Realtime Hybrid Task-Based Control for Robots and Machine Tools

Peter Soetens
Flanders' Mechatronics Technology Centre
Leuven, Belgium
Dept of Mechanical Engineering
K.U.Leuven, Belgium
peter.soetens@mech.kuleuven.ac.be

Herman Bruyninckx
Dept of Mechanical Engineering
K.U.Leuven, Belgium
people.mech.kuleuven.ac.be/ bruyninc

***Abstract*— This paper presents work in the field of hard realtime robotics and machine control. We analyse the requirements of a hybrid realtime control task specification allowing the integration of discrete and continuous control tasks. We propose an application independent task structure providing data flow consistency under simulataneous access by different control layers. We provide an execution flow mechanism to guarantee execution time determinism, yet allowing flexibility to react to a changing environment. We use state machines for process monitoring and a thread-safe realtime event system to communicate changes. The tasks can be distributed over a network and communicate using interfaces or manipulate streams of data in the loop. The presented task structure is applied to a real world example.**

***Index Terms*— realtime control, monitoring, architecture, distribution**

I. Introduction

Realtime systems (machines) are designed to perform tasks on many levels. The specification of the task is tightly coupled with the system, since the system defines the operations it can perform. Consider a hybrid robot-machine tool setup where the robot assists in placing the workpiece. At the lowest level, both robot and machine tool perform a positioning task, using a positional feedback controller. At a higher level, a movement path is planned for robot and machine tool without collisions. Between movements, operations are performed on the workpiece, which requires synchronisation. At an even higher level, the task is to produce a series of these workpieces in a shop floor and so on. Many intermediate tasks may be present: measuring quality of work, evaluating the status of the machines, collecting data for the logs etc.

A. Task Composition

In a simple, non reactive system, the task is no more than a sequence of commands which dictate the system to perform the operations. When the system operates in a non deterministic environment, the task's complexity increases because it must try to react to all possible situations. Furthermore, the task acts as a controller by solving "goals" in one or more domains, e.g. the discrete logic domain or continuous state space domain.

Tasks requiring "intelligence" of the system to reach the goals, need sufficient intelligent controllers. If this intelligence is not present in the system, the task itself must specify how the goal must be reached. The task itself behaves then again like an intelligent system which defines higher level operations it can perform. This cascading of task-system allows to specify high level tasks in reasonable complex environments. Tasks can be executed in parallel on all levels. Well known examples are behaviour based controllers [4], distributed control [15] and manufacturing plants.

B. Task Determinism and Safety

In realtime systems, tasks must meet time deadlines in a deterministic way. Time determinism is measured in two ways: with a maximum latency and maximum jitter. Latency is the time a task needs to complete, jitter is the variation, over time, between the lowest and highest latency. The positional controller, for example, needs low latency to minimise dead time and low jitter to eliminate unwanted excitations.

Process safety is another critical part of the task. The task must be guaranteed that its data are always consistent and correct. A task will want to specify what action must be taken if these guarantees no longer hold and communicate this to a higher level. A well known example is when the controller detects an exceeding tracking error. It might take local actions as well as signal this event to a higher level task.

C. Design

With the addition of each task on any level, various design issues arise. How should a task be specified to define goals in multiple domains? How should a task be specified to allow time and process determinism? How should a task define its operations? How should a task communicate with other tasks? How should a task be specified in order to be scalable, i.e. allow many tasks concurrently? How should a task define how it reacts to changes?

To solve these problems, frameworks [1], [5], [7], [8], [17], [18] have been created with the aim to solve one or more of the above design issues. These issues are mostly called "forces" [9] in software design and have each their influence on the design, pulling it in different directions (Fig. 1). It is hard to balance all the forces,

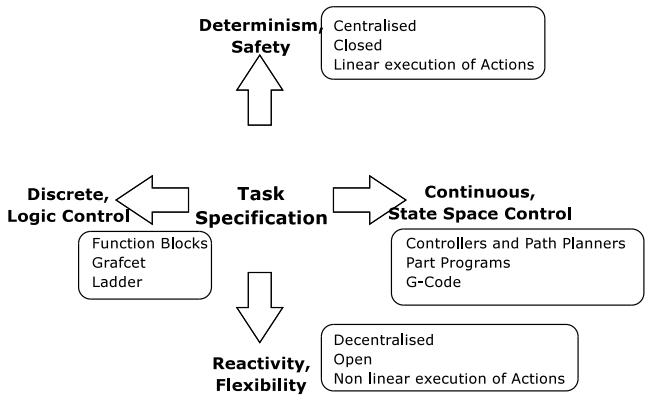

Fig. 1. Four major forces pull the specification in another direction.

since they are not neccessarily orthogonal. When a task is specified in control, four major forces pull the task specification in another direction: Should the specification favour determinism or flexibility? Should it be specified in the discrete domain or the continuous domain? We will present a design that balances these four major forces into a scalable task specification.

The work presented in this paper has been implemented in the Open Robot Control Software (OROCOS) Framework [6] and can verified by downloading the software, distributed with a Free Software license, from the internet.

The Orocos project has the ambition to serve as a common platform for all advanced realtime feedback control applications, where "advanced" means the following (the rest of the text provides more details):

- Strict separation between *data flow* (processing of the control data), *execution flow* (logic decision making), and *configuration flow* (preparing an application to run).
- Absolute attention for data consistency: the same data (from sensors, controllers, motion generators, etc.) can be accessed in different "threads" of a realtime application, but all threads should always see consistent data, i.e., the changes made to the data flow in one thread should only be visible to other threads when the full change is completed, while, at the same time, the time determinism is guaranteed.
- Integration of discrete decision making with realtime control: when one part of the application decides that the control should switch to another "hybrid state", all parts of the application should switch in a coordinated way.
- Synchronous and asynchronous events: different parts of the applications must be able to signal other parts, and react themselves to signals of these other parts. Again, these reactions should take place at moments that are consistent with the whole application.

This paper explains how the Orocos design takes all these advanced control requirements into account.

The rest of the paper is organised as follows. We give a short overview of hybrid control theory and discuss a frequently used hybrid solution for machine and robot control and show its shortcommings. Next we formulate answers for the task design questions. We present a hybrid task design for periodic or non periodic tasks and the integration with discrete events. We apply this design to build a reactive controller from bottom-up and formulate a generic task model.

II. Hybrid Realtime Systems Control Theory

Hybrid realtime systems are controlled in two domains. The first domain contains the continuous characteristics of the system, for example a position or velocity for which control [14] defines a feedback controller which calculates the steering signal, given a reference signal and a measured signal. The second domain contains the discrete characteristics of the system, for example homing signals, break signals etc. When these discrete characteristics influence the continuous characteristics, hybrid control theory [2] defines a hybrid statespace controller which can change the control algorithm on a given trigger. When discrete and continous characteristics are decoupled, the logic control theory defines finite state machines which monitor and control the discrete state of the system.

A task calculates continuous and discrete reference ("goal") signals, which must be reached by the lower level controller. The goal reference signals can change because of external events, or in general, any change in the environment which can be sensed (including elapsed time).

A task specification thus integrates a) continous control algorithms with b) discrete control logic in such a way that it can act in or react safely, intelligently and determistic to a changing realtime system.

III. Modern Hybrid Machine Controllers

Industrial grade hybrid machine controllers often contain a Programmable Logic Controller (PLC) for discrete control and a Numerical Controller (NC) for continuous control.

Logic Control : The last decade, PLCs were programmed in one of the IEC 61131 languages, allowing to formulate input/output relations using a ladder diagram or Graphcet diagram. A recent standard, IEC 61499 added event based function blocks. The tasks in a PLC are defined by the "wiring" of functions. Be it discrete logical components as in the ladder diagram, or more sophisticated IEC 61499 Function Blocks, the task to perform is specified by connecting these parts in a particular order. This allows high determinism, but changes are hard to make, i.e., what a task does is not directly seen from the PLC program, it only shows how the task is done. Loops and branches can not be expressed in the PLC languages and a sequence of operations is hard to express in the ladder PLC languages.

Motion Control : Trajectory planning, kinematics, continous state space control and etc. are executed by numerical motion controllers. A "part" program written in G-code of the ISO 6983 standard is an example of the

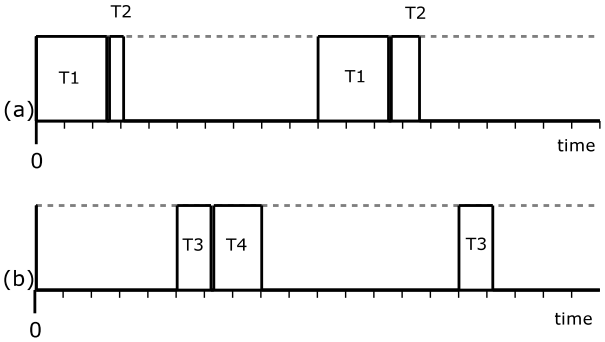

Fig. 2. Thread (a) executes task T1 and T2 which have equal periodicity. Thread (b) has another priority and executes T3. T4 is an aperiodic task with the same priority as T3.

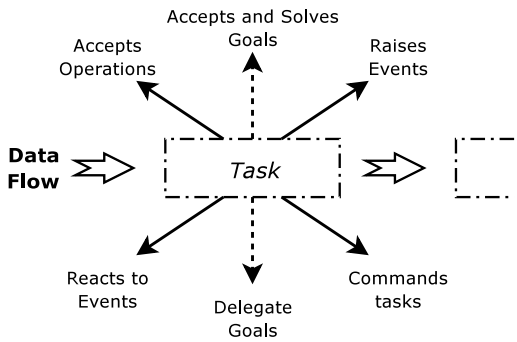

Fig. 3. Task peers can interact in many ways. In the role to accept and to solve goals, it accepts operations and emits events. In the role to delegate goals, it commands other tasks and reacts to their events. The data flow allows tasks to periodically process data in a chain.

specification for a motion control task and is the most common way to specify the operations a machine must execute. Each line in the program contains an operation the machine has to perform, such as moving with a velocity along a trajectory, but without execution branches. It relies on a PLC for delegating tasks with mainly a logic component (like a tool change). Programs written in G-code are fully deterministic, but totally unreactive. Only one program can run in the controller. Errors must be detected by external software or the PLC, which then aborts the program.

Although the PLC-NC controller solution performs well in deterministic environments, it is not suitable for solving online tasks. When looking at how the forces influence the task specification, it can be seen that flexibility is sacrificed for determinism and discrete and continuous control are almost not integrated or interactive.

IV. Task Design

A task was defined as a set of goals which must be reached by an intelligent controller. For example, the Motion Controller presents the tool change as a task to the PLC, while a tool change is programmed as a Grafcet task in the PLC.

A. Parallel execution

Any sufficient advanced realtime system has to complete multiple tasks in parallel, possibly at different priorities. Computer science has invented threads to express parallel "threads of execution" within one process. We have mapped tasks to threads as such: a. Periodic tasks with equal priority are executed in the same thread and in the same order as they are started. b. A periodic task's priority is inversely proportional with the duration of one periodic step, or lower. c. Non periodic tasks can have any priority. d. Non periodic tasks are executed after all periodic tasks. Fig. 2 shows the serialisation of tasks. The advantages of this approach are: 1. Scheduling overhead is minimised between tasks of equal priority. 5. Non periodic tasks do not influence the time determinism of periodic tasks of the same or higher priority. 3. Tasks with equal priority can communicate data lock-free. 4. The order of equal priority task execution is deterministic.

B. Inter Task Communication

Allowing deterministic time task execution is the first step in designing a machine controller. However, when tasks of different priorities need to communicate data, some form of locking is still needed. This would introduce non deterministic execution times (jitter) to the executed tasks. We have applied wait- and lock-free data exchange between tasks [12]. We identify two kinds of data exchange: a. periodical data exchange, which goes over a fixed interface between two tasks, this forms the data flow. b. aperiodic data exchange which is tied to the occurence of events, this forms the execution flow. Fig. 3 shows how tasks communicate using events and an operation interface, constituting the execution flow and periodical data exchange forming the data flow. We call tasks which use these communication forms "Peers", as there is no imposed hierarchy. In the next sections, we will explain the details of those two types of data exchange.

V. Data Flow: Data Objects

When two peer tasks exchange a fixed set of data periodically, in such a way that the second task processes the data of the first task, a flow of data, from one task to another can be observed. According to section IV, periodic tasks of different priority need to guard the data exchange. Lock free buffering techniques have been devised [3] to take care of this issue. Orocos provides data objects to encapsulate such buffering. Its interface has a *set* and *get* method which encapsulate the data exchange and thus provides a classic producer / consumer interface, with the guarantee that the get method always returns the most recent set value.

The kind of data exchanged through a data object is application specific. Data objects are single writer multiple reader.

VI. Execution Flow: Events and Operations

In this section, we define what an event is and how a task can react if an event occurs. The reaction may start a program which may request operations from other tasks.

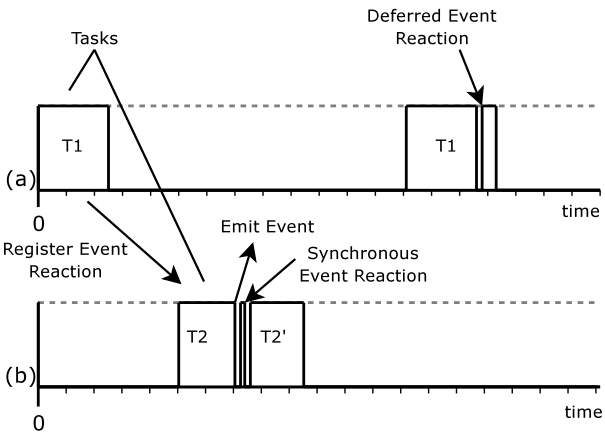

Fig. 4. Task T1, running in thread (a), registers a synchronous and asynchronous reaction (event handler) with task T2, running in thread (b). Both reactions are called on different moments in different threads.

A. Events and Reactions

An event can be described as a means to "publish" any change of the state of a system, which might in turn change the state of an observing system. Data may be associated with events, providing additional information about the state transition. For example, in Graphical User Interfaces [13] the event "Button-Pressed" can contain data denoting which button was pushed. In software, events are implemented using the Observer software pattern [9] and reusable implementations are freely available. We extended [10] to accomodate for thread-safe asynchronous event handling.

A task which is interested in the occurence of an event must associate a reaction, also called handler or hook, with the event. We allow synchronous, i.e. immediate, reactions and asynchronous, i.e. deffered, reactions (Fig. 4). Immediate reactions are executed by the task raising the event, deferred reactions are executed by the task which owns the reaction. The latter thus takes the burden of executing all reactions from the event emitter, and introduces a delay to the reaction, proportional to task's priority. Another advantage of deferred reaction is that the reaction is executed on a safe moment, when it can not corrupt the parallel execution of the task.

When the task's thread of execution is non periodic, the task must specify at which moments it wants to execute its deferred event reactions.

B. Operations and Commands

A task can define an operation interface of which other tasks can make use to solve goals. It is possible that a task defines no operation interface and that it solves built-in goals when it is started. An example of this are the PLC ladder programs, which goals are defined by wiring inputs to outputs using logical operations. All operations must be serialised over time, meaning that two operation requests should never be executed simultaneously. This requires a command infrastructure to do the serialisation.

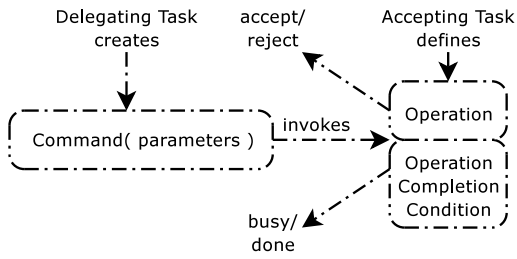

Fig. 5. A delegated command invokes an operation. On execution, it can be accepted or rejected by the accepting task. A completion condition checks if the requested operation is finished.

We define a command (Fig. 5) as the request from one task to another to perform an operation with given parameters. An example is a "move to" command with the location to move to and velocity as parameters. A task may reject the request, if it is still busy processing another command or if the parameters are incorrect.

Because operation requests must be serialised, a task must request an operation through a command and not directly on the other task (like what a traditional functional program would do). Commands are thus objects which store the operation request and are executed after the other commands in the pipeline. The execution is done by a command processor within the task which manages the command queue. The insertion of a command in the queue is again wait- and lock-free [20]. If the command is accepted, it will take some time before it is completed. Each operation defines a boolean condition which, when evaluated, returns true if the command is done. This is called the completion condition of the operation. For example, the command to set the gain parameter of a proportional controller, will most likely be immediately accepted and immediately completed. On the other hand, a command to move to a position may be accepted when there is no other move command in progress and when accepted, is completed when the position is reached. This method thus allows synchronous and asynchronous commands.

VII. The Hybrid Task Model

The previous sections analysed the individual requirements of a task in the discrete and continuous domain. A task (Fig 6), wether it is periodic or non periodic can be decomposed in an interface, state machines, and programs. This section will illustrate a synthesis of the presented solutions in a generic task model with an example robot setup.

A robot needs to pick and place an object, based on a camera detection algorithm. We will implement the positional control task and the pick and place task.

A. Positional Control

Positional control is done by a group of cooperating tasks: reading and monitoring the position sensors, generating an interpolated setpoint, invoking the PID algorithm and sending the results to the axis effectors. These tasks are periodic and always done in this order at the

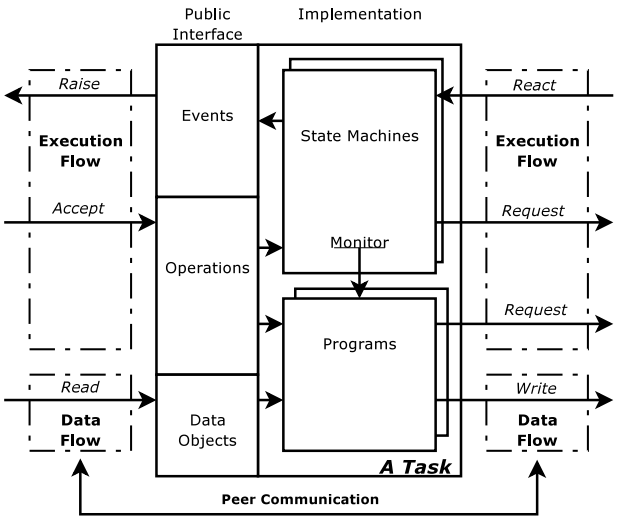

Fig. 6. A Task's public interface is composed of events, operations and data objects. It executes a program which is monitored by a state machine.

same frequency. Following section IV, they are grouped in one thread. They communicate positions and velocities using the data objects of section V. The task's activity is located in a program. The PID task's program will read the values of the data objects, feed them to the PID algorithm and writes the results to the outgoing data object. A program can also contain a sequence of commands and expressions, with branches and loops which are executed by the command processor. Orocos provides a scripting language which can convert programs online to a command object tree which is used for realtime execution of program scripts.

The correctness of the PID algorithm is monitored by a state machine, which runs synchronously after the algorithm. It can detect exceeding tracking errors, but also reset the control parameters on startup. Each task can have any number of programs and state machines running in parallel, given that they work with commands. Like programs, they can be defined in a script and loaded at runtime. The positional control state machines export the *'PID tracking error', 'Interpolator position reached', 'Sensor range error', 'Gripper opened', 'Gripper closed', 'object grabbed' and 'emergency stop'* events.

The first three events form a link from the continuous domain to the discrete event domain. This is an important aspect of the hybrid task specification: the events publish what happened, without exposing how it was detected, allowing a reusable and extendable design.

The operations the tasks support are *switch on, switch off, move to position, reset, open gripper, close gripper* and *emergency stop*. Depending on the application, these operations are called on the individual tasks or on a higher level tasks which groups the positional control tasks. As more programs are added, more operations will appear in the interface, and more events will be exported. As mentioned in section VI-B, these operations will be serialised

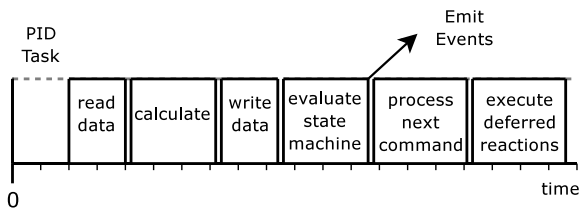

Fig. 7. The order of all functionality in one periodic step of a task. The next command may only be processed if ready.

after the task has done its previous operation and it will be processed right after a periodic execution step, thus not interfering with whatever the task was doing (Figure 7).

B. Camera Detection Task

A Camera sensor is added. A non periodic task grabs the image and detects the type of the presented object. The task then emits the *'object A detected'*, *'object B detected'* or *'no object detected'* events and waits again until it is started. The priority of this task is lower than the controller tasks so it does not interfere with the periodical control. The only operation it supports is *'start detection'* which starts the task's detection program and it uses no data objects.

The Camera Detection Task is suitable for network distribution because the operation is asynchronous and an event reports the result back. Standards like the Realtime CORBA Event Service [11], [16] can be applied here to make the task remote. The task interface allows to distribute them over a network, while keeping time critical functionality, i.e. the state machine, local.

C. Selection Task

This task uses the *Positional Control* and *Camera Detection* tasks. It uses a state machine to monitor the states *'homed', 'moving without object', 'moving with object', 'taking object', 'releasing object', 'detecting object'* and *'error detected'* with each monitoring constraints. For example, *'moving with object'* requires that the *'object grabbed'* event is emitted. If the state detects a failure, it switches to the *'error detected'* state which stops the robot. Of course, more intelligent reactions can be defined if more states are introduced.

The selection task defines the *'run n times'* and *'stop'* operations and exports the *'operation n done'* event, which informs other tasks about completed runs. The task has one main program which is started when *'runs n times'* is requested. Each run, it starts a program which opens the gripper, approaches the piece and closes the gripper. Next, the main program waits for the *'object grabbed'* event and then starts a program which moves the object in front of the camera. The main program then waits for one of *'object A detected'*, *'object B detected'* or *'no object detected'* and invokes a program to drop the piece in the A, B or reject container. After *n* runs, the completion condition of the operation *'run n times'* will evaluate to true and the owner of the command will know that it is finished.

The line between program and state machine is sometimes very thin. When the program needs to perform mostly sequential actions, it may be implemented with a state machine, since multiple state machines can run in parallel or hierarchically.

D. Task states

The task, state machine and program also have a state of their own. The PID controller task can be loaded or unloaded and started or stopped. A task must be loaded in order to be started or accessed by other tasks. A task defines functions which are called when its state changes, like the PID controller task which will start its state machine which monitors the tracking error. Some commands, like the interpolators move command, may be rejected if the task is not yet started.

A program has the same states of a task, but it can only be loaded when the task is loaded and can only be started if the task is started. The program to move the object in front of the camera is started indirectly by the *'object [A—B] detected'* event. This event will first be intercepted by the state machine which, if it occured in the correct state, starts the program.

The state machine can be loaded when the task is loaded and started when the task is started. Our state machines have one initial and one final state and any number of intermediate states. The initial state of the interpolator will read the current positions and write them as reference positions in its setpoint data object. It remains in that state until it is started. After it is started, it periodically checks if a move command was accepted and starts the interpolation program if so. When the state machine is stopped, it makes a transition to the final state, which may cause a safe stop program to run. In the final state, it waits to be reset to the initial state or it is stopped.

E. Task hierarchies

Once tasks rely upon each other to reach their goals, a hierarchy is formed. The selection task uses the interpolator task to move to a new setpoint. If a task uses the operations of another task, the latter exists as a child of the first, and it must be loaded first. This hierarchical constraint is not present if a task relies on the events of another task, since events decouple sender and receiver. If a task processes a data flow, it requires a data object to read from which must be produced by another task. The task itself produces a new data object where other tasks down the line can read from.

These connections between tasks can easily become more complex than parent-child relations. The connections can be set up when all the tasks are loaded. A task will only start if no topological errors are present. This mechanism is part of the *configuration flow*, but is not the subject of this paper. Semantic errors, like out of range sensor readings, can be detected by a state machine.

VIII. Conclusion

The proposed application independent task specification allows advanced, concurrent, realtime tasks to communicate and interact in a deterministic way. The task infrastructure is an enabling technology which makes hybrid controller design easier to implement without enforcing implementation restrictions. The controller tasks are hybrid by design and provide a universal interface for communication. This interface also defines the granularity at which distribution in control is possible.

Acknowledgment

All authors gratefully acknowledge the financial support by K.U.Leuven's Concerted Research Action GOA/99/04, the Belgian State—Prime Minister's Office—Science Policy Programme (IUAP), the Flemish I.W.T. project *Open Machine Control*, and the EU projects *Orocos* and *Ocean*.

References

[1] Open system architecture for controls within automation systems (OSACA). `http://osaca.isbe.ch/osaca`.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 28–37, December1995.

[4] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, Boston, MA, 1998.

[5] J. J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, and D. S. adn N. Turro. The orccad architecture. *International Journal of Robotics Research*, 17(4):338–359, april 1998.

[6] H. Bruyninckx. Open RObot COntrol Software. `http://www.orocos.org/`.

[7] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10), 1997.

[8] S. Fleury, M. Herrb, and R. Chatila. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Proceedings of the 1997 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 842–848, Grenoble, France, 1997.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[10] D. Gregor. The boost signals library. `http://www.boost.org/doc/html/signals.html`.

[11] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time corba event service. In *Proceedings of OOPSLA '97, (Atlanta, GA)*, October 1997.

[12] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[13] T. Inc. Qt 3.3 whitepaper. `http://www.trolltech.com/pdf/whitepapers/qt33-whitepaper-a4.pdf`.

[14] W. S. Levine. *The Control Handbook*. CRC and IEEE Press, 1996.

[15] Ocean. Open Controller Enabled by an Advanced real-time Network. `http://www.fidia.it/english/research_ocean_fr.htm`.

[16] D. C. Schmidt and F. Kuhns. An overview of the real-time CORBA specification. *Computer*, 33(6):56–63, 2000.

[17] S. Schneider, V. Chen, G. Pardo-Castellote, and H. Wang. Controlshell: A software architecture for complex electromechanical systems. *International Journal of Robotics Research*, 17(4):360–380, april 1998.

[18] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *Trans. on Software Engineering*, 23(12):759–776, 1997.

[19] J. D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, 1994.