



Information Technology for European Advancement

# Guidelines for exploiting formal methods in the tools to be developed in WP2

## D 1.7.5

Version 02 - Public

Edited by Luigi Lavazza

# Software Development Process for Real-Time Embedded Software Systems (DESS)

**ITEA COMPETENCES involved:**

- 1) Complex Systems Engineering**
- 2) Communications**
- 3) Distributed Information and Services**

December 2001

## **Release note**

Version 1.0, November 2001

Version 1.1 December 2001

# Table Of Contents

<b>1. INTRODUCTION.....</b>	<b>4</b>
1.1 TASK 1.7: FORMAL METHODS.....	4
1.1.1 Goals .....	4
1.1.2 Methodology.....	4
1.2 OBJECTIVES OF THIS DOCUMENT.....	4
1.3 APPROACH .....	4
1.4 STRUCTURE OF THE DOCUMENT .....	5
<b>2. FORMAL LANGUAGES AND UML .....</b>	<b>6</b>
2.1 FORMAL LANGUAGES AND GRAPHICAL NOTATIONS.....	6
2.2 UML EXTENDED .....	6
2.3 VERIFYING REAL TIME SYSTEM USING EXTENDED UML.....	7
<b>3. DEVELOPMENT PROCESS .....</b>	<b>10</b>
3.1 DESIGN TOOLS.....	11
3.1.1 ArgoUML.....	11
3.1.2 Rose/Esterel Studio gateway.....	14
3.1.3 Rhapsody/Esterel Studio gateway.....	15
3.2 USABILITY REQUIREMENTS .....	15
3.2.1 External tool .....	15
3.2.2 Workflow.....	16
<b>4. REFERENCES.....</b>	<b>17</b>

# 1. Introduction

## 1.1 Task 1.7: Formal Methods

### 1.1.1 Goals

The DESS methodology should be easy to use, because this is important for adoption by industrial developers. However, a methodology based on purely informal definitions would be hardly usable for supporting highly desirable functions such as proving properties, or simulating the run-time behaviour of the specifications. The goal of this task is therefore to provide a formal background that guarantees that the methodology is sound and reliable.

Formalization should be exploited both in the analysis and in design and implementation phases. Generally, formal methods are helpful in the analysis phase, typically to prove desirable properties of the specifications. Nevertheless, the benefits of formal methods can be extended also to the design and implementation phases. While code generation issues are already covered in task 1.5, a useful complement to such activity consists in enabling designers to formalise architectural models and their properties.

### 1.1.2 Methodology

The strategy is to adopt *existing* formal methods (such as temporal logic, timed automata, etc.). However, in order to make these methods accessible by most developers, a relatively simple front-end notation will be developed. This notation is a suitable real-time extension of UML [1,4].

Task 1.7 will also include the study of how formal methods can be incorporated in the tool(s) to be developed in workpackage 2 in a way that is as transparent as possible to the final user. I.e., we want to provide the user with the benefits of formal methods without overwhelming him/her with the burden of learning difficult concepts. Moreover we will focus on the ability of these methods to deliver back to the user valuable diagnostics instead of a simple go/no go answer. Links between the two aforementioned ways of introducing formal methods will be researched : we propose to explore the capability of compositional approach for processes “timing demand” as a counterpart to the compositional method for OO software engineering.

## 1.2 Objectives of this document

This document addresses the specification of abstract requisites for tools that will include and use formal methods to specify real-time systems. It also describes the development process that should be followed to implement the requirements in the chosen tools.

## 1.3 Approach

Formal methods have been proposed and successfully employed in the development of several real-time software systems, where features like safety needed to be formally proved. However, formal notations are generally considered too difficult or too expensive to be used in “ordinary” real-time software development, and it is a matter of fact that they are not widely employed.

This document analyzes the relations between the UML semi-formal specification and formal languages, to understand how to modify UML specification to integrate formal graphical notations to express the formal language of choice.

Understanding how and where extend UML to support formal graphical notations is a prerequisite to understand how and where extend existing UML modeling tools.

In this document we identify the main areas of extension and show how and extended UML editor can be used, together with formal analyzers, to specify and verify the property of a real-time system.

## **1.4 Structure of the document**

In Chapter 2 the relationships between formal languages and UML specifications are examined, to identify where and how UML could be extended to represent a formal language in a graphical notation. The strategy to follow to verify the property of a real-time system is also examined.

In Chapter 3 abstract requisites are defined for the tools to be extended in order to embed the graphical notation of the formal language. We define the steps necessary to successfully implement the extended UML notation in an existing tool. The mapping between the graphical notation and the formal language (passing through XMI) is also explained.

## 2. Formal languages and UML

### 2.1 Formal languages and graphical notations

There exist lots of formal methods created to model concurrent and/or real-time and/or embedded systems. They span from analysis models up to near to code models. These formal methods are very different in purposes, intents and, most important for us, structure.

So, due to these big differences in the formal methods we have considered, a formal graphical notation capable to express coherently all the formal languages is more or less impossible.

Vanishing the pleasant idea of a single formal graphical notation, panacea of lots of formal methods, we have find out graphical translations for each formal method, constantly trying to find out as many commonalities as possible.

It is possible to find a common graphical notation for more formal methods, and also it is obviously possible to find more equivalent graphical notations for a single formal method.

### 2.2 UML extended

As stressed in OMG UML Specification [2], UML is a *semi*-formal graphical notation. This *semi*-formality leads to ambiguities, these ambiguities are probably very useful to maintain UML thin, light, flexible and adaptable, but for our scope they cause lots of problems. So we have resolved all this ambiguities, at least in the areas that we have planned to extend to express a formal language of choice.

The extensions of UML we defined are described in deliverable D1.7.4 and in the other deliverables of task 1.7. Therefore, here we report only some essential information in order to make the report as self-contained as possible. In particular, the study of the generalized crossing system (GRC) reported in D1.7.4 indicated several weak point of UML as a language to model real-time systems, especially concerning the temporal behaviour of state machines. In order to correct these limitations we introduced the following extensions (widely described in D1.7.4) to state machines:

- Transitions can be associated with more than one triggering event are allowed. This is needed to model the reaction to simultaneous events.
- Guards expressions can contain references to events.
- Transitions can be given names, so that it is possible to make reference to a specific transition's execution time.
- A notation was defined to express temporal constraints on the execution of transitions. In particular, instead of the simple After() mechanism provided by UML it was introduced the possibility to specify that a transition has to occur in a given time interval [t1; t2].
- A notation that make clear distinction between various kind of preemptions.

In order to keep to a minimum the changes to the standard UML notation, we tried to identify already existing extensions of the statecharts that could satisfy as closely as possible our needs.

We found that Timed Statecharts [11] proposed by Pnueli and Kesten associate temporal limits to transition execution, moreover they assume a dense time domain (which assumes the possibility of arbitrarily close events). In the Timed Statecharts transitions are classified as immediate and timed (or "waiting"). Immediate transitions are triggered by events, and are thus not directly related to the time flow. Timed transitions, on the contrary, do not depend on any event: they are required to occur in a time interval. We have therefore integrated the notation of Time Statecharts into UML (as defined in [2]).

In particular it was decided to replace the constructs provided by UML to model state machines with those provided by the TS. The resulting notation is however quite close to the original UML, and is performed consistently with the UML metamodeling precepts. We added new elements to the standard UML metamodel, coherently with the TS notation. Moreover, the semantics of a few UML constructs was modified.

The UML Statecharts address semi-formally the constraints of reactive systems which maintains permanent interactions with its environment. Reactive systems involve communication, concurrency and pre-emption. Few models support these three concepts, even less can correctly deal with their coexistence. The Synchronous paradigm allows a rigorous approach to this problem crucial to reactive systems. Synchronous languages are founded on simple and rigorous mathematical semantics. In addition correctness of programs can be proved formally. So in parallel to the extensions concerning Timed Statecharts, it was decided to propose another way of extension for formal verification of UML models related to Synchronous approach. We have therefore introduced new stereotyped UML class <<Capsule>> for the representation of Synchronous reactive class. The behaviour of a <<Capsule>> is no longer represented by Statechart but by SyncChart (SynchronousChart) [14, 15].

SyncChart is a synchronous graphical formalism which many features are inherited from Statechart. SyncChart are state based models which support hierarchy of states, orthogonality, information broadcasting and preemptions. On this last point SyncCharts surpass StateCharts, they make a clear and non ambiguous distinction between various kind of preemptions (abortion or suspension, weak or strong). The SyncChart graphical formalism is fully compatible with the imperative synchronous language Esterel and is especially convenient to express complex reactive behaviour. We have therefore integrated the notation of SyncCharts into UML.

### **2.3 Verifying real time system using extended UML**

Ease of use is of fundamental importance for the DESS methodology, in order to make it easily accessible by industrial developers. At the same time, the DESS notation cannot be just semi-formal, as this would prevent the possibility to base formal reasoning (e.g. to verify properties) on DESS models.

We therefore tried to provide the notation with a formal background that could guarantee rigour and reliability to DESS specifications. For this purpose we provide the extended notation with precise semantics by means of translation rules to notations (like temporal logics or timed automata) that are adequate to represent real-time systems, but are generally considered too complex to be diffusely used in industry. This choice also took into consideration the possibility to use already available tools.

With this approach we made possible the construction of tools based on the DESS notation that provide to the users powerful tools based on temporal logics or timed automata while not requiring the same users to learn the difficult concepts of temporal logics or timed automata.

More precisely, the chosen target notations were TRIO [7] , Kronos [10] and Esterel.[12, 13].

Kronos is a research tool that is able (in certain conditions) to verify whether the behaviour of a given real-time system –modelled as a set of timed automata– satisfies the requirements given in the specification phase. The mathematical base of Kronos is provided by timed automata and temporal logics.

TRIO is a specification language based on first order logic, enhanced with temporal operators. It is particularly suited to specify real-time systems, since it allows to express quantitative temporal constraints and requirements.

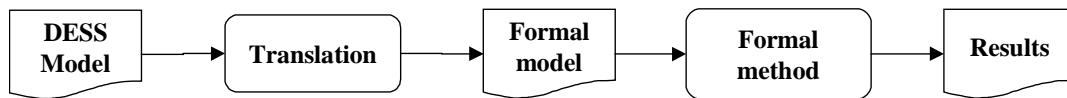


Figure 1 The procedure to verify DESS models.

The procedure adopted to verify properties of the models expressed with DESS extended UML is described in Figure 1. Following the translation rules illustrated in DESS deliverable D1.7.4 the DESS model (stored as an XML file) is converted into a formal model (expressed in Kronos or in TRIO<sup>1</sup>). Properties of Kronos models can be automatically verified by means of a model-checking tool. TRIO models are less tractable, and can be automatically verified only to a limited extent. For instance it is possible to verify if a given evolution of the system (expressed in terms of events and states) is possible according to the model. Of course it would be possible to try to prove the properties of a TRIO model with the help of a theorem prover, but this required sophisticated skills, and is thus not consistent with the hypothesis of keeping the DESS method accessible to most developers.

However, also Kronos has some limits: in order to keep the notation tractable its expressiveness is limited with respect to Timed Statecharts and TRIO. It is thus possible that a DESS model cannot be converted into Kronos.

Of course if the verification highlights a problem with the model, it is possible to modify it and go through the translation and verification again.

Esterel is an imperative synchronous parallel programming language dedicated to control-dominated reactive systems. Esterel programs are input driven: they wait for inputs and compute corresponding outputs in a cyclic manner. An input output computation is called a reaction. Synchrony conceptually means that reaction take no time, or equivalently, that outputs become available as input become available. By abstracting away reaction times, synchrony reconciles concurrency and determinism. Furthermore it is possible to use sophisticated implementation, optimization, and verification techniques commonly used in areas such as process calculi or hardware design and to extend them to software applications. It should be noted that synchronous programming languages share the synchronous model with sequential digital circuits that cyclically “read” their input pins, “compute” and “write” on their output pins, at a given clock speed. Esterel do actually use this analogy by reusing several powerful algorithms used by EDA (Electronic Design Automation). Using this technology enables Esterel to cope with large and complex designs.

The procedure adopted to verify properties of the models expressed with DESS synchronous extended UML is described in Figure 1. Following the translation rules illustrated in DESS deliverable D1.7.4 the DESS synchronous extended UML model (stored generally in dedicated UML modeller tool repository) is converted in Esterel language. With dedicated compiler the Esterel code are transformed in an intermediate electronic circuit representation, onto which powerful hardware circuit analysis and optimization can be applied. Thanks to compact representation of reactive systems using Binary Decision Diagrams (BDD) the code generator is able to deal with large designs. The generated C code is compact and fast. One of very important benefit of Esterel Studio tool environment wich support Esterel language is that apart from graphical instrumentation (used during simulation), the generated reactive kernel is exactly the same for verification and final target. This ensures that what you specify and formally verify is what you execute in your final embedded target!The main novelty brought by synchronous approach for programming real time and reactive systems is the discrete model of time they adopt. Internally the system has no notion of time, it merely

---

<sup>1</sup> In the future it will be possible to generate formal models in further notations.

captures activation. In order to model physical time in Esterel, one has to devise a clock mechanism that will activate the system at regular time intervals and thus to provide it with a notion of sampled time. Synchronous languages assume that reactions should not overlap, that is each reaction should be atomic calculation of the output response to an event (i.e., a set of input signals), that has been sampled, collected and treated. Of course this assumption should be checked; that is one should ensure reactions do not take too long so that no input is lost and the system operates at a speed matching the requirements of the application.

The Taxys tool [16, 17] is dedicated to this verification. One of the major goal of the Taxys tool is to produce a formal model that captures the temporal behavior of the whole application which is composed of the embedded computer and its external environment. The Taxys Tool operates on a formal model that describes the concrete object code behaviour at run time. From the source code of the application (i.e., an Esterel program annotated with temporal constraints), the Taxys tool produces on one hand a sequential executable code and on the other hand a timed model of the application. This model is again composed with a timed model of the external environment in order to obtain a global model that is statically analysed to validate timing constraints.

### 3. Development process

In order to make the DESS methodology applicable in an efficient and effective way it is necessary to provide suitable tools (either new or existing) that can support the development activity.

Figure 2 represents schematically a development environment consistent with the DESS methodology.

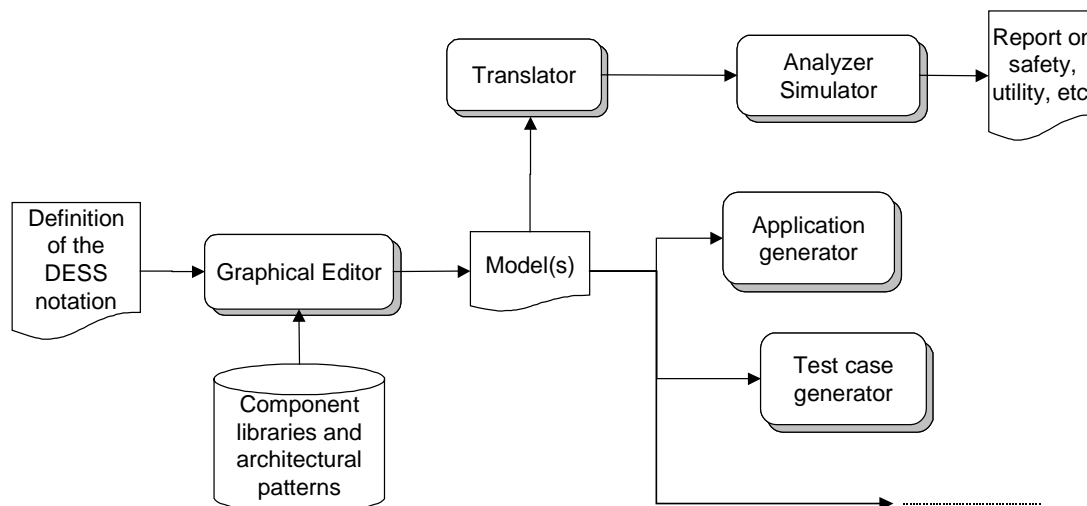


Figure 2 An abstract view of the DESS development environment.

DESS models are specified by means of the extended UML notations mentioned in the previous section. The graphical editor is therefore fully conformant with the DESS notation, e.g. it allows the modeler to associate a time interval to a transition.

The models defined by means of the graphical editor are stored in a format similar to XMI. In fact, the standard XMI will generally not be suited to represent a DESS model, as the DESS notation is richer than the UML notation supported by standard XMI. We shall use an extended XMI notation. Models written in such notation can be given in input to translators that convert the DESS models into Kronos models, TRIO models, into MME executable models or Esterel language.

The obtained models can then be treated by the corresponding tools. In some cases (e.g., when Kronos or Esterel is used) the tools can be invoked automatically in a way that keeps completely transparent to the user the application of the formal tools (except, of course, for the results).

MME (Multi Micro Environment) is an environment for hardware and software configuration and development of distributed concurrent application. MME joins the characteristics of a high level language, MML (Multi Micro Language), with a set of interactive tools enclosed into an integrate environment. MME was based onto the MOOD (MME Object Oriented Design) methodology, a proprietary methodology that shares much similarity with well-known methods such as Booch or HOOD. It requires concentrating on concurrent object first (sequences) decomposing them recursively into lower level objects (complex components) down to the lowest decomposition level (modules). The translation from UML to MOOD and than to MML is obtained mapping the UML component into MME sequences and complex components and the MME modules into the UML classes. The translation process is based on the XMI representation of the UML model and translates it into the relevant MML representation of the same model. After this step our model is ready to be processed by the

MME tool in order to obtain the application code to be deployed into each node of the target environment.

### 3.1 Design tools

In order to develop the graphical editor mentioned above three main approaches are possible:

- to develop the tool from scratch;
- to enhance an existing open source tool;
- to adapt a commercial tool by mean of its API.

#### 3.1.1 ArgoUML

The first possibility is not feasible in practice due to the lack of resources. Between the other two alternatives we decided to exploit an open source tool, as this solution provides the greatest flexibility, and does not create dependencies from a commercial tool provider. In particular, it was decided to extend ARGO/UML, an open source graphical editor for UML diagrams completely written in Java. The available version of ARGO/UML already supported most diagrams of UML in a satisfactory way (although not completely, and not in a very reliable way). ARGO/UML needs to be extended in its ability to support the diagrams that are critical for the DESS methodology, namely state diagrams, class diagrams and object diagrams. In particular it is required that the tool allows the modeller to write state diagrams conforming to the Timed Statecharts notation. For this purpose, the graphical abilities of the tool do not need to be increased, instead the tool should provide a panel where the extended features of transitions (multiple triggering events, time intervals, labels, etc.) can be specified.

The definition of the properties of a transition is exemplified in Figure 3, which shows a possible screenshot of the to-be extended ARGO/UML tool.

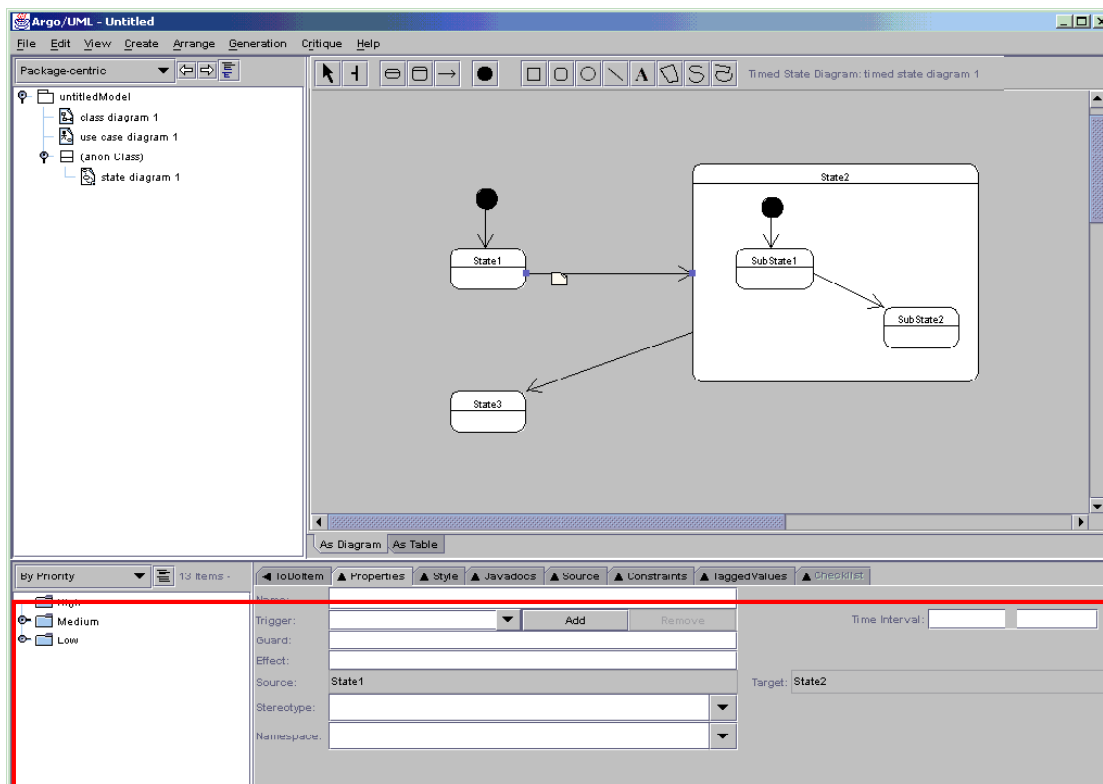


Figure 3. A screenshot of the extended ARGO/UML.

The proposed modifications concern mainly the panel of the properties associated with a transition (bordered in red in Figure 3).

A new field labelled “Time interval” is present, composed of two sub-fields representing the lower and upper time limit for the execution of the transition.

Also the “Trigger” field is modified, allowing the insertion of multiple events via the “Add” and “Remove” buttons.

The “effect” field is also split into two sub-fields, representing the generated event and the target(s): this is needed to facilitate the production of the XMI file representing the model.

Finally, in the “Guard” field negated events can be specified.

Additional modifications concern the graphical specifications of new state diagrams. The following constructs are disabled in the new Time Statecharts-compliant statecharts:

- ShallowHistory
- DeepHistory
- Fork
- Join
- Branch
- Final State
- Internal transition
- Entry e Exit Action

### **3.1.1.1 XMI support**

XML Metadata Interchange Format (XMI) is an open standard, aiming at combining the features of UML and XML.

XMI provides an information interchange model based on the XML language: this allows developer to store in a standardized format data concerning models or diagrams written in UML. XMI does not depend on a particular graphical notation, instead it is strictly connected with the abstract syntax and semantics which underlie the definition of the models’ graphical elements. For this reason, the extension of the UML notation carried out in the DESS project calls for a corresponding extension of the DTD (Document Type Definition) that specifies how to structure in an XML file the metadata defined in a UML model.

The extended version of ARGO/UML belonging to the development environment reported in Figure 2 saves the models that have been written in the DESS extended UML into an extended XMI file without losing any piece of information that is relevant for the translation into a formal notation. In fact the operation of parsing XMI files is the first step of the formalization: in order to perform it successfully the XMI files have to adopt the structure defined in the DTD for the DESS notation described in D1.7.4. As an example, Figure 4 illustrates a fragment of XMI file reporting a piece of the model of the Generalized Railroad Crossing described in D1.7.4.

```

<Behavioral_Elements.State_Machines.StateMachine xmi.id="xmi.3">
<Foundation.Core.ModelElement.name>GateStateMachine</Foundation.Core.ModelElement.name>
. . . . .
    <Behavioral_Elements.State_Machines.StateMachine.transitions>
. . . . .
        <Behavioral_Elements.State_Machines.Transition xmi.id="xmi.10">
            <Foundation.Core.ModelElement.isSpecification />
            <Behavioral_Elements.State_Machines.Transition.trigger>
                <Behavioral_Elements.State_Machines.Event xmi.idref="xmi.20"/>
            </Behavioral_Elements.State_Machines.Transition.trigger>
            <Behavioral_Elements.State_Machines.Transition.stateMachine>
                <Behavioral_Elements.State_Machines.StateMachine
xmi.idref="xmi.3"/>
            </Behavioral_Elements.State_Machines.Transition.stateMachine>
            <Behavioral_Elements.State_Machines.Transition.source>
                <Behavioral_Elements.State_Machines.StateVertex
xmi.idref="xmi.9"/>
            </Behavioral_Elements.State_Machines.Transition.source>
            <Behavioral_Elements.State_Machines.Transition.target>
                <Behavioral_Elements.State_Machines.StateVertex
xmi.idref="xmi.14"/>
            </Behavioral_Elements.State_Machines.Transition.target>
            <Behavioral_Elements.State_Machines.Transition.interval>
                <Behavioral_Elements.State_Machines.TimeInterval>
                    <Behavioral_Elements.State_Machines.TimeInterval.lowBound>
                        <Foundation.Data_Types.TimeExpression>
                            <Foundation.Data_Types.Expression.body>28</Foundation.Data_Types.Expression
                            .body>
                                </Foundation.Data_Types.TimeExpression>
                            </Behavioral_Elements.State_Machines.TimeInterval.lowBound>
                        <Behavioral_Elements.State_Machines.TimeInterval.upperBound>
                            <Foundation.Data_Types.TimeExpression>
                                <Foundation.Data_Types.Expression.body>28</Foundation.Data_Types.Expression
                                .body>
                                    </Foundation.Data_Types.TimeExpression>
                                </Behavioral_Elements.State_Machines.TimeInterval.upperBound>
                            </Behavioral_Elements.State_Machines.TimeInterval>
                        </Behavioral_Elements.State_Machines.Transition.interval>
                    </Behavioral_Elements.State_Machines.Transition>
                </Behavioral_Elements.State_Machines.Transition.interval>
            </Behavioral_Elements.State_Machines.Transition>
        </Behavioral_Elements.State_Machines.Transition>
. . . . .
        <Behavioral_Elements.State_Machines.Transition xmi.id="xmi.85">
            <Foundation.Core.ModelElement.isSpecification />
            <Behavioral_Elements.State_Machines.Transition.trigger>
                <Behavioral_Elements.State_Machines.Event xmi.idref="xmi.94"/>
            </Behavioral_Elements.State_Machines.Transition.trigger>
            <Behavioral_Elements.State_Machines.Transition.stateMachine>
                <Behavioral_Elements.State_Machines.StateMachine
xmi.idref="xmi.78"/>
            </Behavioral_Elements.State_Machines.Transition.stateMachine>
            <Behavioral_Elements.State_Machines.Transition.source>
                <Behavioral_Elements.State_Machines.StateVertex
xmi.idref="xmi.84"/>
            </Behavioral_Elements.State_Machines.Transition.source>
            <Behavioral_Elements.State_Machines.Transition.target>
                <Behavioral_Elements.State_Machines.StateVertex
xmi.idref="xmi.86"/>
            </Behavioral_Elements.State_Machines.Transition.target>
        </Behavioral_Elements.State_Machines.Transition>
    </Behavioral_Elements.State_Machines.StateMachine.transitions>
</Behavioral_Elements.State_Machines.StateMachine>

```

```

        </Behavioral_Elements.State_Machines.Transition.target>
        <Behavioral_Elements.State_Machines.Transition.effect>

            <Behavioral_Elements.Common_Behavior.SendAction>
                <Behavioral_Elements.Common_Behavior.Action.target>

                    <Foundation.Data_Types.Expression.body>Crossing</Foundation.Data_Types.Expr
                    ession.body>

                        </Behavioral_Elements.Common_Behavior.Action.target>
                        <Behavioral_Elements.Common_Behavior.SendAction.signal>
                            <Behavioral_Elements.Common_Behavior.Signal
xmi.idref="xmi.503"/>
                                </Behavioral_Elements.Common_Behavior.SendAction.signal>
                                </Behavioral_Elements.Common_Behavior.SendAction>
                                </Behavioral_Elements.State_Machines.Transition.effect>
                                </Behavioral_Elements.State_Machines.Transition>

                            </Behavioral_Elements.State_Machines.StateMachine.transitions>
                        </Behavioral_Elements.State_Machines.StateMachine>

```

Figure 4. A fragment of DESS model in XMI format.

The most relevant modifications (with respect to standard XMI) are highlighted in bold red. The time interval associated with a transition is represented in the XML element “TimeInterval”, which is itself composed of several sub-elements (including “lowerBound” and “upperBound”).

Another essential modification concerns the representation of data concerning the events generated by a transition. According to the specifications reported in D1.7.4, such data are structured in order to distinguish the generated event from the recipient(s) of the event. This facilitates the parsing process.

Finally, it is important to note that the usage of XML is in line with the trend of several software manufacturers (including Microsoft and Rational) to represent data in XML. We expect that it will be possible to integrate in the DESS environment the tools produced by such manufacturers (at the data level). Moreover, models written in XML will be publishable on the Web, making DESS models available to remotely located people not equipped with the DESS environment.

### 3.1.2 Rose/Esterel Studio gateway

Concerning Synchronous UML extensions it was decided to adapt commercial tools by mean of their API. It was decided to develop a gateway between Rational Rose UML modeler and Esterel Studio. Esterel Studio commercialized by Esterel Technologies (DESS partner) is a visual modeling toolset that provides specification, formal verification, simulation and code generation for real-time embedded systems. Esterel Studio is used for the generation of embedded control applications and it is based on a Esterel language deterministic synchronous approach. Esterel Studio includes editor for SyncCharts the synchronous graphical formalism.

The major extensions for Rose and UML concern the definition of new stereotypes associated to synchronous reactive classes (<<capsule>>, <<port>> and <<protocol>>). The synchronous behavior of a Capsule Class is depicted by a SyncChart. The SyncChart diagram (depicted with Esterel Studio editor) is associated to the Capsule by external file association Rose capability. It was decided to use Rose collaboration diagram to depict the reactive class instances and their communications for Esterel code generation. The collaboration diagram is not convenient for this purpose but Rational Rose modeler don't provide notion of Structure Diagram as in Rational Rose RT to represent these informations necessary for an automatic code generation. Extended synchronous UML model informations are exploited by the gateway for Esterel code generation directly from the Rose model repository via dedicated

API. From Esterel code generated by the gateway, for simulation, formal verification and C code generation of the synchronous reactive control part it is the capabilities of Esterel Studio which are used.

Concerning Esterel Studio it was decided to enhance Esterel Studio to support Taxys (France Telecom DESS partner tool) temporal constraints notations directly in SyncChart and to generate an Esterel code compliant with Taxys requirements.

For more details on Rose/Esterel Studio gateway specification consult the DESS WP2 deliverables.

### **3.1.3 Rhapsody/Esterel Studio gateway**

Various UML modelers exist today on the market. For the greater part, they are documentation tools, which provide a precious help to manage the application structure. Tools as Rhapsody of Ilogix, Rose-RT of Rational or TAU of Telelogic propose in addition a simulation of behavioral modeling (based on state diagrams) and include an automatic generation from this behavioral modeling for final embedded target.

However these tools seem insufficient to verify criteria of real-time embedded applications such as memory constraints, protected from ending in a limited and known time.

It was decided to realized with Ilogix's Rhapsody C++ the same gateway extensions as for Rational Rose modeler.

The major interest to realize this gateway also for Rhapsody is to exploit the Rhapsody code generation features from UML Statecharts to automatically generate the final real-time executable for the embedded target supported by Rhapsody. Inside the same model we can modelise and automatically generate the synchronous control part (with synchronous UML extension and the gateway capabilities) and algorithmic services or asynchronous tasks (called by the control part) with UML Rhapsody capabilities. For example we can define in Rhapsody the main task of the embedded application with an UML class and an associated statechart. This class can depict and automatically generate the clock mechanism that will activate the synchronous reactive kernel at regular time intervals and thus to provide it with a notion of sampled time.

For more details on Rhapsody/Esterel Studio gateway specification consult the DESS WP2 deliverables.

## **3.2 Usability requirements**

### **3.2.1 External tool**

In order to make the DESS environment easily usable even by not particularly skilled people, it is necessary to integrate smoothly the graphical editor and the translator to formal notations.

It should be possible to invoke directly within ARGO/UML the translation into one of the supported formal notations (i.e., Kronos and TRIO) in order to verify properties. The menu of ARGO/UML provides two new entries for this purpose. Of course the translation is preceded by the storage of the model into an XMI-compatible file.

With Rose and Rhapsody a new menu was defined for the invocation of Esterel Studio verifications and code generation capabilities. Only the synchronous extended UML notations of the models are exploited by this menu. So with the same UML environment we can design and generate the synchronous control part and the algorithmic or asynchronous part of large real time and embedded system.

### **3.2.2 Workflow**

The definition of a development methodology and the availability of the associated tools are not sufficient to guarantee that a delicate and complex activity like the development of real-time software is carried out in an efficient and effective way. In order to achieve this goal it is necessary that the development process is suitably supported. On one hand it is necessary to ensure that the methodology is actually adopted and its guidelines followed, on the other hand the methodology has to be supported, obstacles to its adoption must be removed, automated support must be provided, the execution of most complex tasks must be facilitated.

For this purpose a set of process support techniques and tools consistent with the DESS methodology have to be developed. This issue are dealt with in workpackage 5.

#### 4. References

1. Douglass B. P. Real-Time UML, Addison Wesley, 1998.
2. OMG, Unified Modeling Language Specification Version 1.3, First Edition, March 2000.
3. OMG XML Metadata Interchange Specification Version 1.1, November 2000.
4. Selic B., Gullekson G., Ward P.T., Real-Time Object-Oriented Modeling, Wiley, 1999.
5. Morzenti A., San Pietro P., Object-Oriented Logic Specifications of Time Critical Systems, *ACM Transactions on Software Engineering and Methodologies*, 3,1 (January 1994), pp. 56-98.
6. Workflow Management Coalition, <http://www.aiim.org/wfmc/>
7. Ghezzi C., Mandrioli D., Morzenti A., TRIO, a logic language for executable specifications of real-time systems. *The Journal of Systems and Software*, 12, 2 (May 1990).
8. Heitmayer C.L., Jeffords R.D., Labaw B.G., Comparing different approaches for Specifying and verifying Real-Time Systems, In *Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software* (New York May 1993), 122-129.
9. Yovine S. Kronos: A verification tool for real-time systems. In *Springer International Journal of Software Tools for Technology Transfer*, Vol. 1, N. 1/2, October 1997.
10. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, pages 208{219. LNCS 1066, Springer-Verlag, 1996.
11. Y. Kesten, A. Pnueli, *Timed and Hybrid Statecharts and their Textual Representation*, Weizmann Institute of Science, In *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium*, 1992.
12. G. Berry, *The Constructive Semantic of Pure Esterel*, July 1999, <http://www.inria.fr/meije/verification/esterel>
13. G. Berry, *The Foundations of Esterel in Proof, Language, and Interaction*, Essays in honor of Robin Milner. G. Plotkin, C. Stearling, and M. Tofte, Editors. MIT Press, 2000.
14. C. André. *Representation and Analysis of Reactive Behaviors: A Synchronous Approach*, IEEE-SMC Computational Engineering in Systems Applications (CESA), Lille (F), July 1996, pp 19-29
15. C. André, M.A. Peraldi-Frati, J.P. Rigault, *Scenario and properties checking of real-time systems using synchronous approach*, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2001, Magdeburg, Germany, May, 2 - 4, 2001, pp 438--444.
16. E. Closse, M. Poize, J. Pulou, P. Venier, D. Weil, S. Yovine, J. Sifakis, *TAXYS : a Tool for Developing and Verifying Real-Time Properties of Embedded Systems*, Computer Aided Verification CAV'2001, Paris 18-23 July 2001

17. E. Closse, M. Poize, J. Pulou, P. Venier, D. Weil, S. Yovine, J. Sifakis, *TAXYS=ESTEREL+KRONOS A tool for verifying real-Time properties of Embedded Systems*, Conference on Decision and Control CDC 2001, December 4-7, 2001 at the Hyatt Regency Grand Cypress Resort in Orlando, Florida