Task 1.3: Methodology Document for Addressing Resource Constraints Problems in Embedded Systems (D1.3.1)

Version 01 - Public

Edited by Yvan Barbaix & Stefan Van Baelen

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 2000

Table of Contents

1 INTRODUCTION 2

2 THE ROLE AND CHALLENGE OF TASK 1.3 3

2.1 REAL-TIME CONSTRAINTS 3
2.2 MEMORY CONSTRAINTS 4
2.3 THE CHALLENGE OF RESOURCE DRIVEN COMPONENT BASED DESIGN 5

3 METHODOLOGY FOR RESOURCE CONSTRAINTS 7

3.1 REQUIREMENTS FOR SOFTWARE AND ARCHITECTURE ENGINEERING AND THE LIMITATIONS OF UML 7
3.2 REVIEW OF NOTATIONS FOR TIMING CONSTRAINTS 9
3.3 EXTENSIONS OF UML FOR THE SPECIFICATION OF TEMPORAL CONSTRAINTS 10
3.4 INTEGRATION OF TIMING AND SCHEDULABILITY ANALYSIS IN A UML DESIGN FLOW 11
3.5 DEVELOPMENT COMPONENT SOFTWARE WITH PRECISE MEMORY REQUIREMENTS 12
3.6 CONSIDERATION OF MEMORY CONSTRAINTS IN A COMPILER 13

4 CONCLUSIONS 14

5 REFERENCES 16

5.1 EXTERNAL PUBLICATIONS FROM DESS PARTNERS 16
5.2 INTERNAL DOCUMENTS 16
5.3 EXTERNAL MATERIAL 16

1 Introduction

This document is the first internal deliverable of task 1.3 (timing, memory and other resource constraints). It is organised as follows.

In section 2 the role and the challenge of this task in the scope of a component based design methodology are characterised. To meet these challenges all members conducted several individual studies on resource constraints. The individual studies have been synchronised on DESS workshops and meetings. The findings are documented as external publications or as internal reports. To clearly reflect each partners view and to provide a self-contained document, all these contributions are attached to this deliverable in the annex part. For a quick overview short summaries are given in section 3 as a reading guide.

In concluding section 4 a first evaluation of task 1.3 is given. Part of this evaluation is the impact that these results have on other tasks of DESS. This evaluation has to be refined during the second part of the project leading to an updated version of this deliverable.

2 The Role and Challenge of Task 1.3

The main goal of task 1.3 is to provide methodological support to the designer of embedded systems on how to take into account resource limitations of the target environment. Task 1.3 is driven by partner's needs:

- application characteristics as documented in task 1.1
- used tools and methods as documented in task 1.2

On the other hand task 1.3 will have the following impact on the project:

- guidance for the definition of components for embedded systems, especially integration of notations for constraints (WP 1.4)
- methodological support to manage resource limitations during the design process (WP1.5 - WP1.8)

Task 1.3 is clearly based on the results reported for task 1.1 in [4] and in [5]. Both documents contain an elaborated list of characteristics for embedded systems. It soon became apparent, however, that in this task we had to concentrate on the most important characteristics and these are surely timing and memory constraints. They are briefly presented in the following two sections.

2.1 Real-Time Constraints

In [5] three different types of real time are distinguished in the usual manner:

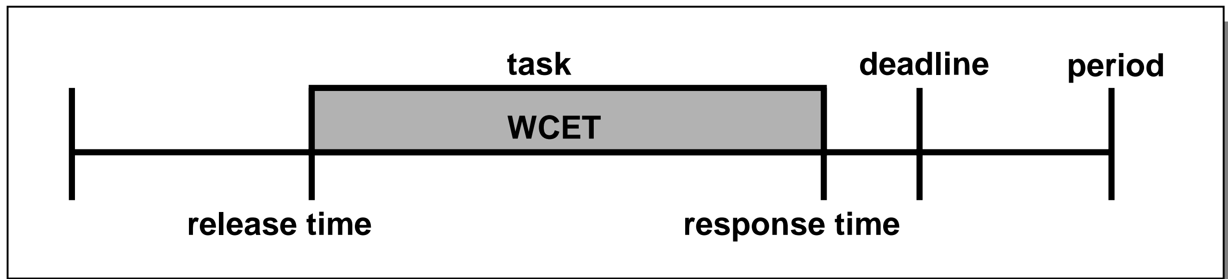
- hard real-time
- soft real-time
- statistical real-time

Real-time requirements can be quantified along the following list of metrics:

- **Period:** As the controller behaves periodically, all necessary computations and adjustments of actuators must be completed within the given time interval. The period is e.g. imposed by a sensor device delivering its value periodically.
- **Deadline:** The deadline of a task is measured relative to the beginning of the period. The task has to be guaranteed to have finished before this deadline. Note that the deadline is less than the period length, if there are several tasks that have to be completed within the given period.
- **Response Time:** This is the longest time ever taken by a task from the beginning of its period until it completes its required computation, i.e. including all possible interferences by higher priority tasks and interrupt routines. The real-time constraint is represented by the fact that the worst-case response time of a task has to be smaller than its deadline.
- **Release Time:** An offset value that represents the arrival (release) of a certain

task with regard to the beginning of the period. The release time of a task must not be later than (deadline - WCET), where WCET is the Worst-Case Execution Time of the task.
- **Jitter:** The amount of time the response time of a task can vary due to interferences and inaccuracies caused to it and its predecessors

The following figure illustrates these constraints[1]:



2.2 Memory Constraints

Quoting [5], section 1.1.3, "memory constraints come in two favours: constraints on the size and on the behaviour. Size constraints are particularly important in products for mass consumer markets. Indeed, if the memory of such products has to be increased, then this will have an immediate influence on the price of the appliance. Even for products where price is not an issue, the size is best determined in such a way that ample space is left for possible future extensions."

And furthermore, "if there is a requirement for predictable, deterministic behaviour of the memory, then this constraint has an influence on the way the memory is used. (...) Deterministic behaviour usually disallows dynamic memory use, as well as time-unpredictable features such as virtual memory or cache."

An additional requirement that is closely related to the size and cost of memory is the type of memory. Memory comes along as ROM/FLASH, RAM and EEPROM. As a matter of cost, usually the amount of ROM or FLASH on a microcontroller is much larger than RAM and especially EEPROM. Also, memory is typically organised in a non-linear way, with short and long address and the use of base addresses.

Memory requirements can be quantified along the following list of metrics:

- size
- load
- cost
- dynamic behaviour

[1] Additional explanations can be found in the contribution "Constraint Specifications" by <partner 1> R&D in the annex part of this deliverable.

- accounting for the data

2.3 The Challenge of Resource Driven Component Based Design

A component based design methodology has very different facets, depending on the involved parties, their roles and expectations. This holds even for a rather focussed topic like "resource constraints".

Right in the beginning of the project it became apparent that there are at least two different major views according to the following complementary terms:

- design of components / design with components
- component level view / system level view
- notational view / analysis view

In the former view the focus is on the single component. Here we are looking for means to describe the constraints which are attached to a component as a building block of a system.

The later view is system centric. The focus is to analyse and validate system level constraints when designing such systems based on components.

The relation between both view leads to the issues of composability and resource control in a component based development[2]. Consider a system under development. On the one hand there are constraints corresponding to the limited amount of available resources. On the other hand, there is a resource demand for supporting the system functions. The design problem can be seen as maintaining the demand lower than the constraint for any considered resource. Such a situation can be compared with the budget management in a complex building project, e.g. the target price (which is a constraint!) must be shared among all the project components .

As soon as components are chosen, the expected price can be easily computed by simply adding the component prices. The predictability of the resulting price of the whole project relies only on the accuracy of the component price estimation.

In a top down design, a system is decomposed into smaller active or passive agents. Any resource constraint to be respected must be partitioned among these agents. Hence, for a given resource, any agent will receive some "budget" corresponding to the amount of the given resource it is supposed to need.

In a component based methodology such as to be DESS methodology, the designer selects among "of-the shelves" or "already developed and checked " components those who fit with the aforementioned agents. This component choice is driven by the function that the component supports but also by the amount of the given resource the component consumes which must be lower than its allocated budget

[2] This very illustrative note on the composability issue has been provided by Jacques Pulou from <partner 1>.

(which is again a constraint .. at the component level). Whatever the considered resource, components that fulfil some function which in turn need some amount of resource to be insured: this is the resource "demand" of the component. In a strictly top down design and for any resource, if the demand of each component is lower than its attributed budget then the methodology must warrant than the resulting system will not exhaust any of these resources.

This crucial condition can be considered as well in a bottom up design process: given individual demand of each selected components, how can we compute the resource demand at the system level in order to verify whether it remains lower than the resource amount (the constraint) or not ?

This mechanism aims to control, along the design process, the amount of resources that will be needed by the resulting system. This predictability of the designed system is of prime importance for DESS methodology. The key issue of such an approach lies in component "resource demand" definition which must fulfil the following crucial property: knowing the component demand, the demand of the whole system can be computed. This property is denoted as the "composability" of the component resource demands. Computing the system demand from the demand of each of its component is denoted as the composition of component resource demands.

At this point, one can consider that predictability can be insured by an over estimating composition operation in order to warrant the resource constraint. In that case, such a naïve approach cannot be followed because the price of predictability will be an unacceptable lack of efficiency.

> Example: Let us consider two processes than run asynchronously each other but cooperate each other e.g. : exchanging each other some data with respect to some synchronisation rules as producer/ consumer schemata. Memory demand of each process can be characterised either by their execution stack (if we restrict to stack memory resource) contains along the process run or only by its maximal size. In this later case, system demand can be simply computed by adding these two maximal size but this can lead to drastically overestimating the required memory size. At the contrary If we choose the former case, the dynamic use of their stack must be characterised and then composed taking into account the synchronisation between the two process execution.

Thanks to these properties and because of the dynamic behaviour of the systems we want to design, "good" resource demand expressions are not easy to define. Most unfortunately, for sound resource demands, composition operation will not be limited to simple number addition like in the building project example!

3 Methodology for Resource Constraints

Following the descriptions in the previous sections it can be stated that the management of resource constraints has a very broad scope. In order to achieve the goals of task 1.3 each partner involved performs individual studies on resource constraints. The proposals are documented as external publications or as internal reports. The results are synchronised on regular project meetings. Any findings are fed into other tasks mainly through personal involvement in these tasks.

The work that has been performed in the first part of the project can be categorised as follows:

Subject	Partner
• Requirements for software and architecture engineering and the limitations of UML	<partner 4>
• Review of notations for timing constraints	<partner 1>
• Extensions of UML for the specification of temporal constraints	<partner 5>
• Integration of timing and schedulability analysis in a UML design flow	<partner 3>
• Development component software with precise memory requirements	<partner 2>
• Consideration of memory constraints in a compiler	<partner 1>

The relations between these efforts can be graphically depicted as shown in Figure 1. The activities are summarised shortly in the following sections. The extended versions of these descriptions are appended in the annex of this document - in the same order as in the table above.

3.1 Requirements for Software and Architecture Engineering and the Limitations of UML

Although UML (Unified Modelling Language) is a well defined and flexible modelling language its origins in the world of commercial computing show up in the lack of certain modelling requirements for real-time systems. For most (if not all) real-time systems, the issues of timeliness and schedulability are crucial. By definition real-time systems are concerned with time and ensuring that certain system activities can be carried out within defined timescales. The modelling techniques required to achieve these requirements involve performance modelling, which in turn, relies upon an ability to model the system architecture (as the actual components and their physical layout can have a significant effect on system performance) and the system concurrency (again the number and the characteristics of the concurrent tasks and the degree of task interaction between them can significantly affect system performance).

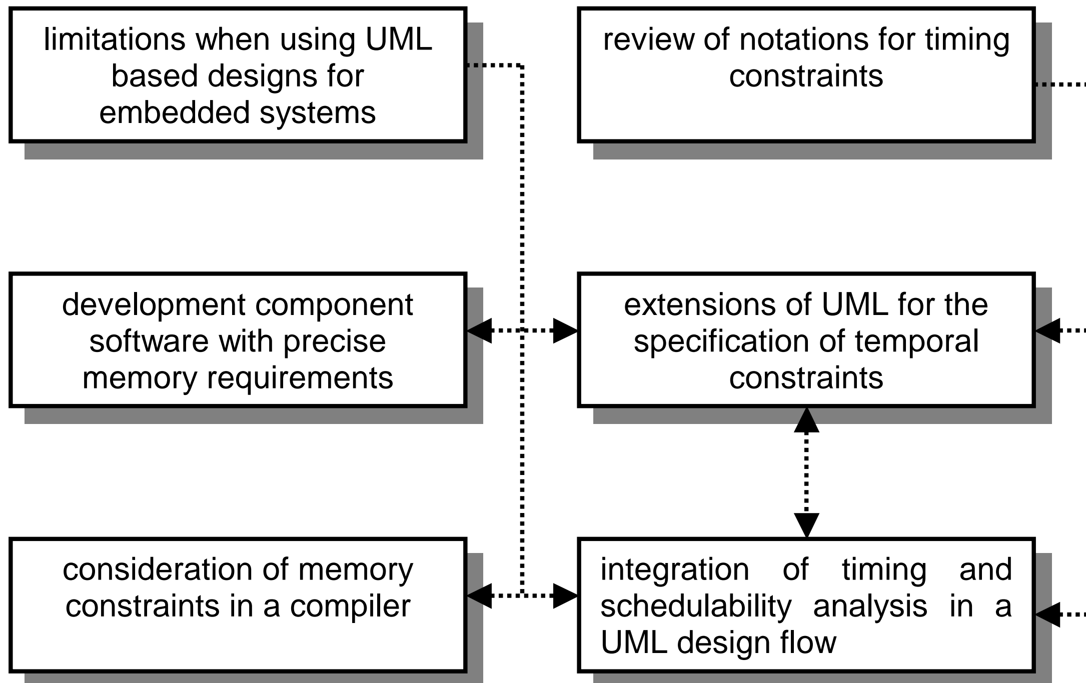


Figure 1: Relations between the activities in WP1.3

Current normalised language UML 1.4 for the real-time development domain is insufficient for the expression of certain critical aspects, as the synchronisation and hardware/software interfaces.

Sequence diagrams can be used to specify the detailed sequence of interaction not simply between actors and system, but between actors, interface devices and system software. It is also possible to annotate the diagram to show where non-functional requirements, in the form of timing constraints, apply. With sequence diagrams, it does not make it easy to distinguish between the different types of element involved with UML. It is not possible to compose a sequence diagram from more primitive sequence diagrams, furthermore it would be useful to have the possibility to annotate global time and synchronisation constraints on a group of treatments.

There is no implicit UML notation to clearly represent tasks or any of the RTOS mechanisms that can be used to protect shared resources or facilitate inter-task communication. There is a need to address the problems of concurrency and timeliness issues in term of task definition and communication, mutual exclusion of shared resources, and in terms of timing details associated with objects communication within and between tasks as well as the timing overheads associated with carried out by the RTOS.

The UML deployment diagram allows to represent the different nodes and their connections of the system, a node contains packages and processes. The connection types and nodes characteristics (processors configuration, memory card …) have an important impact on the architecture constraints of the target system. In UML the mapping between physical elements and other design elements is inadequate. It is necessary for architecture modelling to cover software/hardware information and hardware characteristics such as speeds, memory capacities, bandwidth…

The UML real time limitations are studied within the ATDPF (Analysis and design Platform Task Force) of the OMG (Object Management Group) at the origin of UML. However no standard is still finalised.

By waiting for this normalisation, important actors on the UML tools market (also OMG member) propose UML tools more suitable for real-time development. However these tools, as Rose-RT of Rational or Rhapsody of Ilogix for example, do not fit all the aforesaid needs. These tools allow automatic code generation for embedded targets by using principally notations of current UML1.4. So, such tools don't provide significative solutions to achieve performance and time requirements. The animation possibilities of behavioural diagrams (we can't really called such functionality: simulation) allow an help for the debugging, but in any case don't have the ability to check or better to prove the respect of real time system requirements.
These tools control representations detect today only two kinds of errors:

- Pure syntax error
- Using error due to bad notation or lack of coherence for a given component in the program.

3.2 Review of Notations for Timing Constraints

In the literature various timing constraint expression styles can be found covering either the "scheduling theory framework" or deriving from the "real time programming language area". The review is organised on the classical distinction between time triggered (TT) systems and event driven systems but a special chapter is devoted to the rare attempts towards timing predictability of object oriented real time systems.

In the framework of TT systems (chapter 2) general definitions are introduced and a clear distinction is made between "timed" criterion (to be minimised or maximised) and "timing" constraints that must be respected. At this point, we restrict ourselves to hard real time systems which discard any criterion and focus only on hard timing constraints. Then the more significant results in computing oriented scheduling theory are carefully described with a focus on the involved constraints. Pre-run time scheduling as well as run time scheduling approach are addressed and the most generally used algorithms are given with some insight on their advantages and drawbacks (e.g.: computational complexity for pre-run time scheduling and predictability for run time scheduling). Although rather old, this approach in its more sophisticated developments is still poorly tooled and, despite of their number, its results can hardly be straightforwardly applied to real life systems especially if they have not been built with predictability in mind.

In the event driven system framework (chapter 3), we put the emphasis on the lack of methodology in this approach especially when priority run time scheduling is used. Concerning predictability, we enforce the need to take into account external environment behaviour in specifying timing constraints. To illustrate this fact, we briefly describe the approach followed at FT and show how external environment timing behaviour model is deeply involved in timing constraint specification.

Throughout this contribution, timing constraints are described with the predictability issue in mind. We advocate that "schedulability" checking is of primary concern for the DESS methodology and that this topic is poorly covered by object oriented methods and tools. Consequently a summary of a first glance in this area is attempted in chapter 4.

We conclude on three points:

1. The limited number of timing constraints founded in the literature
2. The difficulties to implement verified event driven real time system because of the gap between this kind of systems and "schedulability" results which address mainly time triggered systems
3. The need of environment behaviour description if predictability of event driven design must be insured.

Another result of this paper lies in the selected bibliography given in the last chapter (chapter 6): out of the enormous literature on these topics (especially scheduling), we propose a choice of reference books or papers that allow the reader to get further insight in this rather wide field.

3.3 Extensions of UML for the Specification of Temporal Constraints

Formal methods[3] have been proposed and successfully employed in the development of real-time (RT) 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" RT software development. On the contrary, UML is becoming extremely popular, essentially because it is a semi-formal notation relatively easy to use (being mostly a graphical notation), and because it is provided with tools that support (although to a limited extend) code generation. However, UML was not conceived for modelling RT software: its application of UML in the real-time domain is limited for lack of constructs to express time-related constraints and properties, as well as by the lack of formal semantics.

We are going to explain some techniques that permit to specify RT systems with the

[3] The description in this section is an expert from the documents "Specification of GRC Problem using UML" and "Specification of ACL 8000" by Gabriele Quaroni and Matteo Venturelli. These two documents are internal studies from <PARTNER 5>. The former GRC problem is a widely used benchmark problem for the specification of safety critical systems. The later refers to a real product from <PARTNER 5>.

UML notation. At the same time we will propose some extension to UML. The solution we have designed is based on the following UML "schema":

- Class diagrams are used to provide the static structure of the system. Classes and their attributes identify elements and values that are considered to be essential in the behavioural definition.
- State diagrams describe the dynamic behaviour of the system and tell how the system has to be developed - correct and complete.
- The OCL syntax is used to express relations between values important to the system as well as properties a system has to respect during its evolutions. This instrument is as powerful as difficult to use. Therefore, we try to use it as few as possible.

In the sequel we will focus our attention time-out constraints as a special type of temporal constraints. UML does not provide any instruments that allow us to express that a transition, leaving a state, has to trigger within x time units after the activation of that state. In order to express such constraints we introduced the "Anomaly" state in which the system goes only if the normal transitions don't perform in time. The trigger of the transition ending in the error state is "After(x)" which is an official UML phrase. Note that the "Anomaly" state here just indicates that the behaviour of the system deviated from the specifications. In an implementation, we would wish to substitute it with an exception handling mechanism, for the sake of robustness.

Another problem with standard UML is that it does not deal with simultaneous events. We discover the necessity to describe the correct priority in handling events. In order to express this constraint we used the UML guard syntax, forcing their semantics to express the check of the occurrence of an event at a given moment. I.e. the triggering of the time-out transition is conditioned to the non-occurrence of the completion event of the critical job.

A third problem has been encountered when we tried to explicitly refer to the time when an event occurs. UML provides some keywords to denote this. However, different transitions can be annotated by such an event and which transition triggers is dependant on the active state. The labelling mechanism of UML can be used in sequence diagrams and collaboration diagrams to identify individual transition instances. It is possible to use it in state diagrams too.

3.4 Integration of Timing and Schedulability Analysis in a UML Design Flow

The position paper in [3] was presented at the workshop on "Formal Design Techniques for Real-Time UML" during UML'2000[4]. The assumption behind this contribution is that a component based design for embedded systems can be based on UML, especially with those extensions of UML such as UML-RT, that already support certain features of embedded systems.

Nevertheless, there are still a number of open issues to be solved for this approach.

---

[4] An extended version of this position paper is currently being written.

We have identified two such areas which we address in our contribution. The first area is a potential inconsistency between different languages of the UML (e.g. sequence diagrams and statecharts). The second area is the missing support for late design phases, especially w.r.t. to the analysis of temporal constraints.

Therefore, we are proposing to define a suitable consistency notion for the semantics of models and for timing constraints. Having such a notion for behavioural aspects and resource restrictions it would be possible to establish bridges to late design phases, especially code generation, timing analysis, and schedulability tests for distributed embedded systems. A sample application is to check whether the code generated for the behavioural statechart of a component fulfils the timing requirements expressed in different sequence diagrams and whether all these objects can be scheduled in such a way that the restrictions of all diagrams can be met.

Although many of the techniques we mention are well established on their own, their integration in a UML based design approach is still challenging.

3.5 Development Component Software with Precise Memory Requirements

In a position paper [1], presented at ECOOP 2000, we described one possible way to develop component software with precise memory requirements. The goal of the paper is to describe one way to use component based development (CBD) for memory aspects as typically required by embedded systems developers.

The presented mechanism used to develop a CBD application requires strong links with the compiler/linker environment as well as the development tool (case tool). While developing a component, the compiler is able to deduce the exact memory requirements of the component based on the target code (see the following section on the consideration of memory constraints in a compiler). This amount consists of a fixed part and a variable part depending on the calling environment. This information is fed back to the component's source-level. Later, while designing an application that uses that component, the tool will extract the memory information and use it to compute the application's total memory requirement.

Note that as the exact memory usage of a component depends on its calling environment (the way a component is used), it is possible to get a component usage that would require unlimited amounts of memory. When this occurs, the compiler/linker mechanism will require the user to add memory parameter knobs to the source to artificially limit the memory usage, i.e. meta-level constructs that limit the upper bound of the memory (optionally checked during runtime). At component-plug-time, the application-builder should set some knobs to case specific values and should link related knobs to each other so that turning one memory knob moves the related memory knobs simultaneously.

This CBD approach for memory requirements has several drawbacks though. First, as there is a strong link between the source code, the target code and the development tool, the implementation of such a development environment would require a

tremendous amount of manpower. In addition, the strong link with the source and target (native) code would require a partial rebuild of the technology for every source language and target platform. Finally, the techniques are limited to statically defined applications - languages like Java would not work well with this approach. Nevertheless, the ideas in this paper provide a good insight into the problems to solve if we really want to achieve CBD with memory constraints in embedded systems.

3.6 Consideration of Memory Constraints in a Compiler

Usually, the execution platform for embedded systems, i.e. microcontrollers and digital signal processors (DSP), comes along with a complex, non-linear memory model. As outlined in deliverables [4] and [5] of task 1.1 predictability of memory usage and overall reduction of the amount of memory are two important design goals. However, to analyze these aspects w.r.t to a given target hardware at design time sophisticated tool support is necessary.

SAXO is a retargetable ANSI C compiler[5] devoted to embedded application development. SAXO can be targeted to any up-to-date architectures including RISC, general purpose register file based architectures. However, SAXO takes special care on highly irregular architectures such as DSP processing cores.

Among the specific features of these architectures (e.g. saturated integer arithmetic support), this contribution highlights the use of distinct memory spaces which are also available on these processing cores. These memory spaces correspond to specialized, simultaneously accessible on chip memory banks. In that DSP oriented architectures, performance of the produced code relies heavily on the proper data dispatching among the different memory spaces which drives the potential parallelism level at run time. SAXO compiler illustrates how the different kind of memory resources can be managed at a high level of abstraction in order to obtain good object code when the whole application code is taken into account at compile time.

Another interest of this paper lies in the description of how SAXO deals with dynamic behavior of the application code for speeding up the local variable access at run time.

---

[5] The SAXO C Compiler is a development of <partner 1>.

4 Conclusions

One of the distinguishing feature of embedded real-time systems is the proper treatment of usually limited and shared resources during the design phase. Therefore, suitable notations for the specification of resource constraints are needed. More important, however, is the consideration of constraints during the whole design process to ensure that the result will fulfil these constraints. As has been pointed out in section 2.3 managing resource constraints in a component based methodology is even more challenging. The design of embedded systems with components has to respect the composability issue, characterised by two different views on resources:

- global resource constraints of a system (top-down view)
- collection of local component demands (bottom-up view)

Another aspect of the composability issue is the predictability of composition of component resource demands without overestimation (the accumulated sum of all demands is often too pessimistic).

Task 1.3 is dealing with these questions in two directions. In the first direction, the focus is on the single component. Here we are looking for means to describe the constraints which are attached to a component as a building block of a system (see section 2.3). This work is mainly performed in an intensive cooperation with task 1.4, eased by the fact, that all partners of task 1.3 actively contribute to that task as well. Through this personnel involvement the work on the use of components has been influenced by the consideration of resource constraints from the very beginning.

As the basic notation for the expression of system functionality and constraints, currently, UML is the language of choice for the intended methodology - although we are aware of several deficiencies of UML for that purpose. With respect to the contents of task 1.3 these are clearly documented in various contributions to this deliverable. Even for task 1.4 it would not have been mandatory to use UML because a component based methodology need not necessarily be an object-oriented methodology. However, UML is currently gaining a lot of attention as a set of languages for the description of systems. Therefore, it appears to be preferable to extend an existing and established language instead of inventing a completely need formalism. Furthermore, the development of UML is still ongoing and international standardisation activities are on the way. There are proposals for a major revision of UML. We will certainly assess the impact of these developments.

Within task 1.3 a number of proposals for concrete representation of constraints have been suggested, among them OCL based notations and tagged values with some formal semantics. These proposals will be evaluated further in the second part of the project taking into account the recent "Response to the OMG RFP for Schedulability, Performance and Time" in [9]. Another major development that is likely to influence the further work in task 1.3 is the expected major revision of UML, giving UML 2.0 in the near future. This revision will eventually ease the definition of profiles, i.e. the extension mechanism of UML.

The second of the above mentioned views is system centric. The focus is to analyse and validate system level constraints when designing such systems based on components. The system level view will mainly be addressed in task 1.5 (code generation), task 1.6 (validation and testing), task 1.7 (formal methods) and task 1.8 (requirements management). The main purpose of the core tasks 1.3 and 1.4 is to clearly express the component point of view in these special interest tasks. The results of this transfer process are documented in the internal deliverables of these tasks [6], [7], [8].

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

[3] 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.

5.2 Internal Documents

[4] Domain Characteristics. DESS Internal Deliverable D.1.1.1. Version 00-Draft E, June 2000

[5] Common Characteristics with Attributes and Metrics. DESS Internal Deliverable D.1.1.2. Version 00-Draft B, June 2000

[6] Code Generation Synthesis Document. DESS Internal Deliverable D.1.5.1. Version 1.0-Draft E, June 2000

[7] Code Generation. DESS Internal Deliverable D.1.5.2. Version 1.0-Draft B, September 2000

[8] DESS Project. Internal Information. http://www.cs.kuleuven.ac.be/restricted-distr/dess/

5.3 External Material[6]

[9] 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

[6] For additional references refer to the bibliography of the contribution "Constraint Specifications" by <partner 1> R&D in the annex part of this deliverable.