THE DASE TOOL: AN ENVIRONMENT FOR THE DEVELOPMENT OF REAL-TIME APPLICATIONS¹

M. Di Natale and A. Domenici
Università di Pisa - Scuola Superiore S. Anna
Dipt. Ing. dell'Informazione
Pisa

In this paper we discuss some design issues related to the handling of timing constraints in embedded systems and present a tool for the analysis of timing constraints at the architectural design level. Most design tools for real-time applications do not perform any kind of real-time analysis and do not allow the definition of explicit timing constraints. Our tool is aimed at real-time schedulability and performance analysis. It guides the decomposition of the application architecture into a set of hierarchical abstractions representing the active components and the cooperation among them (messages and/or use of shared resources). Deadlines can be specified between any couple of events. The real-time schedulability analysis requires a sufficiently accurate description of the (hardware) architecture, the mapping of the software components on the application nodes and the choice of the resource management algorithms.

1. Introduction

The growing complexity of today's embedded applications and the methodologies for the development of complex hardware and software systems strongly affect the way embedded systems are designed. An automated tool for the design of embedded systems should assist the developer in making decisions, estimating and verifying the functional and timing behavior of the system as early as possible and possibly automatically synthesizing some of the system components for him. If we examine the available design tools for embedded systems, only a few are able to support a rea-

¹ This work has been supported by the Ministero dell'Università e della Ricerca Scientifica e Tecnologica of Italy (MURST) in the framework of the MOSAICO project.

sonably extended portion of the development cycle (usually with no guarantees on its efficiency or the timing constraints).

Reactive embedded systems belong to a special application domain. First, in most such systems (known as real-time systems), responses to external events must be bounded in time (control loops, for example, are very sensitive to time delays). Second, the development process of most embedded system requires a performance evaluation phase, a major factor in deciding the partitioning of functionality between hardware and software. These particular requirements generally make the development of real-time embedded software much more difficult than the development of non-real-time software. In real-time systems the application asks for time constraints or deadlines. These deadlines can be critical (or hard), that is if the results are produced after the deadline, the system is not going to work properly (or maybe is not working at all). Time constraints are neglected by most development tools which are often focused on functional properties and do not allow designers to specify deadlines, nor to check their feasibility after the completion of the design stage. Deadlines are imposed by the environment and bound the response of the system to the external stimuli. These (end-to-end) deadlines span the execution of several tasks hosted by the same or different machines and are very complex to manage. In fact most research results show how to handle (and guarantee) a different type of problem: tasks and messages having individual deadlines. Single-task deadlines can be obtained only by overconstraining the original requirements. In this case the design tool should guide the decomposition of end-to-end deadlines to produce an analysable configuration. Finally, while it is relatively easy to maintain functional properties when changing the implementation or the host architecture, the same is not true for real-time properties. The latter are strongly dependent on the implementation and the host architecture which needs to be described with enough detail to afford a first order analysis of the timing behavior.

In our view these are the desirable features for a RT Software Engineering toolset:

- RT SWE tools should support at least the whole design phase and the early coding phase. Support for analysis and requirements specification and for testing is highly desirable. Obviously, support for the whole coding phase is also desirable, but that can be delegated to standard programming environments. The notations used should be easily understandable and yet precise enough to express all the necessary information, at the different levels of detail needed at different stages of development. Ideally, the notation should make it possible to employ formal analysis techniques wherever applicable.
- RT properties and constraints should be dealt with explicitly throughout development. Verification and simulation tools should be integrated in the development environment.
- RT SWE tools should model both SW and HW. HW should be modeled in sufficient detail and yet in a sufficiently abstract manner.

We designed and implemented a toolset composed of a design editor, a schedulability analyser, an emulator and an estimator of the WCET (worst case execution time) of the code. Its main features are:

- The tool contains a design editor that allows the hierarchical description of the active components and their functional relationships. The user can specify real-time constraints between any pair of events.
- It guides the decomposition of end-to-end deadlines (where necessary) and the choice of the most suitable scheduling and resource management algorithm in order to optimize (and check) the schedulability of the application tasks (i.e. the feasibility of the time constraints) as early as possible.
- The real-time schedulability analysis not only guides the assignment of the CPU time, but tells beforehand which tasks are going to make their deadlines and why the others are going to be late. In alternative, the tool can assist the designer in the construction of a cyclic schedule (or generate it automatically) that satisfies all the timing constraints.
- In order to perform the timing analysis the toolset requires an accurate description of the hardware architecture and a description of the mapping of the software into the processing elements.
- The worst case running time of the application tasks can be estimated using another tool which could be useful not only for checking the schedulability of the SW components but also to decide which components are to be implemented in hardware and which ones by software.

The DASE tool can perform schedulability analysis in a fairly general way but it still lacks a lot of the features that would make it a complete design environment. In order to extend it we are considering Object Oriented (OO) methodologies, which are becoming very common in the development of general purpose software. Objects orientation improves scalability, software reuse and management of complexity. OO techniques can prove significantly efficient in modeling embedded systems as well.

The *Unified Modeling Language* (UML) is gaining acceptance as a standard notation for Object-Oriented software development. It draws from earlier well-known notations and brings them together into a consistent framework, aiming to be a general and flexible means of expressing the concepts needed in software design. UML designers realized that timing constraints have not received enough attention from researchers and software developers. Therefore UML claims to be a notation suitable to the description and the validation of timing constraints. However, neither in [3] nor in [9] there is a proper analysis of the real-time constraints. The second reference proposes a good number of stereotypes (extensions to the base language) and notations (a real-time logic) in order to enhance the expressiveness of the base language.

1.1. State of the Art

Software engineering tools are now in widespread use by developers. Typically, they are centered around an interface that the designer uses to draw diagrams representing the application's architecture in a supported notation. The abstraction level of the representation corresponds to the development process phase addressed by the tool: currently, many tools assist developers from the requirements analysis phase (requiring the highest level of abstraction) to the coding phase (requiring the finest level of detail). The best tools, however, do not just produce and store diagrams, but maintain a

design database, or *repository*, where the logical structure underlying the diagrams is kept, along with all the information that must or can be associated to the structure, such as constraints, comments, source code (tool- or user-generated), version control data, and the diagrams themselves. This information is used to check designs for syntactic correctness and consistency, and can be accessed by the designer for reference. Source code and documentation can be produced automatically from the information stored in the repository.

Real-time tools came into life in the last decade, prompted by the increasing use of electronic control systems and the first results in real-time software analysis (which started in the late 70s, prompted by the well known result by Liu and Leyland [6]). Among the most notable examples of real-time tools or methodologies we have:

RT-HOOD: developed at the University of York [2] as an extension to HOOD (Hierarchical Object Oriented Design), adopted by the European Space Agency (ESA) as a design standard. The methodology consists in the description of the application according to a hierarchical OO scheme. Active elements eventually belong to a few stereotyped classes, including hard and soft real time processes and periodic, aperiodic and sporadic processes. The analysis considers the case of a single processor application and is based on Rate or Deadline Monotonic analysis.

PERTS: Allows the modeling of the tasking and resource graph in a real-time application [4]. It analyses the schedulability of the application against the deadlines. The tool performs a limited form of end-to-end analysis for distributed applications. PERTS asks for a description of all the tasks and resources in the application and performs the schedulability analysis on computation nodes. (The end-to-end analysis is obtained by composing the analysis at the node level without considering the communication between nodes).

Both tools ask the user for an estimate of the worst case execution time of the application tasks. The user is free to give figures based on his experience, testing or whatever he likes. The tools rely on these timing assumptions heavily and yet provide no help in figuring them out.

OBJECTIME-ROOM: The Objectime toolset, based on the ROOM (Real-time Object Oriented Modeling) methodology is perhaps the best known commercial solution for the design of RT applications. Although the ROOM methodology and the toolset automate most of the production phases and establish a clear, effective and understandable design paradigm, Objectime does not provide any feature for analysing and verifying the timing constraints (which are not even part of the system description). Without being OO, our design abstractions are very close to those used in ROOM.

The main limitations of the current tools and methodologies are the following:

- A restricted process model: most of the tools are based on the concept of periodic activity; aperiodic tasks are an unpleasant exception that is made to fit into the picture by means of periodic servers. Deadline constraints are attached to tasks. Since application deadlines are often end-to-end, the user has to figure out how to partition the application deadlines into task deadlines. Sometimes, the user must translate the deadlines into priorities.

- No support for checking the schedulability of distributed computations (i.e. end-to-end deadlines). Most tools assume a single processor architecture. Only a very small number consider end-to-end deadlines. Network links are simply neglected.
- No architectural modeling, no tools for estimating the worst case computation time. All these tools assume knowledge of the worst case computation time of the tasks. These figures are essential for embedded systems since they guide the software/hardware partitioning process.

Our tools make use of a number of algorithms and theorems developed by researchers on real-time systems. These results can be used to predict the execution times of tasks and predictably schedule the application tasks. The number of results that can be applied to special cases is very large, ranging from algorithms that produce a static ordering of tasks and messages to those that perform dynamic scheduling. Among the latter some require standard mechanisms (priority scheduling) while others make use of sophisticated procedures. These results make it possible to check the schedulability in a wide range of system configurations, from single CPUs [6,10,1] running periodic tasks to pipelines of tasks running on different nodes and exchanging messages over a network [11].

2 Uml as a standard notation

According to UML, a system is seen under different views, each corresponding to one or more types of diagrams: *use case diagrams* describe the interaction of the system with external agents; *class diagrams* are an Object-Oriented description of the entities composing the system, and of their static properties and relationships; *behavior diagrams* describe the dynamic properties of the entities composing the system, using various notations; and *implementation diagrams* show the physical structure of the software and hardware components. Not all designs require the full array of UML diagrams. For Real-Time applications, the class diagram, some of the behavior diagrams, and the implementation diagrams are particularly useful.

The *class diagram* is the core of a UML specification, as it shows the logical structure of the system. For each class, *attributes* and *operations* are specified. The information associated with relationships between classes may include *role* names for the participants in the association, their *multiplicity*, association *attributes*, and other items. All diagram elements can be annotated with *constraints* of various kinds and *comments*. *Behavior diagrams* comprise *statechart diagrams*, *sequence diagram*, and *collaboration diagrams*. Statecharts [5] describe the evolution in time of an object or an interaction between objects by means of a *hierarchical* state machine. *Sequence diagrams* show the possible message exchanges among objects, ordered along a time axis. The timepoints corresponding to message related events can be labeled and referred to in constraint annotations. *Collaboration diagrams* also show message exchanges among objects, but they emphasize structural relationships among objects rather than time sequences of messages. *Implementation diagrams* are either *component diagrams* or *deployment diagrams*. Component diagrams describe the physical structure of the software in terms of software *components* (modules). Deployment diagrams describe the hardware architecture in term of processing or data storage *nodes* connected by communication associations, and show the placement of software

components onto the hardware nodes. Normally a UML specification consists of several diagrams of each kind, giving several levels of detail. The UML notation includes a few extension mechanisms. In particular, *stereotypes* can be used to tailor the notation to different application domains or design techniques. A stereotype is basically a way to classify and label some recurring uses of modeling elements such as classes or associations.

1.2.1 Real-Time requirements and UML

Real-Time systems specification and design have been prospected as a very promising application of UML since its earliest definition. Recently, the ROOM (*Real-Time Object-Oriented Modeling*) methodology [8] has been shown [9] to be expressible in terms of the UML notation: the ROOM modeling concepts (such as *actors*, communication *ports*, and *messages*) can be seen as stereotyped usages of UML classes and associations. Both ROOM and UML use sequence diagrams where timing constraints can be expressed using annotations in a language chosen by the designer, however neither notation offers any specific, easily identifiable notation for such constraints. Further, sequence diagrams may not be the most appropriate means to specify them, as they can only represent *scenarios*, that are specific instances of interactions. It has been proposed [7] that timing constraints should be expressed in terms of *transactions*, i.e., end-to-end computations that can be extracted (or abstracted) from sequence diagrams.

2 The DASE tool

A designer using the DASE tool describes a RT system by means of a *software* and a *hardware* architecture. A *mapping* between the two architectures associates software modules to processing nodes.

2.1 Software architecture

The basic elements of a software architecture are *tasks*, *resources*, and *messages*. Tasks are active entities that *use* resources, resources provide *services*, and messages are exchanged between tasks. The designer specifies timing properties of the three kinds of basic elements, such as period for tasks and messages, and execution time for services offered by resources. Timing properties for tasks and messages may also be described by statistical distributions used in simulations. Other properties relevant to RT design include criticalness (*hard* vs. *soft*) and periodicity (*periodic*, *sporadic*, *aperiodic*) for tasks and messages, and synchronization semantics (*synchronous*, *asynchronous*, ...) for messages. Messages are characterized also by their data format.

Tasks and resources are grouped into higher-level constructs: *task containers* group tasks and resources, *module containers* group task containers and (recursively) module containers. The term *module* refers to module containers, task containers, tasks, or resources. Modules are represented by various icons, and messages are represented as lines between modules. Figure 1 shows part of a software architecture model. The larger icons are module containers and the small rectangles are mes-

Different types of arrows denote different communication semantics. The lower part of the picture shows data about the top level module.

Figure 1: Part of a software architecture

The DASE tool enforces top-down development. Design starts from two top-level modules representing the system and its environment: first, the interaction between system and environment is modeled as a set of messages, then the system module is expanded into submodules, until the software architecture is entirely specified in terms of tasks, resources, and messages. A module's expansion is represented graphically as a new diagram showing the module's internals. We observe that with this approach we specify messages that are exchanged between modules rather than between tasks. This is required in order to describe the interaction between two subsystems (modules) before they have been decomposed to the level of detail where the tasks originating and using the messages are defined. To maintain consistency across levels of detail, *ports* representing a message's endpoints in the sending and receiving modules are used. Figure 2 shows part of the expansion of module **Real_Time_subs** from Figure 1. The icons at the top are the output and input ports for messages **Model_request** and **Model_reply**, respectively. The other large icons are tasks. The lower part of the screen shows the characteristics of message **do_mng_msg**.

Ports are a convenient way to refer to a message within a module, but, more important, they are also the means whereby a designer can express timing constraints.

Figure 2: Tasks and ports

A *constraint* on an input port **I** and an output port **O** in the same module is the maximum time interval allowed to elapse between the arrival of a message at **I** and the issue of a message at **O**. A constraint **C** on a pair of ports implies that the module containing the ports is supposed to react to a stimulus at the input port with a response at the output port. The behavior between these two events will usually consist in the exchange of a series of messages between submodules. These messages in turn will have constraints associated with the respective ports, and these constraints will have to be consistent with the original constraint **C**. The set of messages generated in response to a stimulus and leading to its response (excluding the response itself) is the *message chain* of the constraint. A message defined on a path between a constrained pair of ports can be added to the message chain of the constraint together with the task originating it. The constraints on the new message and task can then be checked for compatibility against the chain's constraint. Further, the tool can list all the messages associated with a given constraint.

2.2 Hardware architecture and mapping

The building blocks of a hardware architecture are *nodes*, *buses*, *interfaces*, and *nets*. Nodes are the processing elements, that are connected to buses. Buses, in turn, are connected to nets through interfaces. Several parameters are specified for these elements, such as clock frequency, transmission speed, bandwidth, and communication protocol. Figure 3 shows part of a hardware architecture. Several hardware architectures can be defined for the same system. For each hardware architecture, the designer specifies which modules are allocated to each node. The minimum allocable unit is the task container. The tool checks if a physical path exists between communicating modules placed on different nodes. If more than one path exist, they are highlighted

The screenshot shows a Metasploit Meterpreter session with the following details:

- Session Information:**
 - Type: ms-sql
 - Name: OSP
 - Created: 1890-03-12
 - User: galaxy
 - Stable: [x]
- Network Diagram:**
 - The diagram shows a central switch connected to multiple hosts.
 - Hosts include Windows XP SP3, Windows 7, and Windows XP SP3.
 - Network interfaces are labeled with IP addresses (e.g., 10.10.10.10, 10.10.10.11, 10.10.10.12, 10.10.10.13, 10.10.10.14, 10.10.10.15, 10.10.10.16, 10.10.10.17, 10.10.10.18, 10.10.10.19, 10.10.10.20, 10.10.10.21, 10.10.10.22, 10.10.10.23, 10.10.10.24, 10.10.10.25, 10.10.10.26, 10.10.10.27, 10.10.10.28, 10.10.10.29, 10.10.10.30, 10.10.10.31, 10.10.10.32, 10.10.10.33, 10.10.10.34, 10.10.10.35, 10.10.10.36, 10.10.10.37, 10.10.10.38, 10.10.10.39, 10.10.10.40, 10.10.10.41, 10.10.10.42, 10.10.10.43, 10.10.10.44, 10.10.10.45, 10.10.10.46, 10.10.10.47, 10.10.10.48, 10.10.10.49, 10.10.10.50, 10.10.10.51, 10.10.10.52, 10.10.10.53, 10.10.10.54, 10.10.10.55, 10.10.10.56, 10.10.10.57, 10.10.10.58, 10.10.10.59, 10.10.10.60, 10.10.10.61, 10.10.10.62, 10.10.10.63, 10.10.10.64, 10.10.10.65, 10.10.10.66, 10.10.10.67, 10.10.10.68, 10.10.10.69, 10.10.10.70, 10.10.10.71, 10.10.10.72, 10.10.10.73, 10.10.10.74, 10.10.10.75, 10.10.10.76, 10.10.10.77, 10.10.10.78, 10.10.10.79, 10.10.10.80, 10.10.10.81, 10.10.10.82, 10.10.10.83, 10.10.10.84, 10.10.10.85, 10.10.10.86, 10.10.10.87, 10.10.10.88, 10.10.10.89, 10.10.10.90, 10.10.10.91, 10.10.10.92, 10.10.10.93, 10.10.10.94, 10.10.10.95, 10.10.10.96, 10.10.10.97, 10.10.10.98, 10.10.10.99, 10.10.10.100, 10.10.10.101, 10.10.10.102, 10.10.10.103, 10.10.10.104, 10.10.10.105, 10.10.10.106, 10.10.10.107, 10.10.10.108, 10.10.10.109, 10.10.10.110, 10.10.10.111, 10.10.10.112, 10.10.10.113, 10.10.10.114, 10.10.10.115, 10.10.10.116, 10.10.10.117, 10.10.10.118, 10.10.10.119, 10.10.10.120, 10.10.10.121, 10.10.10.122, 10.10.10.123, 10.10.10.124, 10.10.10.125, 10.10.10.126, 10.10.10.127, 10.10.10.128, 10.10.10.129, 10.10.10.130, 10.10.10.131, 10.10.10.132, 10.10.10.133, 10.10.10.134, 10.10.10.135, 10.10.10.136, 10.10.10.137, 10.10.10.138, 10.10.10.139, 10.10.10.140, 10.10.10.141, 10.10.10.142, 10.10.10.143, 10.10.10.144, 10.10.10.145, 10.10.10.146, 10.10.10.147, 10.10.10.148, 10.10.10.149, 10.10.10.150, 10.10.10.151, 10.10.10.152, 10.10.10.153, 10.10.10.154, 10.10.10.155, 10.10.10.156, 10.10.10.157, 10.10.10.158, 10.10.10.159, 10.10.10.160, 10.10.10.161, 10.10.10.162, 10.10.10.163, 10.10.10.164, 10.10.10.165, 10.10.10.166, 10.10.10.167, 10.10.10.168, 10.10.10.169, 10.10.10.170, 10.10.10.171, 10.10.10.172, 10.10.10.173, 10.10.10.174, 10.10.10.175, 10.10.10.176, 10.10.10.177, 10.10.10.178, 10.10.10.179, 10.10.10.180, 10.10.10.181, 10.10.10.182, 10.10.10.183, 10.10.10.184, 10.10.10.185, 10.10.10.186, 10.10.10.187, 10.10.10.188, 10.10.10.189, 10.10.10.190, 10.10.10.191, 10.10.10.192, 10.10.10.193, 10.10.10.194, 10.10.10.195, 10.10.10.196, 10.10.10.197, 10.10.10.198, 10.10.10.199, 10.10.10.200, 10.10.10.201, 10.10.10.202, 10.10.10.203, 10.10.10.204, 10.10.10.205, 10.10.10.206, 10.10.10.207, 10.10.10.208, 10.10.10.209, 10.10.10.210, 10.10.10.211, 10.10.10.212, 10.10.10.213, 10.10.10.214, 10.10.10.215, 10.10.10.216, 10.10.10.217, 10.10.10.218, 10.10.10.219, 10.10.10.220, 10.10.10.221, 10.10.10.222, 10.10.10.223, 10.10.10.224, 10.10.10.225, 10.10.10.226, 10.10.10.227, 10.10.10.228, 10.10.10.229, 10.10.10.230, 10.10.10.231, 10.10.10.232, 10.10.10.233, 10.10.10.234, 10.10.10.235, 10.10.10.236, 10.10.10.237, 10.10.10.238, 10.10.10.239, 10.10.10.240, 10.10.10.241, 10.10.10.242, 10.10.10.243, 10.10.10.244, 10.10.10.245, 10.10.10.246, 10.10.10.247, 10.10.10.248, 10.10.10.249, 10.10.10.250, 10.10.10.251, 10.10.10.252, 10.10.10.253, 10.10.10.254, 10.10.10.255).

Figure 3: Part of a hardware architecture

Among the abstractions that compose the logical design there are the managers of the system resources: processor schedulers for nodes, network protocols and resource access protocols for shared resources. The process of mapping the software architecture onto the hardware involves selecting a set of resource managers for each node and network. The designer selects a set of predefined resource management algorithms and maps it to a node. The DASE tool checks the tasks already allocated to the node for schedulability with respect to the selected algorithm and the messages with respect to the network scheduling protocol.

Figure 4: Module mapping

3 Beyond DASE

If we compare the DASE tool to the tools currently available for general (non RT) applications we notice a few limitations. Most notably, DASE does not support the basic object-oriented concepts, such as classes and inheritance and has no facilities to specify interfaces and encapsulated data. Another significant limitation lies in its restricted scope across the development process. The tool is tailored to the architectural design phase, and should be extended to analysis and to detailed design.

The DASE tool compares better in the area of RT tools, but still we must point out a few drawbacks. First, it enforces a top-down approach, but still this is often desirable for various reasons, including work planning and distribution, it is considered impractical for co-design efforts, where performance evaluation may lead to extensive redesign, including software/hardware repartitioning. Then, the set of modeling concepts is perhaps too restrictive. Finally, there is no way to specify dynamic behavior (states and transitions). Again, we are led to reconsider our choices concerning the modeling language adopted in DASE. The original decision of making a limited number of fundamental abstractions (such as *task* or *resource*) available to designers is useful in the early (architectural) stage of a RT design process where focus is on timing properties, but it is unsatisfactory in the later design and coding phases, where an accurate definition of the software architecture requires greater expressive power.

The modeling language should conform to a standard as far as possible. UML is currently the most promising candidate, and its controlled extension capabilities can be used to provide those specialized abstractions we deem useful in RT design. The modeling approach adopted in DASE is compatible with the UML approach, but establishing a correspondence between them is not immediate. The DASE language deals with objects rather than classes, so a DASE software architecture diagram is a

form of a UML collaboration diagram rather than a class diagram. A further discrepancy is in the modeling of messages. UML associates messages only to events and to data exchanged between objects, while DASE associates them with more parameters, such as period or criticalness. In order to map DASE messages into UML modeling elements, we have the choice of representing them as full-fledged objects or as associations with attributes. In any case, a designer should be aware of the semantic difference between DASE and UML messages. Further, the tool should be able to extract the timing information associated with DASE messages and use it to create sequence diagrams that show object interactions in terms of UML messages. Ports are another DASE modeling element that can be expressed in different ways in UML. They could be expressed as attributes of messages, or tasks, or modules. They could be seen as independent objects, and this would make it possible to map them explicitly to hardware interfaces. Constraints between ports could be expressed as standard UML constraints or as stereotyped associations between ports. We may notice that the concepts of constrained ports and their associated message chains have some similarity to the transactions proposed in [7]. The information DASE associates with hardware nodes and communication links is much richer than the information mentioned in published examples of UML specifications, but in UML these modeling elements are actually stereotypes of classes or objects and of associations. Therefore the hardware modeling elements envisioned in DASE can be easily made to fit the UML framework.

References

- [1] T.P. Baker. (1991) "Stack-based scheduling of real-time processes." *The Journal of Real-Time Systems*, (3):67--99.
- [2] A. Burns. (1991). "Scheduling hard real-time systems: a review." *Software Engineering Journal*, pages 116--128.
- [3] Bruce Powel Douglass. (1998) *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley.
- [4] J.W.S. Liu et al. (1993) PERTS: a prototyping environment for real-time systems. In Proc. 1993 IEEE Real-Time Systems Symposium.
- [5] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231--274, 1987.
- [6] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46--61, Jan. 1973.
- [7] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for automated implementation of executable object oriented models for real-time embedded control systems. In Proc. IEEE Real-Time Systems Symposium, 1997.
- [8] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and sons, 1994.
- [9] B. Selic and J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*. Object-Time, March 1998. (av. from www.objecttime.com).
- [10] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocol: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175--1185, Sept. 1990.
- [11] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing & Microprogramming*, 40(2--3):117--134, April 1994.