MASTER

Performance modeling based on skill primitive nets

Krijnen, M.

Award date:
2009

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Performance Modeling based on Skill Primitive Nets

By
Marijn Krijnen (s0494620)

Supervisors:

Prof. Dr. U. Goltz (Tu-BS)
Dr. Ir. P.J.L. Cuijpers (TU/e)

Eindhoven, July 7, 2009

Abstract

Parallel Kinetic Machines show great perspective to improve industrial tasks and are subject to much research. At the collaborative research center 562 at the technical university of Braunschweig, a robot control system for these parallel kinematic machines has been developed, that provides for hybrid movements and an intuitive programming language. A program in this language is called a Skill Primitive Net and is built from simple robot commands called Skill Primitives. In order for the robot control system to be able to execute such a Skill Primitive Net safely on a Parallel Kinematic Machine, the control system has hard real-time requirements.

Future extensions of the Skill Primitives and the robot control system might endanger the hard real-time requirements. Performance Analysis beforehand of execution of a Skill Primitive Net provides the user with the necessary data about the resources that the Skill Primitive Net will use. This study analyzes four different performance analysis methods with respect to the robot control system, namely holistic analysis, timed automata, real-time calculus and hybrid automata. Based upon different Skill Primitives, different modules are activated within the robot control system. It should be possible to generate a corresponding model from a Skill Primitive Net. We will show that, for now, Hybrid Automata provides the most structure to automate the modeling of this system, but the tooling support for Timed Automata is much better. We will provide recommendations for further development of the methods that can improve their usefulness for this system.

Preface

During my study of Computer Science at the Eindhoven University of Technology, I became interested in modeling work and logic. Creating models of architectures, identifying problems, weaknesses, strengths that can be used during architectural steps or verification steps seemed an interesting task. Also the ideas behind modeling methods appealed to me very much. I would like to continue working with models also after graduation.

Because of my interest for models, I decided to visit the departments that provided most courses on this subject. These involved Formal Methods, Design and Verification of Systems and Information systems. During an initial talk at the formal methods group, I was put in contact with Pieter Cuijpers. He showed me a project running in Braunschweig that involved modeling a special robot. Somehow robots have always appealed to me and this made me choose for that specific project.

The research group in Braunschweig proposed to work on performance modeling of the robot. Performance modeling was still a bit unknown for me since the university does not provide many courses on that subject. Out of curiosity I accepted so I could again learn another way of modeling systems, which finally resulted in this work. The project provided me with great insight in the problems that still exist in modeling distributed systems and I was surprised to discover that formal methods and tools were still at their early stages for this.

There are several people I wish to thank for their support during my graduation project. At the risk of accidently missing out somebody, I would like to express my gratitude to them here:

First of all I would like to thank Dr. Ir. Pieter Cuijpers for his willingness to supply me with both an interesting graduation project and advice during the project.

Secondly, many thanks go out to the Institute For Programming and Reactive Systems (IPS) at the university of Braunschweig for their corporation and their insights in this project. In special I would like to thank Prof. Dr. Ursula Goltz and Dipl.-Inform. Jens Steiner, who made it possible for me to spend a month at the university for the necessary insight in the robot control

system and provided me with valuable information.

Dr. Ir. Reinder Bril and Prof. Dr. Jos Baeten, I thank for their willingness to be part of my examination board together with Prof. Dr. Ursula Goltz and Dipl.-Inform. Jens Steiner. This despite the fact that this thesis would arrive to them on a rather short notice. Hereby, I want to extend my sincerest apologies for that.

Other people I want to thank are;

Jochen Maaß for his tour around the robot facility in Braunschweig. The people at the Laboratory for Quality Software, to provide me with a place to work and help when needed. All students and members of the Formal Methods group, who although I was not much present at that group took the time to view my intermediary presentation and provide feedback. The people at the International Office in Braunschweig, for providing a room to stay for a month and helping me with everything I needed for my stay there.

Last but not least, I want to thank my family for their support during my time at the university. In special my parents, Frans and Hennie Krijnen, who gave me all their support. You all have my gratitude.

Marijn Krijnen
Eindhoven, July 7, 2009

Contents

List of Figures

Chapter 1

Introduction

For many years now, Parallel Kinematic Machines (PKM) have been subject to research. Parallel Kinematic Machines differ from classical serial robots in a way that the end effector of the machine is controlled by multiple limbs that are connected to the base of the machine. The actuators control the limbs at the base of the machine and through a series of passive joints the end effector is controlled. This architecture allows for high stiffness and high accuracy. Machines can reach high speeds and have a fast acceleration. They show great potential to increase production in factories over serial machines, especially in areas where high accuracy and speed are required, like pick and place machines or milling machines. Drawbacks of the Parallel Kinematic machines are a small workspace, the inability to work around obstacles, mechanical limitations and the possibility to reach a singular position. Also calculations that need to be done to perform a desired manipulation are generally harder.

At the collaborative research center 562 at the Technische Universität Carolo-Wilhelmina Braunschweig, research is being done on Parallel Kinematic Machines. A long-term goal of the collaborative research center 562 is to automate assembly planning. One of the products of this research are Skill Primitives, a programming interface for robots. A robot control system has been developed that implements this Skill Primitive language and controls a robot. The high velocities and accelerations that can be reached by a Parallel Kinematic Machine, require short control cycles up to 125 μs. This induces hard real-time constraints on the software. Because of the nature of the machine that is being controlled, it is very important that these hard real-time constraints hold. Failing to do so results in uncontrolled behavior, which might damage the structure of the Parallel Kinematic Machine.

Until now the control software has always been executed on reasonably fast machines. Now questions arise about the minimum system requirements needed to safely control the Parallel Kinematic Machines behavior.

Skill Primitives allow for position controlled, force controlled and velocity controlled robot movement while future extensions, like vision controlled

movement, are being considered. The algorithmic load and bus load to transfer vision data and evaluate this data require more resources. Questions arise if the current architecture will handle this extra load.

Previously, the system has been analyzed by the system analysis tool SymTA/S. However this analysis used the assumption that all modules of the controller are active at the same time. Depending on the Skill Primitive that is active, different modules become active or inactive. In other words, the SymTA/S analysis of the system did not fit the task description of the Parallel Kinematic Machine that is contained within the Skill Primitive Net. It assumes that every motion module becomes active while during run-time this might not be the case.

In this Thesis, we will describe ways to generate models based on Skill Primitives, therefore being more precise. The models should also be easily adaptable or even generated in case of small changes to the system.

- **RQ1**: What is the best method for modeling the robot control system?
- **RQ2**: How can we build a model from a Skill Primitive Net?
- **RQ3**: Can the model be changed easily if the robot control system changes?

Studies of comparison of different performance analysis methods are limited and do not show proper advantages and disadvantages to certain modeling methods [PWT+07]. The approach taken was to choose a model checker or toolkit for every performance analysis method we selected. We build a model of the worst case scenario in every method we selected using the model checker or toolkit. Based on these models, we analyze strengths and weaknesses of the methods selected. We discuss ways to automatically generate a model in these methods and finally see how architectural changes would affect the model.

Chapter 2

Nomenclature

This thesis contains some terms that are frequently used. In order to prevent confusion, we present our interpretation of these terms.

In this thesis we talk about *tasks*. A task can be seen as a block of sequential code that can be executed. These tasks can overlap in time and depending on the scheduling algorithm a task will either get permission to use the resource or not. Tasks will enter different states based upon that. When a task is not competing for the resource, in other words the task is not active, we will call the task *idle*. An internal or external *event* can activate the task, a task will compete for the resource to be able to execute, these tasks are called *active* tasks. An active task can also be in a different state, when it is waiting for the resource to be assigned to him, we call the task either *ready* or *waiting*. Once the task gets the resources it is called *executing* or *running*. A running task might get interrupted if a higher priority task becomes active and claims the resource. In this case the current running task stops running and becomes a *preempted* task. Another possible interruption of a current task is when the task requires a certain resource that is locked by a lower priority task. This is called priority inversion. In this case the task gets blocked until the shared resource is free again. This is a *blocked* task [But04].

We will discuss scheduling methods and algorithms. Many of these are described in [But04]. Scheduling algorithms are used on systems where two tasks might become active at overlapping times and the system has to decide which task gets precedence over the other. The scheduling algorithms are either *preemptive* or *non-preemptive*, meaning they can suspend a currently running task to allow another to execute or they cannot suspend a task and have to wait till the current task finishes its execution. The most commonly used algorithm is *Fixed Priority*, in which tasks have a fixed priority that places them in a hierarchy that can decide which task to execute first. This method is very popular since it is implemented in most operating systems. Other scheduling algorithms we name are *Earliest Deadline First*, which means that the task with the closest deadline will be allowed to run first and *First In First*

Out, were the activation of tasks fall into a large buffer and a task is allowed to execute based on when it was activated.

Finally, *mutual exclusion* will be mentioned. In case two tasks share a resource, mutual exclusion gives exclusive rights to the resource during a *critical section*, the section where the shared resource is in use. When a lower priority task claims a resource and gets preempted that resource stays locked, when the higher priority task wants to claim the resource it gets blocked on that resource by the mutual exclusion algorithm. To prevent that a higher priority task gets blocked longer because the lower priority task is preempted by a task that has a priority in between them, two protocols exist. The *priority inheritance protocol* and the *priority ceiling protocol*. Suppose that a higher priority task is blocked by a lower priority task in its critical section. The priority inheritance protocol requires that the lower priority task executes its critical section at the priority of the higher priority task. This prevents task with a priority in between both to suspend both tasks. The priority ceiling protocol assigns a ceiling priority to resources. Once a task locks this resource it temporarily assumes the ceiling priority as its own until it unlocks the resource again, preventing other tasks to suspend the task while it is in its critical section. More information about these protocols can be found in [But04].

Chapter 3

Robot architecture

In this chapter the general robot architecture and the concept of Skill Primitives will be explained. The architecture consists of three parts that are interesting for our performance model, the robot actuators and sensors, the controller and the communication bus, see figure 3.1 [SHM06].

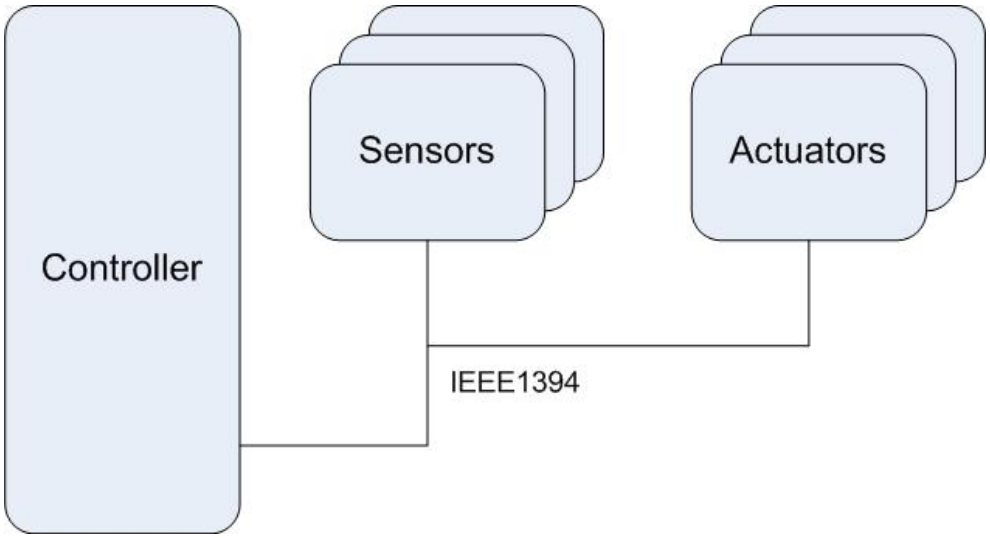


Figure 3.1: Robot control System architecture

We shall also provide the necessary information about Skill Primitive Nets here.

3.1 Skill Primitive Nets

Skill Primitives are simple robot commands. They hide the specific needs for a Parallel Kinematic Machine and provides the programmer with an intuitive programming interface, that is independent of the topology of the Parallel

Kinematic Machine. The main goals of the Skill Primitive Programming language are

- Generality (e.g. an uniform way to program robots with complex sensors)
- Robustness (e.g. able to correct for small uncertainties)

TF = {T1, HF, HF}					
Dx	Control	Dy	Control	Dy	Control
20	POS	40	POS	15	VEL
φx	Control	φy	Control	φz	Control
0	POS	0	POS	0	POS
τ = {}					
λ = (Fz > 15 N) v (Pz > 50 mm)					

Figure 3.2: Example skill primitive

Tool commands are commands that can be executed by the end effector. In case we have a gripper as end effector a tool command can either open or close the gripper.

Definition 1 *[TWMH05] A SP* $:= \{HM, \tau, \lambda\}$ *where*

- HM *is a Hybrid Move*
- τ *represents tool commands*
- λ *is the stop condition that finishes the Hybrid Move when expression evaluates to true*

Skill Primitives currently support 3 types of hybrid movements, these are either position/orientation based, velocity/angular velocity based or force/-torque based. New types of movements like relative position based on vision are currently in research.

A hybrid move contains two parts. A Task Frame (TF) or Reference Frame from which the hybrid move is executed. All movements are relative to this position. Examples of this Task Frame are, the Hand Frame (HF)

which causes the task frame to be placed on the position and orientation of the end effector. The Base Frame (BF) which places the Task Frame on the base of the robot, or the world frame (WF) which allows the Task Frame to be anywhere in the world with any orientation. The Task Frame can be anchored (ANC) during execution of a Skill Primitive, in that case the Task frame will follow the position and orientation of the frame during execution and moves accordingly [FKW05].

The second part is a six degree of freedom system, where the first three variables represent the position, velocity and force, and the last three variables represent orientation, angular velocity and torque.

Definition 2 *[TWMH05] A Hybrid Move* $HM := (P, RF, ANC)$ *where*

- $P \in \mathbb{R}^6$
- $RF, ANC \epsilon (HF, WF, BF)$

A Skill Primitive Net (SPN) is a number of Skill Primitives in a graph like environment. A Skill Primitive Net has an initial and final state and a number of states in between. Every state contains of one Skill Primitive that is being executed. When a Skill Primitive reaches its stop condition, it will jump to the next Skill Primitive. Note that based on which stop condition becomes active a different Skill Primitive might become active and no two Skill Primitives can be active at the same time.

Definition 3 *[TWMH05] A* $SPN := \{\Sigma, \Pi, \Xi, \Upsilon, \Omega\}$ *where*

- Σ *is a finite, not empty set of SP*
- Π *is a finite not empty set of start states*
- Ξ *is a finite not empty set of stop states*
- Υ *is a finite set of error states*
- Ω *is a finite set of guarded directed arcs linking the states together. The conjunction of all the outgoing arcs defines the stop condition of the current SP.*

3.2 Controller

The most important part of the robot control system is the controller. The modular design of the robot control system allows for easy replacement of modules and layers in this controller so other parallel robot topologies, sensors, actuators and skill primitives can be used without the need to replace the whole control software.

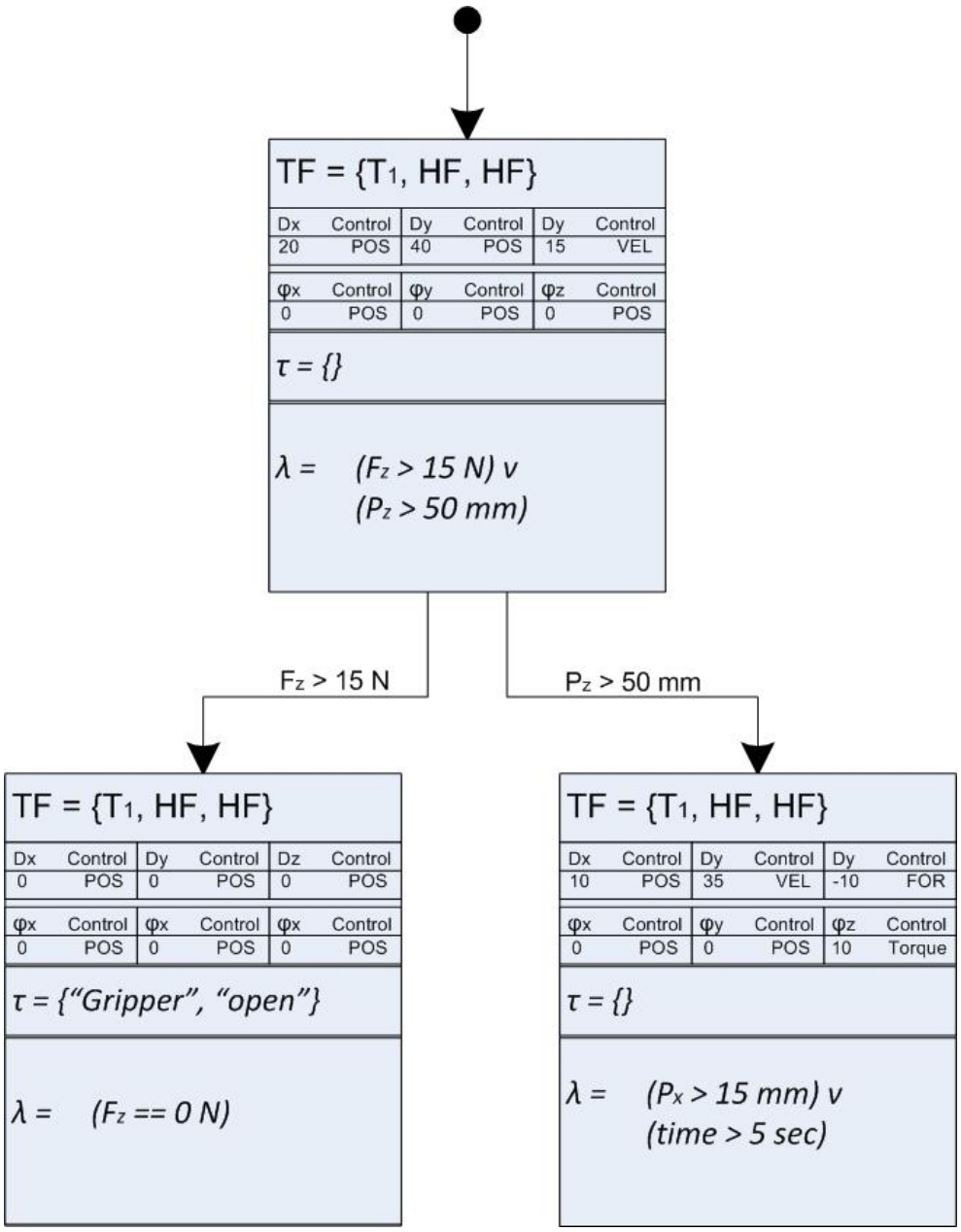


Figure 3.3: Example Skill Primitive Net

This control unit or controller has several layers in its software. The first layer is the inner control loop and can be seen in the first row of figure 3.4. Tasks in this inner control loop have hard deadlines. The inner control loop directly communicates with the DSPs of the robot and sends the nominal values to them. This layer is different for every different Parallel Kinematic Machine, because it contains precise knowledge of the Parallel Kinematic Machine's topology, which is needed to write the right values to the actuators. A cycle of the inner control loop takes 1000μs which is split into two parts. The first 750μs are reserved for the Industrial Automation Protocol (IAP), which communicates new values to the DSPs through a Master Data Telegram (MDT). The DSPs respond by changing actuator values were necessary and calculate new actual values and send those back to the controller through Device Data Telegrams (DDT).

After the 750μs are completed, the inner control loop needs all Device Data Telegrams to have been received and starts calculating new values for the next Master Data Telegram. It starts with the Industrial Automation Protocol again, which will then activate the HardWare Monitor (HWM). The HardWare Monitor is, next to monitoring, also responsible for the shutdown process in case a defect is detected. The HardWare Monitor activates two other tasks of which one also belongs to the inner control loop, the Drive Controller (DC). The Drive Controller will calculate new values for the actuators based on a set of coordinates it gets from the second layer and also activates two other tasks. The inner control task that is activated is the Smart Materials Controller that is charged with the job of reducing vibrations due to the high speeds a Parallel Kinematic Machine can produce. These tasks have to be finished within 250μs, after which a new cycle will start. The inner control loop is activated by a Cycle Start Telegram (CST) that is generated by the communication bus every 125μs. Every first and seventh Cycle Start Telegram will activate the Industrial Automation Protocol. Remaining CPU time will be used by the lower layers.

The second layer or outer control loop consists of tasks that will calculate proper values for the inner control loop to interpret. It will provide a movement value for the end effector of the robot which the inner control loop can translate into the proper actuator values. The outer control loop is activated by the Drive Controller every time the inner control loop has been executed for a fixed number of times. In this thesis the control system activates the outer loop every two inner control loop cycles. Other Parallel Kinematic Machines can have lower requirements for the outer control loop. The first task that gets activated is the Core Controller (CC) and gets activated by every second execution of the Drive Controller. The Core Controller will make sensory data available for the motion modules, these motion modules will be activated by the Core Controller only when needed for the Skill Primitive, that is currently being executed. Due to the modularity of the architecture it is very well possible to add new motion modules that can interpret new Skill Primitive types

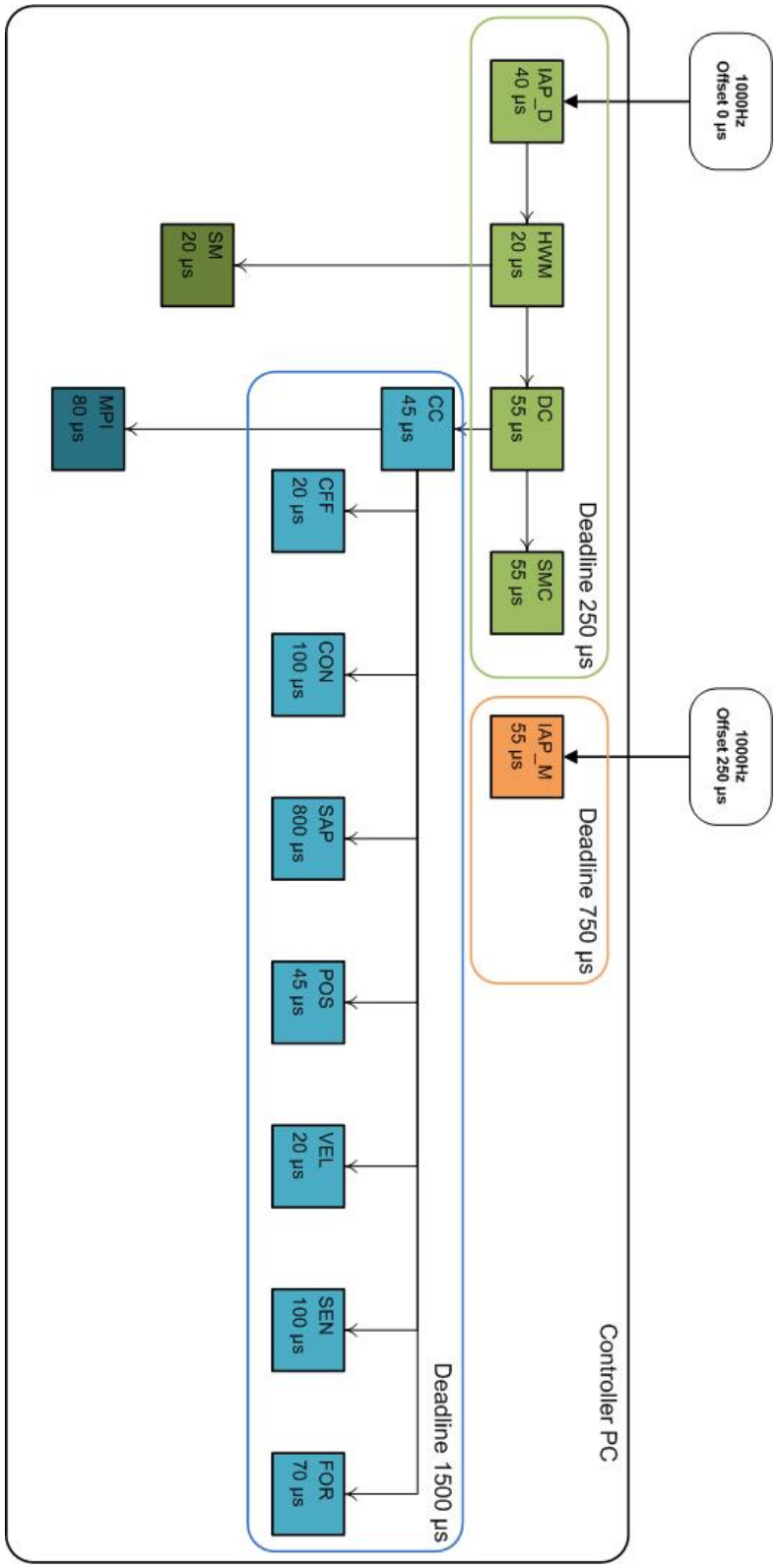


Figure 3.4: Controller tasks, priority decreases downwards

without the need to change the complete architecture. The controller software studied in this thesis, contains a Position module (POS), Velocity module (VEL), Force Module (FOR), Contact Module (CON) and a Force/Torque controller (CFF). Also a Sensor Module (SEN) and a Singularity Avoidance module (SAP) are activated by the Core controller. These modules can interpret skill primitive commands based on positions (POS), velocity (VEL) and Force (FOR, CFF, CON). The Singularity Avoidance Protocol (SAP) is dependent on the topology of the Parallel Kinematic Machine, since its primary job is to prevent singular positions, i.e. positions were due to the passive joints the machine is unable to determine which effect actuators have on the motion of the machine.

The second layer consists of Soft Real-Time tasks. Here, soft means that missing a deadline is not a disaster. In case the outer control loop is not able to finish in time, the inner control loop will enter an error state and tries to calculate stop values for the actuators. It is however preferred that the second layer also meets its deadline, since calculating a stop condition is difficult and should only be done under extreme circumstances. This thesis will therefore view any deadline misses in the second layer as a failure. The deadline of the second layer is harder to determine, it should finish before the inner control loop starts its 250μs tasks again. That would mean that the outer control loop will have a deadline of $2000 - (2 \times 250) = 1500\mu s$. This is however a very negative view on the deadline since the outer control loop is activated half way the inner control loop a deadline of $2000 -$ (Worst Case Execution Time of DC) $= 1945\mu s$ after activation is more precise, but if the outer control loop is not finished before the second inner control loop starts which will be around $1800\mu s$ it will never reach its deadline.

The third layer consists of tasks with no deadline. First there is the Self Manager (SM) which is activated by the HardWare Monitor. The Self Manager tries to improve the execution of the robot control software. The Self Manager does not have a deadline, but if it wasn't able to finish its execution before the HardWare Monitor runs again it will discard his current activation and start over again. The other task is the Manipulation Primitive Interpreter. This module will only be activated once the stop condition of a Skill Primitive has been reached. Once the stop condition has been reached the control system will message the Manipulation Primitive Interpreter through the CC. It will then wait for the Manipulation Primitive Interpreter to deliver the new skill primitive. This module has been omitted from our models, since this job does not break any deadlines and can potentially take indefinitely long to respond.

During this project we use the following worst case response time values. These values were calculated by Symta/S during run time [MSA$^+$08]. These values can change between different Parallel Kinematic architectures and different computing nodes.

The control unit schedules tasks according to a Fixed priority preemptive schedule. Tasks of the same priority are scheduled as First In, First Out. The

Task	**WCRT**
IAP_D	40
HWM	20
DC	55
SMC	55
IAP_M	40
CC	45
POS	45
VEL	20
FOR	70
CON	100
CFF	20
SAP	800
SEN	100
SM	20

Table 3.1: Worst Case Time values of every task in the controller software

tasks within the inner control loop are activated by a token ring. Tasks will release the token once they finish their task completely and pass it along to the next task. The priorities of different tasks have been visualized in figure 3.4. The priority decreases if the task is vertically lower in the overview.

The controller also uses several resources under mutual exclusion. In this case a task claims exclusive rights on a resource, which is also used by another task. Most of these are contained within the inner control loop and due to the token ring used in the inner control loop these will not interfere, since they can not preempt each other which results in never overlapping critical sections. Two however can interfere. The Drive Controller and the Core Controller can both claim exclusive access to the same memory block. Another shared resource under mutual exclusion exists between the Hardware Monitor and the Self Manager. They can also claim access to a certain memory block [MSA+08].

The controller uses priority inheritance to deal with shared resources. Priority inheritance will raise the priority of the task that has claimed the resource to the task that is blocked on the resource.

The controller has been developed to work with multiple computing nodes. The motion modules from the second layer can be assigned to another computing node. Depending on if this computing node is just another core in the current computer or a new computer within the network additional information has to be sent over the communication bus [KKR+04]. We will assume for now that the controller is executed on one processor.

3.3 Bus

In order to execute the movements a number of actuators and sensors need to be controlled. Every actuator and sensor is controlled by a Digital Signal Processing unit (DSP). These Digital Signal Processors interpret the messages from the controller and if necessary change the behavior of their corresponding actuator and calculate values that have to be sent back to the controller to update the internal status of the controller. These values will be sent to the controller by small packets called Device Data Telegrams.

The controller is connected to the different robot actuators and sensors by means of an IEEE1394b (firewire) connection. This bus is often used in audio and video transfer where drops in the quality of service are accepted. However the high bandwidth, the fast cycle time and the nearly jitter free Cycle Start Telegrams are ideal for real-time applications. In order to make use of these advantages the Industrial Automation Protocol was developed. There is an interest to use IEEE1394 more in industrial automation.

IEEE1394 has a cycle time of 125 μs. Every cycle starts with a Cycle Start Telegram (CST) sent by the root node[1], to synchronize the clocks of all devices. The Cycle Start Telegram is also used to start the control cycle of the robot controller. A control cycle should be started every 8 Cycle Start Telegrams. Every first and third Telegram (or first and seventh telegram, depending on which of the two you see as the first) activate a task on the controller computer.

Firewire has two different modes in which it can run, the asynchronous transfer mode and the isochronous transfer mode. Isochronous transfer can be used for transferring data in a real-time environment. The robot control system is able to operate in both modes however.

Isochronous transfer mode guarantees periodic data transmissions, every node gets to send its information periodically. During initialization every node may request a certain amount of bandwidth. A Resource Manager on the root node allocates bandwidth every cycle for the node to send its data based on these requests. Whenever a node has sent its data, there is a small gap on the bus in which no data is being sent. Other Isochronous nodes detect this gap and can then compete to send their data. This gap is called an isochronous gap.

Asynchronous transfer mode allows asynchronous messages to be sent after a subaction gap. A subaction gap is a prolonged isochronous gap, so when all isochronous nodes have sent their data, the isochronous gap will be prolonged to a sub action gap. When a node detects a subaction gap, it sends a request to the root node. The first request to reach the root node is allowed to use the bus. This is most often the node that is closest to the root node in the tree.

[1] devices are connected as in a tree in the firewire protocol with the first device as root node

Task	WCRT
MDT	128
CST	4
DDT	28

Table 3.2: Packet length in bytes

After this the node can send its data and closes with a small gap after which an acknowledgement packet should be received. This completes the asynchronous transfer and the nodes can send a request for the bus again after a subaction gap. A Cycle Start Telegram will be sent at the beginning of the next cycle. If at that time the bus is still busy sending an asynchronous packet, the Cycle Start Telegram will wait till the bus is free again. The IEEE 1394 protocol will make sure however that the next Cycle Start Telegram will be sent exactly 125 μs after this Cycle Start Telegram should have been sent. This delay is only disallowed for the key cycle start telegrams that also synchronize the controller. If those are delayed, the robot will get into a state of uncontrolled behavior.

Firewire also has methods to ensure that fairness is guaranteed in case of asynchronous transfer and that no node can claim the communication bus indefinitely. Once a node sends a message the node gets flagged and can not send anything else before the fairness interval expires. A fairness interval is recognized by the system due to a arbitration interval gap, which is a prolonged version of the subaction gap indicating that every node has been marked as having send their data once, therefore assuring fairness. Since every module in the robot control system only sends a couple of packets every 1000μs, we can ignore the fairness principle in our models for now. In that case asynchronous transfer mode will act like a fixed priority non-preemptive schedule. A proper modeling of the firewire protocol in the different methods was not possible in the scope of this project.

In firewire every node can either send asynchronous or isochronous packets. During the 125 μs cycle, the isochronous packets will have a higher priority and be sent first. Once the isochronous packets have been sent or 80% of the bandwidth has been used, the bus will allow asynchronous packets to be sent. The robot control will either use asynchronous or isochronous transfer mode [And99] [SHG07] [KVSG04].

In order to calculate worst case response times we need to know the time it costs in microseconds to sent a packet of information over the bus. The following values were acquired [MSA$^+$08].

IEEE 1394b has a speed of 800 Mbit/s, which is 100 Mbyte/s, which is almost 105 bytes/μs. The Cycle Start Telegram and Data Device Telegram will complete in almost an instant, while the Master Data Telegram takes a bit

more then 1 μs to complete. The packets listed above take slightly longer to complete, since an acknowledgement packet and a subaction gap are required to finish the transaction. The length of such a gap is less then 0,05μs.

Chapter 4

Performance Modeling

Performance modeling is often used in embedded systems engineering. These systems are generally designed to complete a small amount of tasks that are known during the design process. In other words the end-user cannot and should not change the programming of an embedded system. A performance analysis is able to indicate bottlenecks in the system and can provide minimum requirements for the hardware to minimize the cost or power consumption of the system, while correct behavior is still guaranteed. However in hard real-time systems the designer is generally interested in best case and worst case execution times, due to hard real-time requirements.

To model the robot control system (chapter 3) our performance analysis method has certain requirements.

- The method should provide correct results or in other words, there should not be a case where the system can exceed both upper and lower bounds of the analysis. In this thesis we are only interested in finding upper bounds.
- The analysis should provide accurate results. Some methods might produce approximations of bounds, these should be reasonably close to the true boundaries of the system.
- The method should provide structure to be able to automate the building of a model based on the Skill Primitive and the Parallel Kinematic Machine. Different Skill Primitives activate and deactivate certain tasks in the controller, it should be easy to not include certain tasks without changing the model to much. Also new tasks might be added to the controller or even extra processing units.

There are a number of different methods that can analyze or estimate the performance of distributed systems. The most popular methods in literature are simulation, stochastic performance modeling and formal performance modeling

There are a number of different formal performance analysis methods. They can either be classified as holistic scheduling, in case a whole system is analyzed at once, or modular, where the performance is analyzed per module and results propagate through the system often at the expense of accuracy.

Simulation based approaches, which are popular in industrial environments, can also be used to perform an analysis. Simulation however has the disadvantage that corner cases, the worst case and best case scenario that might happen, are hard to find without trying every possible input combination. Even if a corner case can be found, there is still the requirement of a proof that it indeed is a corner case. The possibility that a simulation does not represent the worst-case situation is a high risk within a safety system. Simulation can still be helpful in combination with a formal performance modeling method. A simulation that returns a bound that is close to a bound found by an approximating method can indicate that the method provides very accurate bounds. However no guarantees can be made if those two values are far apart, since either the approximation is too pessimistic or the simulation is too optimistic.

Another field of research on performance modeling is stochastic performance modeling. Stochastic methods can be used to provide for average values and analyze a certain quality of service, it does not give hard guarantees that are needed in safety systems. Therefor those are not considered within hard real-time analysis [PWT+07].

Up to now scheduling analysis at the collaborative research center 562 has been done in the system analysis tool SymTA/S. SymTA/S uses classical scheduling techniques to analyze modules within a model and uses the outcome to propagate through the system. In the beginnings SymTA/S was only used for verifying the real-time properties within the system. It is now used to work with the Self Manager to analyze the system at runtime. New analysis methods were developed for the analysis of firewire in SymTA/S. The model of the robot control system we use to analyze the system is based on the model that was generated for SymTA/S [SHG07].

4.1 Holistic performance modeling

Lots of different scheduling methods have been developed over the years, like Rate Monotonic scheduling, Earliest Deadline First and many others. Together with these scheduling methods came analyses about worst case response times for these methods.

As distributed systems become more popular, several proposals were made to extend the classical scheduling methods to distributed systems. The communication bus between different computing nodes becomes a factor of importance in these systems, since these can cause jitter inside the system. These proposals to add communication to the classical scheduling have lead to holis-

tic scheduling.

Holistic scheduling is a group of methods that, rather then considering individual modules inside the system, consider the system as a whole. There are many algorithms developed that deal with a certain kind of scheduling. This specialization allows for a more precise analysis.

A holistic performance modeling algorithm is tailor made for a specific kind of scheduling, input event model and resource model. If any of these change, it is likely that the algorithm also has to change. This fact makes it difficult to use holistic modeling as a tool to try out different architectures.

4.1.1 Modeling and Analysis Suite for real-Time applications

The lack of flexibility when using holistic scheduling is reduced by the introduction of the Modeling and Analysis Suite for real-Time applications (MAST). MAST combines a couple of holistic algorithms for the most encountered scheduling problems together in one toolkit. It allows for quick switching between different analysis algorithms and therefore overcoming the initial problems with holistic scheduling. The algorithms are generic algorithms to analyze a certain schedule. Since most models will not precisely adhere to this, the result of the analysis done by MAST might be pessimistic. A pessimistic result is over-approximating and therefore not precise.

The toolkit supports fixed priority and earliest deadline first. It also supports blocking delays due to shared resources, preemptive scheduling as well as non-preemptive scheduling.

The toolkit can be divided into two parts the design part and the analysis part. A system can be defined by a powerful text language that is supported by MAST. The powerful language has as disadvantage that it is also very complex and not very suited for manual creation. Tasks that we define in MAST are referred to multiple times in things like shared resources under mutual exclusion, processor resource, scheduling algorithms, etcetera. The probability that the user forgets something is very high that way. It makes automatic creation of a model very plausible though. Every aspect of the robot control system can be directly translated in this language. For other projects there is a graphical user interface available to build the model in and therefor greatly reducing the modeling effort.

The analysis tool is a program that allows you to set some options like the input file, the output file, the analysis method, slack times, etcetera. After a successful analysis the program returns a statement if the system is schedulable. The program also shows worst case execution times of every activity in a transaction. It does however not return information about buffers. In our case we are not interested in that since our activation buffers only consist of one space. In case the user is interested in buffers for for example burst activation patterns then the MAST toolkit in unsuited.

A model of a real time system is represented by a series of one or more

graphs called transitions in MAST. One of these can be seen in figure 4.1. Every transition is coupled to a scheduling method to represent the internal scheduling of the transition and to a resource like a processor. Every graph consists of a number of activities which represent tasks of the system. Every transition is triggered by an external event (e1 in figure 4.1 is an external event). MAST supports periodic, sporadic, singular, bursts and unbounded aperiodic events. Every activity also generates an event after execution, which can be used to start other activities. In case such an internal event is orange (o1 in figure 4.1) it contains a deadline that has to be met. Grey internal events do not have deadlines. These internal events can be delayed, so that they activate a next task after an amount of time [MMG+02].

Figure 4.1: Example transaction in MAST

Because MAST uses different algorithms, the input for MAST is a rather complex textual format. There is a graphical user interface to ease the modeling work, but this interface is rather unstable, causing it to crash and result in the loss of parts of the model. This leaves the user with considerable modeling effort for larger systems. Although the format is complex it is also very powerful.

4.1.2 Model

The model made in the MAST Toolkit (figure 4.2) is a direct translation of the proposed model of the controller computer in section 3.2. A single processing unit with a fixed priority preemptive scheduler was used to handle this model. Every task has been translated into an activity with their Worst-Case Execution Demands and priorities as properties of that activity. Every activity provides for a single output event, which can be used to activate other activities. Whenever an event needs to activate more than one other activity, like it is the case with the Core Controller, a multicast is used to provide for multiple events.

A rate divisor is used between the Drive Controller and the Core Controller to reduce the number of events sent to the Core Controller by half. A delay block is used to start the messaging part of the Industrial Automation Protocol, delays can be set on certain external events. In this case it was easier to model activation of the Industrial Automation Protocol to be $250\mu s$ after the arrival of the key Cycle Start Telegram, instead of trying to let it activate on another Cycle Start Telegram that follows exactly $250\mu s$ later. It makes no difference for the model.

The toolkit allowed to specify two shared resources, which could be locked for a critical instance by different tasks. As described the tasks Hardware Manager and the Self Manager can lock a shared resource they use. And the Core Controller and Drive Controller can lock another shared resource. For both these resources it can be specified if they use the priority ceiling or priority inheritance protocol to allow continuation of the high priority tasks. In case of the Robot Control System the Priority Inheritance Protocol was chosen. Although this might not be the exact case, we assume for now that the whole task is a critical section for a shared resource. In reality only a part of the task will be a critical section though.

4.1.3 Results

Although the toolkit accepts the use of multicast activities, these blocks allow for activation of multiple tasks by one event, the analysis algorithms are not yet implemented. Every analysis method available in the toolkit responses to the model with the statement: Feasible_Processing_Load not yet implemented for Multiple-Event systems.

We have tried to find workarounds, but it turned out that the controller can not be built as a single pipeline of activations. External events also do not provide for the necessary activation schemes to be able to model the controller in a different way in MAST. Hence, we were not able to analyze the robot control system with the current version of MAST. No further attempts were made to implement the bus and digital signal processors in MAST.

4.1.4 Discussion

The fact that the modeling language of MAST is powerful makes it a good candidate to be a meta language for every performance modeling problem that needs to be specified. However it becomes also more difficult. The user interface will alleviate this problem however the current version is far from stable. During the use it crashed often, resulting in data loss, close to a point that it was absolutely unusable to work with.

MAST is unable to analyze our model, since there is currently no algorithm in MAST that is able to work with multiple events. Multiple events are blocks that either distribute an event to more then one activity or concentrate the events of multiple activities to one activity.

The fact that MAST is unable to use multiple events does not make the method flawed or incomplete though. It still might be a good idea to construct an algorithm, which can be added to MAST also, to calculate the right values. Holistic analysis methods are still subject to a lot of research, finding the right algorithm requires a lot of knowledge and time. The risks that an algorithm becomes unusable after a small change makes the choice between implementing an algorithm and trying another method not hard.

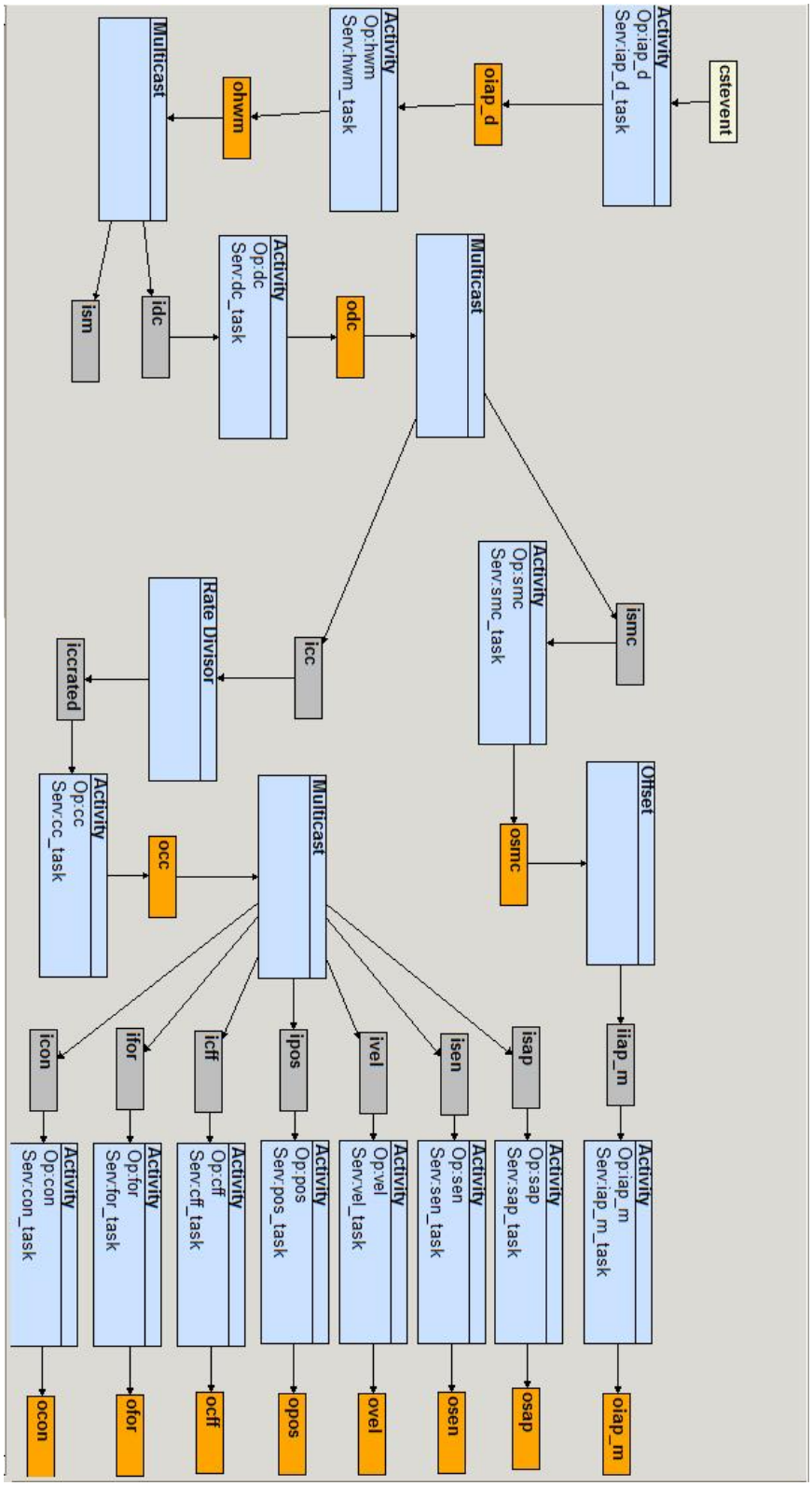


Figure 4.2: Robot Control System implemented in MAST

4.2 Modular Performance Analysis - Real-Time Calculus

Real-Time Calculus has its foundations in the network calculus, a theory of deterministic queueing systems. Because it is deterministic, the calculus provides guaranteed upper and lower bounds. These bounds can be overapproximating though.

Real-Time calculus is a modular performance analysis method. It performs analysis on small parts (modules) of the system. This method allows to calculate timings of a certain task or module within the system and let these timings propagate through the systems.

The model is based on a number of processing blocks that have incoming requests and execute those requests based on the available capacity (see figure 4.3).

The request function R(t) represents the total amount of requests that has been generated for this process up to time t. The amount of requests handled is output in R'(t). The Capacity function C(t) represents the total capacity that is available if the resource generating the capacity is under full load until time t. The capacity that is left for other processes to use is output in C'(t). The value K is used to determine how much capacity is needed to process one request, output can be calculated according to the functions:

$$R'(t) =_{u \leq t}^{min} (R(u) \times K + C(t) - C(u)) \div K$$
$$C'(t) =_{u \leq t}^{max} (C(u) - (R(u) \times K))$$

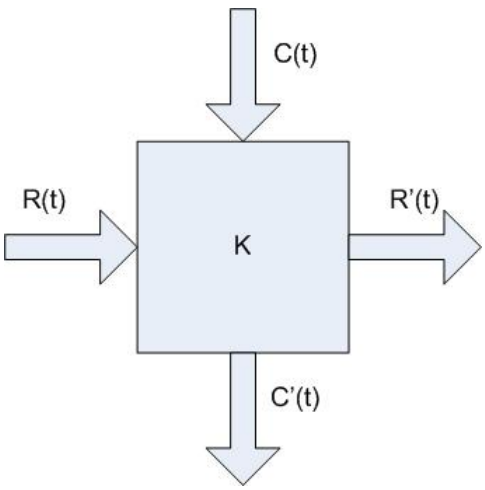


Figure 4.3: A model of a processing component in Real-time Calculus

While the request and capacity functions only describe one possible input model, we can also calculate upper and lower bounds for a system, the request

function will show as two arrival curves that determine the upper bound(α^u) and lower bound(α^l) of the request arrivals based on the type of event. These bounds are absolute so there is no request function for that system that is not contained within these bounds.

$$\alpha^l(t,s) \leq R[s,t) \leq \alpha^u(t,s), \forall s < t$$

The same holds for the Capacity function. An upper and lower bound can be found which we call service curves(β^u, β^l) representing the upper and lower bounds of the Capacity of the service available.

$$\beta^l(t,s) \leq C[s,t) \leq \beta^u(t,s), \forall s < t$$

The upper and lower arrival curves and service curves can then be used to calculate the output arrival and service curves. The most used component used in Realtime calculus is a greedy processing component which tries to process a request as soon as it arrives at the greedy processing component if there is capacity available. An incoming event stream modeled by the arrival curves will enter a First In, First Out buffer in front of the component. As soon as capacity is available represented by the service curves, an event will be processed. An arrival curve will be output which represents the upper and lower bound of the processed requests. The capacity that is still available after processing the events is output as an upper and lower service curve.

The resulting curves that leave the processing component can then be used to model the data flow in case of the request curves or the unused resources in case of the resource curves. These can be propagated to other processing components [TCN00] [WT06] [WTVL06].

4.2.1 Real-Time Calculus Toolbox

The Real-Time Calculus Toolbox is an easy to use framework in which large systems can be modeled by using small standardized building blocks, that implement the equations needed to describe the processing of the event and service streams. The user does not have to know these equations by heart.

The Real-Time Calculus toolbox currently supports Fixed Priority Preemptive scheduling, Rate Monotonic scheduling, General processor sharing, Time Division Multiple Access, Earliest Deadline First and First In First Out scheduling. Research is still going on in non-preemptive schedules.

There are also predefined functions in the toolbox to model arrival streams and service streams. An arrival stream models task activation by providing a worst case upper bound and a best case lower bound, based on a period, jitter and an inter-arrival distance. Resources can be modeled in different ways, but provide for an upper and lower bound based on an amount of bandwidth. This bandwidth can either be delayed or not. Some scheduling types can be modeled by modifying the resource stream, for example time

```
[oIAPD coIAPD delHWM bufHWM] = rtcgpc(keycst, crc1, wcedIAPD);

[oHWM coHWM delDC bufDC] = rtcgpc(oIAPD, coIAPD, wcedHWM);

[oDC coDC delSMC bufSMC] = rtcgpc(oHWM, coHWM, wcedDC);

[oSMC coSMC delIAPM bufIAPM] = rtcgpc(oDC, coDC, wcedSMC);
```

Figure 4.4: Modeling of 4 Greedy Processor Components from the robot control system

division algorithms or periodic servers, by splitting up a resource stream and divide the resources over two tasks.

In order to use the real-time calculus toolbox, the user has to call the appropriate commands within a MATLAB program. A graphical user interface is in development for the Real-Time Calculus toolbox to ease the modeling effort even further, but is not yet available [WT06].

4.2.2 Model

Since the controller considers a fixed priority preemptive schedule, it is perfect for modeling in real-time calculus. Every task will be modeled as a greedy processing component. The processor provides for a constant resource stream for the highest priority task. Any lower priority task will get whatever resources are left from the higher priority tasks. The control system has some tasks of same priority. To simplify the model these tasks have been given a modified priority based on their activation. If task 1 activates task 2 then task 2 will get a lower priority than task 1. Tasks that show no such order have been randomly been given a priority close to their real priority. Since the deadlines of all the tasks with the same priority either do not interfere or are exactly the same, this modification does not influence the system results (figure 4.5).

The communication device is not easy to model. Not one of the scheduling algorithms available in the toolkit provides a close enough algorithm. Based on the First In First Out module however, we were able to develop an algorithm that represents behavior of a Fixed Priority Non-preemptive Schedule [HT07] [CB08], which could be used to represent the communication bus. Based on the transfer mode of Firewire the bus resource stream is either a steady continuous stream or a time division stream that provides for 80% of the $125\mu s$ cycle to be allocated to the communication tasks. The model will however lose the knowledge that a Cycle Start Telegram will immediately follow after the cycle starts again, since there is no way to model correlation between activation patterns and resource patterns. In our analysis we have only considered asynchronous transfer mode.

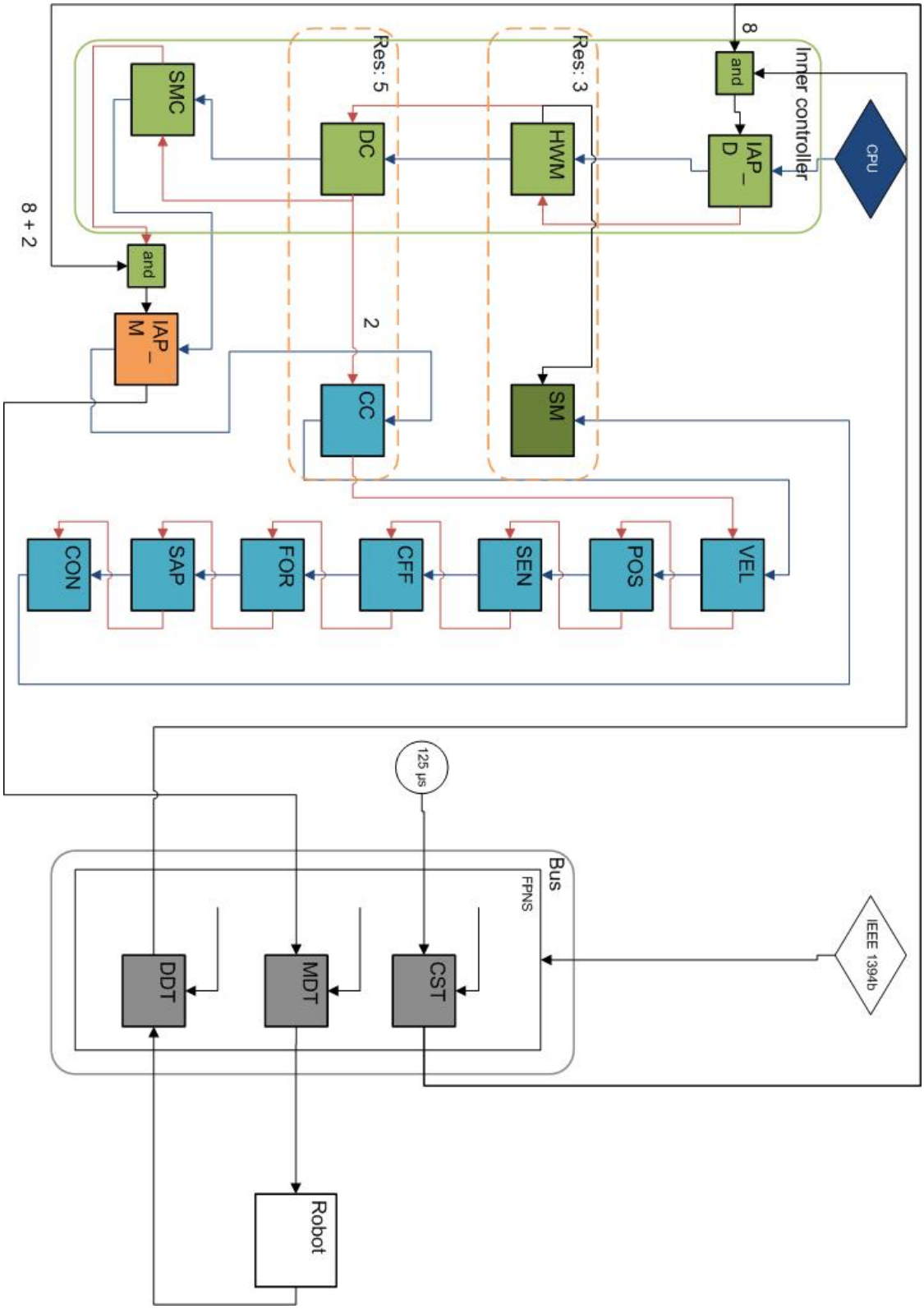


Figure 4.5: RealTime-calculus model of the robot control system

There is only one external activation stream in the model, which is the $125\mu s$ cycle that is generated by the bus. Every one of these events has also an inter-arrival distance of $125\mu s$. The jitter of this clock generated arrival event is less then $0.5\mu s$ allowing it to be negligible. The arrival stream is represented in figure 4.6. Note that due to the fact the inter-arrival time of the stream is as large as the period the upper and lower bound of the stream are exactly the same.

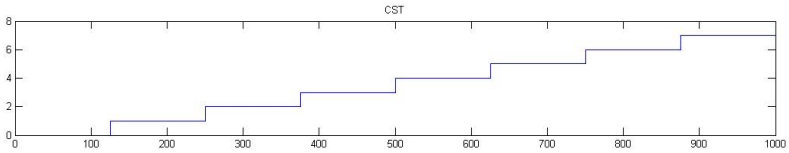


Figure 4.6: Request stream of the Cycle Start Telegram activation

4.2.3 Results

The Real-time Calculus toolbox has a couple of predefined variables it can return. Among these are maximum buffer lengths required for storage of activation requests, end-to-end delays and calculation time of the analysis. It is also dependent on the resources available if the end-to-end delays are bigger or smaller.

At first the outer control loop failed to meet its deadline. This was caused by the pessimistic blocking times we used in the inner controller. The preemption would cause the outer loop to surpass the $1500\mu s$ deadline. A slight relaxations of this deadline as was suggested in chapter 3.2 makes the model feasible again.

Worst Case Response Times calculated by the toolbox are 235 μs for the IAP_D - HWM - DC - SMC path. The IAP_M - MDT - DSP - DDT path resulted in 656,8 μs. The motion modules have a worst Case Response Time according to Real-Time Calculus of 1615 μs.

4.2.4 Discussion

Real-time calculus does not support shared resources and has no way to implement the priority inheritance protocol. It seems unlikely that shared resources will be supported in future releases since dynamically diverting a resource stream is still very hard in Real-Time calculus. Different scheduling algorithms that use dynamic assigned priorities, like Earliest Deadline First, or are non-preemptive lead to problems within the Real-Time Calculus toolbox. Considering blocking times leads to an over-approximating schedule, since the task will not be blocked every cycle of the controller or might never be blocked due to their activation patterns. Also the activation pattern of the Self Manager that can discard an activation when not enough processor time is available cannot be modeled. Since the execution time is already contained within the HardWare Monitor we can leave the Self Manager out of the Realtime Calculus model.

One of the major problems in this model is the cyclic dependency. A cyclic dependency happens when two components both generate input and output for each other. In this particular case the problem exist with the non-preemptive bus. This part can not be solved by greedy components, but all the tasks are added together in one non-preemptive module in which case the output of the Cycle Start Telegram is indirectly an input for the Master Data Telegram which is also an input for the Device Data Telegrams. This problem can be solved with fixpoint calculations at the expense of accuracy [JPTY08]. The fixpoint values are then used as an approximation of the input pattern for the activation stream of the Master Data Telegram. The same holds for the Device Data Telegram task whose activation is based on the output of the Master Data Telegram.

It is possible to let MATLAB automatically figure out the fixpoint values. However since we need at least two of those which influence each other, this might become a very heavy calculation will the activation pattern is quite straightforward. The Master Data Telegram will be sent by the Industrial Automation Protocol. This Task has the highest priority and can not be blocked by any other task. So the activation pattern of the Master Data Telegram will be the same as that of the Industrial Automation Protocol plus the Execution Time of that task.

The second fixpoint we can do on activation of the Digital Signal Processor or the Device Data Telegram. Either way will suffice, however the Master Data Telegram has less chance of being interrupted and has also a quite steady activation pattern. Since this task activates the Digital Signal Processors, we have chosen this task. Resulting in the fact that the Digital Signal Processors are activated every (approximate) 2 microsecond after the activation of the Master Data Telegram task.

Real time calculus uses the horizontal difference between the upper activation curve and the lower resource curve to determine the maximum delay

a task may have. If two tasks are activated at the same time but enough resources were available then both will have no delay which is not really the case. That makes analysis somewhat harder. We started therefore assumed that the control loops start immediately. That way the toolbox will at least take computational time into account and gives us an estimation of the Worst Case Response Times. For the IAP_D-HWM-DC-SMC path, the deadline was initially surpassed according to the toolbox, however the toolbox took blocking times for every task in account of its direct predecessor, this is however not the case since that predecessor has just been executed and will not execute a second time directly after that. It is a clear example that you easily lose activation patterns in real-time calculus due to local analysis and propagation. Something which can cause unreasonable faults in jitter free systems.

The over approximation of the IAP_D-HWM-DC-SMC path is caused by the blocking times that are added due to a missing mechanism to implement mutual exclusion and priority inheritance. This value propagates through the whole system and will cause over approximations in other tasks that are preempted by this path.

Although the timings are correct, the accuracy of this method can become bad very quickly. Cases can be generated that delays are double of what they should be. Also the two fixpoint calculation can make code generation or analysis very difficult. With an analysis time of about 4 seconds it is definitely one of the quickest. Since this method is more focussed on buffers and maximum delays, it does not look like the best method for our needs.

4.2.5 Model Manipulations

The modular performance method allows for different quick model manipulations that can be used to model changes in the robot control system architecture. A small difference could be changing the Parallel Kinematic Machine. Due to the modularity of the software, the only changes from a performance point of view would be different timings for the tasks in the inner control loop and a different number of Actuators and Sensors, which results in a different number of Digital Signal Processors. Since these are activated by the outgoing arrival stream of the Master Data Telegram task and activate the already available Device Data Telegram, the commands that have been used to model a certain Digital Signal Processor, can be copied and reused with different values to represent an additional processor.

The modularity of the software also allows for the addition of new motion modules. These can be represented by new greedy processing components that are added to the list of motion modules that are already in the model, it does not matter at which position they are added, since all motion modules have the same activation time and deadline they all have to finish at the same time. It does not matter for the schedule, if they are executed in a different order the result will stay the same. They will either fail or succeed at executing

within the deadline.

It is also possible to add computing nodes to the robot control system and let some second layer tasks be executed on them. This requires however that new tasks need to be added to the bus which indicate information communicated over the bus between the master node and the additional computing node. In order to do this, new fixpoint calculations have to be done to see if the new packets do not break the old ones. As a result, making this change is extra hard to be automated.

4.3 Timed Automata

Timed automata are an extension of finite state automata with a finite set of clocks. Those clocks can be tested for a certain value and can be reset. The progress of all clocks in the system are the same and the value of the clock indicates how much time has passed since it was reset. System properties can be checked using a logic language.

A timed Automaton consists of at least the following elements [BY04]:

- A finite set of locations or nodes.
- One of these locations is the start location.
- A set of edges between locations.
- Locations have invariants, the automaton can only stay in a location if the invariant holds. These invariants have to be clock related, they always exist as a clock compared to a natural number.
- The edges can also be conditional depending on a clock variable compared to a natural number. They are also allowed to reset clocks and have a label that can be used for synchronization with another automaton.

A number of timed automata can be used to model a system.

4.3.1 UPPAAL

UPPAAL has been used as a model checker for the timed automata. UPPAAL has been developed by UPPsala and AALborg Universities. In UPPAAL a single component can be modeled by a timed automaton and with the use of global variables and synchronization labels those automata can interact with each other. A system can be formed from these timed automata that use synchronization and global variables to interact with each other.

UPPAAL works with templates, every template represents a certain timed automaton and can be instantiated multiple times within the process declaration of UPPAAL with certain arguments the user can add. These arguments

can be used to fill in user defined parameters of the timed automata. Synchronization is normally done binary in UPPAAL. If an edge gets a synchronization label $a?$ it synchronizes with one other edge in an other automaton with synchronization label $a!$. There are also broadcast channels in UPPAAL in this case an emitting channel $a!$ synchronizes with a set of receiving channels $a?$ that is available at that moment. So if a automaton has an outgoing edge that also receives on the broadcast channel that is currently being emitted on and the edge is available (the guard evaluates to true) then the edge has to be taken. In case the edge is unavailable the emitter is still allowed to broadcast and synchronize with other automata.

UPPAAL also provides urgency in different settings. Urgent synchronization require a transition or edge to fire immediately if it becomes available. Note that a urgent synchronization cannot have a timing constraint. Urgent States do not allow time to pass in the state. This is the same as adding a clock to the state and an invariant that states that that clock can never become greater than zero. The system will have to perform one or more transitions before clocks can continue again. Committed states are like urgent states, however when the model is in a committed state the next transition should be a transition out of a committed state.

Different expressions can be used in UPPAAL to either define a *Guard*, which should evaluate to True before an edge is allowed to fire. *Synchronization*, which uses a label to synchronize the firing of two edges in two automata. *Assignments*, to change values of clocks and variables or *invariants*, based on clocks and a certain expression to limit the time that can be spent in a certain state.

UPPAAL provides a graphical user interface in which the user can model the automata, run simulations and verify properties that are coded in a language that shows resemblances with Computational Tree Logic (CTL), that allow to check states as well as paths in the model. The Computational Tree Logic possibilities that are available in UPPAAL are:

- $A\Box\psi$ which checks for a certain ψ in every state.
- $E\Box\psi$ which checks if a certain ψ holds in every state of one path that can be taken from the current state.
- $A\diamond\psi$ which checks if a certain ψ is always reachable for every path the system takes.
- $E\diamond\psi$ which checks if a certain ψ is reachable from our current state (it does not necessarily mean that that state is eventually chosen).
- $\psi \rightsquigarrow \varphi$ which indicates that once a certain ψ is satisfied every possible future action should return eventually in a state where φ is satisfied.

These can be used to check for liveness, safety and reachability properties.

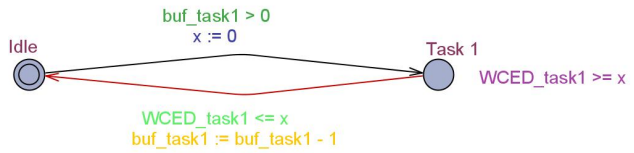
4.3.2 Model

There exist different approaches to model the robot control system. One way is to model the tasks and schedulers. The approach used is based on modeling the resource and environment as was introduced by Hendriks and Verhoef [HV06]. It states precisely in which state the resource currently resides. This approach makes modeling by hand harder and error prone, but can still be automated well due to the approach structure.

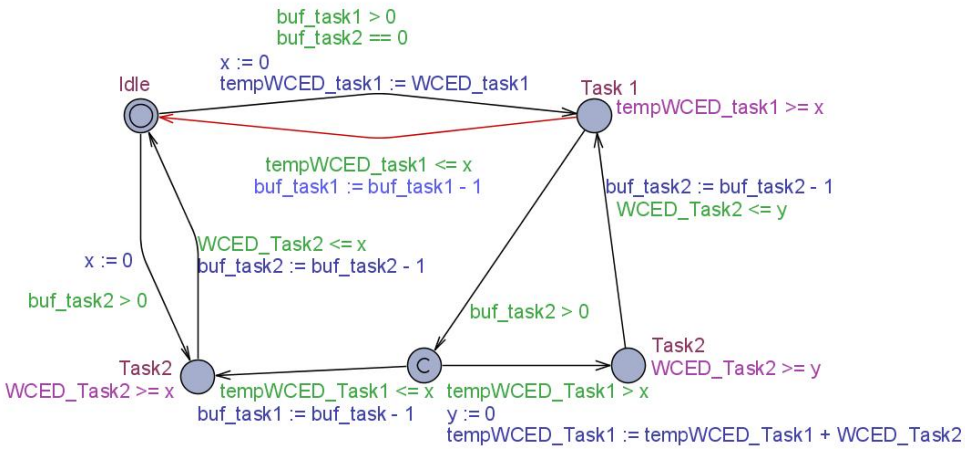
Due to the strict periodic nature of the CST activation, it is easy to model the environment, which is just a loop that activates the CST send task after every 125 μs. After 125 μs it can not wait in its current state anylonger and needs to fire an edge, activating the CST task and resetting the clock for the next loop.

Assuming we have only one computational node, all tasks will be executed on this node. A resource can either be idle or working on a task. An algorithm to build a resource timed automata can be to start with an idle state (4.7(a)) and add tasks from the lowest priority to the highest priority.

(a) Idle state



(b) Added a Task



(c) Added a second task with a higher priority

Figure 4.7: UPPAAL: building a Fixed Priority Preemptive Schedule

In figure 4.7(b) a first low priority task is added. As soon as the buffer for this task contains a value, the task should start. Note that the transition from idle to task 1 is an urgent transmission, so it is obliged to fire as soon as the guards are met. A clock will be set to zero and starts counting. As soon as the clock reaches the Worst Case Execution Demand of the task, it will be forced to leave its state and jump back to idle, thereby removing one value from the buffer.

When a second, higher priority task is added, see figure 4.7(c), preemption can occur. First a second state is added the same way as was done with task 1. Some additional states need to be added to model preemption. The transition from the idle state to the task 1 state is changed so it is only allowed to trigger if the buffer for task 2 is empty. Also a temporal variable for the Worst Case Execution Demand is created and is set to the Worst Case Execution Demand of task 1. Once Task 2 becomes active while Task 1 is executing it will be preempted, hence another urgent transition is used. This preemption can happen at two occasions, hence the committed state at preemption, either the task reached his Worst Case Execution Demand, but did not yet leave the current state. In this case the buffer of task 1 can be reduced and a transition can be taken to the normal execution of Task 2. The other case is that task 1 is in the middle of his job and gets preempted, in that case it should go to a state where task 2 executes while task 1 is on hold. A new clock is needed to monitor progress of the second task, while the temporal variable for the Worst Case Execution Demand of task 1 is increased with the Worst Case Execution Demand of task 2. UPPAAL clocks can not be stopped and can therefore not properly be used as a progress variable for a preemptive schedule. By adding the Execution Demand of the preempting task you take the preemption into account for the task so that it can continue after the preempting task has finished. Every other higher priority task is added in the same way.

The controller software consists of a number of tasks that are of same priority. These can be added as were they non-preemptive, they can not preempt each other. If two or more of these tasks are active then UPPAAL can choose non-deterministically which task will execute first. Hereby we reduce the depth of preemption for the robot control system. Since all tasks of same priority also have the same deadline it is not necessary to implement more difficult schedules.

The communication bus is non-preemptive allowing for a much simpler model. Every task is directly connected to the idle state. A fixed priority decides which task should be executed if more than one task is activated. Once a task starts executing, it cannot be stopped and continues until it completes the task.

A shared resource can be modeled by two state machines synchronized with the activation of the tasks that can potentially lock it. If a task wants to lock the already locked resource the synchronization label is unavailable and the higher priority task is unable to start. The model will also not be able to

use the transition that activates the task.

The robot control system however uses priority inheritance to prevent long blocking times of the higher priority tasks. In case a higher priority task gets blocked on a locked resource, the lower priority task that locked the resource should become active. If the higher priority task arrives when the lower priority task is preempted or executing, there is a transition to a new state that will continue the execution of the lower priority task, but at the same priority as the higher priority task.

4.3.3 Results

In order to verify if the system holds to the said deadlines, we have added a number of additional clocks. These clocks are reset synchronized with the activation event of the first tasks within the model. For our $250\mu s$ this would be the IAP_D task. For the outer control loop deadline, we used every activation of the Core Controller. And for the $750\mu s$ the IAP_M task was used. There are two ways to check if these deadlines are met by the system using a modelchecker. Either we check for failure or we check for success. So we search if a task is not finished within its deadline or we try to determine that the task always finishes within the deadline. The first one is much easier to implement though. Since our buffers are emptied at the moment a task finishes and not earlier, we can check if the buffers are not empty when we reach our deadline, by verifying the CTL formulas:

$E\diamond$ controller.deadlineclock250 $>$ 250 && buf_IAP_D + buf_HWM + buf_DC + buf_SMC $>$ 0
$E\diamond$ controller.deadlineclock750 $>$ 750 && buf_MDT + buf_DSP + buf_DDT + buf_IAP_M $>$ 0
$E\diamond$ controller.deadlineclockoc $>$ 1900 && buf_FOR + buf_VEL + buf_SAP + buf_SEN + buf_CON + buf_CFF + buf_POS + buf_CC$>$ 0

All these formulas evaluated to unsatisfiable indicating that the deadlines are met. The analysis time is less then a minute on a 2 Ghz dual core machine and does not use more then 100 Mb of memory.

We tried to shutdown different motion modules by setting their Worst Case Execution Time to 0. It did not affect any of the above formulas. Only when we increased the time of certain modules like changing the HardWare Monitor to 105 μs, the first formula became satisfiable indicating that there is a way to surpass that deadline. The same holds when we increase one of the motion modules with at least 500 μs.

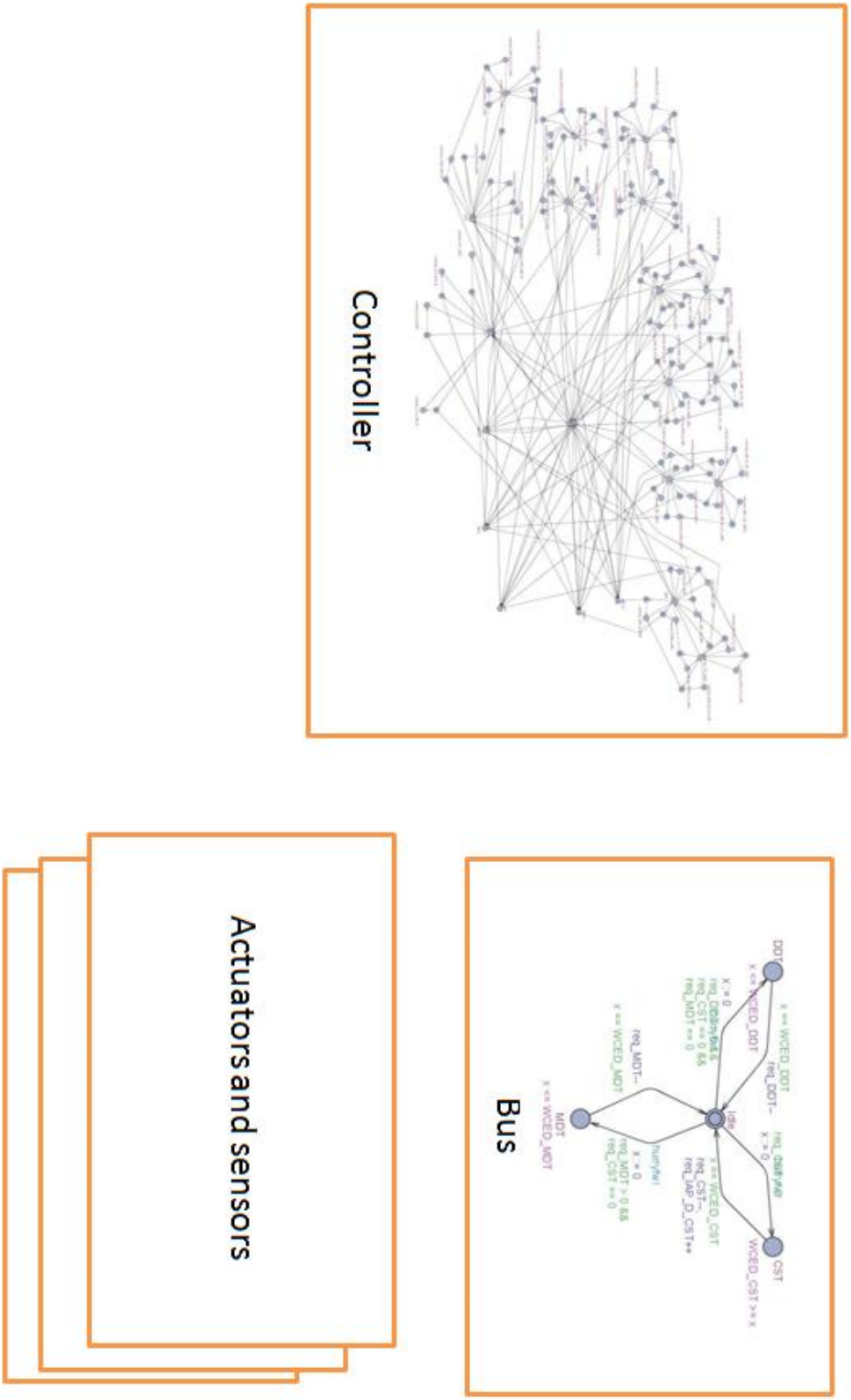


Figure 4.8: UPPAAL model of the robot control system

4.3.4 Discussion

The environment and resource modeling method can ideally be used in non-preemptive schedules like First In, First Out. Preemptive schedules expand exponentially and are hard to make by hand. The main cause for this is the fact that clocks cannot be paused and are therefore not suitable as progress variables in preemptive systems. A system with four or more preemption possibilities gets unreadable. A tool however that generates this model automatically should not have a problem with this. This approach does not use templates to its advantage either, since resources are not likely to be comparable at all, which increases the modeling effort significantly.

Although the answers in 4.3.3 is satisfying, as in we know that the system will hold to its realtime requirements, we might need a bit more information or values that are important for measuring the performance. We can do this by using a different branch of UPPAAL named UPPAAL CORA where it is possible to specify cost functions on actions and delays. However during the project, unaware of UPPAAL CORA, we used CTL formulas and common sense to determine the worst case response times. When activation patterns are straight forward, worst vase response times can be calculated manually and confirmed by using the model checker. When a system contains more difficult activation patterns due to jitter for example CORA might be a good idea. Also in case of our project the cost function of CORA might give the user more insight in what the bottlenecks are of the system.

In order to come up with a good CTL formula we can take the 250 μs tasks for example. Then 250 μs tasks are of the highest priority, so they will probably be executed right after each other which makes it plausible that the tasks will have a Response time of at best 170 μs. Only priority inversion with the Self Manager or the Core Controller can result in a Worst Case Response Time of 235 μs.

In order to determine the Worst Case Response Time we can define two CTL formulas that check for clock values. Of these one should be satisfied while the other is not. If we can find the values that differ precisely 1 μs, we have found the worst case response time. So we defined the following CTL formulas for the 250 μs deadline to determine the Worst Case Response Times.

$E\diamond$ controller.deadlineclock250 > 169 && buf_IAP_D + buf_HWM + buf_DC + buf_SMC > 0
$E\diamond$ controller.deadlineclock250 > 170 && buf_IAP_D + buf_HWM + buf_DC + buf_SMC > 0
$E\diamond$ controller.deadlineclock250 > 190 && buf_IAP_D + buf_HWM + buf_DC + buf_SMC > 0

When we use the values indicated in figure 3.4, the results are that the first one is satisfiable and the second and third are not, leading to a Worst

Case Response Time of 170 μs. Interesting to see is that if we should turn of a number of the motion modules, that the Worst Case Response Time stays 170 μs. The current values for the modules can not cause mutual exclusion. When we change the value of the Singularity Avoidance Protocol from 800 μs to 385 μs, we get that the first two CTL formulas are satisfied while the third is not. In this case priority inheritance will allow the Self Manager, that has started just before the inner control loop of 250 μs starts again, to finish its task before the HardWare Monitor may start its execution. This leads to a higher Worst Case Response Time that can become at most 190 μs.

For the second inner control deadline of 750 μs, it is also relatively easy to find the Worst Case Response Time. The digital Signal Processors have dedicated resource so can never be blocked. We assumed that a Digital Signal Processor will finish within 500 μs. The Industrial Automation Protocol that sends the messages on the bus is of the highest priority so will also not be blocked. The bus is of very high speed which can send packets of around 100 bytes every microsecond. The packets we send are very small, the master data telegram is largest with 128 bytes of information and the Data Device Telegram sends approximately 28 bytes of data. The cycle start Telegram sends approximately 4 bytes of data. In order to keep the values readable, we decided to round the time it would take to send a packet to the next whole microsecond. This suggests that the Worst Case Response Time is somewhere in between 540 and 544 μs if only one Digital Signal Processor is used. More Digital Signal Processors can result in a slightly higher Worst Case Response Time, although this would require the Digital Signal Processors to finish at the same time, which seems unlikely. For one Digital Signal Processor the model confirmed that the Worst Case Response Time is 543 μs.

The outer control loop however becomes preempted a couple of times. At least ones by the inner control loop and a second time by a couple of tasks that activate after the Core Controller becomes active. This leads to a Worst Case Response Time of $1200 + 170 + 80 + 55 = 1505\mu s$. This first guess is confirmed by UPPAAL where we looked for a path where one of the motion modules was still busy while the deadline clock surpassed 1505 μs. This returned unsatisfiable while the CTL formula that asked if a motion module was still active for 1504 μs is satisfiable. The Worst Case Response Time is 1505 μs.

4.3.5 Model Manipulations

The model allows for changes in the architecture. One of the simplest changes is adding or removing actuators and sensors to or from the robot. These are represented by the Digital Signal Processor model template, which is a one task resource automaton, which can be instantiated for every actuator and sensor. Again in case of automatically generating a model this is not a problem. In order to reduce the number of preemptions, tasks with the same priority can be executed in a non-deterministic order. Since tasks of the same

priority have the same deadline or do not overlap this method will not violate the validity of the model. A new global variable should be updated by the Master Data Telegram though to add a request for this new Digital Signal Processor

Adding more computing nodes would simplify the current model, since tasks will be assigned to different computing nodes. Adding a computing node would mean that a new resource is added, so we have to add a new template that represents the node. Tasks have to be assigned to nodes beforehand, so the total model can be based upon this distribution. Additional computing nodes can either be an additional CPU in our master control machine or an additional machine connected to the master machine by the firewire network. Depending on the nature of the additional computing node new communication tasks need to be added for the communication bus. Since the master machine will initiate the contact, these communication task will have the same priority as the Master Data Telegram in our model.

Adding or removing motion modules is not that hard. Since motion modules are all of the same priority they do not preempt each other and only preempt the Self Manager. Adding an extra motion module to the current model would only need to build an additional tree for this module connected to the idle state and a preemption tree in the Self Manager state. There is no need to rebuild the whole model. A tree in this contact is a part of the model that describes the task and all his preemptions below that.

4.4 Hybrid Automata

The timed automata inspired a different approach. The problem that you can not stop a clock in UPPAAL in case of preemption, might very well be solved by using a hybrid automaton instead of a timed automata, since we can model clocks that can be paused (act like a stopwatch) in certain states. In this case we model our system not from a resource perspective, but from a task perspective.

Hybrid models allow for including environment processes in your model. In general systems that show both discrete as well as continuous behavior can be modeled. Think of a system that controls the temperature in a room for example, where the control is discrete in if the heater is on or off and the temperature is a continuous function depending on the state of the heater.

In hybrid automata there are either continuous changes or discrete changes. Discrete changes are modeled by transitions, continuous changes happen in states. In case of multiple hybrid automata, time for continuous changes is equal over every automaton. So if automaton 1 has been in a state for a number of cycles after a synchronized transition with an automaton 2, we know for certain that the same amount of time has passed for automaton 2, allowing for continuous variables to change for the same amount of cycles.

A Hybrid automaton consists of at least the following elements [Hen96]:

- A number of variables that are allowed to take any real number.
- A number of locations that represent a certain state of the hybrid automaton.
- To every location a flow condition should be added. The flow condition over a variable x, indicated by $\dot{x}$ represents the change to the variable over time, while the automaton is in that specific location. $\dot{x} = 1$ indicates for example that variable x is increased by one every cycle in that location.
- Invariants on locations that show in which cases the automaton is allowed to be in a certain location
- A number of initial conditions, comprised of the initial value for variables and the initial location of the automaton.
- A number of transitions between locations, these are directed arcs between two locations that allow for changing the automaton state from one location to another, these transition can be guarded.
- Jump conditions can be specified for these transitions, setting the value of a variable to a certain constant. The primed version of the variable x' indicates the value after the transition and the unprimed variable x indicates the value before the transition.
- Actions that are assigned to the transitions. These actions can be used to synchronize transitions over multiple hybrid automata.

To illustrate a hybrid model, a room that is being heated automatically when the temperature reaches a certain lower bound and stops heating when an upper bound is reached is modeled as shown in figure 4.9

4.4.1 PHAVer

In order to use compositional modeling, we used the Polyhedral Hybrid Automaton Verifier (PHAVer) model checker. PHAVer makes use of hybrid Input/Output automata, which is a type of hybrid automata.

Definition 4 *[Fre05] A hybrid Input/Output-automaton (HIOA)* $H = (Loc, Var_S, Var_I, Var_O, Lab, \rightarrow, Act, Inv, Init)$ *consists of the following:*

- *A finite set Loc of locations.*
- *Finite and disjoint sets of state and input variables,* Var_S *and* Var_I *, and of output variables* $Var_O \subseteq Var_S$*. Let* $Var = Var_S \bigcup Var_I$*.*

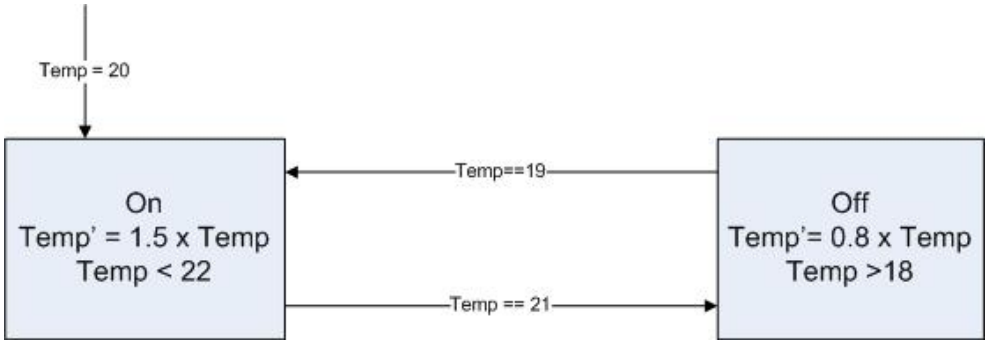


Figure 4.9: A hybrid model of a the temperature in a room with an automatic heater

- *A finite set Lab of synchronization labels.*
- *A finite set of discrete transitions* $\rightarrow \subseteq Loc \times Lab \times 2^{V(Var) \times V(Var)} \times Loc$. *A transition* $(l, a, \mu, l') \in \rightarrow$ *is also written as* $l \overrightarrow{a, \mu}_H l'$.
- *A mapping* $Act : Loc \rightarrow 2^{act(Var)}$ *to time-invariant sets of activities.*
- *A mapping* $Inv : Loc \rightarrow 2^{V(Var)}$ *from locations to sets of valuations.*
- *Initial states* $Init \subseteq Loc \times V(Var)$ *s.t.* $(l, v) \in Init \Rightarrow v \in Inv(l)$.

In order to build an automaton in PHAVer one starts with the name of the automaton, followed by the declaration of state, input and parameter variables. Next, the synchronization labels used in the automaton are declared, after which a number of locations of the automaton will be described. Each location has an identifier and an invariant which must hold for the automata to be in that location. If the invariant does not hold the system locks. If a flow condition in a location is about to break the invariant for that location, the system is only allowed to take a transition until a state is reached in which flow variables are allowed to change again. A derivative that either describes linear dynamics, in which case the variable change with a certain value, or affine dynamics,in which case variables change according to a formula over the variables, is declared. The location also has a number of guarded outgoing edges specified by which the guard must evaluate to True in order for the edge to fire, second a synchronization label is put at the edge, it does not necessarily have a corresponding label in any other automaton though. Also the edge can have assignments to variables and the edge states to which new location it should point. At the end of specifying all locations the automaton requires a initial state.

When a synchronization label in case of a transition is encountered in PHAVer, it is required for every automaton in the composition that has that synchronization label declared to fire at the same time. If one automaton does

not have an edge with that synchronization label available, the transition is not allowed.

Clocks or in this case progress variables can be made by using a flow variable that equals 1 in case the task is running and that is set 0 when a task is blocked, preempted, idle or waiting. These variables are also called stopwatch variables, since clocks always run while stopwatches can be paused.

One of the strong points of PHAVer for this particular area is the support for compositional models [Fre05].

The model checker PHAVer generates the whole state space in order to detect if forbidden states indicated by the user can be reached. This state space is generated by a search for symbolic states that can be reached from the initial state until a fixed point is reached. A symbolic state is a (number of) location(s) with their corresponding values of the control variables. Note that the range of these variables can be infinite. It is still possible for hybrid automata to never reach a fixed point, so the reachability problem here is undecidable. In case of performance modeling, a model that is undecidable is either flawed or not schedulable. Setting buffer sizes and let the model enter a deadlock state when a request causes an out-of-buffer error will provide additional information about the cause of the problem, since a trace can be generated for which the model is not schedulable. This should however not be used lightly, there is no guarantee that a model would never reach a fix point eventually.

4.4.2 Model

This model approaches the problem from a task perspective. UPPAAL used a template per resource, the new method will have a template per task. The PHAVer model consists of three different elements: a hybrid model for every different task, a timer section that generates starting events for the system every $125\mu s$ and a file that provides commands to build the whole state space and checks for forbidden states.

A fully preemptive task can exist in four different states. These states can be formed by two variables. The first variable describes if the task has been activated(represented vertically in figure 4.10). An activated task is queued to run at the earliest possible instance, it is not necessarily running when activated. The second one describes if the task should be waiting for the resource to become available due to activation of an higher priority task (represented horizontally in figure 4.10). This leads to the 4 states, activated and executing, activated and waiting or preempted, idle but a higher priority task is currently executing, idle and no higher priority task is busy, in this last state it is possible however that a lower priority task is being executed. It is still possible for the task to directly start executing and therefor preempt this task.

The transitions that leads to the waiting or preempted state as well as

the ones leading to the idle but a higher priority task is executing state are synchronized with the block transitions of their direct predecessor. In the case of the highest priority task these states are obsolete, since that one can not be preempted or waiting, nor can a higher priority task be active. The edges that are outgoing from these states are synchronized with the idle transition of the direct predecessor of the task. Allowing only to fire when the direct predecessor is in its idle location therefore assuring that no higher priority task is running.

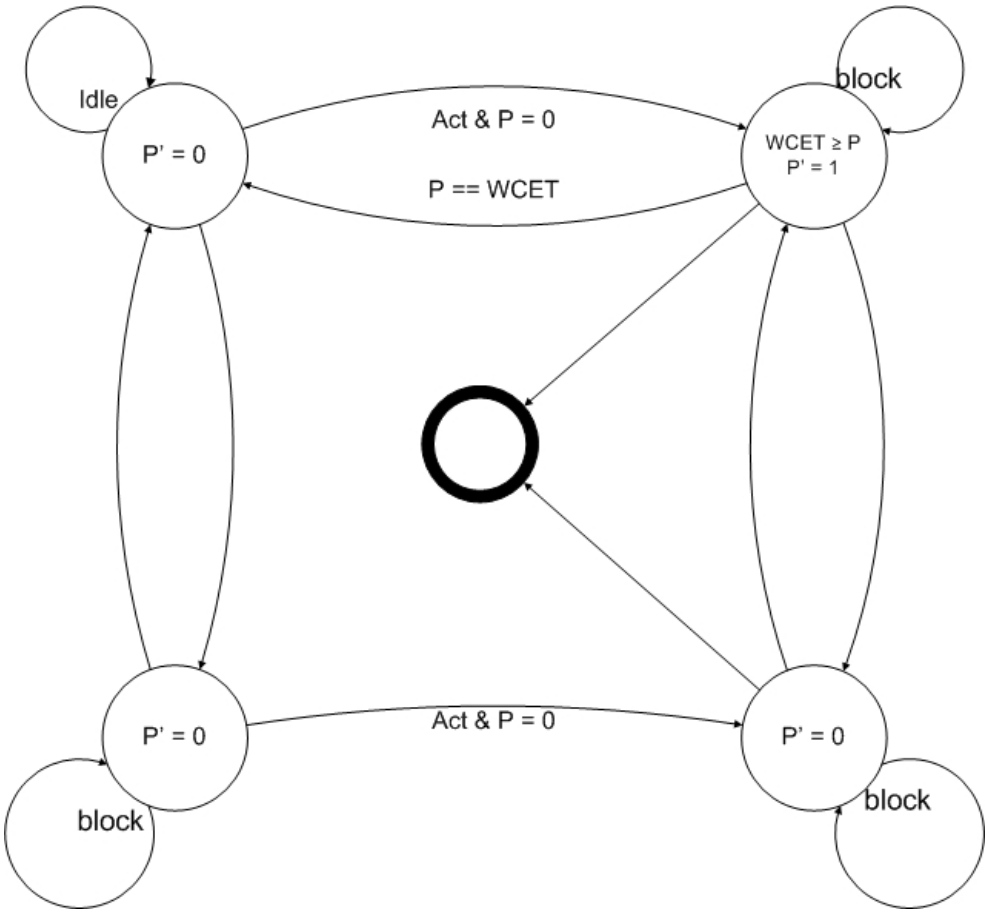


Figure 4.10: A hybrid model of a fully preemptive task

The fifth state that is present is not a required state, but can be used as a deadlock state when the task gets activated too many times, which would indicate a buffer overflow. In case of the robot control system this buffer is set to one place buffers. In the robot control system this state would be reached if a certain task gets activated at least two times without finishing execution in between these activations.

A non-preemptive task can also be modeled as a four states hybrid model.

The model will be similar to the one that is used for preemptive tasks. The main difference is that tasks block each other according to a round robin system. So tasks can start executing based on their priority and once execution starts all other tasks become blocked till the current task is finished. So the lowest priority task is the predecessor of the highest priority task.

To handle shared resources among two tasks, we can use the pessimistic method of just adding the worst-case execution demand of the lowest priority task to the highest priority task. A better approach would be to add this in the model. The priority inheritance protocol can be implemented with extending the four state model with two additional states. These states will be for the lower priority task, one to indicate the executing of the task at a higher priority and one for preemption by another task of higher priority (figure 4.11).

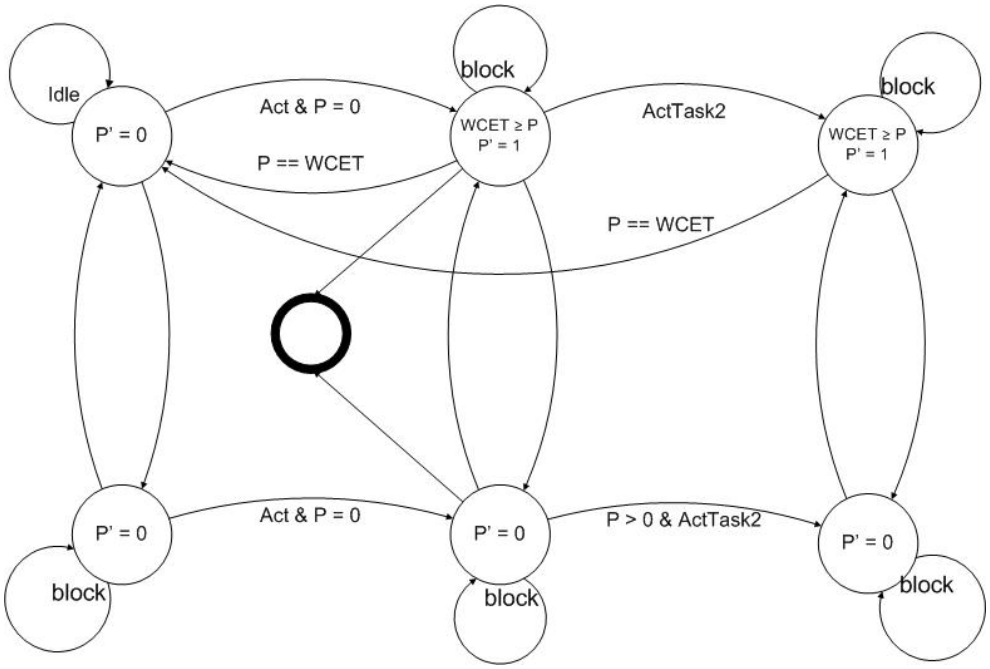


Figure 4.11: A hybrid model of a preemptive task with a shared resource under priority inheritance

The highest priority task in the shared resource model gets a transition to a new blocked state, where it waits till his new direct predecessor finishes. The new direct predecessor will be the task that has locked the shared resource. Instead of calling the idle loop transition for activation, synchronization can happen on the finishing transition of the task or a new loop transition that allows the higher priority task to start again needs to be added.

After the tasks have been modeled in the proper way they can be used to build the composition state space. In order to check if the deadlines are

not breached, a clock can be built in hybrid automata. A clock is a one state automaton with a reset transition to itself that sets the variable to zero. Inside the state the variable is increased by one for every cycle. The reset transition must be synchronized with the event that starts the first task of a deadline group, this is exactly the same as with timed automata. In order to check if deadlines are not met, we can specify forbidden states in PHAVer. A forbidden state is a collection of locations in which the automata are together with values or value ranges for variables. An example is a state in which the clock has surpassed the deadline value and one of the states that should have been idle before this is still in his running or preempted/waiting state. A reachability log can also be printed to see if there are states where a deadline was passed while one of the tasks was not yet finished.

An additional check is to see if any of the deadlock states are ever reached, which indicates that a task is activated two times while not finishing any of these two. The deadlines of the robot control system do not allow this behavior.

4.4.3 Results

Implementation of above mentioned compositional way in PHAVer uncovered one missing fact. There is no properly working urgent transition system in PHAVer, which means that although certain transitions are becoming active, the system might choose not to take that action, while the intention of the model requires a change. To overcome this problem some greedy behavior should be encouraged in the preempted/waiting states and the idle but a higher priority task is running state by summing up all progress variables from higher priority tasks and see if there is still progress being made (there is still a task executing) and by adding different synchronized transitions. This sum should be evaluated only in the preempted/waiting state or the idle but a higher priority task is running state. If the sum is equal to the progress variable C then progress is still being made and the model continues in this state. In case progress in the current state is not possible, either a higher priority task will start executing or the direct predecessor of task A turns to the idle state, which enables the loop transition which is synchronized with the edge that fires to the running resp. idle state of task A (see figure 4.12).

This approach however negates the modularity of the system since every task will need to have knowledge of *all* the higher priority tasks instead of his direct predecessor. Besides that, all lower priority tasks will have an edge from their idle or running state to the idle but a higher priority task is running or preempted/waiting state, for every higher priority task, leaving the user with an additional modeling effort.The edges are synchronized with every transition to the running location of a higher priority task.

The task perspective approach causes a gigantic state space to be built. In order to calculate the full state space of the model, it was necessary to

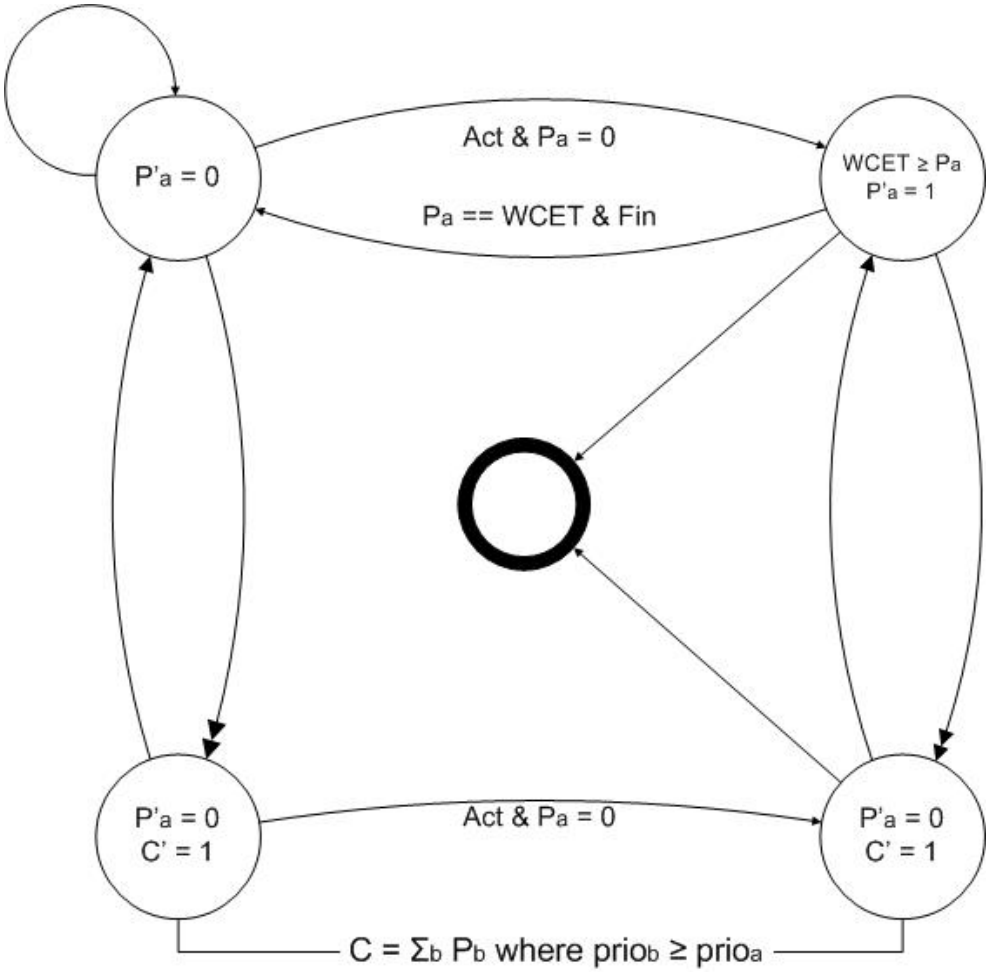


Figure 4.12: A hybrid model of a fully preemptive task without urgency

develop a 64 bit version of PHAVer. We succeeded in adapting the code to be able to run on a 64bit linux system. In order to be able to execute this model a powerful computer and the modified 64 bit version of PHAVer are needed. Together with the time it takes to build the model and the knowledge needed to analyze the state space for worst case timings do not make this method ideal for quick testing of different designs. However tool support for this method can reduce these needs. Automatical generation of these models is still possible.

The model takes around 4 hours to generate on a 2Ghz machine and requires at least 50 Gb of memory to store the whole state space. Note that we did not actively monitor the behavior of the machine so the actual times and memory usage might differ slightly. Also, we defined 2 independent reachability calculation, since we can independently calculate the state space for

```
bad=sys.{
    $~HWMdl~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$ & True,
    IAP_Ddl~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$ & True
    }
```

Figure 4.13: Example forbidden list, states that should not be reachable

```
bad=sys.{
    $~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$ & deadline250 > 250,
    $~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$~$ & deadline750 > 750
    }
```

Figure 4.14: Example forbidden list 2, variable values that should not be reachable

the controller and for the communication part. The Motion modules (the loop CC-SAP-SEN-FOR-CFF-CON-VEL-POS) does not interfere with the communication (the loop IAP_M-MDT-DSP-DDT) and can therefor be calculated independently. Note that if multiple computing nodes are used this statement does not hold, since additional communication will be required, for the modules that are run on the additional computing node.

In order to analyze the system forbidden states can be defined. In any case reaching a deadlock state is undesirable, so for every deadlock state a rule can be added to the forbidden states list. Generally a rule of this kind looks like figure 4.13. Every $ indicates a wildcard, meaning that we do not care in which state that automaton is, every automaton is connected to another via the $\sim$ character. After the & limitations for the variables can be given. In our case there are no limitations so we state True.

Besides the check for deadlocks we can also build three clocks, one for every deadline in the system. These clocks consist of flow variable that are either on or off. A clock starts at the same moment as the first task within a deadlock cycle, by using synchronization labels and stops once the last task within a deadlock cycle finishes. After that those clocks can be checked if they ever reach a value higher the the deadline. If such a state can not be found the system is safe. (see figure 4.14).

The model did not reach any of the forbidden states we defined nor did it enter a deadlock state, which indicates that the tasks that need to end before a certain deadline did so. A second step can be to extract worst case response times. In this case we need to produce a reachability log where we have to find the highest value for a timer. In our case we found the following timings.

After finishing the IAP_D - HWM - DC - SMC tasks at most $170\mu s$ has passed. Note that this is only the case when all tasks execute at worst case

time and all motion controllers are active. When other timings are taken into account or when different motion modules become inactive it is very well possible that priority inversion is encountered at the activation of the HardWare Monitor. It is very unlikely that the Core Controller will ever block the Drive Controller however, due to the activation pattern.

The outer control loop has a Worst Case Response Time of 1505μs. The IAP_M - MDT - DSP - DDT tasks have a Worst Case Response Time of 541,6.

In order to test the forbidden states, we tried the same model and changed the Worst Case Execution Time of the HardWare Manager to 130 μs. PHAVer responds with the message that the intersection of the forbidden list and the state space is not empty. This indicates that a state that is in our forbidden list is reachable. This state can be found by printing this state space in which we see that the deadline timer for the 250 μs deadline surpasses the deadline.

Finally to test priority inheritance we tried to change the Worst Case Execution Time of the Singularity Avoidance Protocol to 385 μs, the same value we used in UPPAAL. It resulted indeed in a slightly higher response time for the inner control loop. This resulted in 185 μs Worst Case Response time. The outer control loop worst case response time is 880 μs

4.4.4 Discussion

Although it is the most versatile method, hybrid automata have the disadvantage of a large modeling and analysis effort. The reachability log that can be generated is useful to gain information about buffers and response times, however it will take the user considerable time to extract this information depending on how large the state space will get. Besides that, there is no proper simulation method in PHAVer, which makes it hard to see if the model is correct and the model will never reach a deadlock. Deadlocks easily happen when a lot of automata make use of the same synchronization variable for example. Especially large systems with lots of different automata, which is the case in the robot control system due to urgency, are susceptible for small faults which are not recognized by the model checker and are hard to find for the user. Urgency would make a parameterized version of the model possible which would greatly improve the readability of the model, although the output will still be hard to interpret.

Also the model checker is still at its early stages. Errors like jumping to a non existent location are possible and might be overseen by the user, when analyzing the reachability file. Also discovery of syntax errors at runtime can be frustrating, when your model building time takes a few hours to complete.

The hybrid model returns as expected the same results as the timed model. Due to the large analysis effort we did not try many different parameters. However the system responds correct in any of them as are shown in chapter **??**.

As is shown in chapter **??** the values produced are precise. So the accuracy of this method is very good. However people should be careful with best and

worst case execution times for different tasks. Different values can result in different worst case response times. A better timing for one task might result in worse timings for other tasks.

4.4.5 Model Manipulations

The compositional nature of the hybrid model allows for some easy changes to it. Digital Signal Processors can simply be instantiated as can be done by the timed automata. If urgency was available in PHAVer, then adding or removing motion modules from the controller would also mean instantiating a new preemptive module into the whole model. The absence of this feature means that adding a new module or removing one propagates through a large part of the model, since progress variables are used in every lower priority task to check if there is still progress to be made. Adding an additional computing node also means that a large part of the modules need to be changed, since modules assigned to the new computing node would not be taken into account for the progress of the first computing node.

Chapter 5

Conclusion

The methods described are ideal for beforehand analysis of a design, without knowledge about what kind of system we are going to use or the implementation method. For mono-processing systems a lot of analysis methods have been developed, but support for distributed systems is still at an early phase. Available methods we used showed that the tools still need much work in order to be usable in design environments. Every method performs very well under certain circumstances while in other cases it falters. To summarize, MAST is unable to analyze systems that use multiple events, while this can be common behavior among many systems. Real-time calculus performs very well for Fixed Priority Preemptive Scheduling problems, which is used in most systems nowadays. The absence of support for mutual exclusion algorithms like Priority Inheritance or Priority Ceiling can be overcome by taking blocking times into account, this is a very pessimistic approach though. However when Non-preemptive schedules are considered or schedules that use dynamic priorities, modeling the system is Real-Time calculus becomes very hard, and in case of cyclic dependencies the user has to do simulations in order to provide good starting points for fixpoint calculations necessary to overcome the cyclic dependencies. The activation patterns for the current tasks are easy to simulate, but can become much harder in case of different architectures. Timed Automata are, contrary to Real-Time calculus, quite able to model non-preemptive systems, while preemptive systems take a considerable modeling effort, due to the fact that clocks can not be used properly as progress variables in this case due to the inability to pause them. Hybrid automata seem to be able to counteract this problem by introducing a stopwatch variable that can be paused when preemption is in effect. However the absence of a proper urgency principle in PHAVer increases modeling effort. Also analysis effort for the task oriented modeling should for larger systems not be underestimated.

5.1 RQ1: What is the best method to analyze real-time requirements of the Robot Control System

In order to judge the different methods we have named a few characteristics that we will look for. Although other characteristics might also be interesting for performance modeling, our focus is mainly on the following characteristics.

Correctness, all discussed methods except for the MAST method showed correct results, since we used only formal method this is no surprise.

Accurate results, the methods that used model checkers have shown the most accurate results. Since they check the state space of the model the results are precise. The price to pay is modeling effort and analysis effort. Since we discussed however that the model would need to be automatically generated, the modeling effort is not a problem that weighs heavy in our conclusion. The analysis effort is not a first priority for this system and may take as long as necessary. Real-Time Calculus is less accurate due to the fact that some input patterns are not entirely recognized, cyclic dependencies and the absence of support for mutual exclusion.

When we look at automatic model generation, again the model checkers win. The fixpoint calculation that is needed to solve the real-time calculus model is hard to come by and can become harder by adding or removing a number of digital signal processors. Also in case of multiple computing nodes over different machines, problems arise when new data needs to be sent over the communication bus, that must be taken into account into these fixpoint calculations. Both modeling approaches show enough structure to be able to automatically build a model out of a system description. Also the two different ways of modeling, task oriented versus resource oriented, that were used could easily be implemented in either hybrid or timed automata.

What remains is the determination which of the two characteristics is more important, urgency or stopwatches. They both cause an equal amount of damage to the modularity of the models. The PHAVer model checker used is still at an early development process and will eventually contain methods to implement urgency allowing it to surpass the possibilities of timed automata. On the other hand Timed Automata are based on clocks, it does not seem likely that stopwatches will ever be implemented. The possibilities of Timed Automata are contained within Hybrid Automata. The algorithms used to calculate the state space of Timed automata are optimized for use with clocks and can produce results more quickly.

Although the tool support of UPPAAL makes it the best tool available, hybrid models show most perspective for the future as a method. Other hybrid model checkers might already support urgent transitions, those model checkers still have to support at least Linear Hybrid Automata. Since we are searching for a best method and not a best tool, we advise to pursue Hybrid Automata

for now, but encourage research in the area of StopWatch Automata, which we will define as Timed Automata in which a clock can be stopped by the user. The main question in this topic would be if analysis of StopWatch Automata can outperform analysis of hybrid automata on memory usage or speed.

5.2 RQ2: How can we build a model from a Skill Primitive Net?

As shown in the introduction the motion modules are the main interpreters of the skill primitive. Based on these skill primitives, the user can enable modules that are required for a certain skill primitive and disable modules that are not in use. This would result in a model being generated for every skill primitive within the Net, or the model could stay the same and the activation of the motion modules should be mapped to the skill primitives. The latter is due to the nature of the robot control system where deadlines do not surpass their periods. The first one is the easiest to implement though.

5.3 RQ3: Can the model be changed easily if the robot control system changes?

For every method except for the MAST method adaptations of the model to change three of the most likely adaptations of the robot control system have been summarized. These are changes to the amount of digital signal processors used, most likely when another parallel kinematic machine is used. This will also mean that the inner control program will be replaced, however the inner control systems appear to be of equal structure with different calculation times for the tasks. Adding new types of skill primitives will need new motion module. Adding additional computing nodes to distribute the controller over the different nodes. This however increases traffic on the communication bus. Digital Signal Processors are independent computational nodes with an independent program that is activated by a broadcast of a Master Data Telegram on the bus. In the hybrid and timed modeling language a template of one digital signal processor can be instantiated multiple times. The return communication along the bus is just an additional activation of the Data Device Telegram task. The same holds for Real-Time Calculus although there is currently no option to instantiate a part of the model in there, only a couple of independent commands need to be copied to provide for another resource curve and a greedy processing block that represents the additional Digital Signal Processor.

Adding new skill types of skill primitives is most likely to need new motion modules, which can be added to the second layer. The real time calculus method would only require an additional greedy processing component to be

added in at the motion modules. Although Real-Time Calculus requires a strict priority policy, every task has a different priority, it does not matter in between which two of the motion modules the new module is inserted since the deadlines for all these modules is the same, they all have to finish their tasks before that and if one fails, all fail.

The last probable change to the robot control system is the addition of multiple computing nodes for the control system. This would require new tasks to be defined for the Bus. It is by far the hardest part to automatically generate. Still, if a proper topology of communication needs for the different modules is defined, generation of these new tasks is not that hard. The Real-time calculus might have most difficulty with new tasks on the communication bus. This is only caused due to the inability of Realtime calculus to model a non-preemptive scheduler in a modular way. For the model checker approaches these methods are much more likely to succeed.

5.4 Future Work

We see needs and opportunities for additional research on the following points.

- What is needed for Fixed Priority Preempted Scheduling lays in between of Timed Automata and Hybrid Automata. A new subset of Hybrid Automata can be defined which we call Stopwatch Automata. Automata that have the same abilities as Timed Automata, but can also stop clocks. Analysis methods can be derived that still might profit from these stopwatches and are therefor faster then Hybrid Automata. Currently, work is being done to implement stopwatches in UPPAAL. The task based hybrid model can be implemented in stopwatch UPPAAL once they finish this work.

- Although we used the assumption that firewire can be modeled by using Fixed Priority Non-Preemptive Scheduling, better methods are required to model this bus type. The modeling of communication busses in all methods is at its early stages and not many bus types are supported yet. Additional research could be done for multiple bus types and their performance analysis.

- Although Real-time Calculus does support other schedules besides Fixed Priority Preemptive, these methods are far from modular, but need any task in this schedule to be available beforehand. Any activation pattern where a task in such a schedule would activate another task in the same model would cause an algebraic loop that needs to be solved. This requires additional input and knowledge from the designer that they may not have. Alternatives that are modular need to be researched.

- Tools for modeling scheduling problems in timed automata have been made in the form of the TIMES tool, which would reduce the large modeling effort. The TIMES tool does not support distributed systems and is still very limited. A tool that can solve schedulability problems based on the task approach suggested in the hybrid model could easily be generated for any system.

- Our PHAVer model can be improved if urgent transitions would exist in this model checker. Research is already in progress for this, but not finished. Once finished or if another hybrid model checker supports urgency, the PHAVer case study should be reviewed again to confirm our initial thoughts.

- The PHAVer method shows great potential to be able to model a large number of different scheduling algorithms. This thesis does not cover any other schedules besides the Fixed Priority Preemptive and the Fixed Priority Non Preemptive schedule. It is currently unknown if schedules like the Earliest Deadline First or First In, First Out could be modeled in a compositional task oriented way. Additional research could result in compositional models for new schedules.

- Due to the different strengths and weaknesses the different methods show, the question arises if a suitable way of combining different methods would be applicable.

- Realtime Calculus allows for different schedules to be run within one resource, questions arise if the other methods can also provide this feature.

Bibliography

[And99] Don Anderson. *FireWire system architecture (2nd ed.): IEEE 1394a*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2004.

[CB08] Devesh B. Chokshi and Purandar Bhaduri. Modeling fixed priority non-preemptive scheduling with real-time calculus. In *RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 387–392, Washington, DC, USA, 2008. IEEE Computer Society.

[FKW05] Bernd Finkemeyer, Torsten Kröger, and Friedrich M. Wahl. Executing assembly tasks specified by manipulation primitive nets. *Advanced Robotics*, 19(5):591–611, 2005.

[Fre05] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *HSCC*, pages 258–273, 2005.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[HT07] Wolfgang Haid and Lothar Thiele. Complex task activation schemes in system level performance analysis. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 173–178, New York, NY, USA, 2007. ACM.

[HV06] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. *Parallel and Distributed Processing Symposium, International*, 0:165, 2006.

[JPTY08] Bengt Jonsson, Simon Perathoner, Lothar Thiele, and Wang Yi. Cyclic dependencies in modular performance analysis. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 179–188, New York, NY, USA, 2008. ACM.

[KKR+04] N. Kohn, M. Kolbus, T. Reisinger, K. Diethers, J. Steiner, and U. Thomas. Prosa - a generic control architecture for parallel robots. In *Proceedings of Mechatronics and Robotics*, pages 55–61, Aachen, Germany, 2004. Sascha Eysoldt Verlag.

[KVSG04] N. Kohn, J. Uwe Varchmin, Jens Steiner, and Ursula Goltz. Universal communication architecture for high-dynamic robot systems using qnx. In *ICARCV*, pages 205–210. IEEE, 2004.

[MMG+02] González Harbour Medina, J. L. Medina, J. J. Gutiérrez, J. C. Palencia, and J. M. Drake. Mast: An open environment for modeling, analysis, and design of real-time systems, 2002.

[MSA+08] Jochen Maaß, Jens Steiner, Ana Amado, Jürgen Hesselbach, Michaela Huhn, and Annika Raatz. Self-management in a control architecture for parallel kinematic robots. In *Proceedings of the ASME 2008 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2008*, Brooklyn, New York, USA, August 2008.

[PWT+07] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, Arne Hamann, Simon Schliecker, Rafik Henia, Razvan Racu, Rolf Ernst, and Michael González Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 193–202, New York, NY, USA, 2007. ACM.

[SHG07] Jens Steiner, Matthias Hagner, and Ursula Goltz. Runtime analysis and adaptation of a hard real-time robotic control system. *JCP*, 2(10):18–27, 2007.

[SHM06] J. Steiner, M. Huhn, and T. Mücke. Model based quality assurance and self-management within a software architecture for parallel kinematic machines. In *Proceedings of the IEEE 3rd International Conference on Mechatronics (ICM2006)*, pages 55–60, 2006.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systeme, 2000.

[TWMH05] U. Thomas, F.M. Wahl, J. Maaß, and J. Hesselbach. Towards a new concept of robot programming in high speed assembly applications. *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 3827–3833, Aug. 2005.

[WT06] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox. http://www.mpa.ethz.ch/Rtctoolbox, 2006.

[WTVL06] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *Int. J. Softw. Tools Technol. Transf.*, 8(6):649–667, 2006.