customisation is needed to have the generated code integrate with existing software libraries.

The proposed prototype application interfaces some existing S2K libraries, using its set of API as well as CORBA.

6 CONCLUSIONS

The MDA technology focuses on the business logic modelling, so that it is possible to validate the business process independently from the concrete implementation and from the platform. There is no specification about how to create a model, nor which UML version to use. The choice of UML 2.0 is the vendor's choice. Furthermore, how to use UML is also not specified by MDA, thus for example, the Telelogic Tau uses UML in a different way of what the Kennedy Carter iUML does.

6.1 MMI
The MMI of the application cannot be really created with MDA, but it can be modelled from the system point of view. The approach for modelling the MMI can be likely similar to a state machine, but MDA remains focused on the business logic, allowing the substitution of different implementation of the MMI to a single business model.

6.2 LEARNING

For the start-up time needed to use MDA and an MDA tool, the most important part is to acquire a solid knowledge of UML. MDA is a new technology, so it has to be learned. This technology is made for huge and complex tasks, for this reason also the technology has a certain complexity. This complexity and the new way of working have as a consequence a learning curve to be taken into account. Furthermore, the learning curve is made steeper by the necessity of learning a new development tool, which also reflects the same complexity.

6.3 APPLICATION SIZE

The application size can have an impact; in fact in very small projects the effort needed, starting from scratch, is not competitive with the traditional software development. Only after the achievement of some development infrastructure and experience, also small projects can take advantage from the MDA.