Timing analysis of a robot controller

Masters' Thesis

Johan Andersson
Mälardalens Högskola
Västerås, Sweden
jan98053@student.mdh.se

Jonas Neander
Mälardalens Högskola
Västerås, Sweden
jnr98003@student.mdh.se

October 10, 2002

Abstract

ABB Robotics has no methods for theoretical verification of the timing behavior of the robot control system. The system is complex, about 2,5 million line of code, distributed over 15 subsystems.

When changing the system, it's hard to predict how that change will affect the temporal behavior. The system was not designed to explicitly support impact analysis. This has resulted in a "trial-and-error" approach regarding temporal issues when adding new functionality. Timing related errors can be very costly, since they might occur only under very special conditions and thus might be hard to find.

This thesis proposes a solution to the problems. A set of tools and methods for verifying the temporal behavior is presented.

The approach of this work is to enable the temporal analysis of the robot controller by enabling the simulation of a model. Since no existing solutions has been found, the development of a modelling language and a simulator is necessary. The model is implemented by describing the robot control system in the developed notation (ART-ML) and inserting data measured from the control system.

The model and the simulator have been used to simulate the system and the results from that simulation are easily comparable with the results from the measurements. The comparison shows as expected similarities. It is not a perfect match, but with respect to the roughness and simplicity of the created model the results are promising.

The created model of robot control system consists of about 100 lines of code. That is a very simple model compared to the complexity of the modelled system. Creating a more detailed model would increase the accuracy of the simulation. Adding more details to the model is very possible since the data that is extracted by the measurements contains a lot of dependencies between the tasks that can be investigated and inserted in the model.

Using these tools/methods with a more accurate model, it is possible to test changes to the robot controller before implementing them.

This method of simulation-based timing analysis is very general and flexible. The modelling language is capable of describing many classes of systems and the simulator is fast, enabling simulations of complex models.

Acknowledgements

We would like to thank our examinator, Christer Norström, for his enthusiasm of this work and the support given to us. We would also like to thank Anders Wall at Mälardalen University, Martin Lembke and all the other developers at ABB Robotics for their support during these six months.

Contents

Chapter 1

Introduction

ABB Robotics develops industrial robots and robot control systems. The control system is very complex; it contains over 60 tasks and several million lines of code, distributed on three processors. It is not trivial to verify that a system of this complexity work as it should.

ABB Robotics has no methods for making timing/impact analysis of changes in their software. The customers have high demands on the robots, especially regarding reliability. A robot that stops working is expensive, since it can stop an entire production line.

Since it is such a complex system it is hard to know if there is room for new/modified functionality, in terms of CPU-utilization, memory etc. The way the developers are working today is to implement the new functionality and perform a lot of tests to verify that the system works as expected. It is very hard to know if the temporal behavior is correct (if there are timing problems) before implementing and testing it. Incorrect temporal behavior can be hard to fix due to the complexity of the system. Using this "trial-and-error" approach is expensive in man-hours and it could take a lot of time to get the product ready for release.

Re-designing the system could reduce the complexity and recent results from research regarding product line architectures [18], task scheduling and monitoring could be used to make the system more robust with a higher testability and ease further development. To re-design the robot controller, i.e. to develop a completely new system, is too expensive and at this time not an option. Methods and tools to verify the temporal behavior of the system are therefore desired.

The goal of this work is to provide these tools and methods mentioned above and use them to verify the temporal behavior of the three prioritized tasks. To achive the goal, the work was divided into the following projects:

- develop a task model including interrupts and all relations between the tasks.
- populate the model with data from the requirements and measured data from the system.

- perform a response time analysis. Not only worst case but also distributions in response times and sampling periods are interesting.
- develop a tool for analyzing the model, preferable Excel-based.
- investigate the results and how the method can be used when adding new functionality into the system. Impact and robustness analysis is especially interesting.

These methods/tools are intended to be used to test changes in the software of the robot controller at an early stage, before they are implemented. The process for conducting such impact-analysis is described in figure 1.1. The result of this work hopefully shortens the time required for the testing and verification and thereby increases the robustness of the system.

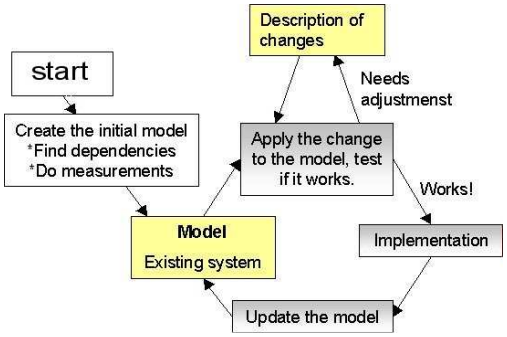


Figure 1.1: The intended use of the tools

The process described in figure 1.1 starts with the creation of the initial model. When this model exists, it can be used to perform impact analysis when changes is to be implemented. The analysis can predict how the changes will affect the temporal behavior of the system. If the analysis indicate that there are problems with the temporal behavior in the altered system, these problems can be investigated prior to implementation. When the changes have been implemented, the model of the new system can be validated. By that, the loop is closed.

This is the public version of the report. The actual names of tasks and classes has been replaced. Any detailed descriptions of the target system has been removed.

Chapter 2

Theoretical Background

A Real-Time System (RTS) is a system where correct behavior is not only dependant on what result it delivers, but also when it does it. [14] A RTS is composed of parallel processes, tasks, that often communicate and depend on each other. The latest time a task can deliver a result without being too late is the deadline of the task. Real-Time Systems can be divided into two categories, hard and soft Real-Time Systems. In a hard real-time system a single deadline is considered a failure. In a safety-critical application such as aerospace or military applications, where hard real-time systems are common, a missed deadline might result in catastrophic damage. There is a need to guarantee that there are no timing problems in such systems, i.e. i.e. their temporal behavior is correct. A soft real-time system allows some occasional deadline-misses, e.g. the requirements do not need to be guaranteed at all time. An upper limit is often defined over a time period.

There exists several scheduling policies for hard real-time systems (most common are Rate Monotonic and Earliest Deadline First, EDF, also referred to as Deadline Monotonic) and a lot of work has been done regarding hybrid task sets, containing both hard and soft tasks [5, 2, 3]. In the robot controller, fixed-priority scheduling is used, i.e. all tasks are manually assigned a priority.

2.1 Fixed Priority Analysis

Fixed Priority Analysis (FPA) is a method for verifying the feasibility of a fixed priority task set [14]. The basic idea is that if the task set is schedulable under worst-case conditions, it will always be schedulable. The worst case condition is when all tasks want to execute at the same time. If a deadline is missed, FPA will show it. It tells however nothing about how probable that scenario is or if other scenarios will result in missed deadlines as well. It only tells if the system works under the theoretically worst-case conditions, which might never occur. A system can be schedulable if the theoretical worst-case scenario never can occur, but the analysis tells the opposite.

A lot of work exists regarding timing and reliability analysis, although nothing found has the same focus as this thesis. Next follows a summary of what was found during the literature studies that was conducted for this work.

2.2 Related Work

In [11], a simulation based methodology for reliability analysis of distributed systems is presented. A tool for this has been developed and is evaluated in a case study. The goal of their work is to enable timing analysis on an early stage when designing a system [11]. The simulation-based approach extends the class of applications that can be analysed, compared to traditional analysis methods. Simulation is extensively used in the industry, so this approach should be attractive. In the simulator, a system is modelled as a set of nodes, interconnected by buses. Each node has a set of tasks that sends and receives messages using the buses. Behavior of tasks in the simulated system is described as sequences of states. Two types of states exists; call states and execute states. In a call-state, the task is using a system-service. When a task is in a call-state, it is not pre-emptable. When a task is in an execute-state, it can be pre-empted. The duration of an execute-state is determined by a probability distribution. The level of abstraction in this approach can be varied to some extent, but since it doesn't contain a modelling language, the lower limit of the abstraction level is quite high. The tool is evaluated in [10] by creating a model of an actual system and simulating it.

A way of analysing task sets with stochastic execution times is presented in [13], but they assume strictly periodic, non pre-emtable tasks, which is not very suitable for this work, as the control system contains pre-emption as well as aperiodic tasks.

In [12] a method is presented for using measurements to derive the worst case execution time (WCET) of the tasks in an existing system. The following quote from the paper describes it quite well: "The essence of the method is to derive a system of linear equations from a limited number of timing measurements of an instrumented version of the considered program." [12]. The equations that are derived describes the different execution times of the program.

A tool-suite called STRESS is presented in [4]. STRESS is developed by Audsley et al. at the University of York. The STRESS environment is a collection of tools for "analysing and simulating behavior of hard real-time safety-critical applications" [4]. STRESS contains a modelling language where the behavior of the tasks in the modelled system can be described. It is also possible to define algorithms for resource sharing and task scheduling. STRESS is primarily intended as a tool for testing various scheduling and resource management algorithms. It can also be used to study the general behavior of applications, since it is a language-based simulator.

DRTSS is related to STRESS. DRTSS is a high level simulation framework that "allows its users to easily construct discrete-event simulators of complex, multi-paradigm, distributed real-time systems" [17]. The DRTSS framework is quite different from STRESS, although they are closely related. It contains a set of algorithms and protocols from which one can pick the appropriate ones and build a simulator. New algorithms and protocols can be added to the original set. It has support for searching for extremes in the timing behavior of the simulated system. DRTSS has no language where task behavior can be specified, so the abstraction level of the simulation is high and fixed.

DRTSS is a part of the PERTS tool-suite, which was developed at the University of Illinois at Urbana-Champaign. The PERTS tool-suite has been commercialised by Tri-Pacific Software Inc.[16].

Chapter 3

System description

The robot control system is based on VxWorks, a commercial fixed-priority Real-Time Operating System developed by WindRiver. The tasks are non-terminating and cyclic (i.e. each task contain a non-terminating main-loop). They communicate through IPC-queues, shared objects and semaphores. The tasks have priorities between 0 and 255, where 0 is the highest priority. There are multiple processors in the system, the main computer and two others. This work focuses on the main computer.

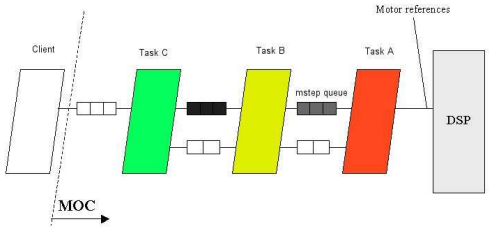


Figure 3.1: The task structure

3.1 System behavior

The main-computer generates the motor-references and brake-signals that the external computer need. The external computer sends requests to the main computer every X'th millisecond and expects a reply in the form of motor references. This depends on three tasks: Task A, Task B and Task C, figure 3.1 [1]. Two of these tasks, Task B and task A, are high-prioritised, periodical and frequently executing. Task C mostly executes in the beginning of each robot movement and has lower priority. The final processing of the motor references is done by Task A. It sends the references to the external computer. Task A is dependent on data produced by Task B. If the queue between them gets empty, Task A can't deliver any references. This causes a system failure, i.e. the robot stops and a reboot of the controller is required. Task C sends data to Task B when a movement of the robot is requested. If the queue between Task C and Task B gets empty, the robot stops. In this state, Task B sends

default-references to Task A.

3.2 Limitations of the Thesis

Although the tools and methods produced by the work should be valid for the entire system, the three tasks Task A, Task B and Task C are prioritised. Verification of other parts of the system is to be done if there is spare time.

The accuracy of the model is not the primary result of this work, the focus should rather be on the quality and usability of the methods and tools, although an accurate model is desirable.

Chapter 4

Problem analysis

4.1 The re-engineering aspect

Since this work focus on temporal verification of an existing system with hundreds of man-years of work put into it, there is a need for a re-engineering of the system, where a model is created, based on the source code. The system is about a decade old and has evolved considerably from its initial design. Temporal analysis has never been an issue when designing or implementing the system. An alterative to model the system by re-engineering is to create a new system that is easier to verify. However, the cost would be extremly high, to introduce support for temporal verification is alone not enough to motivate that.

4.2 Problems with existing analysis methods

Performing a "classic" response time analysis of this system, such as Fixed Priority Analysis (FPA) [14], results in a very pessimistic picture of the system. FPA is based on the fact that if a set of tasks (containing only periodical tasks with deadlines less or equal to their periods) is schedulable under worst-case conditions, it will always be schedulable. The worst case condition is when all tasks in the task-set want to execute at the same time. The result from such analysis is of a binary nature, it doesn't give any numbers on probability of failure, it just tell if the system is guaranteed to work or not. In this work, the result from an FPA would be negative, telling that this system will miss deadlines under worst-case conditions.

The task model of this system is not very suited for Fixed Priority Analysis. The task-model doesn't specify any deadlines for the tasks, although there are temporal requirements on the system. These temporal requirements are not specified in terms of task deadlines. Some tasks can have their deadlines derived from these requirements, but not all tasks can easily be assigned a deadline. To perform an FPA, it is necessary to know the period and worst-case execution time of the tasks. Task C is not periodic, it executes sporadically and with very different execution times. Using the worst case in terms of both execution time (maximum) and inter-arrival time (minimum) is not an option in this case, it would be extremely pessimistic.

FPA is suitable if a single missed deadline is directly catastrophic. In this case, it is not. A missed deadline can under worst case conditions be dangerous, for instance if the emergency stop-button is pressed or a collision is detected, it is very important to stop the robots movement on time. However, the situations where a missed deadline is dangerous are rare, in most cases a missed deadline will only stop the robot and thereby stall the production.

Something that Fixed Priority Analysis can not provide but is very interesting is to get numbers on probability of failure. FPA will only tell that the system doesn't work under worst-case conditions, it tells nothing about probabilities. It is obvious that another method is necessary.

Two methods to achieve the goal were considered. The first approach was based on a formal analysis, the second a simulation-based approach. The analysis-based approach would be to extend FPA to use distributions of execution times instead of a worst-case. The simulation-based approach was to create a simulator, which used the model of the system as input.

4.3 The analysis-based approach

In [13] a method for timing analysis of task models with stochastic execution times is presented, but it could not handle sporadic tasks. A solution for this could not easily be found. Without fixed inter-arrival times (i.e. in presence of sporadic tasks), a least common divider of the tasks inter-arrival time can not be found. The thought occurred to use FPA to analyse every possible scenario, i.e. every execution time and every possible inter-arrival time of all tasks in any combination.

Testing all possible scenarios would be extremely time-consuming, even for simple systems. For example, consider the sporadic task T. It has 3 possible execution times and 3 possible inter-arrival times. The number of possible execution-scenarios that exist for task T is therefore $S_T = 3*3 = 9$. This means that if this is the only task in the system and the number of executed instances is $I_T = 10$ there exists ${S_T}^{I_T} = 9^{10} (\approx 3.48 * 10^9)$ possible execution-scenarios for the entire run. If other tasks are included as well, each having a set of execution scenarios, the state space grows to enormous proportions. Say for example a system consisting of three tasks, each having 9 possible execution scenarios (3 execution times and 3 release times), at every preemption point, there are 27 possible ways the system can continue its execution, so after 10 executed (non-preemptable) instances the state space equals $27^(10) \approx 2 * 10^14$. If a computer could test 10^6 execution scenarios every second, it would take $2 * 10^8$ seconds to test the entire state space, i.e. over 6 years. Even if computers double their computing capacity every 18'th month according to moore's law, it is obvious that it is impossible to analyze longer execution scenarios using this approach.

4.4 The simulation-based approach

The model of the system that is used in an analytical approach of temporal verification (such as Fixed Priority Analysis) does not consider what the task

actually does (the behavior), only task parameters such as inter-arrival time, execution time and priority. In such an approach, it is hard to include tasks with stochastic parameters.

If the behavior of the tasks is included in the model, the system can be analyzed with higher accuracy. The presence of a description of the behavior can reduce the number of possible scenarios, since the behavior controls the execution and many of the theoretically possible cases might never occur due to the behavior of the tasks.

The behavior of the entire system is determined by running the model of the system through a simulator. It creates a trace of the execution (i.e. a log) from which relevant data can be extracted. The behavior of the system can be described in different levels of abstraction. In the most detailed level, the actual code for the tasks is the "description" of the behavior, in this case, a perfect description. There is no point in creating such a detailed description, it would be easier to test the the actual system. The least detailed description of the behavior is to only specify the execution time of the task, no behavior at all is described. This would not be a very accurate model. Obviously none of these two extremes are suitable, but something in the middle is, where only the most important parts of the behavior is described. It is a trade-off between model complexity and accuracy. This approach was selected for the work, since it allows the behavior of the tasks to be modelled without resulting in an extremly complex model.

Chapter 5

The model

5.1 Demands on the modelling language

In order to create a model of the robot control system, there was a need for a formal notation describing the system. A large quantity of work was put into finding an existing notation for suitable for describing the system. The demands on such a notation were defined.

The notation should have the below listed properties.

- It should be able to describe the tasks forming a complex system in different levels of abstraction.
- It should also be able to describe kernel-calls such as message passing and other relationships between tasks.
- The notation should also be able to describe the execution time characteristics of a task in terms of distributions rather than worst-case.
- Furthermore, the notations should be able to describe memory usage, task priorities, deadlines and timeouts, cycle times (of periodic tasks) and allow analysis of task release jitter, task execution times, task response times, end-to-end response times and message-queue sizes.
- The notation should be down-to-earth and simple to use, since it is meant to be used by developers without academic background in real time systems.

Three possible solutions was found, STRESS, DRTSS and a simulator developed at Mälardalen University [11]. STRESS was the most suitable (although not perfect) solution.

5.2 STRESS

"The STRESS environment is a collection of CASE tools for analysing and simulating behavior of hard real-time safety-critical applications" [4] STRESS

is primarily intended as tool for testing various scheduling and resource management algorithm. It can also be used to study the general behavior of applications, since it is a language-based simulator.

The tools included in STRESS are a simulator, a presentation tool and examples of the modelling language. The modelling language allows very complex behavior to be modelled and the presentation tool presents a very nice trace of simulation, but it crashes if the duration of the simulation exceeds about 10000 time-units. STRESS has no support for modelling distributions of execution times or memory allocation. Attempts were made to get the source code for STRESS, but they were unsuccessful, so it was not possible to improve STRESS. These flaws make STRESS unsuitable.

5.3 DRTSS

DRTSS is a simulation framework that "allows its users to easily construct discrete-event simulators of complex, multi-paradigm, distributed real-time systems" [17]. The DRTSS framework is quite different from STRESS, although they are closely related. DRTSS has no language where task behavior can be specified. That it something that is necessary for achieving the goals of the work and excludes DRTSS as a possible solution.

DRTSS is a part of the PERTS tool-suite, which was developed at the University of Illinois at Urbana-Champaign. The PERTS tool-suite has been commercialized by Tri-Pacific Software Inc. `www.tripac.com`.

5.4 The anonymous simulator presented by Lindgren et al.

The work presented by Lindgren et al. in [11] is similar to STRESS and DRTSS. The goal of their work is to enable timing analysis on an early stage when designing a system. They use a simulation based approach as well. In their approach, a system is modelled as a set of nodes, interconnected by buses. There is no support for IPC-type message queues as used in the robot controller. The behavior of the tasks in the simulated system is described as sequences of states, which limits on the lowest level of abstraction. The modelling langauge in STRESS enables more detailed/complex models to be implemented. The focus of [11] is analysis of distributed systems and buses in a higher level of abstraction which is not exactly the focus of this work.

Chapter 6

The Solution

Since no existing solution was found, the decision was taken to implement a modelling language and simulator from start. This resulted in a STRESS-inspired modelling-language and simulator.

A language and three tools was developed:

- The modelling language ART-ML
- A compiler for ART-ML
- A simulator
- A tool for extracting data from the logs generated by the robot controller/simulator

This tool-suite is capable of performing impact-analyses on a system regarding its temporal behavior. By applying a change to a model of the system and run it through the simulator, it is possible see how the change will affect the temporal behavior. The accuracy of the simulation depends on how accurate the model is and the simulation length.

The model can be made very accurate since it is possible to get feedback on the accuracy. The simulator produces log-files of the same type as the control system generates. The simulator's output can thereby be compared with the logs from the actual system and the model refined. When iterating this process, the accuracy of the model increases.

The performance of the simulator is sufficient for the purpose of this work. Simulating a model take about 0.1 - 10 times the execution length depending on the complexity of the model and the performance of the computer.

A performance-test has been conducted where a model consisting of three tasks, each about 10 statements of behavior was simulated. The stop-time of the simulation at $6 * 10^8$ s, 10 minutes. It took about 9 minutes and produced an 366 MB output-file. The computer used was a Pentium II at 400MHz, running Microsoft Windows NT 4. In other words, it is possible to do very long simulations, as long as there are enough disk-space. The size of the output can be reduced if the simulator is changed to produce binary output instead of plain text-files.

6.1 The Modelling Language - ART-ML

The language ART-ML (Architecture and Real-Time behavior Modelling Language) supports multiple processors, inter-process communication, synchronisation and fixed priority scheduling. Each processor in a model can have up to 256 tasks. There is no support for multiprocessor systems in the simulator, even though the modelling language supports it. Implementing such support would introduce a lot of problems that there is no time for solving in this thesis.

It was designed as a mean to describe the robot control system. It is an imperative language, like ANSI C, Pascal and Basic. It's designed to be syntactically close to ANSI C (although simplified) since C is used for the development at ABB Robotics and the developers are used to the syntax. It is based on the concept of tasks. A task in the ART-ML language can be divided into two parts, attributes and behavior. Three attributes are mandatory for a task, Name, Task Priority and Task Type. There are two more attributes, Deadline and Memory, but they are not mandatory.

The behavior part of the task is the "code" of the task, the model of the code in the actual system (the robot controller). The behavior is a list of statements. A statement can be an assignment (the language supports integer variables), a loop, a selection or a kernel-call. ART-ML has no support for function-calls. All expressions must end with a semi-colon ";". If variables are used, they need to be declared, either in the beginning of the behavior-part of the task or on processor-level. A variable that is declared within a task is only accessible within that task. A variable that is declared within a processor is accessible from all tasks within that processor.

An ART-ML model of a system contains one top-level component, the "system". That components owns all the other components in the model, either directly or indirect by owning components that owns other components. The system contains one or multiple processors and other components such as mailboxes,variables, constants and semaphores. A processor can own other components as well. It can not own other processors, but it owns one or multiple tasks, semaphores, mailboxes and constants.

6.1.1 Task Model

The language supports several different ways of activating a task. Periodic and aperiodic terminating tasks can be described, as well as non-terminating tasks. It is possible to explicitly set a task to be activated on an incoming message, and it is possible to assign a task a probability of activation. Such a task is activated by a certain probability every X'th time-unit in the simulation. Every task must have a priority, an activation method and a name. Deadline and memory usage can also be specified, but it's not mandatory. Interrupts are modelled as high-prioritized tasks.

6.1.2 Modelling the behavior

A tasks behavior can be modelled in various abstraction levels. The most abstract model of a task is to only specify its distribution of execution times. On the other extreme is to make a perfect model, i.e. the model have the exact same behavior and temporal characteristics as the real task. The latter alternative is not an option unless the task is very simple and the hardware platform of the simulated system is very deterministic in its temporal behavior.

A level of abstraction has been chosen for the model. The execution time distribution and the inter-process communication are described. It is possible to add various dependencies in the model so that the execution times are dependent of the message passing. It would improve the accuracy of the model considerably.

The way task-behavior is described in ART-ML reminds of ANSI C code. The ART-ML language is a lot simpler though, there are no pointers, functions or arrays and ART-ML contains only integer variables. The ART-ML language contain selection, iteration and assignments, so the language is turing-complete [7].

The language contains a form of stochastic selection, the chance-statement. It works just like an if-statement, but instead of evaluating an expression, it compares a random number between 0 and 99 with the expression. With this statement one can add stochastic selection of behavior in a task. For example, it is possible to specify that there is a 19% chance of sending a message if a message was received, else there is 42% chance of sending the message.

There is a statement in the language that is very important, the execute-statement. It consumes time according to a distribution. That is the only instruction that affects the simulation clock. The execute-statement is not atomic, the task can be pre-empted while executing an "execute"-statement.

A rough model of a typical task can be constructed from three statements. First a "receive" that waits for a message, next one or several "execute" that consumes time and finally, a "send" statement that sends some data to another mailbox.

6.1.3 Types

The modelling language is based on 7 different types, system, processor, task, semaphore, mailbox, variable and constant, figure 6.1. The system is the top-level component, there can only be one system in a model. The system contains one or multiple processors (multiprocessor-support is not implemented in the simulator), and can also contain mailboxes, semaphores and constants. The processors can contain all types except other processors and the system-type. A task can only contain variables and constants. A task always contains behavior, i.e. it must have at least one instruction.

- `variable i;`

  Declares the variable `i`. The variable is initially set to zero.

- `constant c N;`

Declares the constant `c`. The constant gets that value N. A constant is actually a variable, the simulator is not aware of the concept of constants, the compiler transforms the into variables when writing the code. This is why there are no constants in figure 6.1.

- `semaphore s;`

  Declares the binary semaphore `s`. The semaphore is initially unlocked.

- `mailbox m N;`

  Declares the mailbox `m` with the size N. A mailbox has two members, `size` and `maxsize`. They contain the actual and maximum number of messages in the mailbox, respectively. Both are constant, at least in the sense that they are write-protected. The `size` changes it's value since it is updated when a send/recv occur. This typing is to protect the mailbox from corruption from possibly (semantically) faulty behavior code in the model. If the size member was a variable, it could be changed from the model and cause an inconsistency in the simulator, i.e. the size member could give a different value compared with the actual number of messages stored in the mailbox. These two members are automatically created when a mailbox is declared. They can be used to check if there is any messages in a mailbox or it a mailbox is full without calling send/recv.

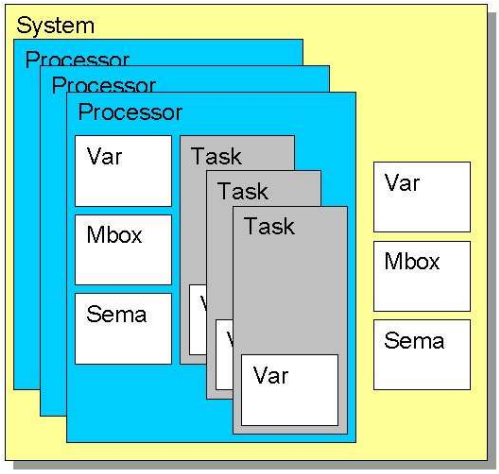


Figure 6.1: The system

6.1.4 Selections

There exist two different kinds of selection in the ART-ML language, the standard if-statement and a probabilistic selection statement called chance. Both selection-statements support an else-block, but it is optional. The chance-statement works just like the if-statement, but the selection is based on the probability that is given as parameter, i.e. chance(10) means that it is a 10% chance of executing the first block and 90% chance of executing an else-block.

- ```
  if (expr)
      statement;
  ```

  If *expr* is non-zero, *statement* is executed.

- ```
  if (expr){
      statement1;
      statement2;
      ...
      statementN;
  }
  ```

  statement can be a list of statements, a block, if braces are used to encapsulate them.

- ```
  if (expr)
      statement1;
  else
      statement2;
  ```

  If *expr* is non-zero, *statement*1 is executed, else *statement*2.

- ```
  chance (expr)
      $statement1$;

  chance (expr)
      $statement1$;
  else
      $statement2$;
  ```

  Like the if-statement, except the condition. A random number between 1 and 100 is generated and compared with *expr*. If the random number is less than the value of *expr*, *statement*1 is executed, else *statement*2. This means that the value of *expr* is the probability (in percent) of execution of *statement*. If an else is used, the probability of execution of the else-block is therefore $(100 - expr)\%$.

6.1.5 Iterations

ART-ML supports while and for-loops. The for-loops doesn't follow the C-syntax however, they are somewhat simplified. They take three arguments, just as in ANSI C, but the arguments have different meanings. The first is the loop-variable, the second is the starting value of the loop-variable and the third is the maximum value.

- ```
  while(expr)
      statement;
  ```

  statement is executed as long as *expr* is non-zero.
  ```

- ```
  while(expr){
      statement1;
      statement2;
      ...
      statementN;
  }
  ```

  The list of statements is executed as long as *expr* is non-zero.

- ```
  for(i,start,stop)
      statement;
  ```

  statement is executed as long as $i < stop$. Initially i is set to *start*. i is (automatically) incremented by one on each iteration. *statement* can be a block encapsulated by braces.

6.1.6 Kernel calls

The functionality for message passing and synchronization and other "kernel-calls" is located in the simulator, the compiler just output a single instructions for each kernel-call. Six kernel-calls exists:

- sleep(T)

  Make the task sleep for T us.

- semtake(S) [timeout T]

  Attempt to lock the semaphore S. If the semaphore already is locked, the executing task gets blocked. If the timeout is specified, the attempt is aborted after T timeunits.

- semgive(s)

  Unlock the semaphore S.

- send(Mbox,S) [timeout T]

  Send the value of S (an immediate value or integer variable) to the mailbox Mbox. If that mailbox if full, the executing task is blocked until there is room for the message. If the timeout is specified, the attempt is aborted after T timeunits.

- recv(S,Mbox) [timeout T]

  Read a message from mailbox Mbox and put the value is S (a variable). If that mailbox is empty, the executing task is blocked until a message arrive. If the timeout is specified, the attempt is aborted after T timeunits.

- execute($(P_1, T_1), (P_2, T_2), \ldots, (P_n, T_n)$)

  Execution time is consumed according to the distribution specified. One or multiple pairs of probability/execution time can be specified. These pairs corresponds to particular execution scenarios in the actual system. This call tightly coupled with the simulator engine, this is described in 6.2.3.

6.2 The Simulator

6.2.1 Interpreter vs. Virtual Machine

When designing the simulator, two different approaches were identified. The most intuitive was to let the simulator parse the model and execute it statement by statement. The other approach was to create a compiler that translated the statements into simple, assembly-like, instructions and create the simulator as a virtual machine that executes the instructions. A test was made to compare the performance of the two approaches. Two very simple versions of the two simulators were created. Their task was to evaluate the expression $1+2*6/4-3$.

The interpreter-based simulator did this by recursively parsing a string containing the expression. It searched the string for operators. If no operator was found, it converted the string to an integer and returned it. If one or more operators were found, the string was split (using pointers, no data was copied), the two parts were evaluated by a recursive call and their results combined with the appropriate operator.

The virtual-machine based simulator had support for four instructions, add, sub, mul and div. It had a register-bank where the result from the instructions was put. It had a list of instruction, a program, evaluating that same expression. It had a simple function called execute, that executed all instructions in the list. It checked the subtype of the instruction, i.e. if the data was immediate or located in the register-bank, applied the appropriate operation to the operands and put the result on the specified register. The program had four instructions with the following semantics $R0 = 2*6, R0 = R0/4, R0 = R0-3, R0 = R0+1$.

Both versions was set to evaluate the expression 50000 times, the latter version was 40 times quicker. That is a significant difference in performance, so the choice was not hard to make. We implemented the compiler/virtual-machine solution.

6.2.2 General behavior

The simulator reads a compiled model from file. The compiled model contains a symbol table (a list with all identifiers, their type and any associated value) and the code for each task, Appendix A 43. The code for the tasks is created by the compiler and is assembly-like low level instructions. The simulator builds up a linked data-structure, containing the complete description of the system. Next it enters the main-loop, selects a task to execute and executes the instructions of that task until a context-switch occurs. The main-loop is terminated when the current time in the simulation exceeds the user-specified limit.

6.2.3 The simulator engine

The engine is based on three parts. The instruction decoder, the scheduler and the event-processing. The instruction decoder executes the instructions. It identifies them and calls the appropriate functions. Some of these functions create events. An event always contains three things, type of event, id of source-task and a timestamp telling when the event is to be activated. The events

generated by the tasks are stored in a global datastructure. The purpose of these events is to notify the simulator when timeouts occur in the simulation.

The scheduler decides what task that is to execute. It is also responsible for calling the functions that creates the output.

The "execute" kernel-call, the consumption of time, is what drives the simulation forwards. First, a time-increase is selected according to the distribution that is passed as argument. The current time is then increased with that time-increase. If the change of the current time makes any events occur, the execution of that task is suspended, the event is taken care of and the scheduler is called. The next time that task is allowed to execute, it will restart the execution of the execute-instruction, but it remembers that it has executed previously and for how long, so the time-increase is adjusted this time.

Since an "execute" kernel call is necessary for pushing the simulation forwards, there must always be a task that is ready to execute and contains such a statement. Due to this it is mandatory to have an idle-task in the simulation that consumes time if no other task is ready.

6.3 Compiler

The compiler translates a model in ART-ML to a binary representation. The code that describes the behavior of the tasks is compiled into assembly-like instructions. The different identifiers are stored in a symbol table and written to the output file together with the code.

Chapter 7

Measurements

7.1 Setting up the model

In order to get more familiar with the control-system documents such as "Motion Control OBM" [1] was studied, but the most efficient method of gaining knowledge of the system was to interview the developers. Three interviews was conducted in order to cover the area this thesis focus on.

With this knowledge, a rough model describing the behavior of the three tasks could be defined using the ART-ML language. There were however no data to insert in the model, there was a need to measure the distributions of the tasks execution times.

7.2 Setting up the test system

To be able to measure the data, a robot system had to be configured. This took a lot of time, since a lot of problems occured. At first the motion-control test-system, moctest, was set up. After some time it was clear that the real system had to be used instead, since moctest doesn't contain everything that the real system does and doesn't have the same temporal behavior. Another problem was that the robot-system that was used had a motor replaced, but the controller was never configured to work with the new motor, so it was not possible to measure anything until the old motor was put back.

An arc-weld configured robot was used for the measurements, since it was considered the worst case in terms of processor utilization. A robot running an arc-welding program makes a lot of small movements that require a lot of calculations.

7.3 The system monitor

The operating system provide functions to hook user defined code to events such as context-switches, task creation and task deletion. Developers at ABB Robotics have used this to implement a system monitor. 7.1.

The system monitor stores relevant information about the previously executing task, which task it was, its status (i.e. cause of task-switch), a timestamp

Taskname	"sched"	"complete"	Status	Totalt	Task_Cts
Task_Cts	619.61us	619.61 us	PEND DELAY	0.00 ns	1
Shell	3.41 ms		READY	619.61 us	0
| Task_Cts	852.26 us	852.26 us	PEND DELAY	4.03 ms	1
Shell	1.40 ms	4.81 ms	PEND	4.88 ms	0

Table 7.1: The old system monitor log.

and the number of messages in message queues. The system monitor can store a fixed number of context switches in the memory. Currently at release 4.0.53 of the robot-system could about 16000 context-switches be stored. The amount of free memory is the limitation. The available space is enough to store about 6 seconds of execution. Measurements of 25 seconds or more is possible in other memory configurations.

A new function has been added to the system monitor to enable the start of the logging at a certain location in the code. The existing method of starting the system monitor was to call the snStart function manually from the shell. The snTrig-function polls a certain variable, the system monitor start-flag, for 30 seconds. When the flag gets the requested value, snTrig stops the polling and calls snStart in order to start the system monitor. If the 30 seconds pass without the flag is set, the snTrig-function finishes. The flag that snTrig poll is system global, it can be used wherever wanted in the system. This solution enables repeating a measurement of a certain scenario. Due to the polling nature of the snTrig function the system monitor doesn't start at the exact same time when repeating a measurement, but very close. This is due to the polling nature of the snTrig-function (the variable is polled every millisecond) and the quite low priority of the task that executes the snTrig-function.

The function system monitor_switch_hook has been modified to store information about multiple IPC-queues. Previously only a single queue was monitored. Even the queue between Task A and Task B is included, 7.2. That queue is not a regular IPC-queue, it's a custom FIFO buffer. A counter was added in the FIFO-class, it holds the number of messages in the queue. The system monitor monitors that counter to record the number of messages in the queue. The amount of IPC-queues the system monitor records can be changed, currently five queues are monitored (the queue of Task C, the two queues of Task B and the atwo queues of Task A). The time used in system_monitor_switch_hook is taken from the software probe class instead of the standard OS clock so that logfiles from the software probes and system monitor could be compared. The clock rate used in the system monitor class had to be changed from OS clock's rate to the software probe class' clock rate. The system_monitor_print_log function that writes stored data to file had to be modified to the new behavior in system_monitor_switch_hook.

The overhead from system_monitor_switch_hook is low, that is because the system monitor doesn't use any dynamic allocation of memory and it doesn't do any operations on strings. system_monitor_switch_hook just store some integers in a static array of structs. All handling of strings are made in the system_monitor_print_log function, which is called afterwards when the system

Taskname	"sched"	"complete"	Status	Tot	Task_C	Task_B	Task_B_pri	Task_A	mstep
Task_C	0.758	0.758	PEND DELAY	0.000	0	0	0	1	12
Shell	3.281		READY	0.758	0	0	0	0	11
| Task_C	0.742	0.742	PEND DELAY	4.039	0	0	0	1	11
Shell	1.682	4.963	PEND	4.781	0	0	0	0	10
timer	0.054	0.054	DELAY	6.464	0	0	0	0	10

Table 7.2: The new system monitor log.

monitor has been stopped.

7.4 Software probe class

A software probe class was used to log values with very low overhead. The probe class store a string and timestamp. It can store a values as well, integer or float. The class has been modified to return the timestamp it put on the event. That timestamp is used in system monitor to get the current time, a probe call is conducted in the system_monitor_switch_hook function. This has a nice side effect: the task-switch is recorded in the probe-log.

To start the logging of software probes the init-function must be called somewhere in the code. The number of probes to log is fixed, a constant is defined in a header file and is currently set log 100 000 events. The call to the init-function is placed next to where the system monitors start-flag is set.

7.5 Selecting probe points

Probes were placed in the tasks of interest. This introduced a probe-effect, a call to the software probe class takes a few microseconds (below 30 s). That is a very small compared to the execution-times of the tasks that are from 500 s up to about 50 ms, so it should not affect the result noticeably.

The location of the probe points is of course very important to get right. They were placed before and after each receive-operation and before each send-operation. An probe was placed in the system_monitor_switch_hook-function to log context-switches. This selection of probe points results in a very detailed log from the probes. Various statistics suitable for setting up the model can be extracted from that log with the tool ART-ML Builder.

A software probe that is located immediately before a receive-operation result in a receive-event, the probe immediately after a receive-operation creates an activate-event and a probe before a send records a send-event. An activate-event marks the start of a task instance and a receive-event marks the end of the instance. With this data it is easy to extract the distributions of execution times and where in the task execution the events occur (i.e. the time between the message-passing events)

Chapter 8

Tools

This thesis resulted in a set of tools that enables creation, simulation and analysis of a model describing a real-time system.

The simulator imitates the behavior of the modelled system and produces an output of the same type as the monitoring function produce when running the real system. This way both the output from the real system and the simulator can be analysed using the same tool, the ART-ML Builder.

The output contains data about all instances of the tasks in the model. The data is for example execution time, time of start and time of finish. It is also possible to see when and why an instance has been preempted.

The model that the simulator uses as input is defined in a modelling language developed for the purpose, ART-ML, Architecture and Real Time behavior Modelling Language.

A model defined in ART-ML needs to be compiled before it can be read by the simulator, as the simulator is a virtual machine, executing assembly-like instructions. The ART-ML Compiler was developed for that purpose.

To simplify the creation of the model, a tool called ART-ML Builder has been developed. It analyzes the output from the system monitor or simulator and creates excel-charts visualizing the data.

8.1 Compiler

The ART-ML Compiler was created with the sole purpose of speeding up the simulator. Two possible ways of implementing a simulator was identified. One virtual machine-based solution and one solution based on runtime parsing. The latter simulator would have to evaluate expressions (such as $a = 8 * (b + 7)/2$) during runtime and spend a lot of time comparing strings, while the virtual-machine solution would have a compiler that broke down expressions into simple instructions. The virtual-machine approach was selected due to the superior performance.

An instruction set for the virtual machine was developed, containing a subset of the Motorola 68000-instructions [8] and a handful of special instructions that performed operating system services such as message passing and semaphore operations.

The tools Lex (Flex) [15] and Yacc (Bison) [6] were used to develop the compiler. Flex generates code for a scanner, i.e. a tool that transforms the input file into a stream of binary codes, and Bison generates code for the parser, i.e. the tool that checks the grammatical and syntactical correctness of the input. A back-end was also developed. It generates the assembly-code for the simulator and is tightly coupled with the parser. The compiler has a symbol-table, so it can check for errors related to undeclared variables and type-mismatch errors. If an error is found in a model, the type and location is reported.

The output of the compiler is a `.cml-file`, Appendix A p.43. It contains two sections. First the symbol-table, containing name, type and value of all identifiers in the model. An identifier is a word that is not a reserved word, so the symbol-table contains the names of every named thing in the system, such as variables, tasks, mailboxes etc. After the symbol table is the code-section. It contains the code for the tasks. The code of two subsequent tasks are separated with the name of the following task (`TASK X` where `X` is the name of the task).

8.2 Simulator

The simulator takes the symbol-table from the compiler as input and builds a system. The system is built up by nodes (C-structs) see figure 8.1. All nodes except System-node has a next member that links to the next node of same type. This to make it easy to link in new nodes, search and delete. All nodes in the system gets a unique id, a number greater than 31 cause assembler instructions have numbers between 0 and 31, the symbol-table have the identifier as characters but in order to speed up the simulator the use of number identifier were chosen instead. It takes considerably longer time to compare two strings than it does to compare two integers. In order to translate the string to a unique id a struct called sysName is used, it copies the string to the sysName database and assign it a numeric, unique id.

The system is built as a tree, with a System-struct as the top-node. The symbol-table has three fields for describing a symbol, name, type and value. The name is unique, if a processor, p1, has a variable, var1, the variables name is p1.var1, see Appendix A p. 43. Depending on what kind of symbol is read from the symbol-table the system builder makes a new struct and creates all necessary linking to other structs. It all result in a structure like the one in the figure 8.2. The top node has a semaphore, mailbox and a processor. The mailbox has two variables (mailboxes always have two variables) and the processor has tree tasks and two variables.

After the system is built the handle of assembler-code starts. Assembler-code is located after the system in the symbol-table, see Appendix A p. 43. A string containing TASK;nameOfTask mark the start of instructions belonging to that task. Once found, all other instructions below belongs to same task until next start. The first instruction after a start is linked to the task's member "code" and the rest will follow the last instruction in a linked list of assembler-struct type. A set of instructions belonging to a task holds together by the "nextassm" member. The last instruction will point to NULL. This lets the

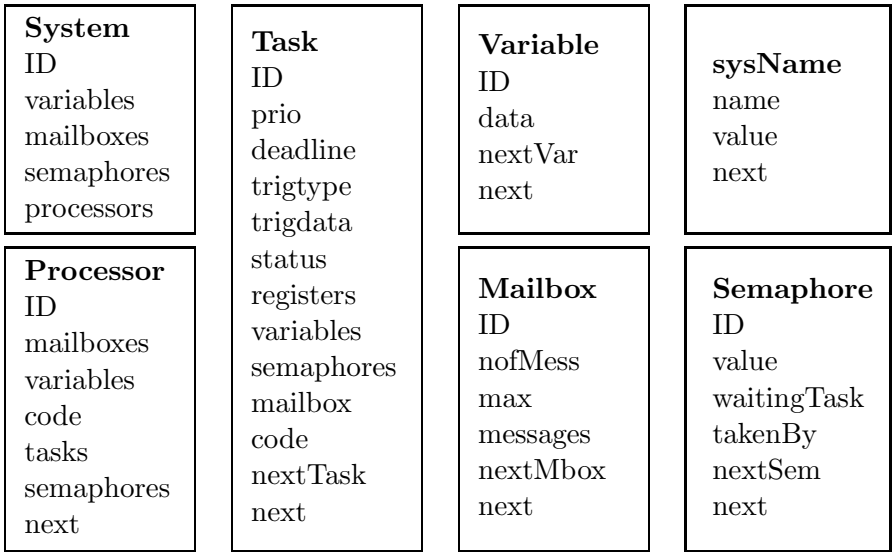


Figure 8.1: The structs forming the system.

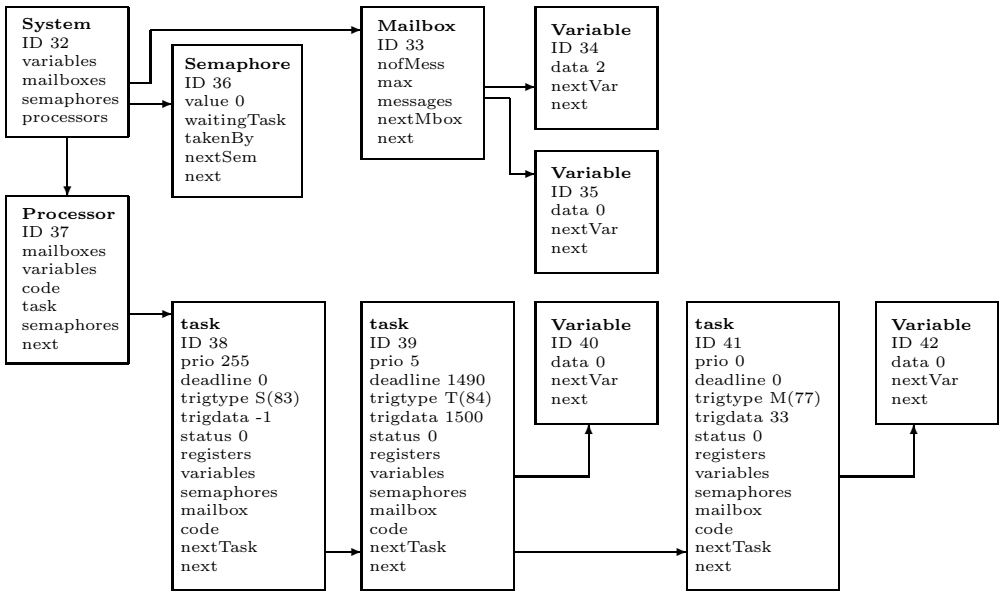


Figure 8.2: A fictitious system.

Time (ms)	Exec-time (ms)
0.018881029	4.176659000
0.171290936	12.814769000
0.435867897	11.962947000
0.460164241	0.371965000

Table 8.1: Execution-time of Task C

Min	Max	Average	n	n/N
0.000287265	0.000420876	(0.000360097)	131	(0.615023)
0.000577448	0.000604320	(0.000590884)	2	(0.009390)
0.004176659			1	(0.004695)
0.004797058	0.005024122	(0.004911885)	12	(0.056338)
0.005177941	0.006829881	(0.005829924)	65	(0.305164)
0.011962947			1	(0.004695)
0.012814769			1	(0.004695)

Table 8.2: Statistical distribution of task Task C

simulator know where the instruction list for the task ends. The assembler-struct has a next-pointer that holds all instructions together regardless of what task they belong to. All the branch instructions have special cases because they have two destinations, one is the next instruction in line and the other is the branch address. The labels that are inserted in the assembler-code are stored in a label-struct. All labels are unique. and if a branch instruction points to a label, the branch instruction inserts the pointer that the label points towards, see Appendix A p. 48.

8.3 Log Compiler

The Log Compiler extracts and compiles data from the system monitor and software probe logs. It compute how long time the task has executed and at what time it occurred, see table 8.1. It will also compute a statistical distribution for each task, see table 8.2. The output from Log Compiler should be used to build the model of the motion controller.

8.4 ART-ML Builder

The ART-ML Builder is a graphical interface for a tool collection. ART-MLs main purpose is to visualize and simplify the work for the user. All the tools mentioned earlier are included in the ART-ML Builder graphical interface. The interface is intended to start the tools with the correct parameters. ART-ML Builder can use Microsoft Excel to make diagrams of the distributions. ART-ML Builder will then open an Excel application and create a workbook, make

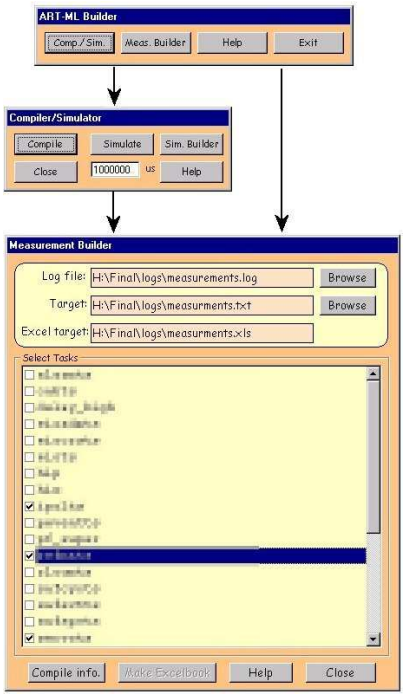


Figure 8.3: The ART-ML Builder application

one spreadsheet for execution time, response time and block time[1]. In each of every worksheet the tasks will be written in columns and a diagram made automatically for each task. See Appendix C p. 53 for more info about how to use ART-ML Builder.

8.5 Verification of the tools

The tools needs to be very carefully tested, since the purpose of the tools is to verify the temporal behavior of a real-time system and the validity of the results depends not only on the accuracy of the model but also on the quality (lack of bugs) of the tools. The compiler, simulator and ART-ML Builder are quite complex programs and very large amounts of output are produced, at least from ART-ML Builder and the simulator. It is very possible that a small bug in any of the tools that has been developed could affect the result without anyone noticing it, if the effect is small. Ensuring the non-existence of such bugs is not trivial and there is absolutely no guarantee that these tools are flawless. That is however a problem for every software developer, there is no methods available that can proof the non-existence of bugs in a program. That is however not in the scope of this thesis.

[1]Execution time is the actual time a task has been executing, response time is the time from task activation to task completion and block time is the difference between response time and execution time,i.e. the sum of the interference from higher prioritized tasks and the time spent waiting for resources.

The tools are of beta-quality but it is doubtful that any major undocumented bug exists. There are some documented bugs. They are listed in Appendix B p. 51 and in the readme-files for the corresponding tools.

8.5.1 Compiler

A large amount of the code in the compiler is machine-generated by the tools Flex and Bison. This have simplified the implementation of the compiler and thereby hopefully improved the quality of the code, since a large part of its behavior is specified in a higher level of abstraction. A mutant-based approach was used [19] to test the compiler. A small program was created to enable this approach. It was called the "error-generator" and was used to create alternated copies of a model-file, i.e. it copied the file and made small alterations. Such an altered version of the original is called a mutant. Not every alternation of the original file is a fault; about 10-15% of the potential mutants was not erroneous. Sets of 100 potential mutants where run through the compiler. If the compiler found any errors, the potential mutant was marked as a mutant (i.e. killed). The error type was inserted at the location of the error and the mutant was moved to a certain directory. If the compiler didn't find any errors, the potential mutant was moved to another directory, intended for potential live mutants. After such a batch-job, there were two directories to inspect. One with 80-90 files marked as faulty and another with 10-20 that approved by the compiler. All 100 files had to be inspected to see if the compiler had made any errors. This took some time, but it was an effective way of finding errors; a lot of bugs were discovered.

As expected, not every error was found. Some bugs in the compiler were found when debugging the simulator as well. This was at least partially due to the fact that the error-generator was a very simple program, it only changed single characters at random locations. This creates errors like removing keywords and references to undeclared symbols, but it doesn't swap order of things or adds extra words. With a more advanced error-generator that could do that, more bugs would have been discovered earlier. The compiler has been used a lot when creating the model and debugging the simulator, so it is not likely that there are any major undocumented bugs left.

8.5.2 Simulator

A set of ART-ML models was developed, each model focusing on certain functionality in the simulator. This had two purposes. Firstly, the models serve as good examples on the language ART-ML. Secondly; they were used as test cases. The simulator was tested with the test cases with relatively short simulation times and the result examined. Some of the models were simulated with longer simulations times, up to 10 minutes. A 10-minute simulation of the model took about 6 minutes to complete on a Pentium II based PC at 400 MHz and produces hundreds of megabyte of output.

A line-by-line verification of such an output is not possible unless highly specialised tools are developed for the purpose. Instead, a statistical verification

method was used. The simulator CPU utilisation was calculated for the models and compared with the value reported by the simulator. They matched exactly. That is not a proof of correct behavior, but it is an indication of it. Statistics on average/minimum/maximum execution time and the number of messages sent per instance where extracted from the output-log, they also matched the model. That was another indication of a correct implemented simulator. The simulator has been throughfully tested, it is not likely that it has any undocumented bugs left.

8.5.3 Log Compiler

The testing of the Log Compiler was not very difficult, as the program has a much larger testability and much smaller state space compared to the simulator and compiler. The testability is a measure of how hard it is to find the bugs in a program. A program with a lot of selections has lower testability than a program with a single path. [9] It basically reads a log-file into a data structure and performs some statistical calculations like in table 8.2.

Since the output format is the same for both the system monitor and the simulator, we could test it using logs from the system monitor and thereby eliminate the simulator as source of any errors. The system monitor actually produces two output files, the log and a file with statistics. The latter file contains information about average execution time, the total CPU-time of the tasks and so on. The Log Compiler produces that information based on the log, since the simulator does not produce any statistics-file. The statistics from system monitor was compared with the one calculated by the ART-ML Builder, the result matched.

8.5.4 ART-ML Builder

The ART-ML Builder was also easy to test because its main function is to start other programs. The diagrams could be tested by compare the diagrams generated by ART-ML with the one's manually done with Excel.

Chapter 9

The Model and the Measurements of the ABB Robotics Control System

In this work a model of the robot control system has been created. The purpose of this was to enable analysis of how changes in the system effects the temporal behavior of system. This impact analysis is conducted using a simulator developed for the purpose to simulate the behavior of the modelled system. A method has been developed for using this method of impact analysis in the development process. This impact analysis can be used to test the feasibility of adding/changing functionality in the system prior to actual implementation. This in return will result in a more effective development of the system since less time will be spent on debugging.

The system version that was used in the work was 4.0.53. The work has focused on the motor reference generation subsystem (subject-area MOC).

To create the model a notation for describing the system was necessary. The ART-ML modelling language was developed for this purpose. The language meets the requirements specified in 5.1 and it is developed to be able to describe a general system, which makes the language suitable for describing future development of the robot controller.

The ART-ML modelling language enables different abstraction levels in different parts of the described system. In this model, the three tasks of interrest are modelled on a lower level of abstraction, i.e. more detailed, and the rest of the system is described on a very high abstraction level.

In order to populate the model with data, methods and tools for measuring and compiling data has been developed. The system monitor was an existing tool that recorded the tasks execution times and a single message queue. New functionality has been added; a new method of starting the monitor, the option to monitor multiple IPC-queues, option to monitor the (non-standard) mstep-queue.

Besides the system monitor, the code of the tasks have been instrumented with probes. This allows measurements of when different key events occurs, relative to activation of the tasks.

The measurements of execution times produce megabytes of data. A tool was therefore developed in order to compile that data into execution time probability distributions. The output of the Modelbuilder tool is equivalence classes of the task instances, based on the execution time and the number of sent messages. Each class is labeled with a probability of occurrence based on the number of times it has occurred in the log. The tool can also export the raw data into Microsoft Excel sheets with scatter-diagrams showing the execution times per task of the recorded task instances. In these diagrams it is easy to see the equivalence classes as well as dependencies between the tasks different execution times.

The simulator that was implemented was able to simulate the model in about 167% of real-time speed, simulating 10 seconds of execution takes about 6 seconds on a Pentium II PC at 400 MHz. The output from the simulator was compared with the measurements and showed strong similarities. It was not a perfect match but very good, especially since the actual system containing 2.500.000 lines of code and the model only 200.

Since the simulator produce output of the same type as the logs from the system monitor, it is easy to compare the model with the actual system. By using the Modelbuilder tool, both the system monitor log and the simulator log can be exported and compiled into Microsoft Excel-diagrams and can thereby be compared visually. This provides a feedback when constructing or changing the model.

The ART-ML model that has been constructed contains 7 tasks, figure 9.1, the three tasks of interest, an idle-task and three dummy tasks. Two of the dummy tasks, *others_high* and *others_low* represents the other tasks in the system on a very high abstraction level. The third dummy task represents the external computer, it sends requests to Task A during the simulation and demands a response within a certain time.

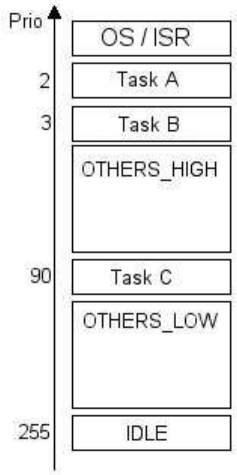


Figure 9.1: The priorities of the tasks in the ART-ML model

The others_high-task represents the tasks with priorities higher (lower in

VxWorks-terminology) than Task C but lower than the Task B. The other dummy-task, others_low, represents the tasks with priorities lower than Task C.

Since thiw work focus on the three tasks Task A, Task B and Task C, they are the only tasks in the model that have their behavior described in the model.

Task A is activated on message arrival on its mailbox. If the message is a mstep-request, it reads an mstep from the mstep-queue, executes for about 700 s and then sends an acknowledgement to the external-computer-dummy-task. If it is any other message, it executes for about 70 s.

Task B is activated by the simulation start. It contains two loops, the first one is entered immediately on startup. That loop waits for the variable *sys-online* to be set. That variable indicate that Task C has started and sent messages to Task B. When it is set, the second loop is entered. The task will stay in that loop until the end of the simulation. For each iteration, Task B reads all messages on its command queue and then sends a certain number of messages to Task A. That send-operation is blocking and most of the time the target queue will be full, so Task B spends most of it's time waiting for a message to be read by Task A.

Task C is activated by a certain probability, a 3% chance every microsecond i.e. about 3 activations per 100 s. When it's activated it sends a varying number of messages to the Task B. This is to emulate the execution of the robot control program.

The external-computer-dummy-task *EXT_DUMMY* is activated every X'th ms and if sys-online is set, it sends a request to Task A. An acknowledgement is expected from Task A, if the reply does not arrive within X ms, the task hits its deadline. If the acknowledgement arrives within deadline, a command-type message is sent to Task A with a certain probability.

A deadline-miss causes the task to be killed and a warning-message to be displayed. Killing the task is not desirable in all situations, so this could be improved by adding options for it in the simulators .cfg file, but since missing the deadline is considered a failure in this case, such option is of limited use.

The simulator presents the average cpu-utilization in the console-window when the simulation is complete.

When measuring the system in order to populate the model, we used the trig command to start the system monitor on a certain system event, in this case when the next movement starts. In the Task C class, a line has been inserted which sets the system monitor start-flag. It is executed if a message arriving indicating the start of a new movement. If the trig command has been started within 30 seconds before that event, it will capture the change in value of the flag and start the snooper.

Chapter 10

Conclusions and future work

In this work a set of tools and a modelling language has been developed. These tools enable the construction of a model of the robot control system. The model has been populated with data measured from the system and has been run through a simulator. The results from that simulation are easily comparable with the results from the measurements.

The approach of this work was to enable the temporal analysis of robot controller by enable the simulation of a model. This required the development of a modelling language and a simulator, since no existing solutions were found. A compiler for the modelling language was necessary, since the simulator was implemented as a virtual machine, executing low level instructions. A tool for extraction and compilation of statistics was developed as well. The model was implemented by describing the robot control system in the developed notation (ART-ML) and inserting data measured from the control system.

Though there are dependencies between the tasks left to include in the model we were able to describe the complex system of 2,5 million LOC with a model consisting of only 200.

This method of simulation-based timing analysis is very general and flexible. The modelling language is capable of describing many classes of systems and the simulator is fast, enabling simulations of complex models. Simulation is frequently used in many industrial projects [11], the developer at ABB Robotics will have no problem to use these simulation-based tools for further developments of the system.

It is easy to extend the different kinds of analyzes that is conducted on the measured data. Implementation of the new methods in the Log Compiler is necessary, but a platform intended for analysis-functions exists, which simplify that implementation.

Future work

- Visualisation

  Create a graphical user interface for the simulator where the output is presented (something like the presentation tool in STRESS)

- Integration

All tools integrated in a graphical user interface. Would lower the threshold for new users and make it easier and faster to use. This exists to some extent today, but not very well implemented. An installation program for the tools would be nice as well, since there are several different programs that depend on each other.

- Extending the simulator
  - Support for distributed systems
    The simulator should be able to simulate a model containing multiple processors, it must be possible to have different temporal behavior for message passing between processors compared to message passing within the same processor.
  - Hidden tasks/processes
    Introducing support for hidden processes within the simulation would improve the capabilities of the simulator. A hidden process executes as a task but is not displayed in simulation, and it must not consume time. Using hidden processes it would be possible to simulate the behavior of external processes and implementing certain system services within the model.
  - Access of task attributes within task-behavior
    If the code describing task behavior can read and write tasks attributes, most importantly task priority, schedulers can be implemented. If support for hidden tasks exists, such a scheduler can be implemented in the model as hidden tasks.
  - Include-functionality
    If system services exist within the model, implemented as hidden processes, it would be desirable to isolate them from the tasks that are visible in the simulation. An include-statement could be used to merge several model-files just before compilation and thereby allow libraries of system services and algorithms to be implemented.
  - Function-calls There is currently no support for function calls in ART-ML. Introducing such support would enable the construction of more complex models, especially in combination with include-functionality.
  - The Execute-statement
    * Only one pair
      If an execute only contains a single 100%-chance, the percentage could be left out. (i.e. execute (750) instead of execute((100,750)))
    * Recursive definition
      To describe more complex distributions, beeing able to write nested execute-statements would simplify things. For example, execute((60,200), (40,((90,500),(10,700)))) instead of execute((60,200), (36,500), (4,700))

    * Linear distributions
      Another thing that would simplify the creation of the model would be to support linear distributions as well. This can be combined with the ordinary distributions. For example:
      execute(100-200) or
      execute((95,500-600),(5,750-760)) or
      execute((90,100-200),(10,((50,225),(40,230-240),(10,270-280))))

- Monitoring of simulation

  Today, the simulator outputs the complete trace of the entire simulation, there is no support for searching for extremes in the behavior as for instance in DRTSS [17]. A nice solution would be to have a language to specify "monitors" and "triggers" for a simulation and perhaps a graphical user interface for that language. A monitor logs something continuously and generates statistics on that certain property, while a trigger takes some action on a certain condition, perhaps show a warning or simply count the "instances" of that condition.

- Refine the model

  - Measurements of delay on send

    All kernel calls executes in zero time as the execution times of these calls are supposed to be included in the execute-statements. The simulator delays the messages that are sent, currently by 100 s. This is to model the time that it takes for the real system to execute the code between the probe and when the message is actually available for other tasks to receive. That delay is not based on any measurement, the 100 s is just an estimation. This should be measured.

  - More tools for analysis of the result

    A platform for this already exists. Adding more functionality to compile statistics would ease the construction of the model.

  - Define more probe-points

    Cover more of the system behavior. Only a few (the most important) paths through the tasks are instrumented with probes. Adding probes to more paths would enable the construction of a more accurate model, in terms of capturing more system states.

  - A more detailed behavior-model

    The model of the tasks' behavior is rough, this could be improved. Preferably done by someone with a lot of domain knowledge.

References

[1] ABB-Robotics. Motion Control OBM. Internal documentation.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. Real-Time Systems Symposium, 1998.

[3] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs proportional share resource allocation. Multimedia Computing and Systems, 1999.

[4] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. STRESS: A Simulator for Hard Real-Time Systems. *Software-Practive and Experience*, 24(6):534,564, 1994.

[5] A. Casile, G. Buttazzo, G. Lamastra, and G. Lipari. Simulation and tracing of hybrid task sets on distributed systems. In *Real-Time Computing Systems and Applications*, 1998.

[6] Charles Donnelly and Richard Stallman. Bison. World Wide Web, `http://www.hrz.uni-giessen.de/hrz/htmldocs/gnuinfo/bison/bison_toc.html%`, 1999.

[7] Andreas Ehrencrona. Turing machines. World Wide Web, `http://cgi.student.nada.kth.se/cgi-bin/d95-aeh/get/umeng`, 2002.

[8] W. Ford and W. Topp. *Assembly Language and Systems Programming for the M68000 Family.* Jones and Bartlett Pub, 2nd edition, 1996.

[9] D. Hamlet and J. Voas. Faults on its sleeve: Amplifying Software Reliability Testing. In *International symposium on software testing and analysis*, 1993.

[10] M. Lindgren. *Measurement and Simulation Based Techniques for Real-Time Systems Analysis.* Lic. Thesis, Uppsala University, 2000.

[11] M. Lindgren, H. Hansson, C. Norström, and S. Punnekkat. Deriving Reliability Estimates of Distributed Real-Time Systems. In *Proceedings of RTCSA2000 (IEEE Computer Society).* Mälardalen Real-Time Research Centre, Mälardalen University, Sweden, 2000.

[12] M. Lindgren, H. Hansson, and H. Thane. Using Measurements to Derive the Worst-Case Execution Time. In *Proceedings of RTCSA 2000 (IEEE Computer Society)*, 2000.

[13] S. Manolache, P. Eles, and Z. Peng. Memory and Time-efficient Schedulability Analysis of Task Sets with Stochastic Execution Time. In *Proceedings of the 13nd Euromicro Conference on Real-Time Systems*. Department of Computer and Information Science, Linköping University, Sweden, 2001.

[14] C. Norström, K. Sandström, J. Mäki-Turja, H. Hansson, H. Thane, and J. Gustafsson. *Robusta Realtidssystem*. Mälardalen University, 2000. A swedish book on realtime systems.

[15] Vern Paxson. Flex - Fast lexical analyzer generator. World Wide Web, `http://www.uni-giessen.de/hrz/htmldocs/gnuinfo/flex/flex_toc.html`, 1995.

[16] Tri-Pacific Software, Inc. RAPID Sim: High-Performance Simulation of Real-Time Systems. World Wide Web, `http://www.tripac.com/`, 2001.

[17] M.F. Storch and J.W.−S. Liu. DRTSS: a simulation framework for complex real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*. Dept. of Comput. Sci., Illinois Univ., Urbana, IL, USA, 1996.

[18] A. Wall. *A Formal Approach to Analysis of Software Architectures for Real-Time Systems*. Lic. Thesis, Uppsala University, 2000.

[19] H. Zhu et al. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, december 1997.

Appendix A

Output from the ART-ML compiler

```
Compiled model out.cml
#SYMBOLS
NAME: mbox1
TYPE: Mailbox
VALUE:0

NAME: mbox1.maxsize
TYPE: Constant
VALUE:2

NAME: mbox1.size
TYPE: Variable
VALUE:0

NAME: mutex
TYPE: Semaphore
VALUE:2

NAME: p1
TYPE: Processor
VALUE:-

NAME: p1.idle
TYPE: Task
VALUE:S;-1;255;0;0

NAME: p1.tt_task
TYPE: Task
VALUE:T;1500;5;1490;0

NAME: p1.tt_task.tmp
TYPE: Variable
VALUE:-

NAME: p1.mbox_task
TYPE: Task
VALUE:M;mbox1;0;0;20000

NAME: p1.mbox_task.v1
TYPE: Variable
VALUE:-

#CODE
TASK;p1.mbox_task
recv;p1.mbox_task.v1;mbox1;-1
stk;mutex;-1
submi;?R0;p1.mbox_task.v1;1
notm;?R1;?R0
cmpmi;?R1;0
beq;skip0
exec;90;100;10;105
bra;skip_else1
label;skip0
exec;90;200;10;205
label;skip_else1
sgv;mutex
TASK;p1.tt_task
exec;75;500;25;600
prbi;?R2;80
cmpmi;?R2;0
beq;skip2
sendi;mbox1;1;-1
bra;skip_else3
label;skip2
stk;mutex;-1
sendi;mbox1;3;-1
slpi;200
sgv;mutex
label;skip_else3
exec;50;120;50;150
TASK;p1.idle
label;while4
cmpii;1;1
bne;while_stop5
exec;100;10000
bra;while4
label;while_stop5
```

The ART-ML language

(TERMINALS, non-termninals)

system: SYSTEM header processor processorlist ENDSYS

processor: PROCESSOR ID header task tasklist ENDPROC

processorlist: empty | processor processorlist

header: empty
| VARIABLE ID ';' header
| MAILBOX ID NUM ';' header
| SEMAPHORE ID NUM ';' header
| CONST ID NUM ';' header

tasklist: empty | task tasklist

task: TASK ID taskheader BEHAVIOUR '{'
header stmntlist '}'

taskheader: trigger prio dl mem

prio: PRIORITY NUM
dl: empty | DEADLINE NUM
mem: empty | MEMORY NUM

trigger: TRIGGER whattrigger
whattrigger:STARTUP
| MAILBOX ID
| PROBABILITY NUM
| PERIOD NUM

stmntlist: empty | stmnt stmntlist

stmnt:lexpr ';'
| send ';'
| recv ';'
| semtake ';'
| semgive ';'
| sleep ';'
| consume ';'
| assignment ';'
| block
| for
| ifelse
| while
| chance

semtake: SEMTAKE '(' ID ')' semtake_timeout
semtake_timeout: empty | TIMEOUT NUM

semgive: SEMGIVE '(' ID ')'

sleep: SLEEP '(' lexpr ')'

ifelse: IF '(' lexpr ')' stmnt ifelse2

ifelse2: empty | ELSE stmnt

for: FOR '(' ID ',' expr ',' expr ')' stmnt

while: WHILE '(' lexpr ')' stmnt

assignment: ID '=' lexpr

block: '{' stmntlist '}'

chance: CHANCE '(' lexpr ')' stmnt chance2
chance2: empty | ELSE stmnt

consume: CONSUME '(' et_pair et_list ')'
et_list: empty | ',' et_pair et_list
et_pair: '(' expr ',' expr ')'

send: SEND '(' ID ',' lexpr ')' send_timeout
send_timeout: empty | TIMEOUT NUM

recv: RECV '(' ID ',' ID ')' recv_timeout
recv_timeout: empty | TIMEOUT NUM

bool_expr:rel_expr
| bool_expr AND bool_expr
| bool_expr OR bool_expr

rel_expr:neg_expr
| rel_expr '>' rel_expr
| rel_expr '<' rel_expr
| rel_expr GTEQ rel_expr
| rel_expr LTEQ rel_expr
| rel_expr EQ rel_expr
| rel_expr NEQ rel_expr

neg_expr:ari_expr
| '!' bool_expr

ari_expr:term
| term '+' ari_expr
| term '-' ari_expr

term:factor
| factor '*' term
| factor '/' term
| '-' term
factor:atom
| '(' bool_expr ')'

atom:NUM
| ID

Compiler output language

Flags: N (negative)
Z (zero)
V (overflow)
C (carry)

Basic Instructions

Code	Types	Name	Arguments	Semantics
0	4	ADD	$D, S1, S2$	$D = S1 + S2$
4	4	SUB	$D, S1, S2$	$D = S1 - S2$
8	4	MUL	$D, S1, S2$	$D = S1 * S2$
12	4	DIV	$D, S1, S2$	$D = S1/S2$
16	2 (m,i)	MOVE	D, S	$D = S$
18	2 (m,i)	NOT	D, S	$D = !S$
20	4	AND	$D, S1, S2$	$D = S1\&\&S2$
24	4	OR	$D, S1, S2$	$D = S1\|\|S2$
28	4	CMP	$S1, S2$	$Flags = S1 - S2$
32	7	Bxx	*Label*	PC = address of instruction following the label (if flags match)
32		BGE		Not N ($S1 >= S2$)
33		BGT		Not (N OR Z) ($S1 > S2$)
34		BLE		N OR Z ($S1 <= S2$)
35		BLT		N ($S1 < S2$)
36		BNE		Not Z ($S1! = S2$)
37		BRA		Branch always
38		BEQ		Z ($S1 == S2$)
39	1	ISP	D, S	$D = (S >= 0)$
40	1	ISN	D, S	$D = (S < 0)$

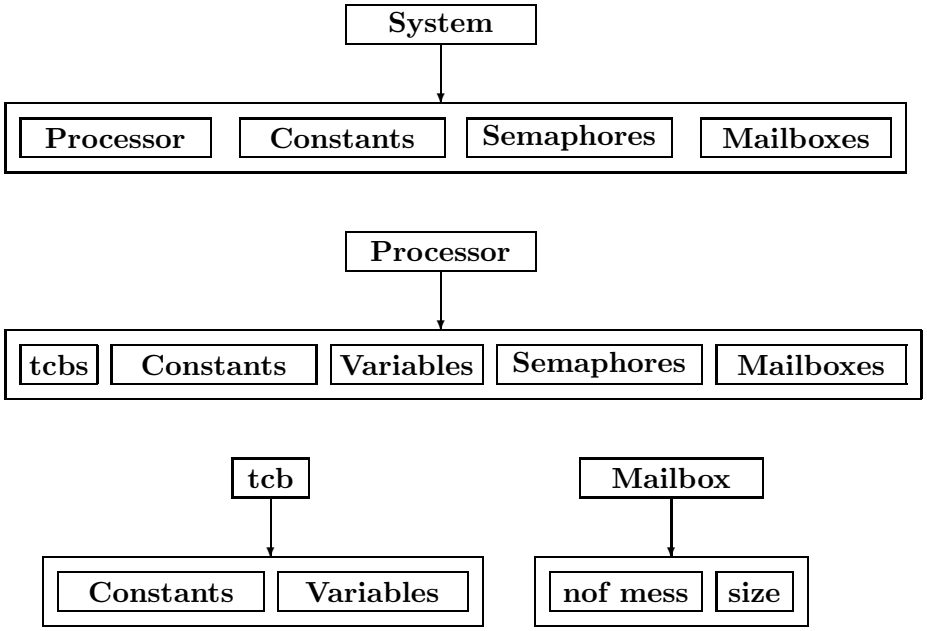
Special Instructions

Code	Types	Name	Arguments	Semantics
41	1	STK	ID	Take semaphore ID
42	1	SGV	ID	Release semaphore ID
43	2 (m,i)	SLP	S	Sleep for S timeunits
45	2 (m,i)	PRB	D, S	If (abs(rand() % 100)) < S) D = 1; else D = 0;
47	2 (m,i)	SEND	ID, S	Send S to mailbox ID
49	1	RECV	D, ID	Recieve from mailbox ID, put value in D
50	1	EXEC	P1, T1, P2, T2... Pn, Tn	Consume time according to the distribution decribed...

Subtypes

Code	Name	Semantics
0	MM	mem[D] = mem[S1] op mem[S2]
1	MI	mem[D] = mem[S1] op S2
2	IM	mem[D] = S1 op mem[S2]
3	II	mem[D] = S1 op S2

Symbol membership



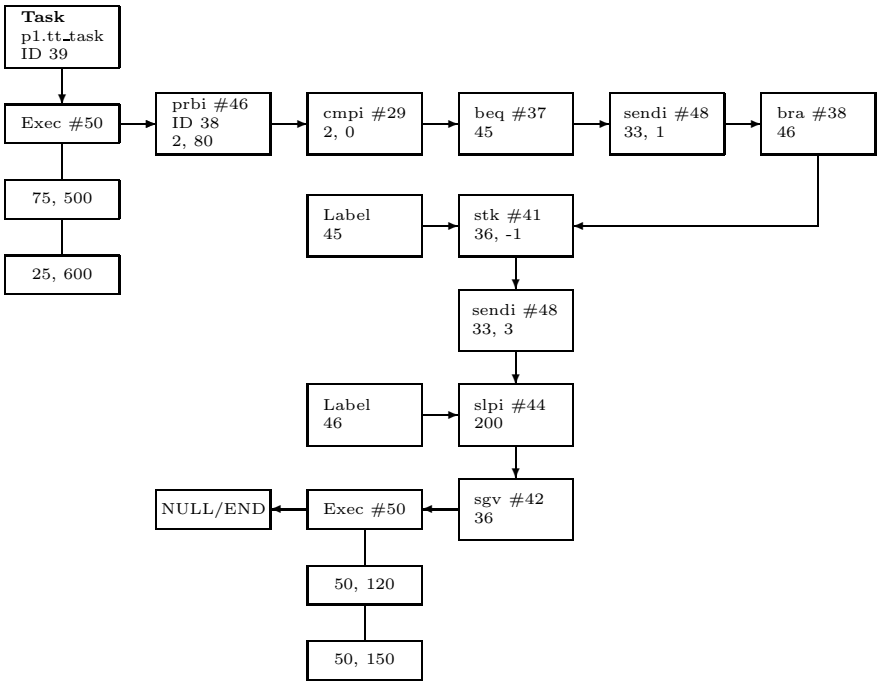
Task example

```
system
mailbox mbox1 2;
semaphore mutex 2;

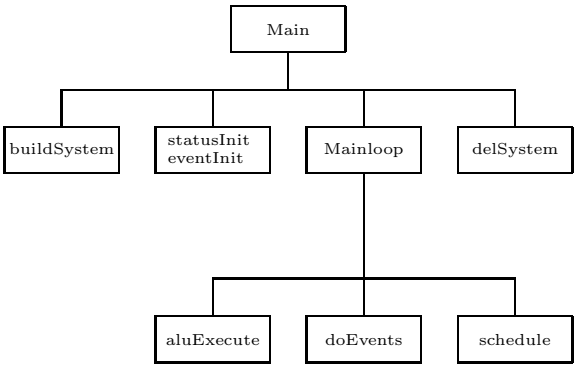
processor p1
    task mbox_task
        .
        .
        .

    task tt_task
        trigger period 1500
        priority 5
        deadline 1490
    behaviour{
        variable tmp; //not used
        execute((75,500),(25,600));
        chance(80){
            send(mbox1,1);
        }else{
            // 20% chance of execution
            semtake(mutex); //take semaphore
            send(mbox1,3);
            sleep(200);
            semgive(mutex);
        }
        execute((50,120),(50,150);
    }

    task idle
        .
        .
        .
```



Simulator - Internal structure



Appendix B - Known Bugs

Bugs in the compiler

The problem with the execute-statement

Description

If more than four pairs of probability/execution-time are specified, the output gets corrupted.

Why this happens

Not investigated.

Proposed solution

Fix the bug or avoid specifying more than four pairs in an execute-statement. An execute-statement containing more than four pairs can easly be tranformed into multiple execute statements, holding at most four pairs each. These execute statements must be grouped within a chance-statement, so only one statement is executed.

The compiler crash

Description

The compiler crashes if the first error is a reference to the size-member of an undeclared mailbox.

Why this happens

Somewhere in the symbol-table functions, the result from a function-call is not compared with NULL before using it.

Proposed solution

Know about the crash and it's cause when compiling or fix the bug.

Bugs in the simulator

The priority problem on send

Description

If multiple task are waiting to send to a mailbox (because it is full) and a another task performs a receive-operation, the wrong task is activated and allowed to send.

Why this happens

A mailbox has two wait-queues, one for tasks that are waiting to receive a message and one for tasks that are waiting to send a message. When a task receives a message from a mailbox, the simulator checks the "waiting-to-send"-queue of the mailbox in order to activate any waiting task.

Today, that is a FIFO-queue, i.e. first-in-first-out, so the first task that wanted to send is the first to be activated when there is room in the mailbox.

Proposed solution

The function "get_from_mbox_send_waitqueue" in kernel.cpp is responsible for selecting the task in the wait-queue that shall be activated. It should select the task with highest priority instead of the first in list. Another solution is to set all tasks in the wait-queue to READY and clear the queue. It is then up to the scheduler to decide what task that shall run.

Bugs in ART-ML Builder

ART-ML Builder crash when making Excel-book

Description

ART-ML Builder just crash.

Why this happens

Excel have probably a book with the same name is opened and locked.

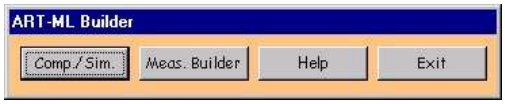
Proposed solution

Simply close all Excel programs.

Appendix C

ART-ML Builder

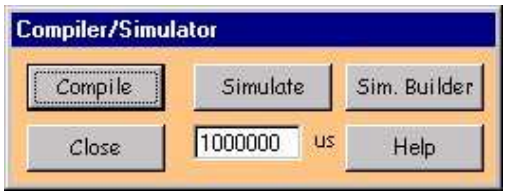
The ART-ML Builder is a tool suite. Its main purpose is to visualise and simplify the work for the user. The tool suite contains the scan-, log_comp-, compiler- and simulator-executable. ART-ML Builder can make Excel-sheets from extracted data.



How to use ART-ML Builder

ART-ML

To start extract data from system monitor-log click on the "Meas. Builder"-button (Measurement Builder). Continue to the "Usage of Measurement Builder/Simulator Builder" section.
To start compile/simulate your model Click on the "Comp./Sim."-button (Compiler/Simulator). Continue to "Usage of the Compiler and simulator".
To get this help press the "Help"-button at any time.
To quit ART-ML press the "Exit"-button.



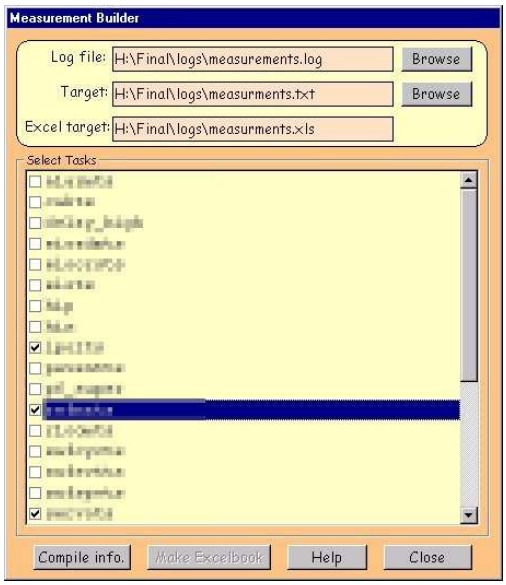
Usage of the Compiler and simulator

If a new model has been created it needs to be compiled with the ART-ML Compiler. Click on the "Compile"-button. Select the model-file to compile and click the "OK"-button. Select a destination filename for the compiled model (the file is used in the simulation of the system), then press the "OK"-button. The ART-ML Compiler will start the compilation of the model.

The ART-ML Compiler will alert if any error occurs.
Once a valid system is compiled you can start the simulation of the model. Select the time (in us) you want to simulate the system. Click on the "Simulate"-button and select the compiled-file created by the ART-ML Compiler and a destination filename for the log-file. The log-file is now ready to be processed by the Simulator Builder, just press the "Sim. Builder"-button.

Usage of Measurement Builder/Simulator Builder

Click on the "Browse"-button to select the log file to use. Then click on "Browse"-button beside the Target-field to select the file to write data to. All tasks from the log file will appear in the Select Tasks-window. Select the one to include in the data file. The content of the output file is the distributions of the selected tasks execution-time, response-time and blocked-time. ART-ML Builder will automatically suggest a name for the Excel-sheet. You can change the name if you like. Just write in the name of the file in the Excel target-field.
Click on the "Compile info."-button to make the output file.
Click on "Make Excelbook"-button to make diagrams of the data file.
The "Help"-button will show this help-file
The "Close"-button will close the Measurement Builder/Simulator Builder.



Note

- In the ART-ML Builder program directory you will find a config file for ART-ML Builder (init.src). It contains paths to executables etc.
- The ART-ML Builder is written in Visual Basic (VB) and VB doesn't like spaces in the directory names so avoid it.

Note to developers

Application notes

- When making Excel-sheets it's important that log_comp makes two newlines after each set of execution-time, response-time and blocked-time. Otherwise it will not work properly.
- Do not put anything else than the line "Statistics" after the blocked-time set. ART-ML Builder stops after finding that line.

Known bugs

- If Excel have a sheet opened with the same name as ART-ML Builder tries to create it will crash.

  Solution: Simply close all Excel programs.

ART-ML Compiler - Users guide

Developed as a part of the Master Thesis "Timing Analysis of a Robot Controller" at ABB Robotics / Mälardalens Högskola

By Johan Andersson and Jonas Neander

This short document describes how to use the ART-ML Compiler (`artmlc.exe`).

The ART-ML Compiler is program that lets you check if an ART-ML-file has correct syntax. When called with an ART-ML-file as argument, the compiler will point out any grammar or symbol related error it might find. If the compiler doesn't generate any error, an output-file is generated with the name specified. The output-file contains a low level description of the system that the ART-ML Simulator can read.

See p. 57 for a detailed description of the compiler.

Usage:

```
artmlc [-debug] [-debug N] [-serie N] path outfile
```

```
artmlc filename outfile
```

Parses the file and tells if it has any errors, and if so, what and where. Creates "output-file" with a binary description of the system.

```
artmlc -debug filename outfile
```

Additional information is presented

`artmlc -debug N filename outfile` (N is a positive integer)

The higher N, the more information is presented. Setting N to 1 is the same as not specifying any N at all.

`artmlc -serie N filename` (N is a non-negative integer)

This option is intended for debugging of the compiler. No output-file is produced since it's not the purpose of this operating mode. The idea is that the parser parses a considerable amount of test cases and copies each tested file to one of two directories named `test_failed` and `test_ok`. When all files has been parsed, one can manually check that all files in the directories have been correctly judged.

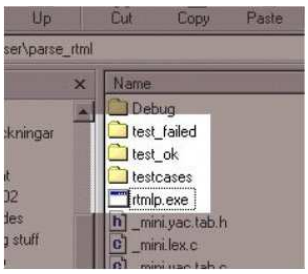
The compiler assumes that there exist three directories named `test_failed`, `test_ok` and `testcases`. It will parse N files located in the directory `testcases` with the name-convention *filenameX.txt* where *filename* is the name specified as the third argument and X an integer between 0 and $N-1$. (See 8.5)

It assumes that all files between 0 and $N-1$ exists. If, say file `qwerty33.txt` is missing when N is 50, the behavior is not defined...(It should stop the parsing...)

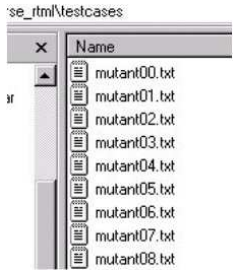
After each file is parsed, the file is copied into one of the two directories `test_ok` and `test_failed`, depending on the result of the parsing. The new files will have the same name as the input files except the files in `test_failed`. They will have an error code attached to their name to make them easier to sort.

Directory structure and name-conventions for the -serie option:

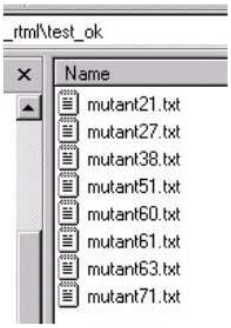
In this example the parser is executed by typing `artmlc -serie 75 mutant`.



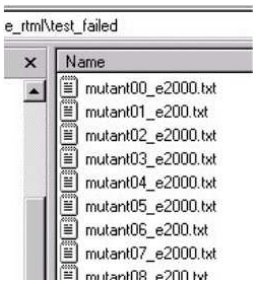
Directory Structure



Directory "testcases"



Directory "test_ok"



Directory "test_failed"

Technical information - artmlc

Capabilities

The compiler checks for two types of errors, grammar and symbol related. Grammar related errors are for example misspelled/missing keywords, missing separators, commas, semicolons and so on.

The grammar checking does not distinguish between different identifiers. If the scanner sees a word that is not a keyword, it is classified as an identifier. The symbol-checking checks that all identifiers that are discovered have been previously declared. A symbol related error is if a symbol is used before it's declared (or misspelled) or if a symbol is of the wrong type.

ART-ML

The grammar for the ART-ML-language can be found in the documents:

`artmlcompiler.yacc` (yacc/bison-code)
or
`artml_language.doc` (simplified description)

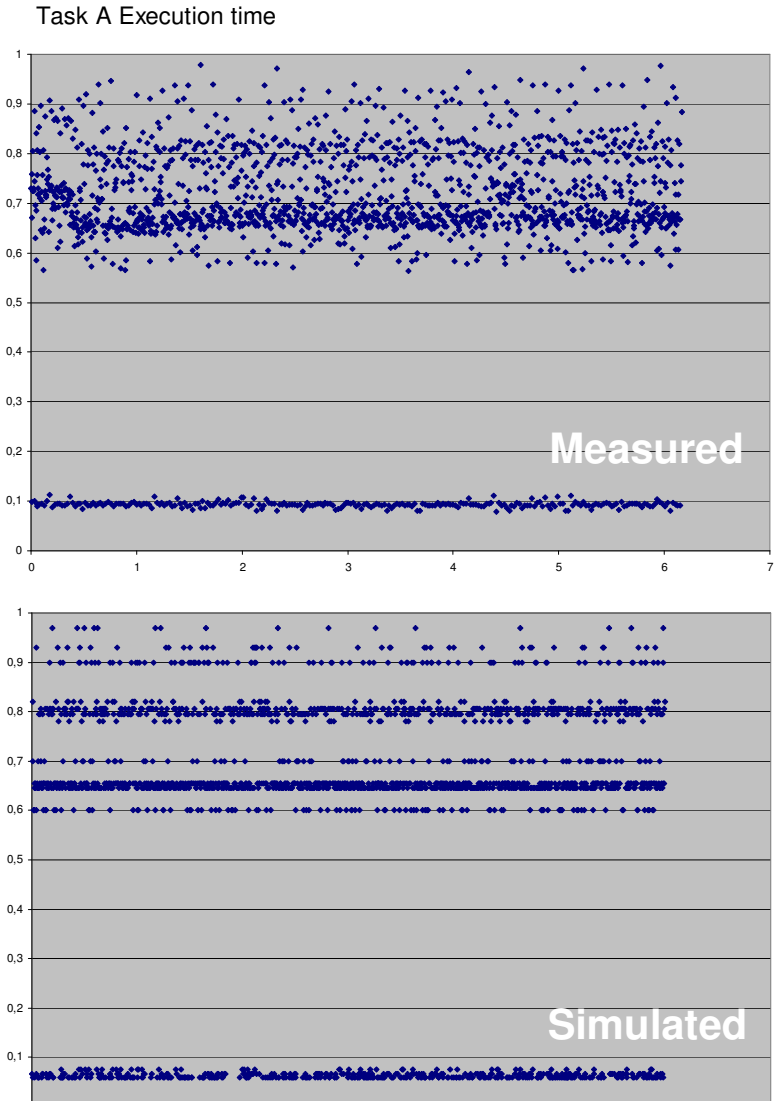
Examples of ART-ML can be found in the directory `artml_examples` in the parser directory.

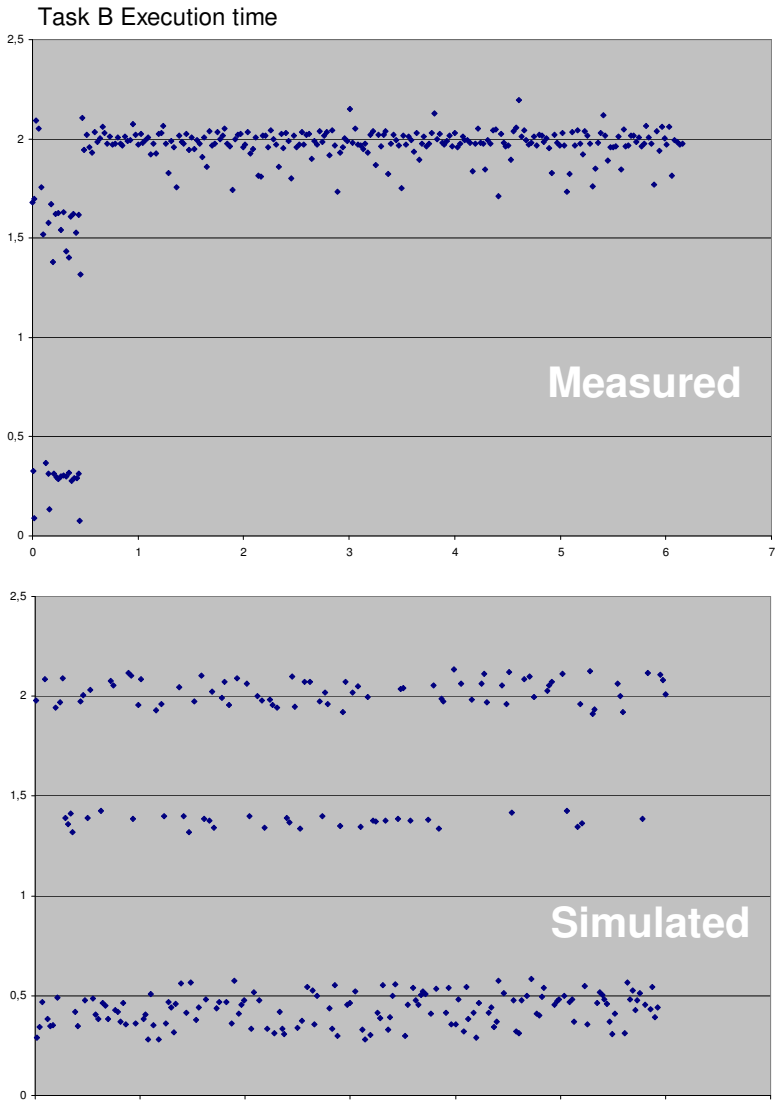
Implementation

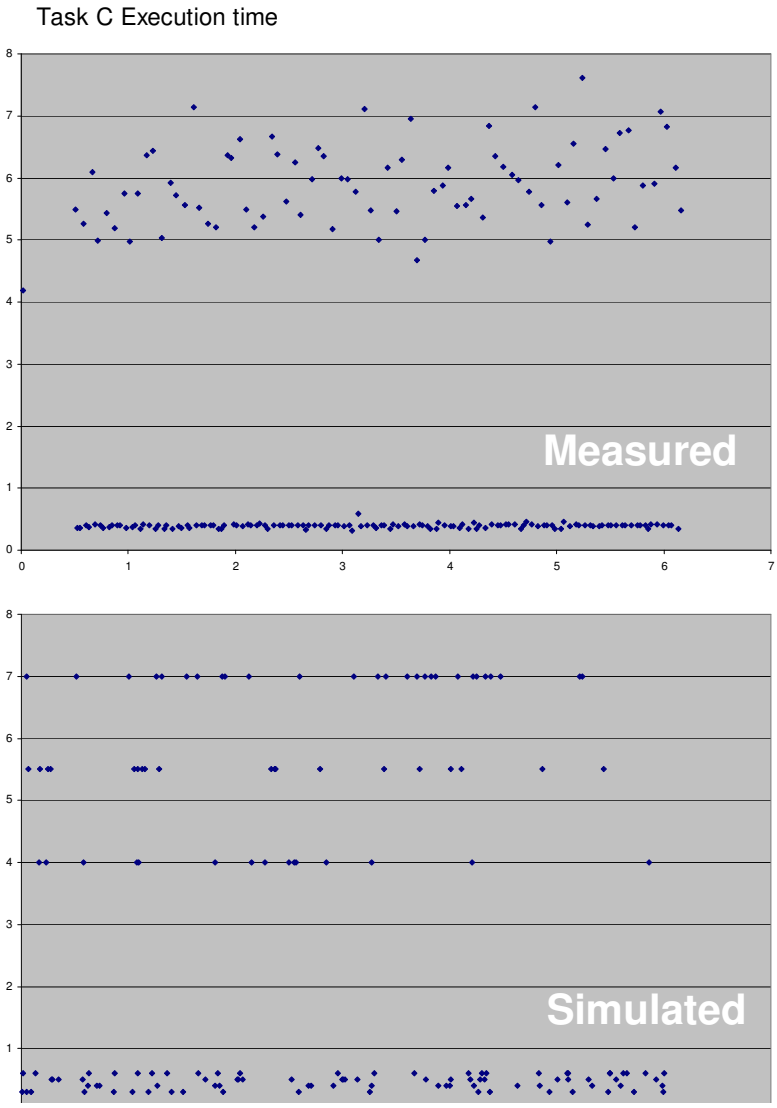
The ART-ML Compiler has been developed using the tools Flex [15] and Bison [6]. They are integrated in the Visual Studio Environment and generate C-code based on a grammar and a set of tokens/reserved words. Flex generates a scanner (the part that reads the source-file and generate binary tokens) and Bison creates a parser that checks if the stream of tokens match the rules that are set up. For more information about the tools see [15, 6].

Appendix D

Measurements and simulation




Task B Execution time
2,5
2
1,5
1
0,5
0
0
1
2
3
4
5
6
7
Measured
2,5
2
1,5
1
0,5
0
Simulated


Task C Execution time
Measured
Simulated