Information Technology for European Advancement

# Task 1.3: Timing, Memory and Other Resource Constraints

## Deliverable D1.3.2
## Version 01 - Public

Edited by Yvan Barbaix, Stefan Van Baelen & Andrew Wills

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

**ITEA COMPETENCES involved:**

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

***October 2001***

# Table of Contents

**1 INTRODUCTION ........ 2**

**2 TIMING CONSTRAINTS ........ 2**

2.1 BASIC CONCEPT: VIRTUAL TIMERS ........ 3
*2.1.1 Semantics of a timer ........ 3*
*2.1.2 Time constraints are instance based ........ 5*
*2.1.3 Time is not a resource ........ 6*
*2.1.4 What about mathematical expressions? ........ 6*
*2.1.5 High-level timing constructs ........ 7*
*2.1.6 Timers are not enough ........ 7*
2.2 EXAMPLES ........ 8
2.3 THE RELATIONSHIP WITH UML ........ 10
*2.3.1 The relationship with UML – first sketch ........ 10*
*2.3.2 Sequence diagrams ........ 11*
*2.3.3 State diagrams ........ 13*
2.4 CONSTRAINT VERIFICATION ........ 14
*2.4.1 Static Analysis ........ 14*
*2.4.2 Run-time Verification ........ 14*
2.5 CONSTRAINT VIOLATION POLICY ........ 15
2.6 FUTURE WORK ........ 17

**3 MEMORY CONSTRAINTS ........ 17**

**4 OTHER CONSTRAINTS ........ 18**

**5 REFERENCES ........ 18**

5.1 EXTERNAL PUBLICATIONS FROM DESS PARTNERS ........ 18
5.2 INTERNAL DOCUMENTS ........ 19
5.3 EXTERNAL MATERIAL ........ 19

# 1 Introduction

This document is the final internal deliverable of task 1.3 (timing, memory and other resource constraints).
Although the initial idea of this work package was to handle many types of resource constraints, the complexity of the problems turned out to be too high to tackle them all in a good way. We therefore decided to work in a two-stage process. In the first stage one particular type of constraint would be fully worked out. We felt that timing was the most important of al constraints and decided to tackle it first. In a second phase, we would use our experience of the first constraint type to tackle other constraints too. Unfortunately, due to the complexity of timing constraints we had to invest more time than initially anticipated. In addition, the experience from timing constraints turned out to be of little use to the other constraint types. As a result, this document will cover timing constraints almost exclusively.
One of the main incentives for the DESS project was the lack of proper support in UML for embedded real-time systems. Despite this deficiency, all partners agreed on the importance of the technology. It was clear to all that UML was a good basis for our work and that it was becoming a standard in the industry. Consequently the DESS project was set to adopt the standard and fill some open wholes in the current UML incarnation. Obviously, UML is omnipresent in DESS and in this document specifically.

# 2 Timing constraints

Timing constraints are different development aspects than the classical functional requirements. They are essentially very hard to scale. This means that is not because a property holds for some partial solution that it will hold once the application is being extended. This behavior is due to the non-functional character of this constraint type. They do not have the well-known functional behavior where the effect of a function does not depend on its external context. In addition, timing constraints are often considered not to be part of the application. The constraints do not some behavior of the software. They just specify that some issues have a timing limitation. So, if all would be perfect, the constraints should not add any code to the application or otherwise interfere with is in any way. Unfortunately, they do. In many cases the interference can be very limited like using a specific scheduling scheme, but in other cases the timing constraints will account for a large amount of code and complexity in the application. From the rough picture described above, it is clear that timing constraints need a specific development strategy.
Let us start by giving some examples of timing constraints first. When developing a new real-time product, engineers encounter a large number of constraints that are directly or (often) indirectly related to timing. Here are some of the typical constraints:

- The value from the sensor must be read every 100ms
- The frequency of the main cycle in 50Hz
- The interrupt should not read the value for at least 20µs after this update…

- The inter-arrival time of the signal is 250ms with a jitter of 10ms.
- The output should be based on the two input values that are sampled within 30ms from each other.
- The WCET (Worst Case Execution Time) of process A is 160ms

It is clear that the types of these timing requirements are extremely diverse. A simple mapping for an informal timing requirement to some (pseudo-) formal timing annotation obviously does not exist. If we want to formally prove that the requirements are not violated or instrument our running application w.r.t. the timing requirements, we must know what these informal requirements exactly mean. We have to know what *exactly* should be verified.

For example, what exactly is the meaning of " the period of a main cycle is 50ms"? Does it mean that the constraint holds for every step of the execution cycle or only at one point of the cycle (for example the start)? In other words, is a varying execution speed of the cycle allowed as long as the constraint is met in one point or does the requirement constrain all steps of the cycle? And if the requirement does not provide some measurement for the jitter, does this prohibit any amount of jitter?

Another classical example of semantic confusion is WCET. Literature has several semantic definitions of the WCET. Which one do we mean?

Facing these and other problems, the DESS consortium has come up with a solution (or at least some answers), which will be discussed in the rest of this document.

## 2.1 Basic Concept: Virtual timers

Our approach is based on the notion of *virtual timers*. Here is the fundamental reason why *virtual timers* are a good starting point:

> Thinking about timing constraints in a fine-grained way, we find that timing always requires two points of an execution trace; a start point and an end point. The start point is the point where an imaginary timer should start counting the time and the stop point is the point that we have to reach and where a given constraint should be valid.

So, we can think of a simple timing constraint as a constraint on some stopwatch or timed alarm that is started at some point and is halted or triggered at a second point -- the notion of a timer.

### 2.1.1 Semantics of a timer

In Figure 1, we show the two things we can do with a timer, resetting some time after its start and letting the timer time out. Note that we depicted this on a UML sequence diagram. As such the virtual timer is just an ordinary object.

When a timer is started, we pass it a timeout time; the time after which we want the timer to time out. In situation (a) some object in the application resets the timer before it times out. This means that the time between the start point (x) and the end point (y) in the sequence takes less than `timeout` time units[1]. This defines the relationship ’<’:

$$\text{ExecutionTime} < \text{timeout}$$

We will refer to this first usage of a timer as the start-reset sequence.

[1] In most cases we will use *ms* or *s* as time units

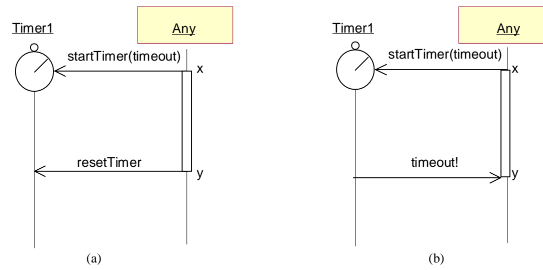


**Figure 1: reset and timeout of a virtual timer**

In situation (b) the timer times out at point (y). This means that the execution from point (x) to (y) in the sequence takes at least `timeout` time units. This defines the relationship '≥':

$$\text{ExecutionTime} \geq \text{timeout}$$

We will refer to this second usage of a timer as the start-timeout sequence.
Note that the relationship '=' is not defined as a stand-alone operator. Though this would theoretically be possible, it would make little sense to do so. It is impossible to indicate exactly at what point in the execution trace we would be at the exact time the virtual timer would time out. And even if it would theoretically be possible to express such a constraint, if would be impossible to verify it for a concrete implementation. One of the main guidelines in the DESS approach was that the constraints should not only be usable for theoretical analysis purposes, but should also be easy to map to run-time verification models.
In addition, equality is added to the start-timeout sequence as this seemed to be the most logical place to put it at. However, from an implementation point of view, it could be more interesting to put it at the start-reset sequence. We would strongly discourage to put the equality to both sequences as this would make allow constraint expressions that could be easy to express but hard (read *impossible*) to verify. If anything, we would prefer to drop the equality altogether. However, this would make some expression needlessly complex, so we leave it at the start-timeout sequence.
In Figure 1, the start and end points are both on the same object (Any). This is however not necessary and will generally not be the case. Timers can be set at one point and reset or receive a timeout at a total different point belonging to a different object and even executed by a different thread of execution. The only limit is that there should be a causal dependency between the two points.

#### 2.1.1.1 Virtual timers are objects

Virtual timers are objects like all other objects in the application. This is important as it means that timers have an identity. If the timing constraints are to be verified at run-time identity of timers becomes very important. This is one of the most important differences between the notion of virtual timers and a mathematical expression for timing constraints (temporal logic).
And because virtual timers are *objects*, they should be modeled in one of the UML object diagrams. The best-suited one is a sequence diagram as this representation explicitly focuses on the timely interaction between entities.

#### 2.1.1.2 Virtual timers are virtual

Virtual timers are *virtual*. This means that they are conceptual things that do not exist in the running application. The net result is that they do not consume resources.

Communication with a virtual timer like setting and resetting it is instantaneous. In addition, one can have as many virtual timers as needed to express the constraints.

2.1.1.3 No what-if relationship for timers

The two only interactions with a timer are the start-reset sequence and the start-timeout sequence. This means that the time between the starting point and the end point of the timer interaction is smaller than (<) or larger than or equal to (≥) the given `timeout`. It is not possible to use a timer in a conditional expression of the kind 'if there is a timeout go this way, otherwise go that way' – at least not in a simple diagram. The prime purpose of the virtual timers is to *annotate* the model with timing constraints. Using the timers in conditional expressions would mean that timers would no longer be notational entities, but would become part of the actual system under development.

### 2.1.2 Time constraints are instance based

Functional properties can be modeled successfully with class-based entities. Class based works well, because the number of instances of the class does not affect the functional aspects. This is a fundamental property of functions. In mathematical terms, the result of a function depends on the parameters of the function only, not on its external context. This makes functions scalable and thus well-suited entities for decomposition. Once the interface of a function is defined, the behavior is fixed and will not depend on any other aspect of the software. Of course, in practice, side effects of imperative languages smut the image somewhat, but proper and limited usage of these more problematic aspects allow developers to do successful incremental development.

Timing aspects are different. They do not have this functional behavior. Timing requirements generally put constraints on the schedulability and the complexity of the tasks to be developed. Ultimately, they impose a restriction on the execution speed. The hardware is responsible for this processing and transmission speed and the software is to be seen as the users of these hardware resources. And as is always the case with resources, their quantity is limited. The net result is that timing constraints can not simply be build with a classical decomposition process with their incremental development strategy. As an example, it is not because one piece of code respects its timing constraints in isolation, that it will do so in an extended context; the additional software can put a load on the processor resource that is too high for the initial code to still run 'fast enough'.

In order to be able to do proper analysis of any resource-based type of constraint, we need to know the total 'context' of the entity under development. This can be the whole application's source, but it can also be something more abstract like some partial code and some model of the higher priority resource consumers. Anyway, we need a picture of the full application and this means that we need to know what the dynamic properties of the application are w.r.t. the resource consumption. In other word, timing constraints can only be validated when the deployment model of the application is known. At this step, realistic estimation of concrete resource usage can be extracted and then annotated back into the behavioural model.

As a result, class-based design of timing constraints (like any other resource based constraint) is not powerful enough (note that is some cases it can be enough, but in general, it is not). With class-based representations, we do not have any view on the dynamic properties of the application w.r.t. the resources. That is why we need to use object diagrams to represent our constraints and that is also a reason why we

use virtual timer *objects*. We say that timing constraints are inherently instance based.

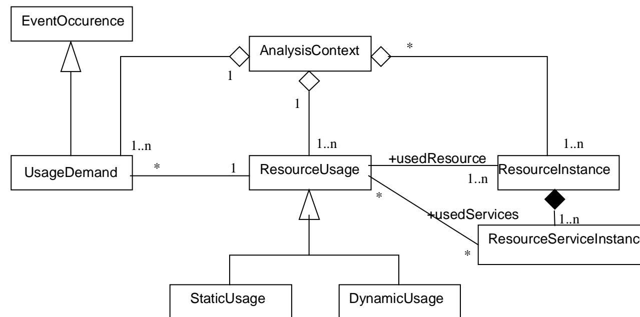


**Figure 2: The resource usage framework from [15]**

In this context, we would like to refer to [15]. In this document, a general resource modeling framework (GRM) is defined that should be the basis of all resource-based constraints. In Figure 2, a copy of the resource usage framework described in this document is depicted. The important thing to note is that the `AnalysisContext` needs a full picture of all resources usages – the list of all `ResourceUsages`. Interesting too is that they have the notion of a Resource Instance. This reflects our observation that timing constraints are instance based.

#### 2.1.3 Time is not a resource

In our search for other attempts to tackle timing constraints, we have encountered several documents where time is said to be a resource. In fact, the title of our own work package, “Timing, Memory and other Resource Constraints” suggests this too. However, Timing is not a resource. Time is a result of resource shortage. The speed of execution (processing as well as communication) will be limited by the hardware resources like the CPU, the network, the system busses, the type and amount of memory etc. The fact that time is not a resource while memory is, is one of the reasons why the research conducted on timing constraints can not easily be reused for these other constraint types.

#### 2.1.4 What about mathematical expressions?

A classical approach on timing constraints is one where a mathematical calculus (like temporal logic) is used to express timing constraints. The advantage is indeed that the expressiveness of such a calculus is strong and elegant. However, such a calculus is not based on objects that exist during run-time. This makes such techniques harder to verify. Especially run-time verification can become difficult, as the evaluation of these expressions can be complex and time consuming. This in turn is unwanted behavior, as the system to be verified should be changed as little as possible by the verification mechanism.

One of the goals of our DESS approach is to provide a simple mechanism that is easy to translate to running code. We believe that virtual timers allow for easy, low overhead implementations and are powerful enough to express the majority of the typical timing constraints. Of course, virtual timers are not powerful enough to tackle all possible timing or timing related constraints. However, the DESS project members believe that it is far more important to be able to handle a big part of the constraints we encounter and have no solution for the remaining ones, than to continue our search for a perfect solution and not have a concrete answer today.

### 2.1.5 High-level timing constructs

With the start-reset and start-timeout sequences, we can only express very simple timing constraints. And indeed, many constraints will be of the kind 'the time to execute this should not exceed *x time*', which can perfectly be expressed with these simple constructs. However, a majority of the timing constraints are more complicated that that. It is unreasonable to expect that the software engineers will (correctly) use the very fine-grained expressions that virtual timers are. What we need is an additional abstraction layer that will hide the details of timing constraint expressions in favor of an easy to understand high-level constructs. As is also observed in [15], it is important to provide the right tools to help general software developers to apply timing and general resource annotations without requiring them to be specialists in these fields. The current state of technology is such that a deep knowledge about these specific constraint types is needed in order to be a good developer. And virtual timers are simply too specific for a general developer to master. Providing high-level timing constraints constructs will provide non-specialist developers with the tools they need to annotate the designs clearly and unambiguously. It is up to the specialists in the timing domain to define a mapping from the high-level timing constraints to the semantics of virtual timers. Those high-level constructs can then be reused while their clear mapping to virtual timers keeps them valuable for analysis and run-time verification techniques.

Obviously, providing a good mapping of a high-level construct like 'period of execution' to low-level virtual timers is not unique. Many such mappings could be possible, but it is up to the specialist to define which one will be used. If multiple mappings of the same high-level construct could be required in different situations, multiple high-level constructs can be defined with the same name and a good accompanying documentation to help the developers in deciding which one to use[2]. Anyhow, we anticipate that the number of useful high-level constraints will be will be limited and more or less fixed across different applications inside a company division. As a result, we believe that such mappings from high-level constraint types to virtual timers should be grouped into UML packages that can be reused from one project to the next.

### 2.1.6 Timers are not enough

So far, we only used the timers to express timing constraints. Some constraints might be difficult or impossible to handle with timers though. If we have a constraint that uses a statistical deviation, we will not be able to map that to virtual timers. It is impossible to set a timeout value for such a constraint. It takes several execution loops to do a statistical analysis that could indicate if the constraint is violated or not.

In such cases, the easiest solution is to measure in the code the actual execution time and to do an analysis on it. Without going into more detail, it is clear that virtual timers alone will not be good enough.

[2] Another alternative and probably a better one would be to allow high-level constraint types to be modeled via UML classes. In this way, we would be able to use inheritance mechanisms to split the responsibilities of the different developers. For example, it would be possible for the generalist developer to select a 'period' constraint as an annotation wile the 'period class/type' would be abstract. The specialist would be responsible for 'instantiating' these abstract types with concrete mappings at the time a proper analysis is to be conducted. However, we did not yet work out this idea in detail

## 2.2 Examples

Now that we have explained the most important aspects of virtual timers, it is time to take a look at some examples.

**Example 1**

Let's take a look at the constraint 'the time to execute the code snippet should be 10ms with 1ms jitter'

Such a constraint actually indicates that there are two bounds for the execution time of the code snippet, an upper bound and a lower bound. The lower bound is then 9ms and the upper bound 11ms. As such, the answer to express the constraint of the execution should be to use two virtual timers like in Figure 3. Here, the indication is that the time between *x* and *y* – the code snippet – is less than 11 ms and at least 9 ms. Note that the two timers now are two. This timing constraint can easily be expressed with the simple virtual timers.

Note that at point x, the two timers are started 'at the same point/time' and that at point y timer2 times out while at the same point/time timer1 gets a reset. This may sound strange at first since a sequence diagram always shows that one object sends a message one at a time. However, our timers are virtual timers and as such have 'perfect behavior'. Communicating with them does not take any time, so it is possible to do multiple virtual timer communications at the same time. So the situation at point x is somehow logical.

However, the situation at point y is more complicated. Note that we asked timer2 to send a timeout message after 9ms. Resetting Timer1 should be competed is less than 11ms. So, the time that elapses between getting the timeout and resetting the first timer could be (almost) 2ms. In this case it still looks strange that we have an arriving and a departing arrow at the same point/time. But here too, the explanation is easy. The semantics of the start-timeout is that the time between the two point of execution (x and y) is larger or equal to (≥) the requested timeout. So the time between the start (x) and the and (y) could in this case well be 10,2ms. We should not read a timeout message to an object as the real appearance time of the timeout, but as a point where the timeout should (will) already have happened. And indeed, is would make little to no sense to put a point on our trace where the timeout would exactly arrive. This would not allow any variation in speed (elapsed time) between 2 executions of the same code. So when using a start-timeout sequence, we have to think about the end point as the point that should not be reached before the timer times out, not the exact point where the timer will timeout.

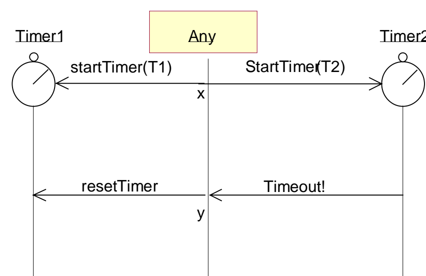


**Figure 3: 10ms with 1ms jitter**

The same reasoning goes for the start-reset sequence, but obviously with the

smaller or equals (≤) relationship. Coming back now to point y in Figure 3, both arrows are linked to the same point/time (y) because it is that point that is to be restricted by the timers.

One interesting aspect is the consistency of the notation. In this example it is obvious that a point y can in theory exist, as there is a timeframe for y between 9 and 11 ms (mathematically: [9ms, 11ms[). However, if we would request a timeout of 12 ms, such a timeframe cannot exist. A good UML-tool supporting virtual timers should be able to warn for such 'basic' design mistakes.

**Example 2**

Here is a more complicated example: 'The period of the main cycle should be 50Hz'. Obviously, one could say that the solution is very simple using a timer with a timeout of 20ms. However, it is not that simple. A first question is, what does this constraint precisely refers to? Does it mean that every step in the cycle should be executed 20ms after its last execution or does it simply mean that for one point in the main cycle, the start of the execution should be 20ms after the previous one. In the last case, some jitter is allowed on the execution cycle of an inner loop point, but on average, it should be 20ms. Another question is, what is the jitter on the period? As we stated earlier, it is in general not possible to use the '=' operator in timing constraints, so an exact 20ms period is not possible. These questions cannot be answered in a general way and will have to be defined per project.

In Figure 4 we give one possible solution. The solution takes the approach of allowing some jitter on the precise invocation time between two successive iterations. Over a number of executions however, the period will be accurate (there is no error drift). Note that in this solution, the amount of jitter is not specified (although this could have been done). In the diagram, we have used some extensions to the basic UML sequence chart. We will come back to UML extensions later in this document, but we will already explain what we need for this example. The first extension is the loop[3] with a name *a*. Everything inside the dashed box is the body of the loop. The start-reset sequence makes sure that the execution of the code inside the loop does not take longer than one period. After completion of the loop body, we must wait some time before we can execute the next sequence. Because time flows from top to bottom, it is not possible to draw the following start-timeout sequence upward from the end of the loop body to the beginning of it. Therefore, we have introduced another UML extension, the loop continuation. It is the dashed box with specification [a: Loop-next]. This construct means that this is the next execution of the loop. The *a* indicates that it is the [a: Loop] that is being executed, so the sequence in this next loop step should be identical to the first one (but details can be omitted). Note here that the 2 *a*'s in the diagram specify the same spot in the code. The rectangle filled with horizontal lines indicates that this part of the code does not contain any code. It is as if the three arrows connected to this box actually point to the same point in the code.

[3] Loops can be annotated in UML, but the way to do it is somewhat vague. In addition, we want to give loops a name for later referencing.

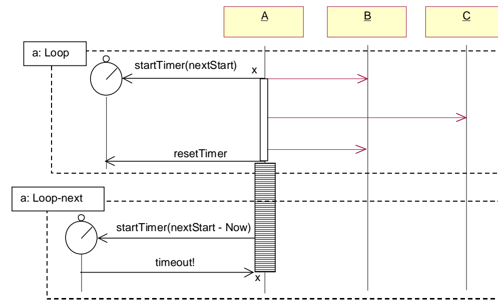


**Figure 4: period constraint**

**Shorthand Notation**

Using the explicit translation from a constraint type in a sequence diagram complicates the understanding of the diagram. This can be solved through the use of a shorthand notation for the constraint type – a high-level construct as we called it. For example, the constraint in example 2 could simply be called the *Period* constraint and could be expressed by general developers as in Figure 5. A good CASE-tool should then be able to switch back and forth between the shorthand view of the diagram and the explicit one. And especially, the shorthand notation should be available to developers, but the correct mapping of the notations to the explicit version should be part of the repository of the software under development. This will allow analysis tools to extract all information from the (XMI) repository.

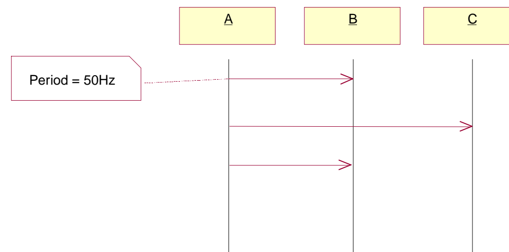


**Figure 5: Short notation of the constraint**

## 2.3 The relationship with UML

In this chapter, we will cover some issues concerning the relationship between our virtual timers and UML as well as weaknesses in the current UML definition. Many of the topics discussed here are of direct concern to real-time development, but some are of more general interest.

### 2.3.1 The relationship with UML – first sketch

As we want to base our virtual timer approach on UML, we have to provide a way to incorporate them into the language. Fortunately, the current UML extensions work well for our needs. This way, we do not have to change the foundation of UML -- the so-called metamodel. Our extension consists of a UML *profile* that contains the basic mechanisms of virtual timers. The profile adds a Virtual Timers package to the existing UML definition whose contents is depicted in Figure 6.

The extension is very simple. We can think of the virtual timer objects as being linked to one clock. Whenever a timer is started, it creates a deadline (Timestamp). That deadline is also passes to the unique clock. The clock keeps track of all timer

deadlines via an ordered queue. The earliest timeout is a special one as the clock will watch for this one to time out first.

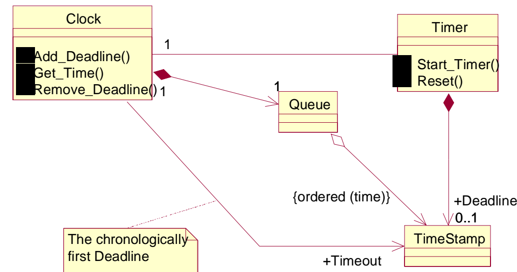


**Figure 6: The relationship between Timers and the Clock**

If the clock times out, the clock signals this. If the timer associated with the timeout would have set the deadline as part of a start-reset sequence, this signal would indicate a timing constraint violation. If, on the other hand, the timer can be reset before it times out, the Timestamp will be destroyed and no signal will be emitted.
If the deadline of the timer would have been part of a start-timeout sequence, a timeout signal would allow verification that the second point of the constraint will not have been past. Note that this is not really a difficult situation as we can allow the task to go to sleep before the end point until the timeout signal emits. Such a signal would then just wake our task up.
In this explanation, it is important that the timers are virtual. As such, the creation of the Timestamp objects and the communication between the timers and the clock consume no time. The whole mechanism is purely conceptual. The §2.3.1 is simply devoted to virtual timer implementation with respect to their semantic (as defined in the §2.1.1) within the UML meta-model.

### 2.3.2 Sequence diagrams

Using virtual timer objects only works well when the syntax of the diagram used to annotate the model provides support for annotating all features. The UML diagram we use as the host of our virtual timers is currently the sequence diagram. A problem with the current sequence diagrams is that it lacks some power and semantic foundation. We will suggest some changes to the current standard. In this sense, we advocate to refine and redefine the current language definition to make it more powerful. However, we believe that this wish is not specifically linked on our needs, but reflects a desire of many current UML users – real-time developers as well as others.
UML specifies that there are two types of sequence diagram, the *instance* form and the *generic* form. The difference is that the instance form represents only one possible snapshot at run-time while the generic for tries to model a template for possible snapshots. One can look at it as if the generic form describes an algorithm that can be used for several executions depending on the run-time situations, while the instance for is one specific trace through such an algorithm. It is clear then that the generic form needs loop constructs and branches while the instance form does not. Unfortunately, the presence or absence of these loops and branches is the distinction property between the two forms. It would be preferable to make an explicit distinction between these two. By not making this explicit, confusion could occur when a sequence diagram would be representing a generic form while it does not contain loops and branches.
However, this is not the only problem with the current incarnation of sequence

charts. First of all, no good mathematical foundation is given for the semantics of sequence charts. This is in sharp contrast to Message Sequence Charts or MSC's for short. They are built on a mathematical definition that excludes ambiguity. In addition, they have high-order sequences, building blocks that allow MSC's to be augmented with looping and branching constructs.

Even better than MSC's are LSC's, Life Sequence Charts. LSC's are MSC's, added with existential and universal quantifiers. This allows us to express that a sequence represents one possible path or all possible paths though a communicating architecture. This is much better than the UML instance form and generic form. Our recommendation to OMG would be to consider LSC's as the basis for the next UML definition.

Still, LSC's do not offer all the features that we would like to see in UML. In UML, the object in a sequence diagrams do not belong to Classifiers (are not instances of classes), but are CassifierRoles. This means that the real objects in an interaction could be any objects that could play the role of the classifier. In some cases, we do want to restrict the object we want to one specific one though.

Another deviation from the current UML definition is the possibility to 'relax' the strict flow of time from top to bottom[4]. If we want to allow several messages to be executed randomly in time, we now have no other choice than to draw diagrams for all possible combinations of these occurrences, or we have to add an informal note to the diagram. We would like to have the option to specify a region in the diagram where time axis is 'switched off' and consequently, arrival times of messages are not specified (other than that sending a message has to happen before arrival of the message). In the same category, we need to have the possibility to stretch one point in time, the idea of what we did in Figure 4. It is as if we halt the time but provide some room for the communication. Note that this is needed because virtual times communicate endlessly fast.

One last point concerns the completeness of a message in terms of object communications. In a sequence diagram, it is not because no message is shown between two successive messages that no such message exists. And in general, this is exactly what we want. But, sometimes, we want to indicate that nothing else can happen between two successive messages. It should be possible to annotate such a constraint.

2.3.2.1 Tool support

In addition to our suggestions for UML extensions, we also would applaud better support from case-tools for manipulating sequence charts.

What comes to mind is the possibility to zoom-in and zoom-out on a sequence diagram in both horizontal and vertical direction. Horizontal zooming would be collapsing or showing the interactions of a method call. If we want to hide the reactions to a message call, the tool should allow the developer to view only the call and return for that message without the details of the sub-interactions of the sequence. The result is that the call takes less screen estate in the vertical direction. And if the result would be that after hiding the internals, some object would not send or receive messages any more, the object can be hidden resulting in more screen-estate in the horizontal direction too.

Zooming-out in the vertical direction would be hiding method calls and all their sub-

[4] …or left to right as is also allowed in UML if the diagram is rotated 90degrees counterclockwise

interactions. Zooming-in would be the other direction. This would help the developer in designing a complex but complete interaction. Once one part of an interaction is defined that part can be collapsed and the rest of the interaction can be modeled without the complexity of the initial interaction part.
Of course, this could lead to overuse of complex interactions, which in turn would result in increased difficulty of understanding the design. This could be remedied by adding 'derived' partial interactions. These are stand-alone interaction diagrams, derived from the 'full interaction', but with selective zoom-in and zoom-out parts. Several of these interactions could help understand the overall mechanisms in the design while still only one full version would be in the UML repository of the model.

2.3.2.2 Relation between sequence and collaboration diagrams

Although the sequence and collaboration diagrams are two different views of the same fundamental notion of interacting objects, they do offer different focus points. While sequence diagrams make the timely dependency of the interaction explicitly visible, collaboration diagrams offer a better view of the architectural dependencies of the communicating objects.
We have only considered sequence diagrams, as it is a more natural notation paradigm for timing constraints. More work on representing timing constraints and the additional topics for sequence diagrams (existential/universal quantifiers, zooming, etc.) is needed. For the time being however, we can live with the definitions in sequence diagrams alone.

### 2.3.3 State diagrams

Besides sequence diagrams, state diagrams can also be used to annotate timing constraints. The relationship between state diagrams and sequence diagrams is that sequence diagrams model inter-object communication while state diagrams model class behavior in terms of internal state transitions and communication with the external world.
A state diagram models the behavior of all objects of a class. In this sense it is very general. A sequence chart models possible interactions between objects, not necessarily any possible objects of the classes. In this sense, a timing constraint that should hold for all possible instances of a class could well be annotated on the state diagram. This will however require that the constraint will involve two points on or in the class, it can not do that between an event in one class and a second event on another class!
In [7] and [8] timing constraints are expressed using UML predefined notations (especially *after*). According to the standard, these predefined keywords are open-ended and it is possible to define additional ones if needed.
In [4], the relationship between sequence diagrams and state diagrams is explored and defined on a mathematical basis. If we allow virtual timers to be expressed in both sequence and state diagrams, such a notion well-defined relationship is needed to ovoid semantic inconsistencies or gaps. Especially, more work needs to be done on the inter-diagram relationship when viewed in the context of inheritance.
Communicating with virtual timers from within a state diagram is straightforward. We just can send out events to the timers. For example, when doing a transition, the action part of the transition could have a signal `^Timer1.start()`. A disadvantage is that all communication from and to a state diagram can only be represented implicitly. It could be a good idea to extend the state diagram notion with an optional explicit communication to other objects. In this case, instead of writing

`^Timer1.start()` on a transition, we could draw the object `Timer1` and make a connection from the transition's action part to the timer object with the start signal. Of course, such a notation would very soon become clumsy, but with good usage, this explicit notion could make the state diagrams easier to understand. Note that this explicit communication would be a general notational alternative. It could be used to express communication explicitly to any kind of object, not just timers. If such an extension would not be appreciated, an alternative could be to sum up all objects that a state diagram communicates with in a note linked to the complete diagram. Such a note could be generated by the UML tool and derived from the explicit actions in the diagram. Here too, the connection between the state diagram and other (external) objects would become easier to grasp, as it would be more explicit.

## 2.4 Constraint verification

Of course, annotating a design with timing info is nice, but if that is all there is, why bother? Things get interesting once we are able to check the consistency of our solution with the constraints. Fundamentally there are two ways of doing this, statically (prior to running the code) and dynamically (while running the code).

### 2.4.1 Static Analysis

Static checking allows us to do the analysis before running the (final) application and, if all things are OK, to prove that the constraints will be met. On the downside this requires information about the whole system. This does not mean that every detail of the system needs to be known, but that all behavioral relevant, potentially interfering aspects need to be known. Only with such a global picture of the full application will the analysis technique have the necessary input to make correct conclusions. For example, the exact behavior of the scheduler needs to be known, including context switch times. Or if an algorithm is used containing a loop, the complexity and the upper bound of the loop needs to be known to allow the analyzer to make a valid assumption about the duration of the worst-case execution. And to be able to predict this time, the analyzer must know what hardware the application will be running on. It should be clear that this *global* information makes static analysis much harder to do and puts some severe constraints on the whole system and their developers. As a result, static analysis is only used when their advantages outweigh their disadvantages. That is generally the case for safety critical applica-tions. Here it is of utmost importance to know that nothing can go wrong because it was proven correct statically. Dynamic run-time verification can only provide a high degree of confidence in the correctness of the application. However, it cannot prove the absence of errors.

To allow some form of static analysis, we cannot just put some constraint annota-tions in our system. We need to provide an analyzer with all possible paths through the system, which means that we have to provide all sequence diagrams of the system. This is currently not feasible with the semantics of UML., but with the suggestions we made in 2.3, this should no longer be a problem.

Still, the basic notions of virtual timers are simple but powerful enough to allow static analysis methods.

### 2.4.2 Run-time Verification

The second way of checking constraint compliance is through dynamic run-time verification. This method cannot provide a compliance proof, but only a high degree

of confidence in the solution. On the positive side, run-time verification is relatively easy to perform.
The timers we discussed until now are virtual timers. Communication with timers does not consume any time nor do timers require any system resources. This makes them ideal for annotation and analysis purposes.
However, if we want or need to do run-time verification of timing constraints, we have to map the timing constraints into run-time timing mechanisms.

#### 2.4.2.1 Concrete run-time objects

The most obvious way to do such a mapping is to translate the virtual timers into concrete timer objects that live at run-time. On the downside, we have to realize that some properties of the virtual timers can only be approximated by concrete timer objects. For example, communication with timer objects does consume some time. The number of timers allowed in the running system should also be limited.
While concrete timers are the most obvious way of handing our virtual timer constraints, other solutions are possible too. One possible way would be to use a triggering mechanism that would allow an external observer (possibly a hardware one) to surveil the system, while the running software would have almost no performance degradation. However, a more detailed discussion on this and other mechanisms is outside the scope of this document.

#### 2.4.2.2 Tool support

To support run-time verification of the constraints, tools should be extended to support automatic code generation. Although concrete code generators do not yet exist, we can give some issues generators should deal with. One such issue is that the developer should be able to activate and deactivate the verification of individual constraints. An application can have so many constraints that verifying them all could put such a weight on the system that the reason for constraint violations would be the load of the verification mechanism itself. Toggling individual verifications on and off would allow dealing with such overload problems.
Another issue is that the timers normally need to be passed as a hidden parameter to functions. As timers are objects, resetting a timer needs to act on the right object[5]. Passing these hidden parameters changes the (source) code. Good tools should be able to verify when such a parameter would not be needed and do some optimization without changing the interface. Finally, it is interesting to note that a start-timeout sequence will probably be mapped to a potential system call to a sleep function. This would then enable the OS the opportunity to schedule some other process.

## 2.5 Constraint violation policy

One issue we did not speak about is the constraint violation policy of an application. Detecting a timing constraint violation is something that is tightly coupled with run-time verification of the timing constraints. If we can statically prove that the constraints will be met, there is no point in verifying these constraints at run-time. As a consequence, there is no need to provide timing verification mechanisms that could detect constraint violations.
However, once we do run-time verification, we have to decide what to do when a

[5] We will not go into more details here, but concurrency is the main reason why a hidden parameter is necessary.

violation is detected. Here, there are two alternatives. The first one is to build into the software a mechanism of handling the violations. The second one is an approach where violations are not handled but only reported or logged, or where the software goes into a ‘graceful degradation’ mode.
The impact of choosing a violation handling mechanism is considerable. The reason for this impact is that the question what to do when something goes wrong now becomes part of the application. This in turn needs to be analyzed, designed and implemented. The total amount of software increases. The situation gets even more complicated when the constraint violation policy tries to bring the software back in a ‘normal’ state after having dealt with the problem, as this will most probably impose some timing constraints on the violation handler as well. And obviously, this constraint could be violated too. Anyhow, the timing constraints are now more than a non-functional requirement, but become part of the functional requirements of the software. The result is that annotating the designs is not enough. Remember that the semantics of the virtual timers is clear and that they do not have a check whether or not the annotation holds at run-time (no what-if test). In these cases, virtual timers are probably not what you need. Virtual timers are pure notational entities that can be checked statically or dynamically if desired. If virtual timers are to be used in a context with a constraint violation policy, additional mechanisms need to be defined to detect and report the violations. It is however hard to define one single one that will work for all purposes. In addition, more research on this aspect is needed in order to increase our knowledge on this problem domain.
The other alternative is of course not to deal with violations. In this case, we take somehow the design-by-contract approach[6]. In this approach, we assume that extensive testing has happened and did not reveal any problems. If something would still go wrong at deployment time, there is little we can do, as we did not anticipate this situation. In contrast to the design-by-contract idea where all checks are switched off at deployment time, timing constraints could well be left on just to log the problems that would still occur. Even a basic constraint violation handling could be kept, but it would be one where the system is for example safely switched off and maybe restarted. The prime difference to the violation handling approach is that timing constraints do not migrate to functional entities of the software under development, but stay non-functional. Therefore they do not influence the development strategy more than what is needed to meet the timing constraints. The overload of the timing constraints is greatly reduced.
In our DESS project, we have build some ideas about handling such violations based on the conceptual ideas of design-by-contract. The case tool could provide a lot of support for such a policy. The important message here is that you have to know what your violation policy will be before rushing to code. The policy for the constraint violation handling could become part of the application in which case the design will be greatly affected

[6] Note that Meyer’s design-by-contract approach is based on mathematical properties and relies on functional behavior which timing constraints clearly do not. However, the idea of how to deal with a problem is very comparable to the design-by-contract idea.

### 2.6 Future work

In this chapter we have defined the fundamentals of the DESS-approach on timing constraints. Although the fundamentals seem simple and sound, a lot of work can and should be done in the future.
First of all, message sequence charts should receive more mathematical and fundamental attention so that the diagram type gets both more powerful and more rigorously defined. This is needed to allow unambiguous communication between the UML-tool on one hand and the analysis tools as well as the developers on one other hand.
More work is also needed on the relationship between the different diagram types like sequence and state diagrams, but collaboration diagrams and activity diagrams as well.
Code generators should also be built for testing this approach and giving feedback on the usability.
Finally, an effort should be made to come up with a set of 'standard' high-level constraint with mapping to virtual timers. This standard set could then be a good base for extension towards the specific needs of the different partners.

## 3 Memory constraints

After our initial work on timing constraints, we found that memory constraints where still difficult to tackle. Especially the quest for a general approach turned out to be extremely complicated. There is a fundamental difference between timing constraints and memory constraints. This difference is the fact that timing is *not* a resource while memory is. Time is a unit that has nothing to do with resources. There is no way to control time. Rather, the execution speed of our technology is time-limited and this makes our hardware a resource. However, timing constraints are never expressed like 'should not take more than x execution cycles on a CISC-processor. The constraints say that a job needs to be done within a certain timeframe.
In contrast, memory is a resource. It is the application itself that will determine how much memory will be needed. Looking at the resource usage framework from [15] (see also Figure 2), it is clear that we need a global view on the memory usage in order to do a memory consumption analysis.
One way we would like to handle memory consumption is to limit the amount of resources that a component[7] is allowed to use. However, this becomes complicated once the topology of inter-component connections is not a pure hierarchy, which it normally is not. The problem is that ownership of a component is sometimes not unique or that it can be move from one object to another. For example, if a component `a` asks a component `b` to create an object `o` and pass it back, who is the owner of `o`? Is it `a` or is it `b`? Or is it `a` during creation and `b` after passing `o` back? And if both `a` and `b` keep a reference to `o`, who is the owner then? In this context, it is rather difficult to say that we will allow component `a` to use no more that some fixed amount of memory. Sure, we could define some rules for the ownership, but the rules should make sense so that developers understand them and obviously, the

[7] Here 'component' is used in a very broad sense. It could be dess-components, but it can be classes or other units as well.

total amount of memory should be correct in order for the memory constraint analysis to make sense. An analysis that would say that all memory is consumed while there still is some amount makes little sense (which makes sharing ownership difficult!).
Another way of limiting the amount of memory is based on following the execution path of the application. Such an approach is described in [1] where memory is requirements are computed for each invocation of a function and each execution of a statement. The memory usage is split into a stack amount and a heap amount. In cases where the memory requirements cannot be computed, some help from the programmer is requested to limit the general applicability to a limit that would work in the application at hand (the tuning parameters). The biggest drawback of this approach is that we do not have memory constraints based on the components we would use to decompose our application. As a developer tends to think in units of decomposition, he would like to say that some unit should not consume more than this amount of memory so that the overall memory consumption can be estimated well. Such a split does not work well with this approach.

# 4 Other constraints

The consortium is convinced that we first have to tackle timing and memory constraint before digging into other constraint types. The reason is of course that these constraints are the most important ones for the consortium partners. Tackling these constraint types in the project too turned out to be too optimistic. The complexity of constraint types is simply too high to do all that in one project.
One interesting constraint type is bandwidth. As bandwidth (throughput) is defined as memory per time units, it is clear that it has some relation with both timing and memory constraints. However, components would probably prefer to deal with bandwidth rather than with memory and time if throughput is what they care about. So, although there is a relation to the two other constraint types, it is a valid constraint type on its own.

# 5 References

## 5.1 External Publications from DESS Partners


[1] Y. Barbaix, K. Hermans, T. Holvoet and Y. Berbers. Tuning Parameters for Component Based Design with Memory Constraints. ECOOP2000, Workshop on Pervasive Component Systems. Sophia Antipolis and Cannes, France, June 2000.

[2] Y. Barbaix, S. Van Baelen, K. De Vlaminck. Handling Timing Constraints with Virtual Timers. ECOOP2001, Workshop on Specification, Implementation and Validation of Object-oriented Embedded Systems. Budapest, Hungary, June 18-22, 2001.

[3] V. Bertin, M. Poize, J. Pulou and J. Sifakis. Towards Validated Real-Time Software. Proceedings of the 12$^{th}$ Euromicro Conference of Real Time Systems. Stockholm, Sweden, June 2000, p. 157-164.

[4] J. Küster and J. Stroop. Towards Consistency of Dynamic Models and Analysis of Timing Constraints. 3rd International Conference on the Unified Modeling Language (UML'2000), Workshop on Formal Design Techniques for Real-Time UML. York, October 2000.
[5] J. Küster and J. Stroop. Consistent Design of Embedded Real-time Systems with UML-RT. The 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing – ISORC 2001. Magdeburg, Germany May, 2 - 4, 2001
[6] Jacques Pulou. Timing Constraint Specifications in the TAXYS CASE Tool.
[7] Quaroni Gabriele, Venturelli Matteo. Specification of GRC Problem using UML
[8] Quaroni Gabriele, Venturelli Matteo .Engine of Rotor during Acquisition Phase

## 5.2 Internal Documents

[9] Domain Characteristics. DESS Internal Deliverable D.1.1.1. Version 00-Draft E, June 2000
[10] Common Characteristics with Attributes and Metrics. DESS Internal Deliverable D.1.1.2. Version 00-Draft B, June 2000
[11] Code Generation Synthesis Document. DESS Internal Deliverable D.1.5.1. Version 1.0-Draft E, June 2000
[12] Code Generation. DESS Internal Deliverable D.1.5.2. Version 1.0-Draft B, September 2000
[13] DESS Project. Internal Information. http://www.cs.kuleuven.ac.be/restricted-distr/dess/

## 5.3 External Material[8]

[14] Response to the OMG RFP for Schedulability, Performance and Time. OMG document number ad/2000-08-04, Version 1.0. August 14, 2000. Available as http://cgi.omg.org/cgi-bin/doc?ad/00-08-04.pdf
[15] Response to the OMG RFP for Schedulability, Performance and Time – Revised submission. OMG document number ad/2001-06-14, June 18, 2001. Available as http://cgi.omg.org/cgi-bin/doc?ad/01-06-14.pdf
[16] S. Mauw The formalization of Message Sequence Charts.
[17] Hanêne Ben-Abdallah, Stefan Leue. Timing Constraints in Message Sequence Chart Specifications.
[18] Gopal Raghavan, Maria M. Larrondo-Petrie. A formal Package for Specifying Real-Time System Constraints
[19] F. Dietrich, X. Logean, S. Koppenhofer, J-P Hunaux. Modeling and Testing Object-Oriented Distributed Systems with Linear-time Temporal Logic
[20] D. Harel, W. Damm. LSC's: Breathing Life into Message Sequence Charts.

[8] For additional references refer to the bibliography of the contribution "Constraint Specifications" by France Telecom R&D in the annex part of this deliverable.