CONTRIBUTIONS TO THE DEVELOPMENT OF REAL-TIME PROGRAMMING TECHNIQUES AND TECHNOLOGIES
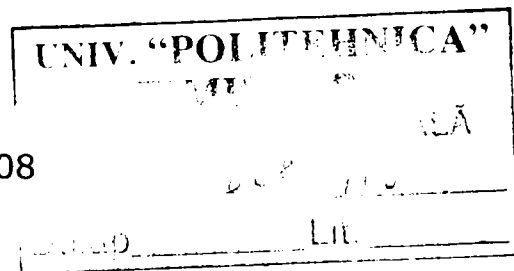
A thesis submitted for the degree of
Doctor of Philosophy *(PhD)* Engineer
University "Politehnica" of Timisoara
COMPUTER SCIENCE domain
by

Ing. Daniel Iercan

Scientific advisor: prof.univ.dr.ing. Nicolae Robu
Scientific referents: prof.univ.dr.ing. Ioan Dumitrache
                           prof.univ.dr.ing. Tiberiu Leţia
                           prof.univ.dr.ing. Toma-Leonida Dragomir

UNIV. "POLITEHNICA"

Day of dissertation: 27.09.2008

Word before

This PhD thesis is the result of my scientific work in the Department of Automation and Applied Informatics, Faculty of Automation and Computers at „Politehnica" University of Timisoara

In this thesis there have been analyzed new programming techniques and technologies for developing real-time applications and there have been presented examples of real-time control applications.

I would like to express my gratitude for all those that have offered their moral and scientific support and helped me finalize this thesis.

For the support and good advice I would like to thank to may scientific supervisor prof. dr. ing. Nicolae Robu. I would like to thank to prof. dr. ing. Christoph Kirsch for offering me the chance to work in an international research group.

I express my consideration for the members of the PhD committee: prof. dr. ing. Octavian Pro□tean, chair of the committee and dean of the Faculty of Automation and Computers at „Politehnica" University of Timisoara, prof. dr. ing. Ioan Dumitrache from University Politehnica of Bucharest, prof. dr. ing. Tiberiu Le□ia from Technical University of Cluj-Napoca, and prof. dr. ing. Toma-Leonida Dragomir from „Politehnica" University of Timisoara, for accepting to be in the committee and for the time they have spent in analyzing my thesis.

I would also like to thank to dr. ing. Arkadeb Ghosal for all the support he had offered me. I thank also to my colleges from B624, to Elza and Fedo for all their help.

Timisoara, September 2008   Daniel Iercan

To my wife and to my parents!

Summary,
In the last two decades there has been an accelerated growth in the complexity of real-time applications. This growth leads to the evolution of the real-time programming. Thus, if in early age of real-time programming both functionality and timing were expressed in the same programming language, nowadays real-time programming requires separation of timing from the functionality. Thus in the last few years there has been a great interest in developing new programming constructs for specifying temporal behavior of an application. The work in this thesis is based on Hierarchical Timing Language, which is a language for specifying timing and interactions between sets of periodic tasks. After HTL language is presented (Chapter 2), there are described two possible implementations of this language (Chapter 3 and Chapter 4). An experimental as well as an analytical comparison of the two implementations is presented in the last part of Chapter 4. In Chapter 5 it is described a real-time control application for the Three Tanks System implemented in HTL. Chapter 6 presents an implementation of HTL for Exotask; Exotask is a new programming construct that uses Java. The HTL implementation for Exotask has been tested by developing a real-time application for a quad-rotor helicopter, named JAviator. In Chapter 7 it is presented an implementation of HTL that targets a microcontroller, implementation has been tested through a control application for JAviator, a detailed analysis of the performance is also presented. In Chapter 8 it is discussed a solution for modeling an HTL description in Simulink. Thesis ends with a short summary of the obtained results.

Contents

List of Figures

List of Tables

Glossary

BET   Bounded Execution Time

E machine Embedded machine

EDF   Earliest Deadline First

GC   Garbage Collector

HLC   High-Level Control

HTL   Hierarchical Timing Language

JVM   Java Virtual Machine

LET   Logical Execution Time

LLC   Low-Level Control

PET   Physical Execution Time

RTGC   Real-Time Garbage Collection

RTSJ   The Real Time Specification for Java

SST   Super Simple Tasker

WCET   Worst-Case Execution Time

WCTT   Worst-Case Transmission Time

WRT   WebSphere Real Time

ZET   Zero Execution Time

1. Introduction

A real-time application is an application that processes information on-line (i.e., it reads the inputs directly from the source that produces them and writes the outputs directly to the destination that uses them), that controls live a plant, and that respects some temporal constraints to ensure that: input data is read often enough so that no significant change is lost, input data processing finishes on time, and output data is written fast enough so that it is well-timed and efficient [1].

Nowadays real-time applications can be found everywhere: from the flight control of an airplane to the steering control of a car, and from the applications that provide stock information in real-time to the game consoles. One important category of real-time applications is represented by the real-time control applications, which can contain periodic, sporadic, and aperiodic tasks. In this thesis the focus is on real-time control applications that contain only periodic tasks. This type of applications typically consists of reading sensor values from a plant, performing some computations with those values (e.g.: filtering input data, computing control laws, etc.), writing commands to the plant actuators, and repeating previous steps with a frequency that depends on the controlled plant. Thus, it is obvious that for such an application to work properly, each task has to respect strict timing constraints. Many of the control applications that exist today require multiple modes of operations that may have similar or different timing behavior, thus expressing timing behavior of such an application using a general purpose programming language (e.g.: C, Java, etc.) may result in a very complicated source code, which is very difficult to be maintained and more important very difficult to be checked formally for temporal behavior correctness. Thus, in this thesis will be presented new high-level programming constructs for describing timing behavior of real-time control applications; the focus is on how such programming constructs can be implemented. Examples of real-time control applications developed using this new programming constructs will be presented also.

Hierarchical Timing Language (HTL) [2; 3; 4] is a time-triggered language for specifying temporal behavior and interactions between periodic tasks, HTL can not express functionality of an application, which has to be implemented in a different programming language (e.g.: C, Java, etc.). HTL is considered to be the successor of Giotto [5]. HTL can express temporal behavior for multi-mode applications through the sequential composition of sets of tasks. Some control solutions have to be implemented in applications that contain periodic tasks that run at different frequencies; HTL support this through parallel composition of sets of tasks. Many of the plants that need to be control have a variable mathematical model in which case the control solution consists of using different control laws for different mathematical models, thus in such a scenario the control application consists of multiple modes having similar temporal behavior but different functionality. HTL addresses this issue by allowing the specification of an abstract timing specification, which is inherit by all the modes that have similar temporal behavior. Thus in HTL a hierarchical structure of timing specifications can be constructed. An HTL description can have only one root timing specification. The hierarchical structure of an HTL description
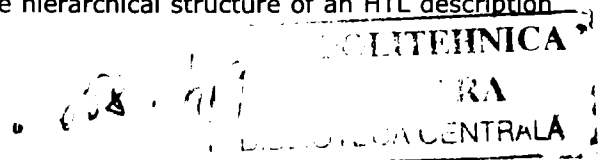
is useful especially when properties like schedulability have to be checked, i.e., only the root timing specification has to be checked, and if the property is true for the root timing specification it will be true for the rest of the specifications in the HTL descriptions, since all of them inherit the root timing specifications.

The entire work presented in this thesis is based on HTL. Thus, after an informal presentation of the syntax of HTL, there are presented two implementations of this language. As in the case of its predecessor, HTL descriptions are not compiled directly into machine code, but into *E code*, which is interpreted by a virtual machine, namely, *Embedded Machine (E machine)* [6]. This approach makes timing behavior that is expressed in HTL portable, i.e., it will be the same regardless of the platform on which the HTL description is executed, provided that there exists an E machine implementation for that platform. The first HTL implementation targets the original E machine, nevertheless since the original set of E code instructions has not been designed to support hierarchical structure, the HTL compiler, named, flattening HTL compiler, needs to flatten the hierarchical structure of an HTL description before compiling it intro E code. Since the flattening algorithm may cause an exponential blow in the number of modes, a second implementation of HTL is presented. The second implementation of HTL targets an extended version of the E machine, named, *Hierarchical E machine (HE machine)* [7], which can handle hierarchical structure at runtime. The instructions set that can be interpreted by the HE machine is a super set of the original E code instructions set, named, HE code instructions set. The compiler that transforms an HTL description into an HE code program, named, *hierarchy-preserving HTL compiler*, is also described in this thesis. The two HTL implementations are then compared both analytical and experimental. The comparison considers the size of generated E code program and HE code program for the same HTL description, and the runtime overhead introduced by interpreting an E code program and an HE code program, which have been generated for the same HTL description. Both the E machine and the HE machine are implemented in C for Unix. The compilers that target the two virtual machines are implemented in Java.

Given that in the last few years there has been a growing interest in using Java as a programming language for real-time application, in this thesis there is also presented an HTL implementation for Exotask [8; 9]. Exotask is a new programming construct for developing real-time applications using Java. One of the advantages of Exotask is the support for extension, i.e., it is possible to define new timing grammars and new schedulers that can understand the new grammars. The HTL implementation for Exotask consists of a timing grammar that can express HTL syntax and a scheduler that implements the hierarchy-preserving HTL compiler, which converts an Exotask timing specification that uses HTL grammar into an HE code program represented in Java, and a Java implementation of the HE machine, which can execute the compiled Exotask timing specification.

All the HTL implementations discussed above can be used to develop real-time applications that are designed to run on hardware platforms on which a real-time operating systems is running (e.g., Unix). However, there are many embedded systems that are based on less powerful hardware, namely, a microcontroller, on which there are not enough resources to run a real-time operating system. Thus in this thesis a fourth HTL implementation, which targets a microcontroller, is presented. The implementation consists of an optimized version of the HE machine and a modified version of the hierarchy-preserving HTL compiler that targets the optimized HE machine. For scheduling tasks released by HE machine a small real-time executive, which uses EDF as a scheduling algorithm, has been implemented.

Knowing that it is a common practice for control engineers to use modeling tools like Simulink [10] to develop and test control algorithms, it is presented a possible mapping of an HTL description to a Simulink model and the implementation of a tool that can be used to covert an HTL description into a Simulink model using this mapping. The

Simulink model of an HTL description can be used to simulate both timing behavior and functionality of a real-time control application before it is implemented. Another advantage is represented by the possibility to generate C code from Simulink models, which can be used as implementation of tasks functionality.

All the ideas discussed in this thesis have been validated and tested by implementing real-time control applications for two plants: JAviator[11; 12], and Three Tanks System (3TS).

1.1. Related Work

In the last two decades real-time programming model has evolved from the physical-execution-time (PET), to bounded-execution-time (BET), to zero-execution-time (ZET), and to logical-execution-time (LET) [13].

PET [13] programming consists of wringing a real-time application in a sequential (usually low-level) programming language (e.g., assembly, C, etc.); the entire timing of the application depends on knowing exactly the execution time of each instruction. Thus real-time applications can be developed using PET programming only for architectures that have instructions that execute in constant time (i.e., microcontrollers). Timing of PET application is very accurate; nevertheless any small change in the source code can change the entire timing of the application, thus PET is not suitable for developing complex real-time applications.

BET [13] programming unlike PET uses concurrent programming. Thus a real-time application consists of a set of periodic, sporadic, and aperiodic tasks that interact with each other. For the development of such application there are used programming languages associated with mechanism of real-time programming, which are usually included in a real-time executive [1] or a real-time operating system (i.e., Real-Time Linux [14]). A real-time scheduler (i.e., Earliest Deadline First (EDF) [15]) is used to schedule the set of tasks. Nevertheless, adding a new task to the existing set of tasks will affect the timing of the entire program and might cause the program to miss its deadlines.

ZET is the programming model used by synchronous languages [16] (i.e., Lustre [17], Esterel [18], etc.). Synchronous languages assume that the hardware configuration is powerful enough to execute tasks in *zero time*, e.g., a task reads its input and writes its output in zero time. Many of the synchronous languages allow using of formal verification in order to prove certain properties of the program. Thus synchronous languages have been used to develop safe critical control applications, i.e., Lustre has been used for implementing flight control for Airbus A380 [19].

LET [5] is the newest programming model for developing real-time application. LET has been introduced by Giotto [5], a language that can specify timing behavior and interactions between periodic sets of tasks. LET model assumes that a task reads its inputs, does some computation, and the result will be available after a period of time, called the LET of the task, even if the tasks finishes execution earlier. LET model allows sequential composition, parallel composition, and refinement of sets of tasks [2; 5].

Temporal behavior of a real-time application is either specified implicitly in written source code or explicitly using special high-level programming constructs [20]. Implicit specification of temporal behavior is often used for developing relatively simple real-time application, e.g., applications that run on a microcontroller, where there is an infinite loop that does all the computation and that can be interrupted by interrupt event handlers, which are used to communicate with the environment, the temporal behavior of the application is implemented using timers. One very important disadvantage of using implicit specification of temporal behavior is that formal analysis can not be used in order to test certain properties of the application. Thus high-level programming languages have been

developed to allow specification of temporal constraints, i.e., Lustre [17] and Esterel [18] have mathematically formalized semantics in order to allow verification of temporal behavior, they can specify both timing and functionality, Giotto [5] and Hierarchical Timing Language (HTL) [2] are two programming languages for specifying temporal behavior of real-time applications.

The work presented in this thesis focuses on the implementation of high-level programming constructs that are based on the LET programming model and that allow specification of temporal behavior explicitly.

1.1.1. Timed Languages

Giotto [5] is the pioneer of timed languages, it has introduced the concept of LET. After Giotto, other languages have been developed based on the concept of LET: Timing Definition Language (TDL) [21], Timed Multitasking (TM) [22], xGiotto [23], Timing Specification Language (TSL) [24], and Hierarchical Timing Language (HTL) [2]. TDL extends the structure of Giotto with the notion of parallel composition. TM and xGiotto are based on the notion of LET, nevertheless, unlike Giotto, they are not timed-triggered languages, but event-triggered languages. TSL relaxes the notation of LET of a task, namely, the LET of a task is not implicitly defined by the period of the task, but by the period, the offset, and duration of the task; TSL also supports direct communication between tasks in the same mode. HTL is the newest timed-triggered language, which is based on the concept of LET. HTL supports parallel composition as TDL, relaxes the LET of a task and supports direct communication between tasks in the same mode as TSL, supports hierarchical structure, and introduces the notion of communicator. In HTL the LET of a task is defined by the period of the task, the latest read input port and the earliest written output port.

The focus in this thesis is on the implementation of HTL. Semantics of HTL has been discussed in details in [3]. The implementation of HTL consists of designing and implementing a compiler and of designing and implementing a virtual machine, on which compiled HTL descriptions can be executed. As in the case of Giotto and all the languages based on it, the target platform for HTL is the Embedded Machine (E machine) [6], which is a virtual machine that interprets the so-called E code. In this thesis are presented two HTL implementations. One implementation is based on the original E machine [6], which has been introduced for Giotto. Since the E machine has not been designed to support hierarchical structure at runtime, the compiler that converts an HTL description into an E code program has to flatten the hierarchical structure of an HTL description, which leads to an exponential blow in the number of modes, and in the end to an exponential blow in the number of generated E code instructions and in the runtime overhead introduce by interpreting such an E code program. Thus, a second HTL implementation, which targets an extended version of the E machine that supports hierarchical structure at runtime, is also presented. The extended E machine is called the Hierarchical Embedded Machine (HE machine), and the code interpreted by it is called the Hierarchical E code (HE code). In this thesis it is presented a complexity analysis of the compilers designed for the two HTL implementations and an analysis of the runtime overhead introduced by interpreting E code and HE code. Such an analysis has not been conducted before for any of the other LET based languages. The HTL implementation that is based on the HE machine has been done for two different types of platforms, e.g., for Unix based platforms and for a microcontroller based platform on which there is no operating system running. The HTL implementation for microcontroller (micro HTL) is new to LET based languages; all the existing implementations target either Unix, OSEK, or Windows based platforms. Two control applications have been implemented using micro HTL, for both applications a detailed timing analysis, which considers both the worst case execution time of tasks and the runtime overhead introduced by the HE code interpretation, has been conducted. Timing

analysis that considers E machine runtime overhead has not been done before for any of the LET based languages.

1.1.2. Java and Real-Time Programming

Java is one of the most used programming languages for regular (non-real-time) applications. Nevertheless, when it comes to real-time application low-level programming languages (e.g.: assembly) and medium-level programming languages (e.g.: C) are still intensively used. The reason for not using high-level programming languages for developing real-time application is represented by the need for high speed of computation (many real-time application requires latencies below one millisecond), and more important, the need for deterministic temporal behavior. Although the need for height speed of computation is mainly solved by the powerful hardware that exists today, the second requirement still remains.

Attempts to make Java suitable for developing real-time applications have been recorded since 1997, e.g., creating an operating system that has Java support built in [25], or creating an microprocessor that is able to interpret Java bytecode [26]. Nevertheless before 2000 when The Real Time Specification for Java (RTSJ) [27] has been introduced, the work in this field lacked clear direction. RTSJ defines some guide lines for making Java a programming language for real-time applications, e.g.: backward compatibility with non real-time Java programs, support the principle "Write Once, Run Anywhere", require no syntactic extensions to the Java language, allow implementer's flexibility, etc. [28].

Recently, one important source of no-determinism in a Java application, which was introduced by Garbage Collector (GC), has been eliminated, since a real-time garbage collection (RTGC) for Java [29; 30; 31] has been proposed. This made possible the use of Java programming language for developing real-time application. Nevertheless, implementing in Java periodic task that have a frequency higher then 1Khz is currently impossible due to the overhead introduced by the RTGC (which is around 1ms). Different solutions have been proposed to solve this problem, and all of them suggested that a special type of tasks, which have a private heap and which can not access the global heap, should be used (e.g., Eventrons [32], Reflexes [33], and StreamFlex [34]). This way a real-time task could interrupt the GC; thus the GC does not influence the temporal behavior of a real-time task. However, there are and there will be embedded system that can not afford to use Java as a development language, this is mainly due to that fact that although JVM has become predictable it will still require a relatively powerful hardware (in terms of power of computation and memory), i.e., the interpretation of byte code introduces an overhead that makes it impossible to run a JVM on a microcontroller, which is a hardware platform that is often used for embedded systems. Thus the use of Java as a programming language is feasible for complex embedded system for which spending money on a powerful hardware is less important than the benefit of being able to use an object-oriented programming language, e.g., next generation battleships [35], whereas for simple embedded systems using a microcontroller and C as a programming language is less expensive and thus more feasible, i.e., for embedded system that are produced in huge quantities every penny matters.

Exotask [8] is a new programming construct for developing real-time applications using Java. Exotask addresses low latency problem in a less restrictive way as compared to previous solutions, but still using a similar approach, e.g., tasks that have their own private heap. Exotask system support pluggable schedulers. An interface for developing third-party schedulers has been defined, which is different from RTSJ where only virtual machine vendors can define new schedulers. But, maybe the most important achievement of Exotask is that Exotask applications are not only portable from the functionality point of view but also from the temporal behavior point of view, which is not the case for many other

currently existing solutions since they rely on platform dependent information. Timing portability is achieved in Exotask by using LET programming model [5].

In this thesis it is presented a new timing grammar for Exotask that is based on the HTL semantics. A new scheduler that understands the new timing grammar is also presented; unlike the scheduler that comes with the Exotask distribution, which uses a single physical thread to run all the tasks in an Exotask application, the new scheduler supports single threading as well as multithreading execution of tasks.

1.2. Overview

The remaining of this thesis is structured in eight chapters. Chapter 2 describes informally the semantic of HTL and the features of this language: sequential composition, parallel composition, refinement, and distribution. Target platform for a real-time application developed with HTL is represented by the E machine, which makes HTL application portable in terms of timing behavior; in Chapter 3 are presented two version of the E machine: the original E machine [6] that was initially developed for Giotto [5], and an extended E machine, which offers support for handling hierarchical structure at runtime, namely, HE machine. In Chapter 4 are presented two HTL compiler algorithms, i.e., flattening HTL compiler, which targets E machine, and hierarchy preserving HTL compiler, which targets HE machine. The two compilers are then analyzed and compared both formally and experimentally. Next, Chapter 5, depicts Three Tanks Systems (3TS) plant, which is one of the two plants that are used as case studies in the thesis. An HTL implementation of a control application for the 3TS plant is also presented; the implementation uses all HTL features. Chapter 6 presents the Exotask programming constructs, then an implementation of the HTL grammar for Exotask and a scheduler that understands the new grammar. The chapter ends with a presentation of the JAviator [11], which is the second plant used as a case study in the thesis. Finally, the new Exotask-HTL grammar is used to develop a controller for the JAviator. Many embedded systems are based on microcontrollers, thus Chapter 7 describes an implementation of HTL that targets a microcontroller; the implementation is named micro HTL. The micro HTL is then used to develop two control applications: one for 3TS plant and the other one for the JAviator plant. The simulations show that the efficacy [1] of the micro HTL is between 75% and 80% for the JAviator control application and about 95% for 3TS control application. Chapter 8 presents a way of modeling an HTL description in Simulink and how such a model can be used to improve development of real-time application. The chapter ends with a case study for the 3TS plant. Finally, Chapter 9 concludes the thesis.

2. Hierarchical Timing Language

Hierarchical Timing Language (HTL) [2; 3; 4] is a timed triggered programming language that is based on the concept of Logical Execution Time (LET), which has been first introduced for Giotto [5]. HTL is a high level programming language for specifying timing of a real-time application. It can only be used to define the timing of a real-time application, and not the functionality, which has to be specified in regular programming language (e.g., C, Java, etc.). Besides of separating the functionality of a real-time application from its timing, HTL also separates, in the spirit of platform-based design [36], the timing specification of a real-time application (i.e., release time and termination time of a task), from the implementation (i.e., WCET of the task). Furthermore, HTL has been recently extended to support separation of concerns for the reliability of a real-time application [3; 37].

2.1. Overview

HTL can express interactions between periodic non-blocking tasks. Tasks in HTL are pure functional and finite blocks of code; they do not use any synchronization mechanisms or blocking I/O operations. A task has it's own private memory space, and the only way to communicate with a task is through its input and output ports.
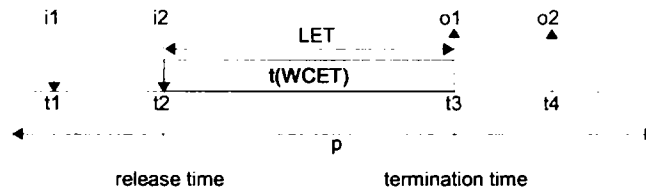
Fig. 2.1: HTL task model

A task in HTL consists of a set of input ports and a set of output ports. In Fig. 2.1 it is presented an example of an HTL task. An HTL task has a period (e.g., p); the entire timing of a task is relative to the period of the task. The moment in time when the latest input port is read (i.e., $i2$), defines the release time of the task, while the moment in time when the earliest output should be written defines the termination time of the task (i.e., $o1$). The time interval between task release time and task termination time specifies the LET of the task. In between this two moments the task should be executed, but from HTL point of view it is not important when the task physically executes (i.e., it can be preempted and resumed several times), the only thing that matters, is that task execution has to complete before termination time. The release time, termination time, and LET are considered to be platform independent information, while WCET is platform dependent information.
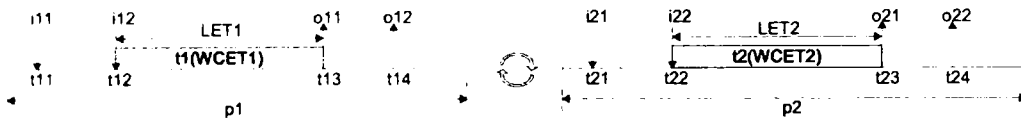
Fig. 2.2: Sequential composition

HTL supports **sequential composition of sets of tasks**, i.e., a set of tasks, s, can be replaced with another set of tasks s'. In Fig. 2.3 are presented two task, $t1$ and $t2$, that are composed sequentially, i.e., only one of the two tasks will be executed at a particular moment in time, the task being executed is chosen based on a condition. The only constraint that HTL requires for a task to switch to another task, is that both tasks have the same period.

Fig. 2.3: Parallel composition

Parallel composition of tasks is also supported by HTL. In Fig. 2.3 are presented two tasks, e.g., $t1$ and $t2$, that are executed in parallel. The two tasks can have different periods. The platform on which the two tasks are executed has to ensure that the timing requirements of both tasks are met (i.e., the program is schedulable for the given platform). Currently existing HTL runtime system uses *Earliest Deadline First* (EDF) [15] algorithm in order to schedule tasks that are composed in parallel.
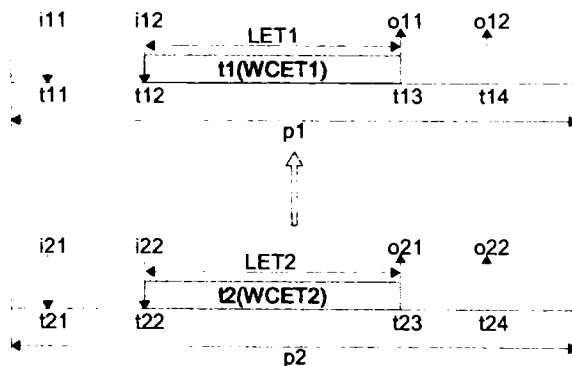
Fig. 2.4: Refinement

Refinement of tasks is one of the strongest features of HTL. In HTL a task can be either abstract or concrete. An *abstract task* is a task with no functionality; it is used as a placeholder for a concrete task. A *concrete task* is a task that has functionality. Only abstract tasks can be refined by concrete tasks or by other abstract tasks. In HTL a hierarchical structure of tasks can be defined. Abstract tasks have no impact on the functionality of a program; they are used only to specify abstract timings, which can

be refined into concrete timings. The main benefit of the hierarchical structure of an HTL description is represented by the fact that testing properties like schedulability and reliability, for an HTL specification is simplified, e.g., if certain constrains (Section 2.3) are imposed on the refinement, it is enough to test the property only for top most abstract timing, and if the top most abstract timing has the property, then the entire HTL description has the property, due to the refinement constraints.

Fig. 2.5: Direct task communication

HTL supports two models of communication between tasks: (a) **direct inter-tasks communication**(Fig. 2.5) and (b) **inter-tasks communication through communicators** (Fig. 2.6). Direct inter-task communication means that a task can read the output of another task; this kind of communication can only happen between tasks that have the same period. Direct inter-task communication introduces a dependency relation between the two tasks, e.g., if a task t reads the output port of another task $t`$, then task $t$ can not execute unless task $t`$ has completed execution.
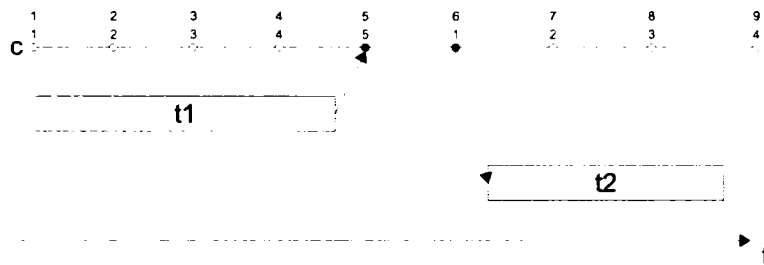
Fig. 2.6: Inter task communication through communicators

Inter-tasks communication through communicators can happen between tasks that have the same period as well as between tasks that have different periods, the only constraint being the fact that the period of the communicator that is used for the communication, should be harmonic to the periods of the two tasks that communicate. A communicator is a typed variable that has a period associated with it. A communicator can be accessed only at specific moments in time, which are determined by the period of the communicator. In Fig. 2.6 it is illustrated how task $t1$, which has a period of ten time units, communicates with task $t2$, which has a period of five time units, through communicator c, which has a period of one time unit. Since task $t1$ has a period of 10 time units, there are ten instances of communicator c that can be accessed by task $t1$; in this case task $t1$ writes to the fifth instance of communicator c. On the other hand for task $t2$ there are only five instances of communicator c that can be accessed; task $t2$ reads the first instance of communicator c. If two tasks that have different periods communicate through a communicator, then if the task that writes to the communicator has a higher period than the task that reads from the communicator, then the task that reads will end up reading same value multiple times (i.e., in the example presented in Fig. 2.6, task $t2$

will read twice the same value). On the other hand if the task that reads the communicator has a higher period than the task that writes the communicator, then the task that writes the communicator will end up overriding values that have not been read. Thus when this type of communication is used, one should be aware of the two situations.

Many embedded system require the software to be distributed over a set of hosts. HTL supports **distribution** of an HTL description over a set of host by allowing tasks to be mapped to different hosts. In HTL only for root tasks can be specified a host mapping, while the refining tasks implicitly will be mapped to the host to which parent task is mapped.

2.2. Syntax

The basic elements of an HTL description are the ports, communicators and tasks. A port is a typed variable; declaration of a port consists of a type, a name, and an initialization driver. An initialization driver is a function written in a different language than HTL (i.e., C) that is called by the E machine when something (i.e., port, communicator, etc.) needs to be initialized. In Alg. 2.1 it is declared a port p, of type int, and which is initialized by initialization driver $zero$.

Alg. 2.1 Example of port declaration (concrete syntax)
---
port
 int p := zero;

A communicator is a typed variable that can be accessed at particular moments in time; it is defined by a name, a type, a period, which restricts the access (i.e., read or write operation) to the communicator, and an initialization driver. In Alg. 2.2 it is presented an example of a communicator declaration, c, which has a period of 100 time units, is of type int, and it is initialize by the initialization driver $init$.

Alg. 2.2 Example of communicator declaration (concrete syntax)
---
communicator
 int c **period** 100 **init** zero;

A task declaration consists of a name, a set of input ports, a set of output ports, a set of state ports, and a function. An input/output/state port declaration consists of a type and a name. Declaration of a state port specifies an initialization driver also. The input and output ports are formal ports, i.e., they are replaced at invocation time with actual ports, while state ports are actual ports. Task input ports represent the input for the task function, and task output ports represent the output of the task function. The state ports are used to store internal state of the task function. If a task declaration has no function associated with it, then the task is considered to be abstract, otherwise it is concrete. In Alg. 2.3 it is declared a task that has one input port, i, of type int, one state port, s, of type int, which is initialized by initialization driver $zero$, one output port, o, of type int, and the function that implements the functionality of the task is f.

Alg. 2.3 Example of task declaration (concrete syntax)
---
 task t **input** (int i) **state** (int s:=zero) **output** (int o) **function** f;

One or more tasks form a mode. A mode consists of a name, a period, a program name, a set of task invocations, and a set of mode switches. The period of a mode specifies

how often the tasks that are invoked in the mode are executed and how often the mode switches are tested. The program name specifies the program that refines the mode; it can be empty if there is no such program. A task invocation consists of a task name, which identifies the task that has to be executed, and a mapping of actual input ports to formal input ports and actual output ports to formal output ports. The type of the actual port has to match the type of formal port to which it is mapped. As an actual port it can be used either a port, in which case it has to be specified the name of the port being used, or a communicator instance, which consists of a communicator name and a number, which identifies the communicator instance number, relative to the period of the mode, that will be used. A communicator can be used as an actual port in a mode only if the period of the communicator is harmonic to the period of the mode, i.e., the period of the mode is a multiple of the period of the communicator. The number of communicator instances that are available for a communicator c that has a period of π_c time unit, in a mode m that has a period of π_m time units it is equal to $\frac{\pi_m}{\pi_c}$. A task invocation can also specify the name of the parent task, if any.

In Alg. 2.4 it is declared a mode, m, which has a period of 1000 time units, invokes three tasks, i.e., $t1$, $t2$, and $t3$, and contains one mode switch, which changes execution to mode m`, when condition $cond$ evaluates to true. The mode is the structure through which HTL supports sequential composition, i.e., in HTL it is possible to specify a set of modes, out of which a single mode can execute at a particular moment in time, which is called the active mode, and that can switch between each other.

Alg. 2.4 Example of mode declaration (concrete syntax)

```
mode m period 1000{
  invoke t1 input ((c1,1)) output((c2,1));
  invoke t2 input ((c2,1)) output(p);
  invoke t3 input (p) output((c1,4));
  switch(cond(c)) m` ;
}
```

Tasks in a mode can communicate to each other either through ports or through communicators. In Alg. 2.4 task $t1$ and $t2$, and tasks $t3$ and $t1$ communicate through communicators, i.e., task $t1$ writes to the second instance of communicator $c1$, and task $t2$ reads the same instance of the same communicator, while task $t3$ writes to the fifth instance of communicator $c1$ and task $t1$ reads the second instance of communicator $c1$. In Alg. 2.4 there is also an example of communication through ports, e.g., task $t2$ writes to port p, which is read by task $t3$, this communication imposes a dependency relation between tasks $t2$ and $t3$, i.e., task $t3$ can not execute before task $t2$ has finished its execution.

The LET of a task that is invoked in a mode is determine by both the period of the mode and the communicators that are read or written by the task, and it is not influenced by the ports that are read or written by the task. Thus, the LET of the task is the time interval between the latest read communicator or the beginning of the period if no communicator is read, and the earliest written communicator or the end of the period if there is no written communicator.

Assuming that communicator $c1$ has a period of 100 time units and that communicator $c2$ has a period of 200 time units, Fig. 2.7 depicts timing analysis for tasks invoked in mode m. Task $t1$ has an LET of 100 time units, i.e., it reads second instance of communicator $c1$ and updates second instance of communicator $c2$. Tasks $t2$ and $t3$ have an LET of 200 time units, i.e., task $t2$ reads the second instance of communicator $c2$, and although it does not write to any communicator, it communicates through port p with task $t3$, which writes to the fifth instance of communicator $c1$, thus the LET of $t2$ can not be higher than 200 time units, similar for $t3$ it can be shown that it can not have an LET higher then 200
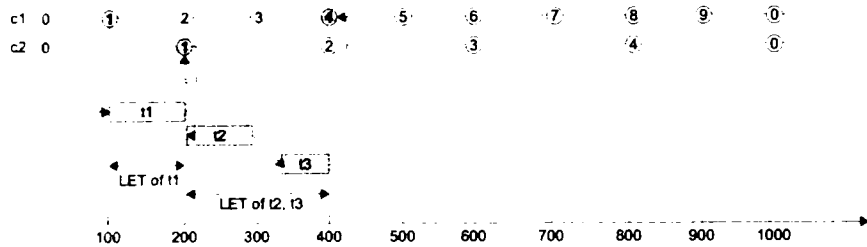
Fig. 2.7: Timing analysis of tasks in mode m

time units.

A set of modes that are composed sequentially form a module. A module declaration consists of a name, a start mode name, a set of port declarations, a set of task declarations, and a set of mode declarations. The start mode name represents the name of the mode that is active first time the module is executed. The set of port declarations specifies the ports that can be used to communicate between tasks that are invoked in modes from the module. The set of task declarations specifies the set of tasks that can be invoked in the modes from the module. The set of mode declarations specifies the modes that are composed sequentially in the module, i.e., only one of these modes can be executed at a particular moment in time. Modules in HTL represent the structures that are used to support parallel composition, e.g., in HTL there can be defined a set of modules, which are executed in parallel.

In Alg. 2.5 it is declared a module, which contains one port declaration, i.e., p, three task declarations, $t1$, $t2$, and $t3$, and two mode declarations $m1$ and $m2$, with $m1$ being the start mode.

Alg. 2.5 Example of module declaration (concrete syntax)

```
module M start m1{

    port
      int p:=zero;
    task t1 input(int i1) state() output(int o1) function f1;
    task t2 input(int i1) state() output(int o1) function f2;
    task t3 input(int i1) state() output(int o1) function f3;

    mode m1 period 1000{
      invoke t1 input ((c1,1)) output((c2,1));
      switch(cond(c)) m2;
    }

    mode m2 period 1000{
      invoke t2 input ((c1,1)) output(p);
      invoke t3 input (p) output((c2,1));
      switch(cond(c)) m1;
    }
}
```

A set of modules that are composed in parallel form a program. A program declaration consists of a name, a set of communicator declarations, and a set of module declarations. The set of communicator declarations represents the communicators that can be read or written by task that are invoked in a mode in any of the modules that are

in the program. The set of module declarations specifies the modules that are composed in parallel in the program.

In Alg. 2.6 it is declared a program, P, which contains two communicators, i.e., $c1$ and $c2$, and two modules, i.e., $M1$ and $M2$.

Alg. 2.6 Example of program declaration (concrete syntax)

```
program P{
 communicator
   int c1 period 100 init zero;
   int c2 period 200 init zero;

 module M1 start m1{
   task t1 input(int i1) state() output(int o1) function f1;
   mode m1 period 1000{
     invoke t1 input ((c1,1)) output((c1,4));
   }
 }

 module M2 start m2{
   task t2 input(int i1) state() output(int o1) function f1;
   mode m2 period 1000{
     invoke t2 input ((c2,0)) output((c2,2));
   }
 }
}
```

In HTL the program is the structure through which refinement is supported. An HTL description consists of a set of programs; one of them being the root program, while the rest of the programs refine modes from the root program. In other words, the root program specifies an abstract timing of a real-time application, which is refined into concrete timings by the rest of the programs in an HTL description.

In Alg. 2.7 it is presented an HTL description that consists of two programs, i.e., P and $P1$. Program P is the root program, it contains one module, which contains one mode, which invokes two tasks, e.g., $t1$ and $t2$, and which is refined by program $P1$. Program $P1$ contains two modules: $M11$, which refines task $t1$ from the refined mode in two tasks, i.e., $t11$ and $t13$, and $M12$, which refines task $t2$ from the refined mode into task $t12$.

Program P directly refines another program $P\grave{}$, if there is a mode m in $P\grave{}$ that specifies P as its refinement. Program P indirectly refines another program $P\grave{}$, if there is a set of programs $P1 \ldots Pn$ so that $P1$ directly refines $P\grave{}$, P directly refines Pn, and for any two consecutive programs Pk and $Pk+1$ in $P1 \ldots Pn$, program $Pk+1$ directly refines program Pk.

Module M contained in a program P directly refines module $M\grave{}$ contained in a program $P\grave{}$, if P directly refines $P\grave{}$. Module M contained in a program P indirectly refines module $M\grave{}$ contained in a program $P\grave{}$, if P indirectly refines $P\grave{}$.

Mode m contained in a program P directly refines mode $m\grave{}$ contained in a program $P\grave{}$, if P directly refines $P\grave{}$. Mode m contained in a program P indirectly refines module $m\grave{}$ contained in a program $P\grave{}$, if P indirectly refines $P\grave{}$.

A program, P, which (directly or indirectly) refines a mode from another program, $P\grave{}$, is a child program or a sub-program of $P\grave{}$, while $P\grave{}$ is the parent program or the super-program of P. A program P is the sub-program as well as the super-program of itself. A module, M, which (directly or indirectly) refines a mode or a set of tasks from a mode from another module, $M\grave{}$, is a child module or a sub-module for $M\grave{}$, while $M\grave{}$ is the parent module or the super-module of M. A module M is the sub-module as well as

Alg. 2.7 Example of HTL description with refinement (concrete syntax)

```
program P{
  communicator
    int c1 period 100 init zero;
    int c2 period 200 init zero;

  module M1 start m1{
    task t1 input(int i1) state() output(int o1);
    task t2 input(int i1) state() output(int o1);
    mode m1 period 1000 program P1{
      invoke t1 input ((c1,1)) output((c1,4));
      invoke t2 input ((c2,1)) output((c2,2));
    }
  }
}

program P1{
  module M11 start m11{
    task t11 input(int i1) state() output(int o1);
    task t13 input(int i1) state() output(int o1);
    mode m11 period 1000{
      invoke t11 input ((c1,1)) output((c1,4)) parent t1;
      switch(cond1(c1)) m13;
    }
    mode m13 period 1000{
      invoke t13 input ((c1,1)) output((c1,4)) parent t1;
      switch(cond2(c1)) m11;
    }
  }
  module M12 start m12{
    task t12 input(int i1) state() output(int o1);
    mode m12 period 1000{
      invoke t2 input ((c2,1)) output((c2,2)) parent t2;
    }
  }
}
```

the super-module of itself. A mode, m, which refines a set of tasks from a mode, $m`$, is a child mode or a sub-mode for $m`$, while $m`$ is the parent mode or the super-mode of m. A mode m is the sub-mode as well as the super-mode of itself.

Parts of an HTL description can be distributed over a set of hosts. The distribution works by annotating modules in the root program with the information about the host on which each module will run. All the modules in the rest of the programs will be automatically mapped to the host to which the refined module from the root program is mapped.

2.3. Well-Formed and Well-Timed HTL Descriptions

An HTL description is **well-formed** if it meets to the following constraints:

Constraints on program: (1) there is only one top-level program; and (2) each mode (other than those of the top-level program) has an unique parent mode.

Constraints on communicators: (1) if a communicator is declared in program P, then it is not redeclared in any other sub-program of P; (2) if a communicator c is accessed (read or written) by a task invocation or a mode switch in a mode of module M

in program P, then c is declared in one of the super-programs of P; and (3) if a communicator c belongs to the hierarchical write-set of a module M, then c does not belong to hierarchical write-set of any sibling module of M.

Constraints on task invocations: (1) for every task invocation, the read time is earlier than the write time; (2) the precedence relation on task invocations is acyclic; (3) two task invocations in a mode cannot write to the same port or to the same instance of a communicator; (4) if a task invocation reads from or writes to a communicator c (resp. port p), then the type of c (resp. p) complies to the type of the corresponding formal parameter in task declaration; (5) if a task invocation in a mode m reads from or writes to a communicator c, then period π_m is a multiple of the communicator period π_c.

Constraints on refinement: (1) if program P refines a mode m, then the period of all modes in P is equal to π_m (this ensures that when there is a mode switch, there is no unsafe termination of tasks in lower-level modes); (2) every task invocation of a mode m that does not belong to the top-level program has an abstract parent task invocation in the parent of m (this ensures that the parent task acts as a placeholder for its children during chedulability analysis); (3) any two distinct task invocations in two modes of two (possibly identical) sibling modules have distinct parent task invocations (this ensures that all tasks that can potentially execute in parallel have unique parents); and (4) if inv' is the parent task invocation of inv, then the read time of inv is not later than that of inv', the write time of inv is not earlier than that of inv', and every invocation that precedes inv refines a task invocation that precedes inv' (this ensures that the parent invocation is more constrained in time than the child task invocation, which is used in the schedulability analysis).

An HTL description is **well-timed** if the worst-case execution time (WCET) of any task invocation is less or equal to the WCET of the parent task invocation and the worst-case transmission time (WCTT) of any task invocation is less or equal to the WCTT of the parent task invocation. Well-formedness is independent of the implementation platform, well-timedness is not.

3. Embedded Machine

Embedded machine (E machine) is a virtual machine that mediates interactions between tasks. The E machine can release a task for execution but it can not execute it; the task has to be executed by the operating system. The code interpreted by the E machine is the so-called E code. The E code contains instructions for copying values from one variable to another, for releasing tasks, and control-flow instructions. Since an E code program is interpreted by the E machine, it can be executed on any platform for which there is an E machine implemented, thus an E code program is portable. The E machine was first introduced as a target platform for Giotto [6]. Then the E machine has been extended [7] to handle hierarchical structure at runtime, in order to support execution of E code generated for HTL descriptions that contain at least one level of refinement. The extended E machine with hierarchical support is called the *hierarchical E machine* (HE machine). The set of E code instructions has been extended also; the extended E code being named *hierarchical E code* (HE code). In this chapter, first the original E machine will be presented, then the HE machine.

3.1. Original Embedded Machine

Fig. 3.1 depicts visually the structure of the E machine. The E machine is divided in three parts: core E machine functionality and data structures, E code program related data structures, and application functionality.

Application functionality consists of: task functions, driver functions, condition function, and state. *Task functions* implement the functionality of tasks that are released by the E code program. Each task has its own private memory. Tasks communicate to the environment only through task input ports and task output ports, which are variables declared in the global memory. *Driver functions* implement the functionality that is invoked by the E code program to transfer information between two variables, a variable and a task input/output port, or between an input port and an output port of two different tasks. A task can not directly access the input port or the output port of an other task, thus the communication has to be implemented through drivers. *Condition functions* represent functional code which evaluates conditions based on some inputs, and returns a boolean value, which represents the result of the evaluation. *State* is a set of variable that can be accessed only through driver functions and condition functions.

E code program related data structures consists of: list of E code instructions and for each type of function (i.e., task, driver, and condition), there is a table, which associates an index to a task, driver, or condition, respectively. The list of E code instructions represent the E code program that has to be interpreted. The index of a task (driver or condition) in the tasks (drivers or conditions) table is used to refer to that task (driver or condition) from an E code instruction.

The central part of the E machine core is represented by the *E code interpreter*. The E code interpreter accesses the list of E code instructions through a pointer, i.e., *program*
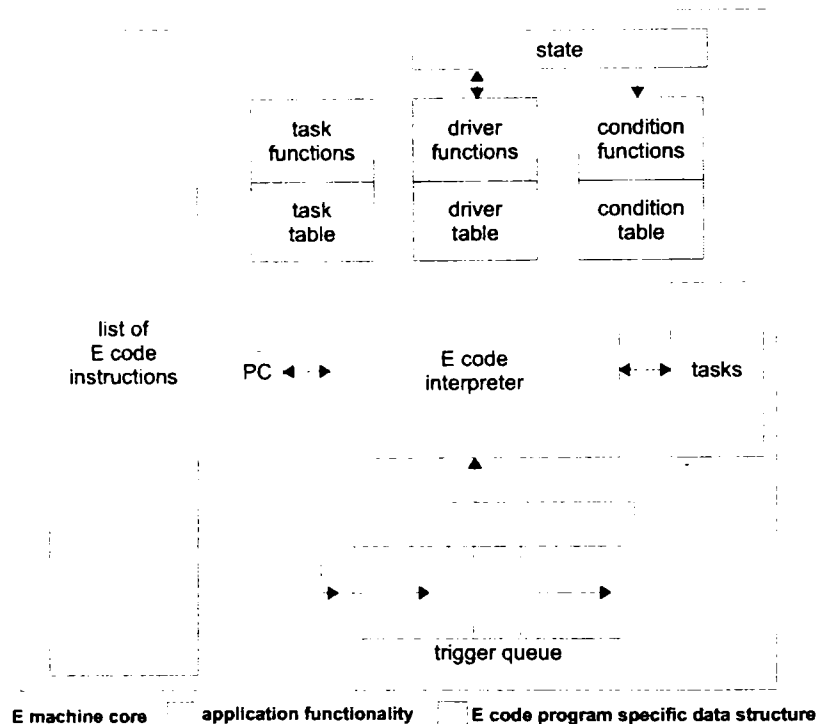
Fig. 3.1: E machine overview

counter (PC), that indicates which is the next instruction in the list of E code instructions that has to be interpreted. The program counter is either incremented after an instruction has been interpreted or is set to an explicit value, if the interpreted instruction is a control flow instruction (e.g., jump). Another data structure which is used by the E machine core is the *FIFO queue of triggers*; there is only one queue of triggers. A trigger, *g*, is an association between a list of events and an address of an E code instruction, e.g., $g = (le.a)$. The list of events, *le* can contain at most one time event and zero or more task completion events, i.e., $listofevents = (n.cmps)$. A time event, *n*, represents a period of time that has to elapse from the moment when the trigger was added to the trigger queue. A task completion event is fired when a task has finished its execution. A trigger is considered to be active if all the events (e.g., time event and task completion events) on which it depends have triggered. The trigger queue is checked every time an event happens and the first found active trigger is remove from the queue and the PC is set to the address associated with the trigger, thus the E machine will start interpreting E code at that address. There is also a *list of tasks* that have been released for execution; whenever a task has been released by the E machine its index in the task table will be inserted into the tasks list, and it will be removed from the list when task finishes its execution.

The E machine always starts interpreting instructions at address zero. During E code interpretation tasks can be released and triggers can be added to the trigger queue, when the E code interpreter encounters a *return* instruction, the E machine enters the waiting state. When the E machine is in waiting state, no E code is interpreted, the only thing that the E machine does in the waiting state is to observe for time events and task completion events and to update its trigger queue. When a trigger in the trigger

queue becomes active, the trigger is removed from the queue and the E machine starts interpreting E code at the address specified in the trigger.

Formally the E machine is a tuple (*TF, DF, CF, state, TT, DT, CT, Eprogram, interpreter, PC, Q, tasks*), where *TF* represents task functions, *DF* represents driver functions, *CF* represent condition functions, *state* represents the state, *TT* represents the table of tasks, *DT* represents the table of drivers, *CT* represents the table of conditions, *Eprogram* represents the list of E code instructions, *interpreter* represents the E code interpreter, *PC* represents the program counter, *Q* represents the queue of triggers, *tasks* represents the list of released tasks.

Nevertheless only some parts of the E machine will undergo changes during E code execution, whereas some part will not change at all. Thus the E machine configuration is defined as tuple (*state, PC, Q, tasks*). First E machine configuration is (*state* $= state_0$, $PC = 0$, $Q = \emptyset$, *tasks* $= \emptyset$).

3.1.1. List of E code Instructions

E code interpretation can be seen as a sequence, $u0, u1, \ldots$, of E machine configurations. Where $u0$ is the initial configuration. In the remaining of this subsection, each E code instruction will be explained based on the effect it has on the E machine configuration. Thus before executing an instruction the configuration is (*state, PC, Q, tasks*), and after the instruction has been executed the new configuration is (*state`, PC`, Q`, tasks`*).

call(*d*): invokes diver function identified by driver index *d* in the drivers table; the configuration after executing this instruction is (*state`* $<>$ *state, PC`* $= PC + 1$, *Q`* $= Q$, *tasks`* $=$ *tasks*); this instruction affects the state of the E machine since a driver always copies a value from a variable into another variable;

release(*t*): releases task identified by task index *t* in tasks table for execution; the configuration after executing this instruction is (*state`* $=$ *state, PC`* $= PC + 1$, *Q`* $= Q$, *tasks`* $=$ *tasks* $\cap$ *t*); the released task is added to the list of released tasks;

future(*n, cmps, a*): adds a trigger, $g = (n \cap cmps, a)$, to the trigger queue, the list of events the trigger depends on is represented by a time event that fires at *n* time ticks after the trigger has been appended to the queue, and on the set of task completion events represented by *cmps*, when the trigger becomes enabled E machine will start interpreting instructions at address *a*; the configuration after executing this instruction is (*state`* $=$ *state, PC`* $= PC + 1$, *Q`* $= Q \circ g$, *tasks`* $=$ *tasks*)

jump(*a*): executes an unconditioned jump to address *a* ; after executing this instruction the configuration is (*state`* $=$ *state, PC`* $= a$, *Q`* $= Q$, *tasks`* $=$ *tasks*);

if(*cnd, a*): executes an conditioned jump to address *a*; the condition that has to be evaluated in order to determine whether the jump should be performed or not is identified by the index *cnd* in the conditions table; after executing this instruction if the condition was evaluate to $true$ the configuration is (*state`* $=$ *state, PC`* $= a$, *Q`* $= Q$, *tasks`* $=$ *tasks*), otherwise the configuration is (*state`* $=$ *state, PC`* $= PC + 1$, *Q`* $= Q$, *tasks`* $=$ *tasks*);

return: interrupts E code interpretation and brings the E machine into waiting state; after executing this instruction the configuration is (*state`* $=$ *state, PC`* $= \perp$, *Q`* $= Q$, *tasks`* $=$ *tasks*).

3.1.2. Handling Parallelism in E code

One of the features of HTL, is parallel composition of sets of tasks, thus when an HTL description is compiled, the generated E code has to ensure that several blocks of E code

are executed in parallel (cvasi-parallel). In other words, the parallel composition of sets of periodic tasks, reduces, at E code level, to cvasi-parallel composition of blocks of E code. In order to achieve this, a future instruction with zero time ticks as time event and an empty list of task completion event (this *future* instruction is farther referred as *future* zero instruction) is generated for each block of E code that has to be executed in parallel. This will ensure that execution of all blocks is started in parallel; it is up to each block to ensure periodicity after the block was started.


```
                      future(0, adr1)
                      future(0, adr2)
                 1    return


            ▼                              ▼

    adr1:                        adr2:
        call(d11)                    call(d21)
        future(0, adr12)             future(0, adr21)
        return                       return
    adr12:                       adr21:
        call(d12)                    call(d22)
        release(t1)                  release(t2)
        future(5, adr1)              future(10, adr2)

    2                            3
```
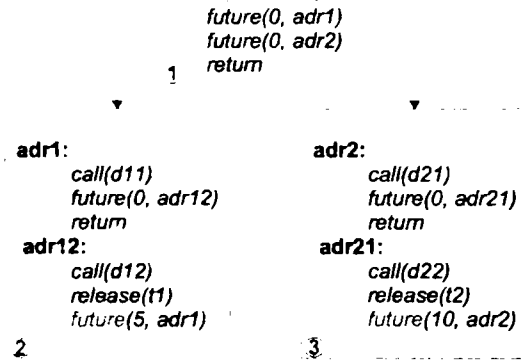
Fig. 3.2: Parallel composition in E code

In Fig. 3.2 it is presented an example of an E code program that composes in parallel two tasks: $t1$, which has a period of 5 time units, and $t2$, which has a period of ten time units. Both tasks have one input port and one output port, and the output port of $t1$ is copied into input port of $t2$, and the output port of $t2$ is copied into input port of $t1$. The E code program consists of three blocks. The first block is the initialization block, i.e., the block that will be executed first. After initialization block is executed, two triggers will be added to the trigger queue; both triggers depend on a time event of zero time ticks. One of the trigger ensures that the second block of E code will be executed, and the second trigger ensures that the third block of E code will be executed.

The initialization block is executed only once, while blocks two and three are executed periodically, i.e., every 5 and every 10 time units, respectively. The first instruction in blocks two and three is a *call* instructions, which ensures that the values produced by tasks $t1$ and $t2$, respectively, in the previous invocation are copied into two global variables, i.e., $v1$ and $v2$, respectively. The copy operation is done through drivers $d11$ and $d21$, respectively. Since the values of the global variables $v1$ and $v2$ are also written into input ports of task $t2$ and $t1$, respectively, by drivers $d22$ and $d12$, respectively, the E code program has to ensure that first drivers $d11$ and $d21$ are executed and then $d12$ and $d22$ are executed. Thus the *call* instructions that updates global variables $v1$ and $v2$ (e.g.: *call*($d11$) and *call*($d21$), respectively) have to be executed prior to the *call* instructions that read global variables $v1$ and $v2$ (e.g.: *call*($d22$) and *call*($d12$), respectively). This is way a *future* zero instruction is generated after drivers $d11$ and $d21$ are executed and then a *return* instruction is generated. The *future* zero instruction will place a trigger in the trigger queue that will resume execution of corresponding E code block. The *return* instruction will bring the E machine into waiting state, thus allowing the other block of E code to start/resume its execution. After driver $d11$ and $d21$ have been executed, driver $d12$ will be executed, task $t1$ will be released and a trigger will be placed in the trigger queue to start again execution of block two after 5 time units; similar for the third block. Thus the actual execution of the three blocks is: block one, block two, block three, block two, and

block three. Thus we can say that block two and three are executed cvasi-parallel.

3.2. Hierarchical Embedded Machine

Although the E machine offers support for sequential composition and parallel composition of set of task, it does not offer support for handling refinement (i.e., hierarchical structure) at runtime. Thus before compiling an HTL description into E code, the description needs to be flatten. As shown in Chapter 4 flattening an HTL description can lead to an HTL description that has exponentially many modes in terms of number of modes of the original description. Thus the E machine has been modified so that it offers support for handling hierarchical structure at runtime, the modified E machine is called *Hierarchical Embedded Machine* (HE machine). The set of E code instructions has been extended with new instruction that can exploit the new structure of the E machine; the extended E code is called the *Hierarchical E code* (HE code).
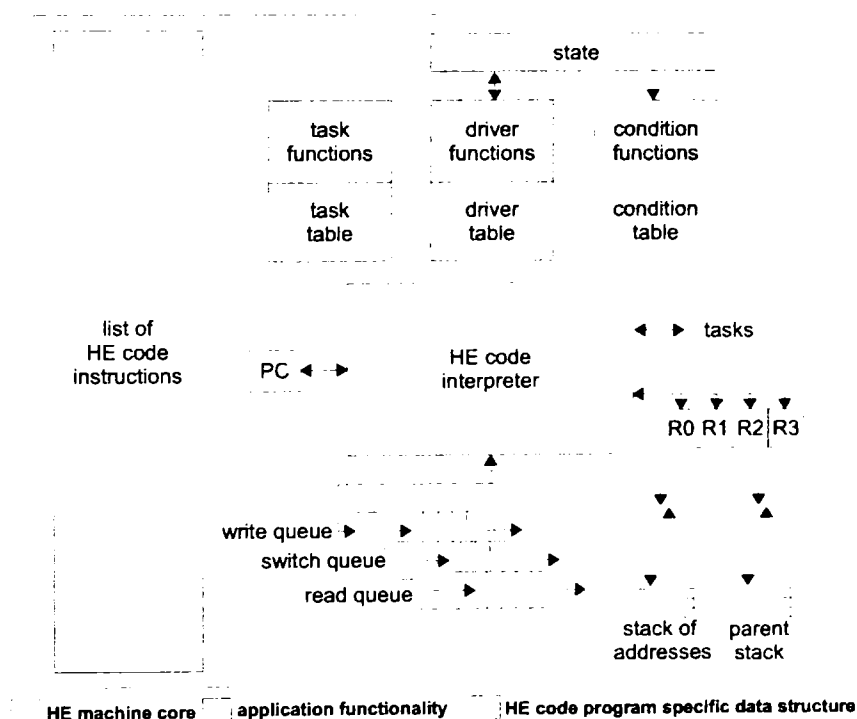
Fig. 3.3: HE machine overview

Fig. 3.3 offers an overview on the structure of the HE machine. The HE machine is also divided in three parts: core HE machine functionality and data structures, HE code program related data structures, and application functionality. The application functionality and HE code program related data structures are the same as in case of the E machine. Only the core HE machine has been changed, i.e., there is not one trigger queue, but three trigger queues, there are also two stacks, and four trigger registers.

The three trigger queues are named based on the trigger that will be added into each queue. Thus *write queue* is the trigger queue in which are added triggers for HE code

blocks that update global variables with values from tasks output ports. *Switch queue* is the trigger queue in which are added triggers for HE code blocks that handle switches between sets of tasks. *Read queue* is the trigger queue in which are added triggers for HE code blocks that read global variables.

The HE machine core contains two stacks: one stack stores HE code addresses and the other stack stores trigger references. The *stack of addresses* is used to implement subroutine calls. The stack of trigger references is used for handling hierarchical structure, i.e., in case of HE code program generated for HTL description, the stack of triggers is used to store references to parent triggers, this is why this stack is called *parent stack*.

The HE machine contains four trigger registers, e.g.: *R0, R1, R2,* and *R3,* which can store references to trigger. The trigger register are used to implement operation with trigger, e.g., setting the parent of a trigger, changing the children list of a trigger or clearing the children list of a trigger.

The concept of trigger has been extended; a trigger g is not only a pair of a list of events and an address, but a trigger also has a parent trigger, and a list of child triggers. Thus in the case of HE machine a trigger g is tuple $(le, a, par, clist)$, where *le* contains events on which trigger activation depends (it can contain one time event and one or more task completion events), a is the address of the E code that will be interpreted when the trigger gets enabled, *par* is a reference to a trigger, which is considered the parent of g, the *clist* is a list of trigger references, which are considered children of g. Thus through the two new elements that have been added to a trigger it is possible to construct an hierarchical structure of triggers, in which each trigger knows who is its parent and which are its children. A trigger can have at most one parent (i.e., it can have no parent if it is a root trigger), and can have none, one or many children.

When the HE machine is started, it will interpret HE code starting at address zero. The HE machine will interpret HE code until a return instruction is encountered and no address is in the stack of addresses. The semantics of the return instruction in the case of HE machine has been overloaded, i.e., it can be used both to return from a subroutine call and to bring the HE machine in a waiting state. Thus if a return instruction is interpreted and there is at least one address in the stack of addresses, an address will be popped from the stack and HE code interpretation will continue at that address. On the other hand if there is no address in the stack of addresses, the execution of the return instruction will cause the HE machine to enter the *write waiting state*. In the write waiting state, the HE machine checks the write queue for enabled triggers. If a trigger is found, then the trigger is removed for the write queue and the HE machine resumes execution, interpreting HE code at address corresponding to the enabled trigger. If no enabled trigger is found in write queue, the HE machine enters the *switch waiting state*. In the switch waiting state, the HE machine checks for enabled triggers in the switch queue. If an enabled trigger is found, then that trigger is removed from the switch queue and the HE machine will continue interpreting HE code at address associated with the enabled trigger. If no enabled trigger is found, the HE machine will enter the *read waiting state*. In the read waiting state, the HE machine checks for enabled triggers in the read queue. If an enabled trigger is found, then it will be removed form the read queue, and HE machine will start interpreting HE code at address associated with the enabled trigger. If no enabled trigger is found, then the E machine will start waiting for events like time ticks or task completion, when the trigger queues has to be updated and checked again for enabled triggers. Since the three trigger queues are priorities, i.e., write queue has the highest priority and the read queue has the lowest priority, the triggers are priorities not only by the position in a queue but also by the queue in which they are.

Formally the HE machine is a tuple (*TF, DF, CF, state, TT, DT, CT, HEprogram, interpreter, PC, writeQ, switchQ, readQ, addressStack, parentStack, R0, R1 , R2, R3, tasks*), where *TF* represents task functions, *DF* represents driver functions, *CF* represent

condition functions, *state* represents the state, *TT* represents the table of tasks, *DT* represents the table of drivers, *CT* represents the table of conditions, *HEprogram* represents the list of HE code instructions, *interpreter* represents the HE code interpreter, *PC* represents the program counter, *writeQ*, *switchQ*, and *readQ* are queues of triggers, *addressStack* represents the stack of addresses, *parentStack* represents the stack of trigger references, *R0*, *R1* , *R2*, and *R3* are tigger registers, and *tasks* represents the list of released tasks.

The parts of the HE machine that change during HE code execution form the HE machine configuration. Thus the E machine configuration is defined as tuple (*state*, *PC*, *writeQ*, *switchQ*, *readQ*, *addressStack*, *parentStack*, *R0*, *R1* , *R2*, *R3*, *tasks*). First HE machine configuration is (*state* = $state_0$, *PC* = 0, *writeQ* = $\emptyset$, *switchQ* = $\emptyset$, *readQ* = $\emptyset$, *addressStack* = $\emptyset$, *parentStack* = $\emptyset$, *R0* = $\bot$, *R1* = $\bot$, *R2* = $\bot$, *R3* = $\bot$, *tasks* = $\emptyset$).

3.2.1. List of HE code Instructions

HE code interpretation can be seen as a sequence, $u0$, $u1$, ..., of HE machine configurations. Where $u0$ is the initial configuration. In the remaining of this subsection, each HE code instruction will be explained based on the effect it has on the HE machine configuration. Thus before executing an instruction the configuration is *state*, *PC*, *writeQ*, *switchQ*, *readQ*, *addressStack*, *parentStack*, *R0*, *R1* , *R2*, *R3*, *tasks*), and after the instruction has been executed the new configuration is (*state*`, *PC*`, *writeQ*`, *switchQ*`, *readQ*`, *addressStack*`, *parentStack*`, *R0*`, *R1*` , *R2*`, *R3*`, *tasks*`).

call(d): executes driver identify by driver index *d* (which updates variable state to *state'*), *PC* is incremented; the HE machine after interpreting this instruction is (*state*`, *PC*` = $PC + 1$, *writeQ*` = *writeQ*, *switchQ*` = *switchQ*, *readQ*` = *readQ*, *addressStack*` = *addressStack*, *parentStack*` = *parentStack*, *R0*` = *R0*, *R1*` = *R1* , *R2*` = *R2*, *R3*` = *R3*, *tasks*` = *tasks*);

release(t): releases for execution task identify by task index *t*, *PC* is incremented; the HE machine after interpreting this instruction is (*state*` = *state*, *PC*` = $PC + 1$, *writeQ*` = *writeQ*, *switchQ*` = *switchQ*, *readQ*` = *readQ*, *addressStack*` = *addressStack*, *parentStack*` = *parentStack*, *R0*` = *R0*, *R1*` = *R1* , *R2*` = *R2*, *R3*` = *R3*, *tasks*` = $tasks \cup t$);

writeFuture(e,a): creates a new trigger $g = (e, a, \bot, \emptyset)$, adds the trigger to the write queue, stores a reference to *g* in *R1*, and increments program counter; the HE machine configuration after executing this instruction is (*state*` = *state*, *PC*` = $PC + 1$, *writeQ*` = $writeQ \circ g$, *switchQ*` = *switchQ*, *readQ*` = *readQ*, *addressStack*` = *addressStack*, *parentStack*` = *parentStack*, *R0*` = *R0*, *R1*` = *referenceOf(g)* , *R2*` = *R2*, *R3*` = *R3*, *tasks*` = *tasks*);

switchFuture(e,a): creates a new trigger $g = (e, a, \bot, \emptyset)$, adds the trigger to the switch queue, stores a reference to *g* in *R1*, and increments program counter; the HE machine configuration after executing this instruction is (*state*` = *state*, *PC*` = $PC + 1$, *writeQ*` = *writeQ*, *switchQ*` = $switchQ \circ g$, *readQ*` = *readQ*, *addressStack*` = *addressStack*, *parentStack*` = *parentStack*, *R0*` = *R0*, *R1*` = *referenceOf(g)* , *R2*` = *R2*, *R3*` = *R3*, *tasks*` = *tasks*);

readFuture(e,a): creates a new trigger $g = (e, a, \bot, \emptyset)$, adds the trigger to the read queue, stores a reference to *g* in *R1*, and increments program counter; the HE machine configuration after executing this instruction is (*state*` = *state*, *PC*` = $PC + 1$, *writeQ*` = *writeQ*, *switchQ*` = *switchQ*, *readQ*` = $readQ \circ g$, *addressStack*` = *addressStack*, *parentStack*` = *parentStack*, *R0*` = *R0*, *R1*` = *referenceOf(g)* , *R2*` = *R2*, *R3*` = *R3*, *tasks*` = *tasks*);

jumpIf(cnd, a): executes condition function represented by condition index *cnd*, if condition function return true, then program counter is set to *a*, else program counter is incremented; after executed this instruction there are two possible configuration: if condition function returns true, then configuration is (*state`* = *state*, *PC`* = *a*, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack*, *parentStack`* = *parentStack*, *R0`* = *R0*, *R1`* = *R1* , *R2`* = *R2*, *R3`* = *R3*, *tasks`* = *tasks*), else configuration is (*state`* = *state*, *PC`* = *PC* + 1, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack*, *parentStack`* = *parentStack*, *R0`* = *R0*, *R1`* = *R1* , *R2`* = *R2*, *R3`* = *R3*, *tasks`* = *tasks*);

jumpAbsolute(a): performs an unconditioned jump to address *a*; the configuration after executing this instruction is (*state`* = *state*, *PC`* = *a*, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack*, *parentStack`* = *parentStack*, *R0`* = *R0*, *R1`* = *R1* , *R2`* = *R2*, *R3`* = *R3*, *tasks`* = *tasks*);

jumpSubroutine(a): executes a subroutine jump to address *a*; incremented value of the program counter is pushed onto the stack of addresses and the program counter is set to *a*; the configuration after executing this instruction is (*state`* = *state*, *PC`* = *a*, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack* $\circ$ (*PC*+1), *parentStack`* = *parentStack*, *R0`* = *R0*, *R1`* = *R1* , *R2`* = *R2*, *R3`* = *R3*, *tasks`* = *tasks*);

copyRegister(R_x, R_y) **where** $x, y \in \{0, 1, 2, 3\}$ **and** $x \neq y$: copies the content of register R_x to register R_y and program counter is incremented; configuration after executing this instruction is (*state`* = *state*, *PC`* = *PC* + 1, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack*, *parentStack`* = *parentStack*, *R0`*, *R1`* , *R2`*, *R3`*, *tasks`* = *tasks*), the values of trigger registers *R0`*, *R1`* , *R2`*, and *R3`* depends on the arguments of the instruction;

pushRegister(R_x) **where** $x \in \{0, 1, 2, 3\}$: pushes the content of the register R_x on to *parentStack*, the program counter is incremented; configuration after executing this instruction is (*state`* = *state*, *PC`* = *PC* + 1, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack*, *parentStack`* = *parentStack* $\circ$ $R_x$, *R0`* = *R0*, *R1`* = *R1* , *R2`* = *R2*, *R3`* = *R3*, *tasks`* = *tasks*);

popRegister(R_x) **where** $x \in \{0, 1, 2, 3\}$: pops content from *parentStack* to register R_x, i.e., $R_{x'}$ = *pop(parentStack)*, and increments program counter; configuration after executing this instruction is (*state`* = *state*, *PC`* = *PC* + 1, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack*, *parentStack`* = *parentStack* \ {*pop(parentStack)*}, *R0`*, *R1`* , *R2`*, *R3`*, *tasks`* = *tasks*), the values of trigger registers *R0`*, *R1`* , *R2`*, and *R3`* depends on the argument of the instruction;

getParent(R_x, R_y) **where** $x, y \in \{0, 1, 2, 3\}$ **and** $x \neq y$: loads the reference of parent trigger of trigger pointed to by R_x into register R_y and increments program counter; configuration after executing this instruction is (*state`* = *state*, *PC`* = *PC* + 1, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* = *addressStack*, *parentStack`* = *parentStack*, *R0`*, *R1`* , *R2`*, *R3`*, *tasks`* = *tasks*), the values of trigger registers *R0`*, *R1`* , *R2`*, and *R3`* depends on the argument of the instruction;

setParent(R_x, R_y) **where** $x, y \in \{0, 1, 2, 3\}$ **and** $x \neq y$: the trigger name in R_y is stored as the parent of the trigger pointed to by register R_x and the program counter is incremented; configuration after executing this instruction is (*state`* = *state*, *PC`* = *PC* + 1, *writeQ`* = *writeQ*, *switchQ`* = *switchQ*, *readQ`* = *readQ*, *addressStack`* =

addressStack, parentStack`* = *parentStack, R0` = *R0, R1`* = *R1* , *R2`* = *R2, R3`* = *R3, tasks`* = *tasks*);

copyChildren($Rx.Ry$) **where** $x, y \in \{0, 1, 2, 3\}$ **and** $x \neq y$**:** the children list of the trigger pointed to by Ry is stored as the children list of the trigger pointed to by register Rx and the program counter is incremented; configuration after executing this instruction is (*state`* = *state, PC`* = *PC* + 1, *writeQ`* = *writeQ, switchQ`* = *switchQ, readQ`* = *readQ, addressStack`* = *addressStack, parentStack`* = *parentStack, R0`* = *R0, R1`* = *R1* , *R2`* = *R2, R3`* = *R3, tasks`* = *tasks*);

updateChildren($Rx.Ry$) **where** $x, y \in \{0, 1, 2, 3\}$ **and** $x \neq y$**:** sets the trigger name in Ry as the parent of all the triggers in the children list of the trigger pointed by register Rx and increments the program counter; configuration after executing this instruction is (*state`* = *state, PC`* = *PC* + 1, *writeQ`* = *writeQ, switchQ`* = *switchQ, readQ`* = *readQ, addressStack`* = *addressStack, parentStack`* = *parentStack, R0`* = *R0, R1`* = *R1* , *R2`* = *R2, R3`* = *R3, tasks`* = *tasks*);

deleteChildren(Rx) **where** $x \in \{0, 1, 2, 3\}$**:** for all trigger references in children list of trigger referred by register Rx: (recursively) deletes the triggers pointed by the children list and remove the triggers from the queue and increments the program counter; configuration after executing this instruction is (*state`* = *state, PC`* = *PC* + 1, *writeQ`* = *writeQ, switchQ`* = *switchQ, readQ`* = *readQ, addressStack`* = *addressStack, parentStack`* = *parentStack, R0`* = *R0, R1`* = *R1* , *R2`* = *R2, R3`* = *R3, tasks`* = *tasks*);

replaceChild($Rx.Ry.Rz$) **where** $x, y, z \in \{0, 1, 2, 3\}$ **and** $x \neq y \neq z$**:** in the children list of trigger pointed to by register Rx, replaces the trigger reference in Ry by the trigger reference in Rz and increments the program counter; configuration after executing this instruction is (*state`* = *state, PC`* = *PC* + 1, *writeQ`* = *writeQ, switchQ`* = *switchQ, readQ`* = *readQ, addressStack`* = *addressStack, parentStack`* = *parentStack, R0`* = *R0, R1`* = *R1* , *R2`* = *R2, R3`* = *R3, tasks`* = *tasks*);

cleanChildren(Rx) **where** $x \in \{0, 1, 2, 3\}$**:** deletes the children list of trigger pointed by register Rx and increments the program counter; configuration after executing this instruction is (*state`* = *state, PC`* = *PC* + 1, *writeQ`* = *writeQ, switchQ`* = *switchQ, readQ`* = *readQ, addressStack`* = *addressStack, parentStack`* = *parentStack, R0`* = *R0, R1`* = *R1* , *R2`* = *R2, R3`* = *R3, tasks`* = *tasks*);

return()**:** pops an address from address stack into the program counter; *PC`* = *pop*(*addressStack*); configuration after executing this instruction is (*state`* = *state, PC`* = *pop*(*addressStack*), *writeQ`* = *writeQ, switchQ`* = *switchQ, readQ`* = *readQ, addressStack`* = *addressStack, parentStack`* = *parentStack, R0`* = *R0, R1`* = *R1* , *R2`* = *R2, R3`* = *R3, tasks`* = *tasks*);

3.2.2. Handling Hierarchy in HE code

Sequential composition in E code is supported through conditional jump instruction, parallel composition is supported through future zero instruction, the only HTL feature that is not supported by the E code is the refinement. Thus the E code has been extended into HE code in order to support refinement. In this subsection it is presented how hierarchical structure is handled in HE code.

The main idea is to build an implicit tree of triggers. The hierarchical relations between the triggers reflect the hierarchical relations between the HTL modes for which the HE code that created the triggers has been generated. The implicit tree structure is needed only to stop executing modes that refine a mode **m** that has switched to another mode **m`**. Since in the switch queue, for each mode that is executed it is placed one

and only one trigger, which will ensure that the mode is executed periodically, and all the triggers related to a mode that are in the write queue or read queue will activate by the end of the mode period, it is enough to create a tree like structure only with the triggers that are in the switch queue. Thus for each mode m there will be a trigger in the switch queue, which has as a parent the trigger from the switch queue that corresponds to the direct parent mode of m, and as children all the triggers that corresponds to the direct child modes of m. The trigger of the parent mode is added to the trigger queue before the trigger of the child mode. Thus if the parent mode switches to a different mode, the triggers corresponding to the child modes will be removed recursively from the queue using the tree information, this will ensure that if the parent mode switches, then all its child modes stop executing.
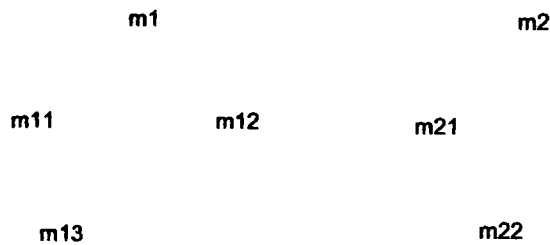
Fig. 3.4: Handling hierarchy: HTL description example

Fig. 3.4, depicts in simplified (i.e., only the modes and hierarchical relations between them are presented) visual syntax the structure of an HTL description that is used as an example to explain how hierarchical structure is handled in HE code. The description consists of modes $m1$ and $m2$, which are composed sequentially. Mode $m1$ is refined by modes $m11$ and $m12$, which are composed in parallel. Mode $m13$ refines mode $m11$. Mode $m2$ is refined be mode $m21$, which is farther refined by mode $m22$.
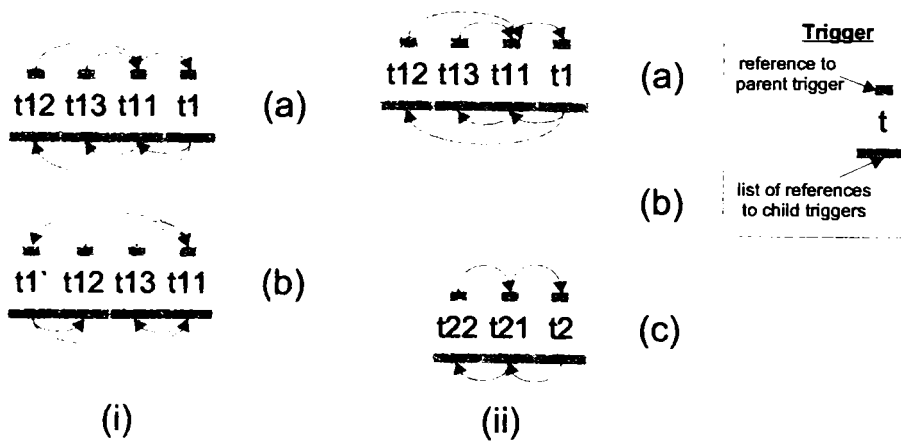
Fig. 3.5: Handling hierarchy: implicit tree evolution

In Fig. 3.5 it is presented the evolution of the switch queue in two situations that

can happen when HE code for mode $m1$ is interpreted: mode $m1$ does not switch to $m2$ and mode $m1$ switches to $m2$. For both scenarios has been considered that mode $m1$ is the start mode. Thus initially in the switch queue there is a trigger for mode $m1$ and all its child modes, e.g., triggers $t1$, $t11$, $t13$, and $t12$ correspond to modes $m1$, $m11$, $m13$, and $m12$, respectivelly. In the first situation (Fig. 3.5-I), since the switch to mode $m2$ is not enabled during HE code interpretation, a new trigger $t1`$ is created; $t1`$ replaces trigger $t1$ in the switch queue and will preserve all its connections to other triggers. On the other hand if the mode switch happens (Fig. 3.5-II), then all the triggers that are children of trigger $t1$ are removed from the switch queue, i.e., in this case all the triggers are removed, and new triggers, corresponding to the new active mode and its child modes, are created and added to the switch queue. When a sub-tree of modes from an HTL description is started for the first time the modes are executed in the top-down and left-to-right order, this is way trigger $t13$ is added to the switch queue before trigger $t12$. When a trigger that is added to the switch queue is created, its parent will be automatically set to the parent of the trigger referred by register $R0$. During the initial execution of a sub-tree of modes, the parent stack is used to store the parent triggers, as the execution goes down the sub-tree of modes, e.g., when trigger $t13$ is created the parent of trigger referred by $Rzero$ si $t11$ and in the parent stack there is one trigger reference, which points to trigger $t1$. When the execution returns the parent of the trigger referred by $R0$ is restored from the stack, i.e., when trigger $t12$ is created the parent of the trigger referred by $R0$ is trigger $t1$.

4. HTL Compiler

Fig. 4.1 depicts visually the actions that happen at compile-time and at runtime. In general, the compilation of any HTL description involves a specific analysis of the description and then code generation. The analysis ensures that the input description is well-formed, i.e., it satisfies the constraints on parallel composition of modules and refinement of modes presented in Section 2.3, and that the description is schedulable relative to the target platform (well-timed). For the schedulability analysis, the WCET/ WCTT information for tasks is provided by an external tool. If all the constraints are satisfied, code generator generates code for a distributed implementation. Code generation is done by compiling the whole description for each host. For each host it is maintained a set of copies of all communicators and ports. However, tasks are executed on the host only if the corresponding mode (in which the task is invoked) is mapped onto that host. At runtime, whenever a task completes execution, all output ports of that task are broadcasted to all hosts; each host will store the value locally. When a communicator (on a host) is to be written with the value from a task output port, the locally stored value of the output port is copied to the communicator. Released tasks are dispatched for execution by an EDF scheduler; the scheduler is external to the E machine (and HE machine).
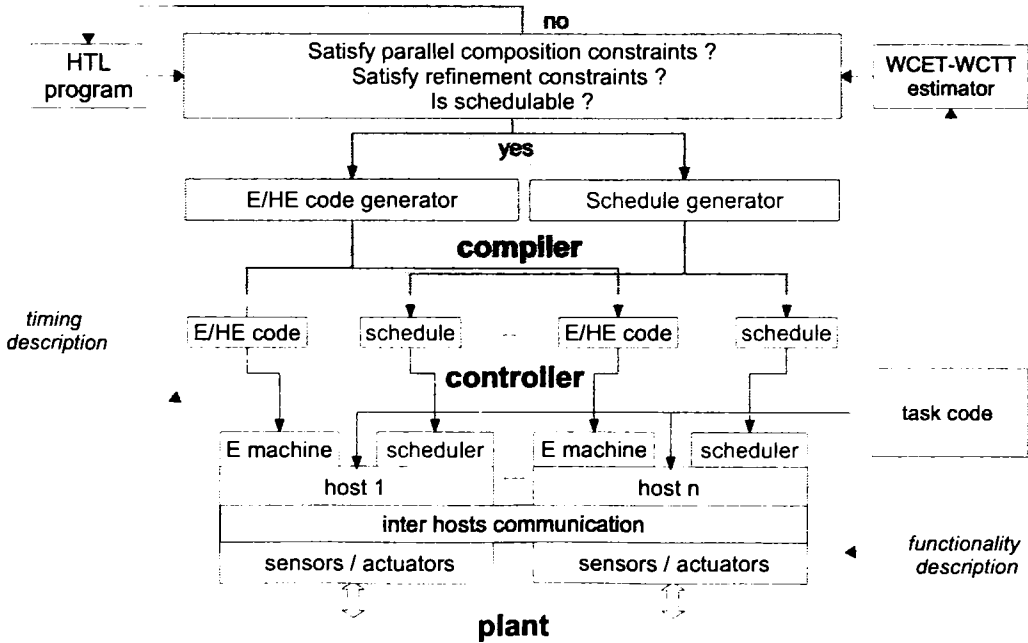
Fig. 4.1: Structure of compiler and runtime system

In this chapter there will be presented two HTL compilers, i.e., flattening HTL compiler [2] and hierarchy-preserving HTL compiler [7]. Flattening HTL compiler needs to flatten the hierarchical structure of an HTL description before generating E code for it; the target platform is represented by the E machine (Section 3.1). Hierarchy-preserving HTL compiler generates HE code directly from the input HTL description without altering its structure, it relays on hierarchical support at runtime; the target platform is represented by the HE machine (Section 3.2).

4.1. Flattening HTL Compiler

After parallel composition constraints, refinement constraints, and schedulability have been checked on the input HTL description, the flattening HTL compiler transforms the input HTL description into a flat HTL description by applying a flattening algorithm. The flattening algorithm consists of traversing the hierarchical structure of the input HTL description bottom-up and left-to-right, merging all the modules in the same child program into a single module that contains all possible combination of modes from the merged modules, and then the resulted module is merged with the parent mode of the merged modules and the resulted set of modes will replace the parent mode. Modules in the root program are not merged. The above presented flattening algorithm is possible since all the modes in child programs have the same period as the parent mode from the root program.

Given a well-formed, well-timed, schedulable, and flattened HTL description and a mapping of its top-level modules to hosts, the HTL compiler generates E code for the description and mapping by invoking Alg. 4.1 on its root program (which is also the single program in a flatten HTL description) for each host to which modules from the description are mapped; the algorithm invokes Alg. 4.2 to generate E code for each module of the program, which finally invokes Alg. 4.3 to generate E code for each mode of each module. The compiler conceptually divides each mode into uniform temporal segments called units. The *unit* of a mode is the smallest time interval at which any two consecutive communicator instances are accessed in the mode. Given a mode m, the duration of its unit is denoted by $\gamma[m]$, which is the gcd of all access periods of all communicators accessed in m. The total number of units of m is $\pi[m]/\gamma[m]$, where $\pi[m]$ is the period of m. The compiler generates separate E code blocks for each unit of a mode. The address of an E code block corresponding to unit i of a mode m is denoted by *unit_address*$[m, i]$. This is a symbolic address to which instructions may forward reference and therefore may need fix up during compilation. Similar notation for other symbolic addresses, are used.

The following auxiliary operators are used in order to present compiling algorithm for the flattening HTL compiler. The driver $\texttt{init}(x)$ initializes the communicator or task port x. The set $\texttt{readDrivers}(m, i)$ contains the drivers that load the tasks in mode m with values of the communicators that are read by these tasks at unit i. The set $\texttt{writeDrivers}(m, i)$ contains the drivers that load the communicators with the output of the tasks in mode m that write to these communicators at unit i. The set $\texttt{portDrivers}(t)$ contains the drivers that load input ports of task t with the values of module ports on which t depends. The set $\texttt{complete}(t)$ contains the events that signal the completion of the tasks on which task t depends, and that signal the read time of the task t. The set $\texttt{releasedTasks}(m, i)$ contains the tasks in mode m with no precedences that are released at unit i. The set $\texttt{precedenceTasks}(m)$ contains the tasks in mode m with precedences.

Alg. 4.1 generates instructions to initialize all communicators and modules. Here *future*$(0, module\_address[M])$ instruction is used to start parallel execution of all modules in the root program. The actual mechanism is relatively complicated: for a *future*$(0, a)$ instruction the E machine appends the already enabled trigger-address pair $(\texttt{true}, a)$ to the

---

Alg. 4.1 GenerateECodeForProgramOnHost(P.h)

// initialize communicators
$\forall c \in$ communicators(P):*emit*(*call*(init(c)))
// initialize and start each module
$\forall M \in$ modules(P):*emit*(*future*(0.*module_address*[M]))
// end initialization phase
emit(*return*)
// generate code for each module
$\forall M \in$ modules(P):*GenerateECodeForModuleOnHost*(M.h)

---

trigger queue and then proceeds to the next instruction. Only when the E machine reaches a *return* instruction, the machine checks the trigger queue again and eventually removes the pair (true.a) from the trigger queue and executes the E code at the address a, but not before it has executed the E code of all other enabled trigger-address pairs occurring before (true.a) in the queue.

Alg. 4.2 GenerateECodeForModuleOnHost(M,h)

---

set *module_address*[M] to *PC* and fix up
// initialize task ports
$\forall p \in$ taskPorts(M):*emit*(*call*(init(p)))
// jump to the start mode at unit 0
emit(*jump*. *unit_address*[start[M].0])
// generate code for each mode
$\forall m \in$ modes(M):*GenerateECodeForModeOnHost*(m,h)

Alg. 4.2 generates instructions to initialize all task ports in a module, and to start the execution of the module by jumping to the E code of the first unit of the start mode of the module. *PC* denotes the program counter of the compiler.

Alg. 4.3 generates the E code for all units of a mode. Only unit 0 contains instructions to check mode switching because mode switching may only occur at the beginning of a mode period. When a mode switch occurs, E code execution continues at the *mode_address*[m'] of the target mode m', not the *unit_address*[m',0], since only at most one mode switch per time instant may occur. At each time instant, the generated E code uses *future*(0,a) instructions to write communicators always before any communicator is read making sure that the latest communicator values are used across all modules. Communicator and port values do not need to be buffered since tasks are invoked at most once per mode period and communicator-to-port transactions are done as soon as possible while port-to-communicator transactions are done as late as possible. It is therefore sufficient to have a single copy of each communicator and task port on each host.

4.2. Hierarchy-Preserving HTL Compiler

The hierarchy-preserving compiler generates code for programs, modules, and modes by invoking Alg. 4.4, Alg. 4.5, and Alg. 4.6, respectively. The compiler uses symbolic addresses to refer to different parts of the code. For each program P, *programInit*[P] (resp. *programStart*[P]) denotes the address of the HE code block that initializes (resp. executes) P. For each module M, *moduleInit*[M] (resp. *moduleStart*[M]) denotes the address of the HE code block that initializes (resp. executes) M. For each mode m, *modeStart*[m] is the

---

Alg. 4.3 GenerateECodeForModeOnHost(m, h)

$i := 0$
while $i < \pi[m]/\gamma[m]$ **do**
  set *unit_address*$[m, i]$ to *PC* and fix up
  // update communicators with task output
  $\forall d \in$ writeDrivers(m, i):*emit*(*call*(d))
  // continue after other modules updated communicators
  emit(*future*($0, PC + 2$))
  emit(*return*)
  if ($i = 0$)
    // check mode switches
    $\forall(cnd, m') \in$ switches(m):*emit*(*if*(cnd.*mode_address*$[m']$))
    set *mode_address*$[m]$ to *PC* and fix up
  end if
  if (mode m is contained in a module on host h)
    // read communicators into tasks
    $\forall d \in$ readDrivers(m, i):*emit*(*call*(d))
    // release tasks with no precedences
    $\forall t \in$ releasedTasks(m, i):*emit*(*release*(t))
    if ($i = 0$)
      // release tasks with precedences
      $\forall t \in$ precedenceTasks(m):
        // wait for tasks on which t depends to complete
        emit(*future*(complete(t), $PC + 2$))
        emit(*jump*($PC + 5 + |$portDrivers$(t)|$))
        // release t after other modules updated communicators
        emit(*future*($0, PC + 2$))
        emit(*return*)
        // read ports of tasks on which t depends, then release t
        $\forall d \in$ portDrivers(t):*emit*(*call*(d))
        emit(*release*(t))
        emit(*return*)
    end if
  end if
  // continue mode after $\gamma[m]$ time
  emit(*future*($\gamma[m]$, *unit_address*$[m, i + 1$ mod $\pi[m]/\gamma[m]]$))
  emit(*return*)
  $i := i + 1$
end while

---

address of the HE code block that starts m, and *targetMode*[m] is the address of HE code block that is executed when another mode switches to m. Each mode m is divided in uniform units corresponding to the smallest period between two time events (i.e., write of a communicator or read of a communicator) in m. Given a mode m, the duration of an unit $\gamma[m]$ is the greatest common divisor of all access periods of all communicators read or written in m; the total number of units is $\pi[m]/\gamma[m]$, where $\pi[m]$ is the period of m. For each unit i of every mode m the compiler generates separate code blocks for updating communicators, checking switches (and related actions), and reading communicators (and releasing tasks): the address of the HE code block that writes communicators is *unitWrite*[m, i], the address of the HE code block that checks switch condition is *unitSwitch*[m, i], and the address of the HE code block that reads communicators is *unitRead*[m, i]. HTL semantics constraints that at any instance, communicator writes, mode switch checks, communicator reads, and task releases should be done in the above order to maintain consistency of communicator values across all modules. The address of the HE code block that sets up the execution order of communicator writes, switch checks, and communicator reads (and task releases) is *modeBody*[m]. Instructions may forward reference to any of the above symbolic addresses which may require fix up during compilation.

Alg. 4.4 GenerateECodeForProgramOnHost(P, h)

---

set *programInit*[P] to *PC* and fix up
// initialize communicators
$\forall c \in$ communicators(P):*emit*(*call*(init(c)))
// initialize all the modules in P
$\forall M \in$ modules(P):*emit*(*jumpSubroutine*(*moduleInit*[M]))
// return from initialization subroutine of P
emit(*return*)
set *programStart*[P] to *PC* and fix up
// start all the modules in P
$\forall M \in$ modules(P):*emit*(*jumpSubroutine*(*moduleStart*[M]))
// return from start subroutine of P
emit(*return*)

Alg. 4.4 generates code for a program P on a host h. The code at address *programInit*[P] initializes all communicators declared in P by calling corresponding initialization drivers (init(·) denotes the initialization driver for a communicator or a port), then it calls initialization subroutine of each module in P. Code at address *programStart*[P] calls the start subroutine of each module M in P.

Alg. 4.5 GenerateECodeForModuleOnHost(M, h)

---

set *moduleInit*[M] to *PC* and fix up
// initialize task port
$\forall p \in$ taskPorts(M):*emit*(*call*(init(p)))
// return from initialization subroutine of M
emit(*return*)
set *moduleStart*[M] to *PC* and fix up
//start the start mode of M
emit(*jumpSubroutine*(*modeStart*[start[M]]))
// return from start subroutine of M
emit(*return*)

Alg. 4.5 generates code for a module M on host h. Code at address *moduleInit*[M] initializes all task ports (denoted by taskPorts(M)) of the tasks in M by calling respective initialization drivers. All tasks maintain two sets of local ports, called *task input ports* and *task output ports*, which are not accessible by other tasks. Before a task is released, values of the communicators and ports that are read by the released task are copied into task input ports, which are used by task at execution. At completion, the task output ports are updated; communicators and ports are written from the task output ports when the writing is due. Code at *moduleStart*[M] calls the start subroutine of the start mode, start[M], of M.

The following auxiliary operators are used for Alg. 4.6. The set readDrivers(m, *i*) contains the drivers that load the tasks in mode m with values of the communicators that are read by these tasks at unit *i*. The set writeDrivers(m, *i*) contains the drivers that load the communicators with the output of the tasks in mode m that write to these communicators at unit *i*. The set portDrivers(*t*) contains the drivers that load task input ports of task *t* with the values of the ports read by *t*. The set complete(*t*) contains the events that signal the completion of the tasks on which task *t* depends, and that signal the read time of the task *t*. The set releasedTasks(m, *i*) contains the tasks invoked in mode m, with no precedences, that are released at unit *i*. The set precedenceTasks(m) contains the tasks in mode m that depend on other tasks.

Alg. 4.6 GenerateECodeForModeOnHost(m, h)

---

0 GenerateECodeToStartModeOnHost(m, h);
1 $i := 0$
2 **while** $i < \pi[m]/\gamma[m]$ **do**
3   set *unitWrite*[m, *i*] to *PC* and fix up
4   // write communicators with the values of task output ports
5 $\forall d \in$ writeDrivers(m, *i*):*emit*(*call*(*d*))
6   // wait for other triggers to become enabled
7   *emit*(*return*)
8   **if** ($i = 0$)
9   GenerateECodeToTestSwitchesForModeOnHost(m, h)
10   GenerateECodeToPreserveHierarchyForModeOnHost(m, h)
11   *emit*(*return*)
12   **end if**
13   GenerateECodeToReleaseTasksForModeOnHost(m, h, i)
14   **if**($i < \pi[m]/\gamma[m] - 1$)
15   // jump to the next unit of mode m
16   *emit*(*writeFuture*($\gamma[m]$, *unitWrite*[m, *i* + 1]))
17   *emit*(*readFuture*($\gamma[m]$, *unitRead*[m, *i* + 1]))
18   **end if**
19   // wait for other triggers to become enabled
20   // OR return from body subroutine of m
21   *emit*(*return*)
22 $i := i + i$
23 **end while**

Alg. 4.6 starts by invoking Alg. 4.7, which emits HE code to start a mode. Alg. 4.7 emits code (at address *modeStart*[m]) for checking all the mode switches (lines **1 – 3**) in a mode m, so that they are tested first time m is invoked. Next, code is generated (at address *targetMode*[m]) to handle the case when no switch is enabled: a call to code at *modeBody*[m], followed by a call to the refinement program (if any). This sets the execution

Alg. 4.7 GenerateECodeToStartModeOnHost(m, h)

0 set *modeStart*[m] to *PC* and fix up
1 // check mode switches
2 $\forall(cnd.m') \in$ switches(m):
3   *emit(jumpIf(cnd.targetMode*[m']))
4 set *targetMode*[m] to *PC* and fix up
5 *emit(jumpSubroutine(modeBody*[m]))
6 **if** (program P refines m)
7   //increment the level
8   *emit(getParent(R0.R3))*
9   *emit(pushRegister(R3))*
10   *emit(setParent(R0.R2))*
11   *emit(cleanChildren(R0))*
12   *emit(jumpSubroutine(programStart*[program[m]]))
13   //decrement the level
14   *emit(popRegister(R3))*
15   *emit(setParent(R0.R3))*
16   *emit(cleanChildren(R0))*
17 **end if**
18 // return from start subroutine of m
19 // OR wait for other triggers to become enabled
20 *emit(return)*

of a mode before the execution of the refinement program. Code emission at lines **6 – 17** checks whether a refinement program exists and subsequently updates the hierarchy information if there is one. Before the code for refinement program is invoked (line **12**), the hierarchy is updated (lines **7 - 11**), as refinement adds one level of hierarchy; once the invocation of the refinement program completes, the level is restored (lines **13 - 16**). Update of hierarchy consists of pushing the parent of the trigger that is referred by the trigger name in register *R0* onto the stack (lines **8 - 9**); the parent of the trigger pointed by *R0* is then changed to the trigger pointed by *R2* (which contains a pointer to the last trigger added to the switch queue), and children list is reset (code for refinement program has yet to be invoked, thus there is no children information). In effect, for the execution of the refinement program, parent of trigger pointed by *R0* points to the parent trigger of all the triggers that will be added in the switch queue for that program. To restore the hierarchy level, the parent of trigger pointed by *R0* is updated by popping the parent stack.

After the call to Alg. 4.7 returns, Alg. 4.6 generates E code for each mode unit *i*. Lines **4 - 7** generates E code that calls the driver for each communicator being written at unit *i*. Next Alg. 4.8 is invoked for mode unit zero to generate code that tests mode switch conditions, and takes necessary action when a switch is enabled. In HTL, modes can switch only at period boundaries; thus the switches are checked only for unit zero. If no mode switch occurs (line **4**), then the code jumps to *modeBody*[m]. If a mode switch occurs, then all children of the last enabled trigger in the switch queue (its name is stored in register *R0*) are removed from switch queue (lines **5 - 8**). The removal of children is recursive, thus all children of subsequent children are also removed. Once the children are removed, the code execution jumps (lines **9 - 10**) to the target address of the destination mode *targetMode*[m'], where m' is the destination mode.

After code for mode switches has been generated, Alg. 4.9 is invoked to sequence the execution order of communicator writes, switch checks, and communicator reads (and

---

Alg. 4.8 GenerateECodeToTestSwitchesForModeOnHost(m, h)

0 set *unitSwitch*$[m, 0]$ to *PC* and fix up
1 // check mode switches
2 $\forall (cnd, m') \in$ switches(m):
3   *emit(jumpIf(cnd.PC + 2))*
4   *emit(jumpAbsolute(PC + 4))*
5   // cancel all triggers related to the refining
6   // program of m, and its subprograms
7   *emit(deleteChildren(R0))*
8   *emit(cleanChildren(R0))*
9   // switch to mode m'
10 *emit(jumpAbsolute(targetMode*$[m']$*))*

---

subsequent task release), for unit zero of mode m. This is done by emitting a future instruction (line **1**) for *unitWrite*$[m, 0]$ (trigger will be added to *writeQ*), a future instruction (line **2**) for *unitSwitch*$[m, 0]$ (trigger will be added to *switchQ*), and a future instruction (line **9**) for *unitRead*$[m, 0]$ (trigger will be added to *readQ*). Whenever a trigger is created and added to a queue, the relevant trigger pointer is stored in register *R1*. Once a trigger is added in the switch queue, the hierarchy information has to be updated (lines **3 - 8**). Depending on how the code is reached, there are two scenarios: first, the code is invoked by handling an enabled trigger in the switch queue, i.e., a mode switch has occurred or a mode is being reinvoked, and second, the code is invoked when a mode is executed for the first time. In both scenarios register *R0* records the relevant hierarchy information. In the first scenario, it stores the name of the last trigger in the switch queue that was handled (by semantics, if any trigger is handled the name is stored in *R0*). In the second scenario, it stores the name of the last trigger in the switch queue that was created. Code at lines **3 - 7** sets the parent of the trigger pointed by *R0* as a parent for the trigger pointed by *R1*, and copies the children list from the trigger pointed by *R0* to the trigger pointed by *R1*. A new trigger for the read queue may remove the information of the last trigger added to the switch queue from *R1*; so a copy of *R1* is stored in *R2* (line **8**).

Alg. 4.9 GenerateECodeToPreserveHierarchyForModeOnHost(m, h)

0 set *modeBody*$[m]$ to *PC* and fix up
1 *emit(writeFuture($\pi[m]$.unitWrite$[m, 0]$))*
2 *emit(switchFuture($\pi[m]$.unitSwitch$[m, 0]$))*
3 *emit(getParent(R0.R3))*
4 *emit(replaceChild(R3.R0.R1))*
5 *emit(updateChildren(R0.R1))*
6 *emit(setParent(R1.R3))*
7 *emit(copyChildren(R1.R0))*
8 *emit(copyRegister(R1.R2))*
9 *emit(readFuture(0, unitRead$[m, 0]$))*

Once the call to Alg. 4.9 returns, Alg. 4.10 is invoked. Alg. 4.10 generates code (at *unitRead*$[m, i]$) that reads (lines **2 - 3**) all communicators (by calling drivers that copy from communicators into task input ports) that are to be read at unit i, and releases all tasks (with no precedences) that should be released at unit i. For unit zero (line **6**), code is generated to release precedence tasks (lines **7 - 15**). For each task t with precedences, a trigger is added to *readQ*: the trigger is activated at the completion of preceding tasks

of t, and the subsequent code writes input ports of t and releases t.

Last stage in code generation for an unit of mode m consists of emitting code to jump from one unit to the next (lines (**14 - 18**), Alg. 4.6). The generated code adds triggers to the write queue and the read queue only. Switches are not possible in other units except unit zero. For unit zero the future instructions that ensure continuity of execution are generated by Alg. 4.9.

Alg. 4.10 GenerateECodeToReleaseTasksForModeOnHost(m, h, i)

---

0 set *unitRead*$[m.i]$ to *PC* and fix up
1 **if** (mode m is contained in a module on host h)
2   // read communicators into task input ports
3 $\forall d \in \texttt{readDrivers}(m, i): emit(call(d))$
4   // release tasks with no precedences
5 $\forall t \in \texttt{releasedTasks}(m, i): emit(release(t))$
6   **if** $(i = 0)$
7   // release tasks with precedences
8 $\forall t \in \texttt{precedenceTasks}(m):$
9   // wait for tasks on which t depends to complete
10 $emit(readFuture(\texttt{complete}(t). PC + 2))$
11 $emit(jumpAbsolute(PC + 3 + |\texttt{portDrivers}(t)|))$
12   // read ports of tasks on which t depends,
13   // then release t
14 $\forall d \in \texttt{portDrivers}(t): emit(call(d))$
15 $emit(release(t))$
16   // wait for other triggers to become enabled
17 $emit(return)$
18   **end if**
19 **end if**

The code generation algorithm for programs/ modules/ modes accesses other programs, modules, or modes through symbolic addresses and does not influence the code generation of other programs, modules, and modes. Thus parts of HTL programs can be compiled in any order separately.

4.3. Compilers Analysis

Complexity of the two HTL compilers, flattening and hierarchy-preserving, are analyzed in terms of efficiency of code generation and runtime overhead. Efficiency of code generation is measured by the number of instructions generated (by a compiler) for a given HTL description. The runtime overhead is the average time spent at each instant in executing instructions and searching for enabled triggers; thus the overhead is measured as the number of instructions that have to be interpreted per instant, and the number of triggers in the trigger queues.

In this rest of this section the two HTL compiler algorithms are compared both analytically and experimentally. First, the lower bound and upper bound, of the complexity of each compiler algorithm are identified intuitively, next a more detailed analytical comparison is presented, and in the end there are presented some experimental results, which confirm the analytical statement.

4.3.1. Overview on the Complexity of the Two Compiler Algorithms

Given an HTL description that specifies, $p \in \mathbb{N}_{>0}$ programs, $m \in \mathbb{N}_{\geq p}$ modules, and $n \in \mathbb{N}_{>0}$ modes per module, Table 4.1 presents the space complexity and the runtime complexity of both flattening HTL compiler and hierarchy-preserving HTL compiler. The bounds presented in Table 4.1 represent upper bounds. However for the hierarchy-preserving HTL compiler these bounds are also lower bounds. For the flattening HTL compiler the upper bounds for the generated code size and for the number of instructions that have to be interpreted per instant are also lower bounds for an HTL description that consists of $p \in \mathbb{N}_{>0}$, $m \in \mathbb{N}_{>p}$ modules, and $n \in \mathbb{N}_{>0}$ modes per module, for which all the HTL programs except an HTL program P that is a direct child of the root HTL program, contain one module, and for which program P contains the remaining $m - p + 1$ modules. The upper bound for the number of triggers in trigger queue presented in Table 4.1 for the flattening HTL compiler is a lower bound for the same compiler for an HTL description that consists of $p \in \mathbb{N}_{>0}$, $m \in \mathbb{N}_{\geq p}$ modules, and $n \in \mathbb{N}_{>0}$, and for which all the HTL programs except the root HTL program contain one module, and the root HTL program contains $m - p + 1$ modules.

The focus of the analysis is to understand the handling of hierarchy by the two compilation algorithms. The code generation for handling communicators, ports, and task invocations is the same for both the compilers, thus the non-hierarchy description is kept to a minimum as follows: the root program contains one communicator declaration; non-root programs do not contain communicator declarations; and modes do not invoke any task. Any mode m in a module can switch to any other mode m' in the module.

compiler	code size	instructions/instant	triggers in queue(s)
flattening	$\mathcal{O}(n^{m-p})$	$\mathcal{O}(n^{m-p})$	$\mathcal{O}(m - p)$
hierarchy-preserving	$\mathcal{O}(mn^2)$	$\mathcal{O}(mn)$	$\mathcal{O}(m)$

Tab. 4.1: Comparison of the two compilers

The generated E code size increases exponentially with the number of modules for the flattening compiler, while the size of generated HE code is linear bounded for the hierarchy-preserving compiler. The exponential explosion in the case of the flattening compiler is caused by the flattening process, which in order to merge the refining program into the refined mode computes all possible combinations between the modes in parallel modules from the refining program. The combination of modes generates an exponential number of new modes and mode switches, which in the end leads to an exponential growth of the number of instructions. On the other hand the number of generated instructions decreases exponentially with the number of programs; this is due to the fact that as the number of programs increases, for a constant number of modules, the number of parallel modules in the refinement decreases, thus there are fewer modules to be merged together, and less modes will be generated. The hierarchy-preserving compiler generates the same number of instructions for each mode no matter where it is declared in the hierarchy. Thus for n modes per module (where each mode can switch to any of the other $n - 1$ modes), the number of generated instructions is $\mathcal{O}(mn^2)$.

The second column in Table 4.1 compares the number of triggers in trigger queues. In case of the flattening compiler, there will be one trigger in the trigger queue for each parallel module in the root program of an HTL test description; i.e., in the worst case there can be $m - p$ triggers in the trigger queue. For the hierarchy-preserving compiler, there will be one trigger in each of the three trigger queues (i.e., write queue, switch queue, and read queue) for each mode. Thus in the worst case (when all modules are executed) the number of triggers in the queues is $\mathcal{O}(m)$ for the hierarchy-preserving compiler.

The number of instructions that have to be executed per instant is given by the

number of mode switches that have to be checked in the worst case. Since the number of mode switches in a mode for a flattened program is exponential in terms of number of parallel modules in the refinement, the number of instructions that have to be executed per instant is also exponential in terms of number of parallel modules in the refinement. For the hierarchy-preserving compiler, the E machine checks (in the worst case) all modes switches for the active mode in each of the modules, which means that in the worst case the number of instructions is bound by $\mathcal{O}(nm)$ for each instant.

4.3.2. Detailed Complexity Analysis

For the detailed analysis, an HTL test description that contains $p \in \mathbb{N}_{>0}$ programs, $m \in \mathbb{N}_{\geq p}$ modules, and $n \in \mathbb{N}_{>0}$ modes per module, has been considered as an input for both compilers. All the modes in the test description contain no task invocation and any mode can switch to any other mode in the containing module.

For this kind of HTL descriptions, formulas will be presented for calculated the worst case generated code size, worst case number of instructions interpreted per instant, and worst case triggers in queues; the formulas will get as an input parameters p, m, and n, which describe a class of HTL test description that have p programs, m modules, and n modes per module, but which vary in the hierarchical structure.

4.3.2.1. Worst Case Generated Code Size

Flattening HTL compiler. In the case of flattening HTL compiler finding a formula that can describe the worst case generated code size for an arbitrary number of programs is very difficult, due to the fact that the flattening algorithm is highly nonlinear. Thus for this compiler it has been considered only the case when the HTL description has two programs, a similar analysis can be performed for any number of programs not just for two, but combining the results in a single formula that will apply for an arbitrary number of program is very difficult. As presented in Section 4.3.1, the flattening HTL compiler generates less instructions as the number of programs increases due to the fact that the degree of parallelism in the refinement decreases as the number of program increases, and since for an HTL description with a single program, the flattening algorithm does not affect at all the code generation (i.e., the program is already flat), it is obvious that for the flattening HTL compiler the worst case generated code size is for an HTL description that has two programs and for which all the $m-1$ modules are in the refining program (e.g., highest degree of parallelism in the refinement).

In order to determine the size of generated code for an HTL description with the flattening HTL compiler, two analyses have to be performed. In the first analysis the flattening algorithm is analyze in order to determine the number of programs, modules, and modes in the flatten HTL description. In the second analysis the compilation algorithm is analyzed in order to determine the size of the generated code.

(a) Flattening algorithm. After an HTL description, which has two programs and m modules, out of which $m-1$ are in the child program, is flatten, the number of programs is reduce to one, and the number of modules is reduced to the number of modules in the root program. Nevertheless the number of modes and modes switches in the flatten description is still unknown.

The first step in flattening an HTL description consists of merging all the modules in the refining program into a single module, this merging process will generate a mode for every possible combination between the modes in the $m-1$ modules from the refining program, thus the number of modes in the merged module is n^{m-1}. Knowing that in each of the $m-1$ modules, any mode can switch to any other mode, then any mode generated from merging together the $m-1$ modules should also be able to switch to any other

generated mode, this gives a total of $n^{m-1} - 1$ mode switches per generated mode. Thus after first step the refining program will contain a single module with n^{m-1} modes and $n^{m-1}(n^{m-1} - 1)$ mode switches.

The second step in flattening the program consists of replacing the refined mode with the modes in the refining program. After this step, the number of modes in top program will be the sum between the number of modes from the previous step and the number of modes that are already in the top level module (i.e., n) minus one (e.g., the mode that will be replaced with the modes from the refining program). As for the mode switches, since the refined mode could switch to $n - 1$ modes it means that all the modes that replace it, also have to be able to switch to any of the $n - 1$ modes in the top module, this gives $n - 1$ more mode switches for each mode that comes from the refining program (thus number of mode switches in a modes that comes from refining program is $n^{m-1}(n - 1) + n^{m-1}(n^{m-1} - 1)$). If we now consider the mode switches that are in the top level modes (i.e., $(n - 1)^2$) we get a total of $(n - 1)^2 + n^{m-1}(n - 1) + n^{m-1}(n^{m-1} - 1)$ mode switches.

In conclusion after flattening the program we get $n^{m-1} + n - 1$ modes and $(n - 1)^2 + n^{m-1}(n - 1) + n^{m-1}(n^{m-1} - 1)$ mode switches.

(b) Compiler algorithm. The compiler algorithm for the flattening HTL compiler is mode up of three algorithms (Section 4.1). Algorithm 4.1 generates one *call* instruction for each communicator declared in a program (there is one communicator in the considered HTL description), one *future* instruction for each module in a program, and one *return* instruction, thus for the considered HTL description, this algorithm will generate three instructions. Algorithm 4.2 first generates instructions to initialize task output ports, since in the considered HTL description there is no task output port, this will generate no instruction, next the algorithm generates a *jump* instruction to the start mode, since in the flatten program there is only one module, there will be only one instruction generated by this algorithm for the considered HTL description. Algorithm 4.3 generates one instruction for each mode switch in a mode, and for each mode it will generate four more instructions (since there are no tasks in any of the modes, no E code will be generated to release tasks or read/write communicators). The compiler also generates an instruction that invokes the root program. Thus total number of instruction generated by the flattening compiler algorithm is $3 + 1 + 4n' + \beta + 1 = 5 + 4n' + \beta$, where β is the number of mode switches in the flatten HTL description, and n' is the number of modes in the flatten HTL description. Now if we use the results from the flattening algorithm analysis we get that for flattening HTL compiler the worst case generated code size for an HTL description with 2 programs, m modules, and n modes per module is $n^{(2(m-1))} + n^m + 2n^{(m-1)} + n^2 + 2n + 1$.

Hierarchy-preserving HTL compiler. The compilation algorithm for the hierarchy-preserving HTL compiler consists of three algorithms, one for each structural unit (i.e., program, modules, and mode) of an HTL description. Algorithm 4.4, which compiles a program, generates one instruction for each communicator, i.e., 1 instruction for the HTL test description (there is only one communicator declared), two instructions for each module in a program, i.e., $2m$ for the HTL test description, and two return instructions for each program, i.e., $2p$ for the HTL test description. Algorithm 4.5, which compiles a module, for the HTL test description, generates three instructions for each module, and one instruction for each output port declared in a module, thus it will generate $3m$ instructions. Algorithm 4.6, which compiles a mode, can be split in two parts: (a) mode initialization code generation and (b) mode unit code generation. For mode initialization, there will be generated one instruction for each mode switch in a mode, two instructions for each mode, and for each mode that is refined there will be generated eight instructions to start refining program. Knowing that for the HTL test description there are $n - 1$ refining programs, mn modes, and $n - 1$ switches per mode, the total number of instruction generated for initializing modes is $((n - 1) + 2)mn + 8(p - 1)$. For mode units, there are generated five instructions for each mode switch in a mode and twelve instructions for each mode. Total

number of instructions generated for mode units when compiling HTL test description is $((n-1)5+12)mn$. In order to start executed HTL program there has to be generated an instruction to call each program initialization section, one instruction to start executing root program and a return instruction. Thus the hierarchy-preserving HTL compiler generates $6mn^2 + 8mn + 11p + 5m - 5$ instructions when compiling the HTL test description.

4.3.2.2. Runtime Overhead

In order to analyze runtime overhead both the worst case number of instructions that have to be interpreted per instance and the worst case number of triggers in queue have to be computed.

Flattening HTL compiler. The worst case number of triggers in the trigger queue, when executing an E code program that has been generated for the HTL test description using flattening HTL compiler, is influenced only by the number of parallel modules in the root program, thus in the worst case for an HTL test description there can be at most $m - p$ triggers in the trigger queue. The worst case number of instructions that have to be interpreted per instant depends on the number of instructions that have to be executed at the beginning of the period of a mode. Since in the worst case there can be $n^{(m-1)} + n - 1$ mode switches in a mode in a flatten program, it means that together with two instructions from the beginning of a mode unit and with the two instructions from the end of a mode unit, a total of $n^{(m-1)} + n + 3$ instructions have to be interpreted per instance in the worst case for an E code program that was generated using flattening HTL compiler for an HTL test description.

Hierarchy-preserving HTL compiler. For the hierarchy-preserving HTL compiler the degree of parallelism is always equal to the number of modules, i.e., m modules are executed in parallel. Since all the modes in all the modules are similar, it is enough to count maximum number of triggers for a mode and then multiply it by number of modules that are executed in parallel (e.g., m) in order to find the highest number of triggers in trigger queues. Since, for an HTL test descriptions the modes contains no task invocation, it means that the maximum number of triggers in trigger queues is three. Thus when executing HE code generated using the hierarchy-preserving HTL compiler for an HTL test description, the worst case number of triggers in trigger queues is $3m$. In order to evaluate the highest number of instructions that have to be interpreted per instant it is important to notice that the degree of parallelism for an HTL test description is n, and that all the modes have the same period, thus in the worst case the HE machine will need to interpret HE code for unit zero for all the active modes. Thus in this case the worst case number of instructions that have to be interpreted per instant is $(n + 7)m$.

4.3.3. Experimental Analysis

This subsection compares the efficiency of both flattening compiler and hierarchy-preserving compiler, experimentally. In order to compare the runtime overhead introduced by interpreting E code and HE code generated by flattening compiler and hierarchy-preserving compiler, respectively, the time spent in interpreting E code and HE code for the 3TS case study HTL description (presented in Chapter 5) has been measured; the delay introduced by code interpretation is below 1% for both E code and HE code. This result proves that for relatively simple HTL descriptions (i.e., with no parallel modules in the refinement), the runtime overhead introduced by interpreting HE code is not significantly higher than the runtime overhead introduced by interpreting E code.   ◥

The code size is compared for HTL descriptions with p programs (i.e., one top-level program and $p - 1$ refinement programs) and m modules ($p \leq m$), where each module has $n = 2$ modes switching between themselves. For each such scenario there are a number of
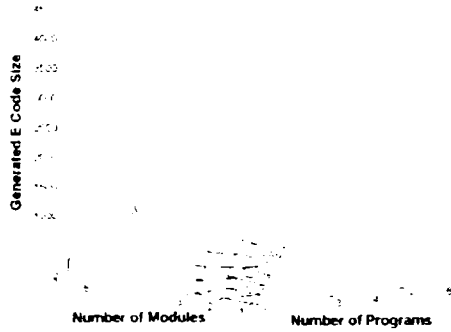
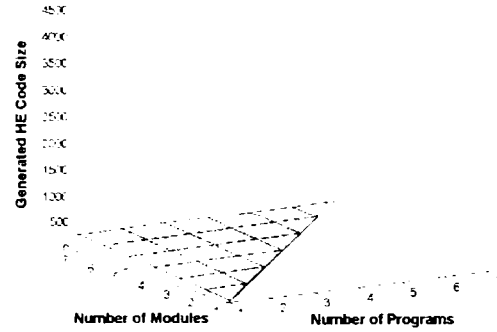Fig. 4.2: Number of E code instructions

Fig. 4.3: Number of HE code instructions

possible HTL descriptions. All the HTL descriptions for which $1 \leq p \leq 7$ and $1 \leq m \leq 7$ have been automatically generated, and compiled with both HTL compilers. For each scenario the highest generated code size for each compiler has been recorded. In Fig. 4.2 it is plotted the highest generated code size for flattening compiler, and in Fig. 4.3 it is plotted the highest generated code size for the hierarchy-preserving compiler. The two figures show that the results from the previous section are correct. Thus in the worst case the size of generated E code (4364 instructions) is an order of magnitude larger than the size of generated HE code (387 instructions).

5. Case Study: Three Tanks System

All the case studies that are presented in this thesis consists of implementing a real-time control application for one of the following two plants: Three Tanks System (3TS) [38] and JAvaiator [11]. Both plants are non-linear plants, which require multi-mode control strategies. This chapter consists of two sections: in the first one it is described the 3TS plant and one possible control strategy, in the second section it is presented an HTL implementation for the 3TS control strategy, which is presented in the previous section.

5.1. Three Tanks System Overview

The Three Tanks System (3TS) plant consists of three interconnected tanks, e.g., T_1, T_2, and T_3; in Fig. 5.1 it is presented schematically the 3TS plant. Each of the three tanks is connected to an evacuation tap, e_1, e_2, and e_3 respectively. Tank T_2 is connected to an additional evacuation tap, g.
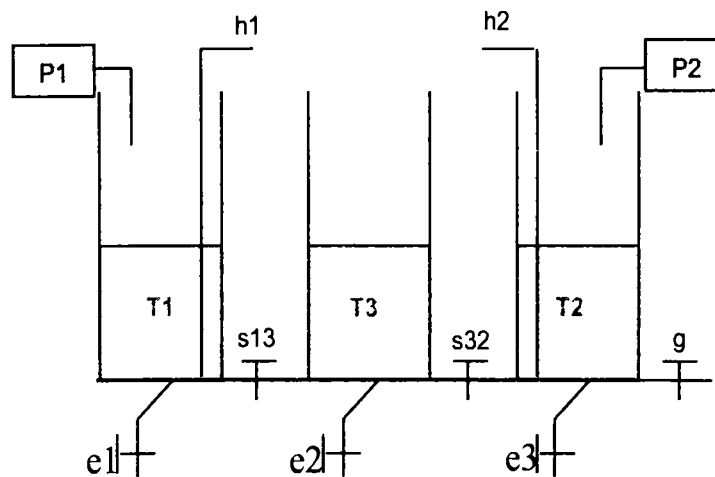

Fig. 5.1: Schematic representation of 3TS plant

Tank T_3 is interconnected with both tank T_1 and T_2. The interconnection between tanks T_3 and T_1 is done through tap s_{13}, and the interconnection between tank T_3 and T_2 is done through tap s_{32}. Evacuation taps and interconnection taps are used to introduce perturbation in the system. The 3TS plant also contains two pumps, e.g., P_1 and P_2 connected to tanks T_1 and T_2, respectively. In Fig. 5.2 it is presented the real 3TS plant.
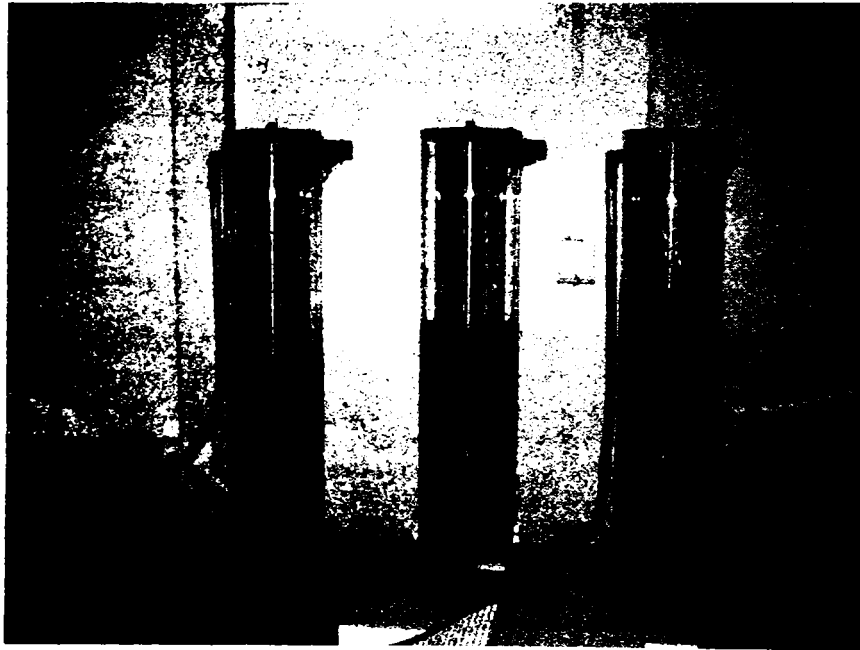
Fig. 5.2: 3TS plant

Mathematical model. For one tank, depending on if there is perturbation in the tank or not, there are two mathematical models:

1. one tank with no perturbation behaves like a pure integrator:

$$H_P(s) = k_P \frac{1}{s} \tag{5.1}$$

2. one tank with perturbation behaves like a plant proportional with first order temporization (PT1):

$$H_P(s) = k_P \frac{1}{Ts + 1} \tag{5.2}$$

For a detailed mathematical model of the 3TS plant please refer to Appendix A.

Controller design. The goal of the controller is to control water level in tanks T_1 and T_2 by controlling pumps P_1 and P_2 based on the feedback information obtained from three sensors, which measure the water in each of the three tanks. For the controller design it has been considered that there are two distinct controllers, i.e., one controller for tank T_1 and another one for tank T_2. Also it has been considered that tanks T_1 and T_2 are two independent tanks, i.e., the dynamics of the water level in each of the tanks can be described by equations (**5.1**) or equation (**5.2**), depending on if there is perturbation or not in that tank. Thus a controller for one tank in general has be designed and then used for both T_1 and T_2. When there is no perturbation in a tank, a proportional (P) controller is a good solution. On the other hand when there is perturbation in the tank, a proportional controller is not enough anymore and a proportional-integrator (PI) controller is needed. Thus the final control strategy consists of switching between the P and PI control law, based on the presences or absence of the perturbation.

5.2. HTL Implementation of the Three Tanks System Controller

In this section it is presented the HTL description that implements the 3TS controller. The hierarchical structure of the HTL description is presented in Fig. 5.3.
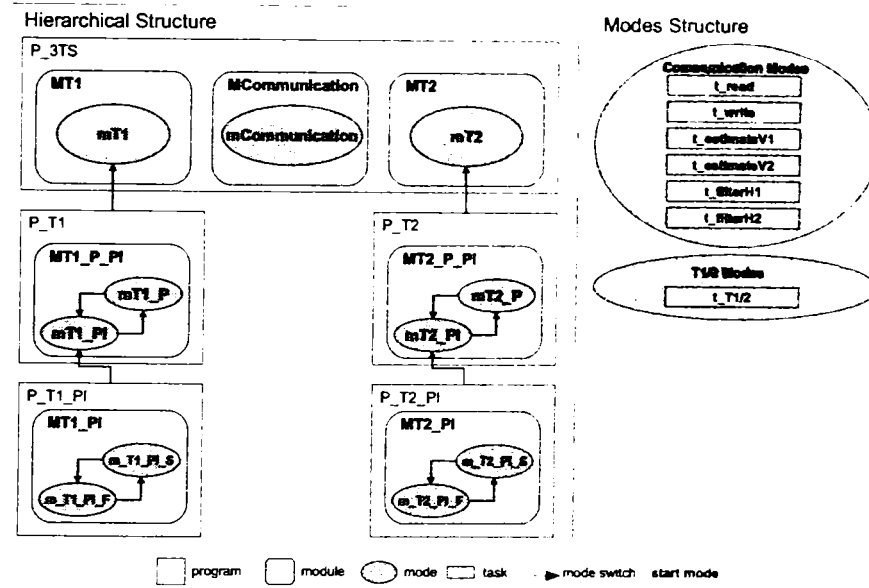

Fig. 5.3: 3TS Controller: Hierarchical Structure

The top level HTL program contains three modules, i.e., $MT1$, $MT2$, and $MCommunication$. Module $MCommunication$ specifies the timing for the communication; it contains one mode $mCommunication$, which invokes six tasks: task t_read, which reads water level sensors, task t_write, which updates pump commands, tasks $t\_estimateV1$ and $t\_estimateV2$, which compute if there is any perturbation in tank T_1, and T_2, respectively, the last two tasks $t\_filterH1$ and $t\_filterH2$, compute a Butterworth filter for the two sensed signals, i.e., water level in tank T_1 and water level in tank T_2, in order to reduce the noise. The modules MT1 and MT2 specify the timing for the first tank controller and the second tank controller, respectively. Each module contains one mode, which invokes one task (e.g., the controller task for the corresponding tank). In both cases the controller task is refined into two tasks, one that implements the P controller and one that implements the PI controller. The PI controller task is further refined into a fast PI and a slow PI, the difference between this two consists in the amplification factor, which is higher for the fast PI controller. The fast PI controller is used for a high control error, while the slow PI controller is used for a low control error. The strategy of switching between a fast and a slow PI controller is meant to reduce overshooting.

In Fig. 5.4 it is presented the data-flow graph for the root HTL program that specifies the abstract timing for the 3TS controller. Task t_read reads the two sensors that measure the water level in tank T_1 and T_2, respectively, and writes those vales into communicator h_1 and h_2. The two communicators are read by tasks $t\_filter1$ and $t\_filter2$, respectively, which compute the filtered water level in tank T_1 and T_2, respectively. The communicators h_1 and h_2 are also read by tasks t_T1 and t_T2, respectively, which compute the control law for tanks T_1 and T_2, respectively, and update communicators u_1 and u_2.
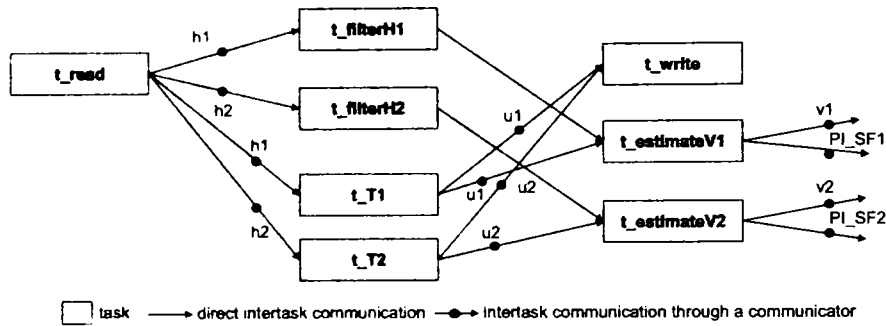
Fig. 5.4: 3TS Controller: Data-Flow

Tasks $t\_estimateV1$ and $t\_estimateV2$ estimate if there is perturbation in tanks T_1 and T_2, respectively, it also compute if a fast PI or a slow PI control law should be used, and updates communicators v_1 and PI_SF1, and v_2 and PI_SF2, respectively. Finally tasks t_write reads from communicators u_1 and u_2 and sends the new command to the two pumps P_1 and P_2, respectively.

Timing behavior of the 3TS controller is presented in Fig. 5.5. The program that implements the 3TS controller consists of running in parallel the functionality that implements the communication with the 3TS plant, the functionality that controls tank T_1, and the functionality that controls tank T_2. In general the program invokes eight tasks every 500ms. The first task to be invoked is the t_read task, which has an LET of 300ms, this task will write to the fourth instance of communicator h_1 and h_2. Tasks t_T1 and t_T2 have an LET of 100ms; they will read the fourth instance of communicators h_1 and h_2, respectively, and in the end they will update fifth instance of communicators u_1 and u_2, respectively. Tasks $t\_filterH1$ and $t\_filterH2$ have an LET of 200ms; these tasks read fourth instance of communicators h_1 and h_2, and their output port will be read by tasks $t\_estimateV1$ and $t\_estimateV2$, respectively. Tasks $t\_estimateV1$ and $t\_estimateV2$ have an LET of 100ms; these tasks read from fifth instance of u_1 and u_2, and from tasks $t\_filterH1$ and $t\_filterH2$ and writes to the second instance of communicators v_1 and v_2, and PI_SF1 and PI_SF2.

5.2.1. Architecture

The 3TS system plant is connected to a Windows 98 machine through a DAC98 acquisition board. Since for this implementation the Unix version of the E machine has been used, a TCP server has been implemented, which sits on the Windows 98 machine, communicates to the 3TS plant and with the machines on which the 3TS controller runs. The HTL program that implements the controller was distributed over three Unix machines: first Unix machine runs the controller for T_1, second machine runs the controller for T_2, and third Unix machine runs the communication module. The communication between two different instances of E machine is done through UDP/IP protocol, and the communication with the server that is connected to the 3TS plant is done through TCP/IP protocol. In Fig. 5.6 it is presented the architecture on which the 3TS controller has been implemented.

5.2.2. Results

In order to evaluate the 3TS HTL controller that has been presented, a number of three experiments have been conducted. All the experiments have been run on the real 3TS plant. In the first experiment the implemented controller has been modified so that a
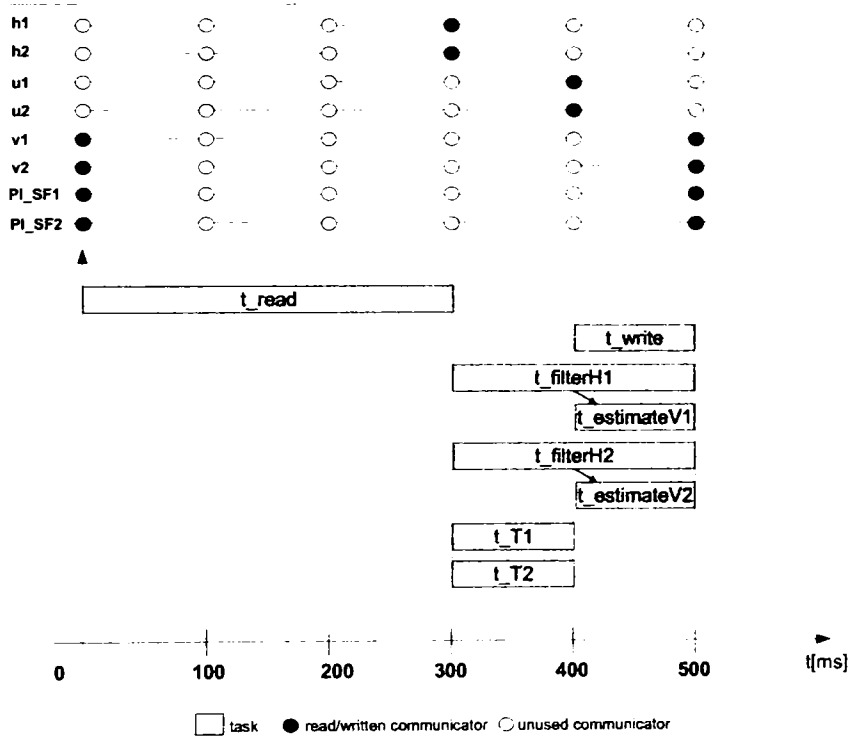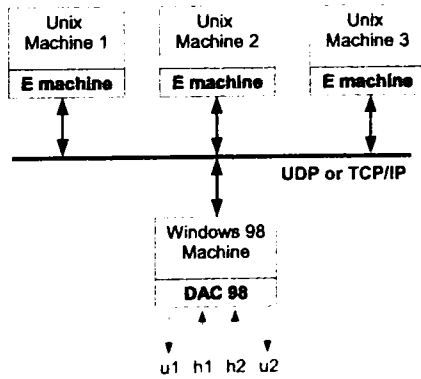
Fig. 5.5: 3TS Controller: Timing

Fig. 5.6: 3TS Controller: Architecture

proportional (P) controller is used for both tank T_1 and T_2. The target for tank T_1 was set to 30cm, and the target for tank T_2 was set to 40cm. In tank T_1 there was no perturbation while in tank T_2 there was perturbation. In Fig. 5.7 it can be seen the water level in the two tanks. In the two diagrams one can observe that for tank T_1 (where is no perturbation) the P controller is good enough and the target water level is reached in about 110s, but for tank T_2 (where is perturbation) the target water level is never reached, which means that the P controller is not good in this case (i.e., it can not compensate for the perturbation).
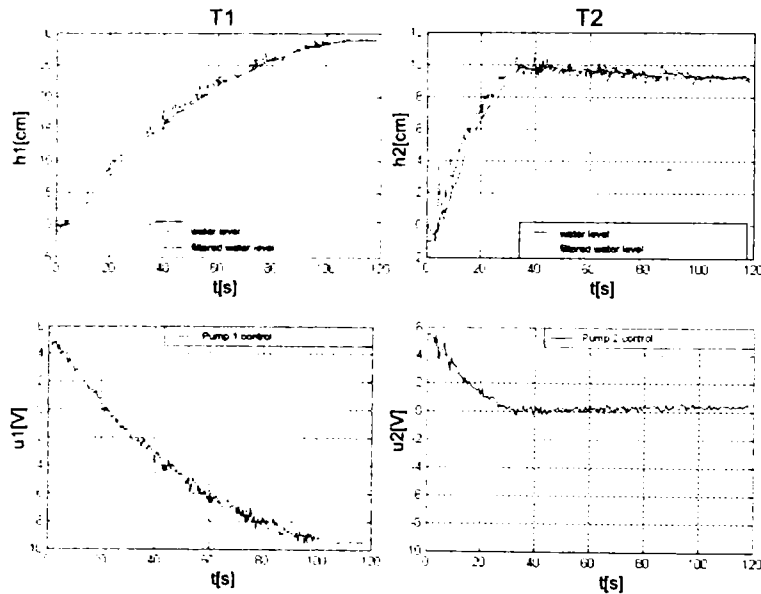

Fig. 5.7: P Controller for both T1 and T2 (T1 without perturbation, T2 with perturbation)

In the second experiment the controller has been modified so that a proportional-integrator (PI) controller is used for both tank T_1 and T_2. The target for tank T_1 was set to 30cm, and the target for tank T_2 was set to 30cm. In tank T_1 there was no perturbation while in tank T_2 there was perturbation. In Fig. 5.8 it can be seen the water level in the two tanks. From the two diagrams one can observe that for tank T_1 (where is no perturbation) the PI controller is not good, i.e., the target water level is overshoot, nevertheless for tank T_2 (where is perturbation) the target water level is reached in about 100s.

In the third experiment either a P or a PI controller is used for both tanks, depending on the presence or absence of perturbation in the controlled tank. The target for tank T_1 was set to 30cm, and the target for tank T_2 was set to 40cm. In tank T_1 there was perturbation, while in tank T_2 there was no perturbation. In Fig. 5.9 it can be seen the water level in the two tanks. From the two diagrams one can observe that for tank T_1 (where is perturbation) the P-PI controller is good enough and the target water level is reached in about 150s, and for tank T_2 (where is no perturbation) the target water level is overshoot, but the difference between the reached level and target level is only around 2cm, the control time is 200s.

From the three experiments that were presented above nor a P controller neither a PI controller is good enough to control water level in tank T_1 and T_2 in all the possible scenarios, but if it is used a controller that switches between a P control law if there is no perturbation in the controlled tank, and a PI control law if there is perturbation in the
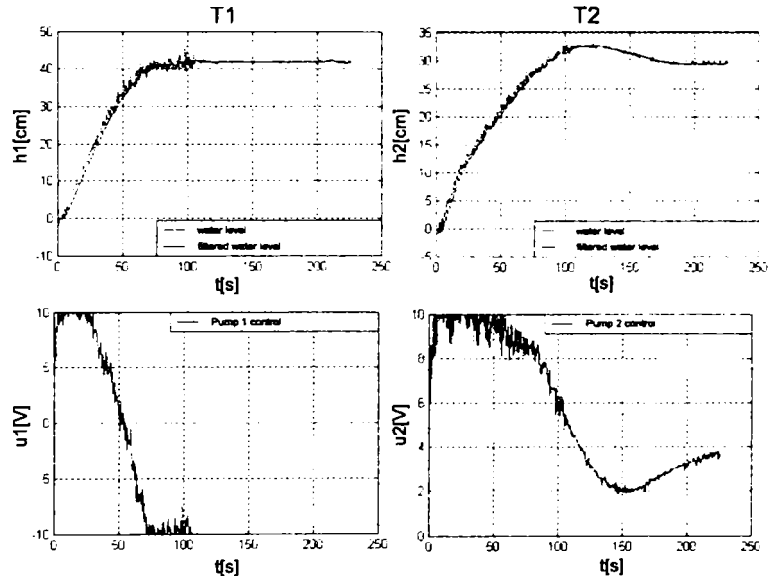
Fig. 5.8: PI Controller for both T1 and T2 (T1 without perturbation, T2 with perturbation)
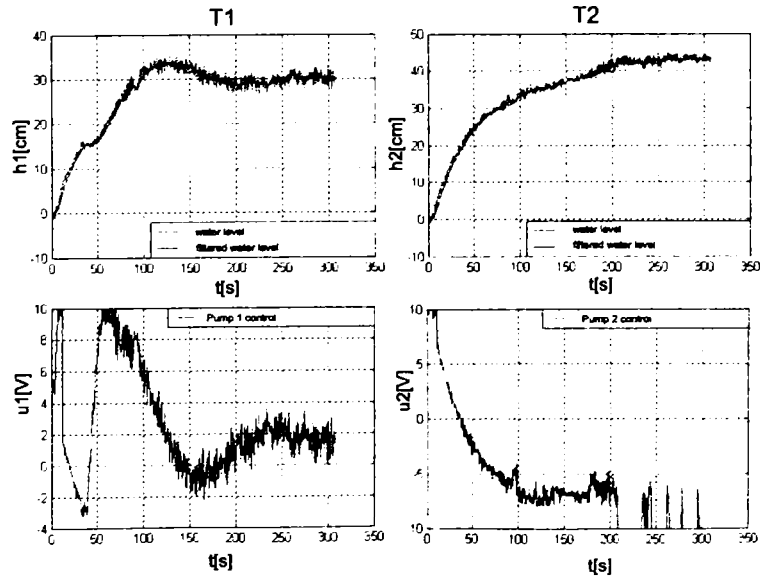

Fig. 5.9: P-PI controller for both T1 and T2 (T1 with perturbation, T2 without perturbation)

control tank, the results become very good and we can say that this controller can control the 3TS plant in any scenario.

Controller	T1			T2		
	$t_c[s]$	$\sigma_1[\%]$	$\gamma[\%]$	$t_c[s]$	$\sigma_1[\%]$	$\gamma[\%]$
P	110	0	0	-	-	-77
PI	75	11	36	150	10	0
P-PI	150	10	0	200	0	5

Tab. 5.1: Control quality indicators

In Tab. 5.1 it is presented the control time (t_c), overshoot (σ_1), and the difference between the target value and the value at which water level stabilizes (γ), for the three experiments. As shown in the table only the P-PI controller is able to control the plant in any scenario. The P controller is not able to compensate for perturbation, while the PI controller is not able to control water level when there is no perturbation.

6. Exotask HTL

The constant growth in complexity of embedded systems requires development of new real-time programming languages and tools that improve development of such systems. In this context more and more attention is paid to Java, which has been shown to be very efficient for non-real-time applications. The main advantages of Java over other object-oriented languages, i.e., C++, are: automatic memory management, namely, in Java programmer does not have to explicitly allocate/de-allocate memory since these operations are done behind the scene; Java programs are portable, i.e., once a program was compiled it can be run on any platform for which there exists a Java virtual machine implemented; code reusability. All this advantages are not for free, the price that is paid for them is represented by the lost in efficacy (the ratio between the time the processor is available for executing application specific code in a period and the period for which the efficacy is analyzed [1]) and non-determinism. The lost in efficacy is due to the fact that Java programs are not directly compiled into machine code but into bytecode, which is interpreted by the Java virtual machine. The non-determinism is due to the garbage collector (GC). Although the non-determinism due to the GC has been removed, thus making possible the use of Java for developing real-time applications, the former problem still remains and limits the categories of real-time applications for which Java can be used. Thus, Java can be used for developing complex real-time applications for embedded systems that use a relative powerful hardware (over 300MHz processor and at least 32MB of memory), whereas for embedded systems that use microcontrollers, C or C++ are still the best choice.

       Exotask [8] is a new programming construct for developing real-time application in Java. It addresses three key problems: low latencies, pluggable schedulers, and deterministic timing behavior. Tasks in Exotask can be run at frequencies below the barrier of one millisecond; this is not something new for Java, nevertheless Exotask is less restrictive as compared to other solutions (Eventrons [32], Reflexes [33], and StreamFlex [34]). In the Exotask system, a task can be annotated with information that defines its timing and that is used by the scheduler when task has to be scheduled. The set of Java classes that are used to specify timing information form the so called *timing grammar*. Although Exotask comes with a predefined timing grammar and a predefined scheduler, which can interpret the predefined grammar, it is not limited to the predefined trimming grammar and scheduler, since Exotask specifies an interface through which a new timing grammar can be specified and a new scheduler, which can interpret the new timing grammar, can be registered into the system; the new scheduler has full control over the task execution. Although Java provides functional portability across platforms, not all real-time programming methodologies that have been developed for Java support timing portability, this is because they relay on platform dependent characteristics in order to tune the application. In the case of Exotask, timing portability is guaranteed, with the condition that there are enough resources. In order to achieve timing portability, Exotask uses LET [5] model of computation. Exotask supports distribution of a program over a set of hosts and composition of Exotask graphs specification.

Exotask framework consists of a hierarchy of Java classes that are used to specify and to execute an Exotask program, and an Eclipse [39] plug-in, which provides a graphical editor for the Exotask program. In order to run an Exotask program IBM WebSphere Real Time (WRT) product JVM [40] its needed. The WRT includes RTSJ [27], the Metronome real-time garbage collector [29], and an ahead-of-time (AOT) compiler, which is used to eliminate non-determinism due to JIT compilation [41].

An Exotask program consists of an *Exotask specification graph*, which can specify *data-flow* between the tasks invoked in the program and *timing* of the tasks. The program also contains a set of Java classes that implement the functionality of the nodes in the Exotask graph specification. In Fig. 6.1 it is presented an overview of how a real-time application is developed using Exotask. The nodes in an Exotask graph specification are represented by specifications of Exotasks (Exotask specification), while the edges represent specifications of connections (Exotask connection specification) between the Exotasks. Thus an Exotask graph specification depicts the data-flow in an Exotask program. In order to specify the timing of an Exotask graph, each node and each edge of an Exotask graph specification has to be annotated with timing information, e.g.: the period, the offset with in the period, etc. The timing information that can be used to annotate nodes and edges in an Exotask graph specification, depends on the selected grammar. The Exotask system comes with two predefined grammars: Timed-Triggered (single mode) and Timed-Triggered (multiple modes). The difference between the two grammars consists in the fact that the second grammar supports sequential composition of sets of Exotasks, whereas the first one does not support such composition.
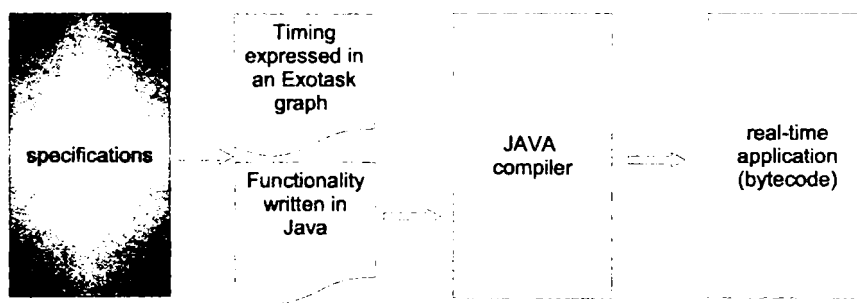

Fig. 6.1: Overview of Exotask programming model

There are three types of Exotasks specifications: communicator specification, task specification, and predicate specification. A *communicator* is a system provided Exotask, which has one input port, one output port, and exposes an execute method, which copies the value from communicator input to communicator output. A communicator acts like a buffer, it is inspired from HTL, and it can be used to communicate between tasks that have different frequencies. A communicator specification in an Exotask graph specification consists of a name, which can be used to refer the communicator, a data type, which specifies what kind of values can be buffered by the communicator, an initial value, and communicator's timing. A *task* is an Exotask written by the user. A task specification consists of a name, an implementation class, a list of input ports, a list of output ports, and timing annotation. The name is a string that is used to refer to the Exotask. The implementation class, is a Java class that implements the functionality of the task; it has to obey certain restrictions [8]. The list of input ports and the list of output ports

represent the interface through which the task can communicate with other Exotasks in a program. Each input/output port has a type associated with it; it can handle only values of the associated type. A *predicate* is still an Exotask written by user that is invoked by the scheduler and which computes a boolean value based on a set of inputs. The boolean value computed by a predicate is interpreted by the scheduler according to the timing grammar, i.e., in the case of Timed-Triggered (single mode) predicates are not used at all, but the Timed-Triggered (multiple modes) uses predicates in order to implement mode switching. A predicate is specified through a name, an implementation class, a list of input ports, and timing annotation. The name is a string value used to refer to the predicate. The implementation class is a Java class that implements the functionality behind the predicate; as in the case of a task, the class has to obey some restrictions. The list of input ports are used to read values from other Exotasks, values that influence the boolean result. The timing annotation for all types of Exotasks specifies when an Exotasks has to be executed.
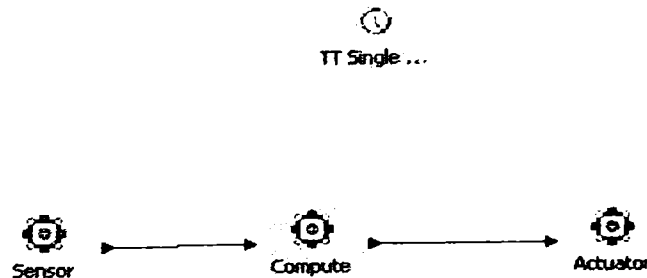

Fig. 6.2: Example of Exotask graph in graphical editor

An Exotask connection specification consists of a name, a data type, a source and a target. The name is a string value, which is used to refer to the connection. The data type identifies the type of values that can be exchanged though the connection. The source identifies the source Exotask and the source port, and the target specifies the destination Exotask and destination port. Both the type of the source port and the type of the destination port have to match the type of the connection. The timing annotation for an Exotask connection specification specifies when the value from the source port is transferred to the destination port.

There are three ways to create an Exotask graph specification: using the java classes for specifying an Exotask graph, which are provided by the Exotask framework, editing an XML file that can be parse by an XML parser provided by the Exotask framework, or using the graphical editor, which is provided by Exotask framework. In the case of the graphical editor, the graph is saved in an XML file. Exotask framework provides functionality for switching between the Java representation of an Exotask graph specification and an XML representation. In Fig. 6.2, Fig. 6.3, and Fig. 6.4 is represented an Exotask graph specification, which contains three tasks and which uses Timed-Triggered (single mode) grammar, in the three possible ways: graphical editor, XML, and Java code, respectively.

The graphical editor is very expressive when it has to represent data-flow, nevertheless there is no view for illustrating timing of the entire program; timing of an Exotask

```
<ExotaskGraph>
 <TimingProvider kind='simple'
   parser='com.ibm.realtime.exotasks.timing.simple.SimpleTimingDataParser'
 |   graphics='60 60 300 45' period='5'/>
 <Task id='Sensor' implementation='test.example0.Sensor' graphics='60 60 124 172'>
  <Output id='out0' type='java.lang.Integer'/>
 </Task>
 <Task id='Actuator' implementation='test.example0.Actuator' graphics='60 60 429 171'>
  <Input id='in0' type='java.lang.Integer'/>
  <Timing offsets='5'/>
 </Task>
 <Task id='Compute' implementation='test.example0.Compute' graphics='60 60 267 171'>
  <Input id='in0' type='java.lang.Integer'/>
  <Output id='out0' type='java.lang.Integer'/>
 </Task>
 <Connection id='Compute_Actuator' source='Compute' target='Actuator'>
 </Connection>
 <Connection id='Sensor_Compute' source='Sensor' target='Compute'>
 </Connection>
</ExotaskGraph>
```

Fig. 6.3: Example of Exotask graph in XML

```
ExotaskGraphSpecification ans = new ExotaskGraphSpecification();
ExotaskTaskSpecification Sensor=new ExotaskTaskSpecification();
Sensor.setName("Sensor");
Sensor.setImplementationClass("test.example0.Sensor");
Sensor.setInputPortTypes(new String[]());
Sensor.setInputPortNames(new String[]());
Sensor.setParameterType("");
Sensor.setParameterValue("");
Sensor.setOutputPortTypes(new String[]("java.lang.Integer",));
Sensor.setOutputPortNames(new String[]("out0",));
Sensor.setWeaklyIsolated(false);
ans.getTasks().add(Sensor);
ExotaskTaskSpecification Actuator=new ExotaskTaskSpecification();
Actuator.setName("Actuator");
Actuator.setTimingData(new com.ibm.realtime.exotasks.timing.simple.SimpleTimingAnnotation(
        new String[](null), new long[][](new long[](5000000L,),)));
Actuator.setImplementationClass("test.example0.Actuator");
Actuator.setInputPortTypes(new String[]("java.lang.Integer",));
Actuator.setInputPortNames(new String[]("in0",));
Actuator.setParameterType("");
Actuator.setParameterValue("");
Actuator.setOutputPortTypes(new String[]());
Actuator.setOutputPortNames(new String[]());
Actuator.setWeaklyIsolated(false);
ans.getTasks().add(Actuator);
ExotaskTaskSpecification Compute=new ExotaskTaskSpecification();
Compute.setName("Compute");
Compute.setImplementationClass("test.example0.Compute");
Compute.setInputPortTypes(new String[]("java.lang.Integer",));
Compute.setInputPortNames(new String[]("in0",));
Compute.setParameterType("");
Compute.setParameterValue("");
Compute.setOutputPortTypes(new String[]("java.lang.Integer",));
Compute.setOutputPortNames(new String[]("out0",));
Compute.setWeaklyIsolated(false);
ans.getTasks().add(Compute);
ans.getConnections().add(new ExotaskConnectionSpecification("Compute_Actuator",null,Compute,0,Actuator,0));
ans.getConnections().add(new ExotaskConnectionSpecification("Sensor_Compute",null,Sensor,0,Compute,0));
ans.setTimingData(new com.ibm.realtime.exotasks.timing.simple.Period(5000000L));
```

Fig. 6.4: Example of Exotask graph in Java

or a connection can be view only by clicking on it. The XML and Java code representations of an Exotask graph specification are more intended to be automatically generated and not edited manually.

Once the Exotask graph has been specified, the resulted Exotask graph specification is validated by the scheduler associated with timing grammar that has been used for specifying the Exotask graph. As a result of validating the Exotask graph specification, an Exotask graph is generated, which can be used to control execution of the Exotask program.

In the remaining of this chapter I will present a timing grammar for Exotask which implements the HTL semantics. In order to achieve this, the following three steps had to be performed: definition of new timing annotations and extension of the graphical editor in order to support new timing annotations, implementation of a scheduler that can interpret HTL grammar, the scheduler also compiles an Exotask graph specification into an HE code program represented in Java, and finally implementation of an E code interpreter, which is invoked when Exotask graph is started.

6.1. Exotask HTL Grammar

In this section are presented the timing annotations that have been defined in order to express HTL semantics in an Exotask graph specification. First the hierarchical structure of an HTL program has to be specified as global timing annotation for an Exotask graph specification that uses HTL grammar. The HTL global timing annotation consists of a list of HTL program declarations, a list of HTL module declarations, and a list of HTL mode declarations. An HTL program declaration consists of specifying the name of the HTL program. An HTL module declaration consists of specifying the name of the module, the name of the start mode, and the name of the HTL program that contains the HTL module. An HTL mode declaration consists of specifying mode name, the period of the mode, the name of the HTL module that contains the HTL mode, and the name of the refining HTL program if any. In Fig. 6.5 it is presented an example of a global timing annotation for an HTL description that contains two HTL programs, i.e., $P1$ and $P2$. Program $P1$ contains one module, $M1$, which contains two modes $m1$ and $m3$. Program $P2$ refines $m1$ and contains one module, $M2$, which contains one mode, $m2$.

```
<TimingProvider kind = 'htl' parser = 'at.uni_salzburg.cs.exotasks.timing.htl.HTLTimingParaParser
    graphics= 60 60 245 45'>
  <ProgramList>
    <Program name = 'P1'/>
    <Program name = 'P2'/>
  </ProgramList>
  <ModuleList>
    <Module name = 'M1' start = 'm1' program = 'P1'/>
    <Module name = 'M2' start = 'm2' program = 'P2'/>
  </ModuleList>
  <ModeList>
    <Mode name = 'm1' period = '5s' module = 'M1' refine = 'P2'/>
    <Mode name = 'm2' period = '5s' module = 'M2' refine = ''/>
    <Mode name = 'm3' period = '10s' module = 'M1' refine = ''
  </ModeList>
</TimingProvider>
```

Fig. 6.5: Example of global timing annotation for HTL grammar

HTL communicators are mapped directly to Exotask communicators, thus a communicator is annotated with the program in which the communicator is declared and with

the access period. In Fig. 6.6 it is presented an example of a communicator annotation; communicator $c1$ is declared in program $P1$ and has a period of $1s$.

```
<Communicator id='c1' type='java.lang.Integer' initialValue='' graphics='60 60 362 133'>
 <Timing period = '1s' program = 'P1' />
</Communicator>
```

Fig. 6.6: Example of communicator timing annotation for HTL grammar

HTL tasks are mapped directly to Exotask tasks, thus a task has to be annotated with the name of the HTL mode in which the HTL task is invoked. Since in HTL a task can be abstract or concrete, a task in the Exotask graph specification has to be marked as abstract or concrete, also for tasks that refine abstract tasks it has to be specified the name of the parent task. In Fig. 6.7 it is presented an example of a task timing annotation for tow task: $t1$, which is an abstract task invoked in mode $m1$ and $t11$, which is a concrete task, it refines task $t1$ and it is invoked in mode $m2$.

```
<Task id= t1' implementation='test.simple.htl.T1' isolation='strong' graphics='60 60 526 122'>
 <Input id='in0' type='java.lang.Integer'/>
 <Output id='out0' type='java.lang.Integer'/>
 <Timing isAbstract = 'true' parent = ''>
   <ModeAssignment mode = 'm1'/>
 </Timing>
</Task>
 <Task id= t11' implementation='test.simple.htl.T11' isolation='strong' graphics='60 60 515 20'>
 <Input id='in0' type='java.lang.Integer'/>
 <Output id='out0' type='java.lang.Integer'/>
 <Timing isAbstract = false parent = 't1'>
   <ModeAssignment mode = 'm2';>
 </Timing>
</Task>
```

Fig. 6.7: Example of task timing annotation for HTL grammar

HTL mode switches are mapped to Exotask predicates, thus a predicate it is annotated with the source mode and target mode. In Fig. 6.8 it is presented an example of predicate annotation; the annotated predicator is mapped to the mode switch that switches from mode $m1$ to mode $m3$.

```
<Predicate id='m1_2_m' implementation='test.simple.htl.Switch_m1_2_m3' isolation='strong' graphics='60 60 771 237'>
 <Input id='in0' type='java.lang.integer'>
 Timing targetMode='m3'>
   <ModeAssignment mode = 'm3'>
 </Timing>
</Predicate>
```

Fig. 6.8: Example of predicate timing annotation for HTL grammar

HTL tasks can communicate either directly or indirectly through an HTL communicator. In an Exotask graph specification that uses HTL grammar a direct communication between two tasks is represented through an Exotask connection that connects an output port of one of the tasks and an input port of the other task. Indirect communication is

represented through two Exotask connections; one that connects one task and a communicator and the other one that connects the other task and the same communicator. Thus an Exotask communication is annotated with the following information: the name of the mode in which connection is used, the instance number of the communicator read/written, and a flag to specify if the communicator is read or written. If the connection is used to communicate between two tasks then the instance value has to be set to -1 and the flag does not matter. In Fig. 6.9 are presented two connections: $t2\_t3$, which is used to communicate between two tasks, and $c1\_t1$, which is used to communicate between a communicator and a task.

```
<Connection id='t2_t3' source='t2' target='t3'>
 <Timing instance='-1' writesCommunicator='false'>
   <ModeAssignment mode = 'm3'/>
 </Timing>
</Connection>
<Connection id='c1_t1' source='c1' target='t1'>
 <Timing instance='1' writesCommunicator='false'>
   <ModeAssignment mode = 'm1'/>
 </Timing>
</Connection>
```

Fig. 6.9: Example of connection timing annotation for HTL grammar

The Eclipse plug-in that implements the Exotask graph specification graphical editor has been extended to recognize the new annotations, and an XML parser has been implemented to parse HTL grammar annotation.

6.2. Exotask HTL Scheduler

As presented in Chapter 4, an HTL description is first compiled into HE code, then the resulted HE code program is executed on HE machine. Thus in order to executed an Exotask graph specification that uses HTL grammar, a scheduler that translates the HTL annotations of an Exotask graph specification into a form of HE code, which is designed to work with the corresponding instantiated Exotask graph, had to be designed. The compiler algorithm used here is based on the hierarchy-preserving HTL compiler (Section 4.2). When the execution of the Exotask graph is started, the generated HE code is interpreted by a Java HE machine.

The compiler and the E machine together play the formal role of a pluggable Exotask scheduler. E code instructions that release tasks cause those tasks to be assigned exclusively to a scheduler thread responsible for running it once (in general, the binding of tasks to threads is temporary and dynamic, but the compiler has determined the maximum concurrency level and the scheduler then requests enough scheduler threads to ensure that there will always be a thread available to run each released task). E code instructions whose purpose is to copy values between ports, or between ports and communicators, use the Exotask system interface made available to schedulers for performing the deep copying between Exotask heaps. E code instructions that perform mode switches interrogate condition nodes in the graph, just like any other scheduler.

In adapting the latest HTL compiler to work with the Exotask system, compilation

is always done "on demand" at the point where the Exotask system invokes the HTL scheduler. That is, despite the opportunity for separate compilation when using the hierarchical strategy, there is no ability to save code artifacts across successive runs.

6.3. Case Study: JAviator

The JAviator [11], [12] is a quadrotor helicopter [42]. Fig. 6.10 presents an overview of the real JAviator. The helicopter was developed at University of Salzburg in order to test different methodologies for designing and developing real-time applications.
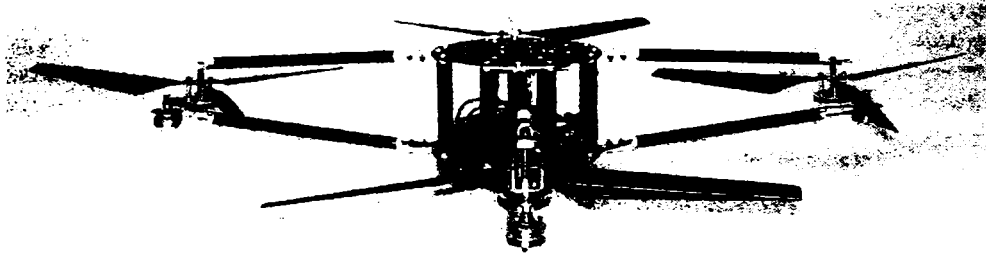
Fig. 6.10: JAviator

JAviator hardware configuration consists of:

- one Gumstix [43], which is a full-function miniature computer, on top of which runs Linux operating system, it is powered by an XScale processor at 400MHz and it has 64MB of RAM, there is also a version of IBM WebSphere Real Time (WRT) product JVM running on it, this makes possible development of Java real-time application for the Gumstix; for JAviator the Gumstix is used to run either parts of the control program or even the entire control program, and to communicate with the ground station over TCP or UDP;

- one Robostix [44], which is a board based on the ATmega128 processor (clock frequency 16MHz, flash memory 128KB, EEPROM data memory 4096B, SRAM data memory 4096B, two timers/counters on 8bit and two on 16bit, two full duplex USART, I2C interface, support for in-system programming, etc.) ; it is used for sensing, actuating, and for running parts of the control program, it is possible that the entire control is implemented on the Robostix, in which case the Robostix also has to communicate with the ground station; the communication can be either directly over an RS232 communication channel, or indirectly through the Gumstix; the communication between Gumstix and Robostix is performed over RS232;

- one Microstrain gyroscope [45],which provides information about Euler's angles, derivative of Euler's angles, and accelerations on x, y, and z axies;

- one SFR10 ultrasonic range finder [46], which provides information about the altitude.

For the JAviator there are at least two control problems that have to be addressed: *low-level control*(LLC) problem, which consists of controlling the altitude and the attitude (i.e., roll, pitch, and yaw) of the helicopter, and *high-level control*(HLC) problem, which consists of controlling the x and y position of the helicopter. In Fig. 6.11 it is presented on overview of the two JAviator controllers.
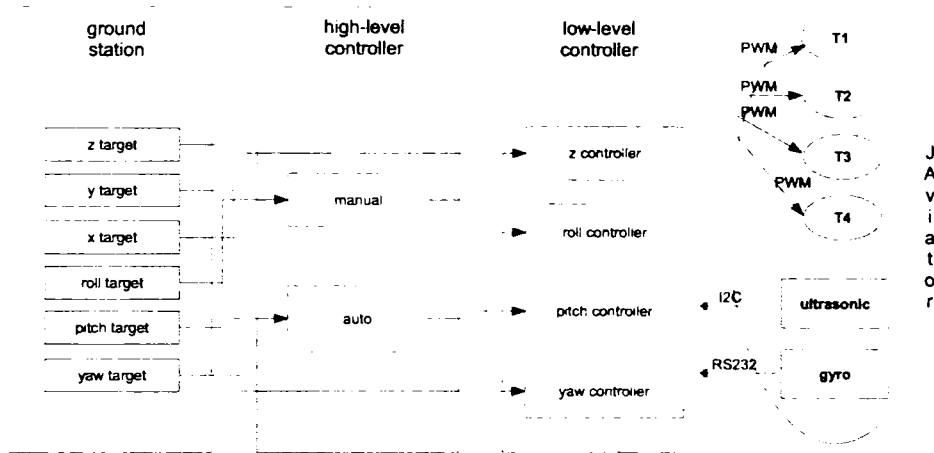
Fig. 6.11: JAviator control overview

The low-level controller receives sensed data, from the gyroscope and from the ultrasonic sensor, and target values for z, roll, pitch, and yaw from ground station or from high-level controller, and computes the command for each of the four rotors of the JAviator (i.e., T_1, T_2, T_3, and T_4), which is send to the helicopter as a PWM signal. The low-level controller consists of: z controller, which controls the altitude, roll controller, which controls the roll angle, pitch controller, which controls the pitch angle, and yaw controller, which controls the yaw angle. The z controller is a PID controller, while the roll, picth, and yaw controllers are PD controllers. The four controllers have been designed using the pole allocation method. For designing the controller the simplified mathematical model of the JAviator (Chapter B) has been used.

The high-level controller has two modes of operations, e.g., the manual mode and the auto mode. In the manual mode the high-level control only passes the roll and pitch references received from the ground station to the low-level controller. In the auto mode the high-level controller computes the roll and pitch targets for the low-level controller so that the requested x and y target position is reached. Currently there is no sensor on the JAviator that can provide information about the x and y position, thus the only way to get such information is to estimate it based on the acceleration received from the gyroscope.

So far there has been implemented a Java low-level controller using Exotask [8] framework, and a C low-level controller. In this thesis I will presented two more implementations, one that uses Exotask-HTL (Subsection 6.3.1), and the other one that uses micro HTL (Chapter 7).

6.3.1. Exotask-HTL Implementation of the JAviator LLC

In this section it is presented an Exotask-HTL implementation of the low-level controller. Figure 6.12 depicts, in visual syntax, the HTL program that implements the the low-level control. The program consists of running in parallel functionality that implements the low-level control as well as functionality that implements the communication with the JAviator and with the ground station.

The top-level program contains three modules, namely, $MLLControl$, $MJAviatorComm$, and $MGroundComm$. The $MJAviatorComm$ module specifies the timing of tasks that implement communication with the JAviator. It also computes the
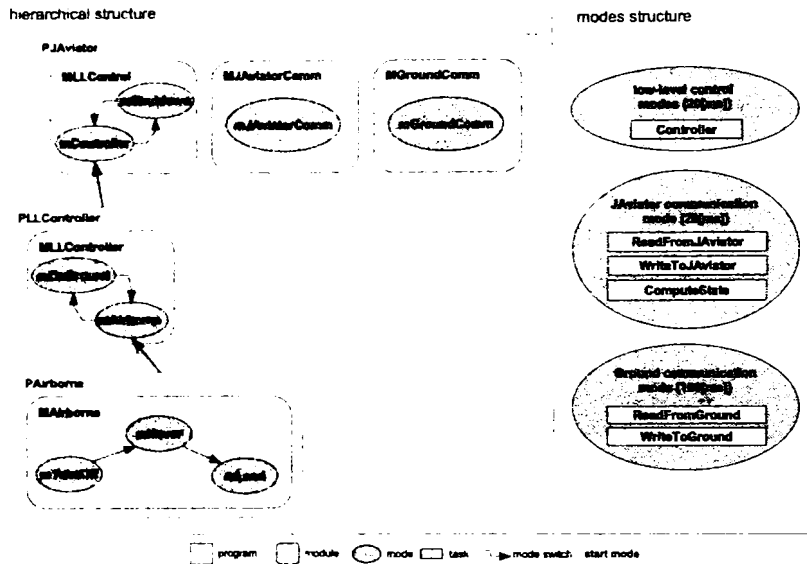
Fig. 6.12: The HTL Program Structure of a JAviator Flight Controller

next state for the altitude and attitude controllers. The module consists of a single mode, which has a period of 20ms. The mode invokes the $ReadFromJAviator$, $WriteToJAviator$, and $ComputeState$ tasks. The $MGroundComm$ module specifies the timing of tasks that implement communication with the ground station. This module also contains only one mode that has a period of 100ms and invokes the $ReadFromGround$ and $WriteToGround$ tasks. Module $MLLControl$ contains the $mController$ and $mShutdown$ modes, which both have a period of 20ms. The $mShutdown$ mode specifies the timing of the emergency shutdown, while the $mController$ mode specifies the timing of the altitude and attitude controllers. The $mController$ mode invokes the $Controller$ task, which implements the altitude, roll, pitch, and yaw controllers.

The $Controller$ task is refined by two tasks in the HTL program $PLLController$: one task is a concrete task and is invoked in the $mOnGround$ mode, while the other one is an abstract task and is invoked in the $mAirborne$ mode. The abstract task is further refined in the HTL program $PAirborne$ by three other concrete tasks; one for each of the three possible states of a flying helicopter, i.e., take-off, hover, and land.

Figure 6.13 presents in visual syntax the Exotask graph that specifies $mController$ mode, and the Exotask graph that specifies both communication modules, i.e., with the sensors and actuators, which are connected to the JAviator, on one hand, and with the ground station on the other. The XML source of the two Exotask graphs can be found in Section E.5. All the modes that refine the $mControl$ are specified in Exotask graphs that have a similar structure with the Exotask graph that specifies mode $mControl$, before the Exotask program it is translated into HE code, all the Exotask graph specifications are composed into a single Exotask graph specification.

Both the $MJAviatorComm$ module and the $MGroundComm$ module contain two tasks one for reading data from the sensors and the ground station, respectively, and the other one for writing data to the actuators and the ground station, respectively. In addition to the read and write task, module $MJAviatorComm$ contains the task $ComputeState$, which

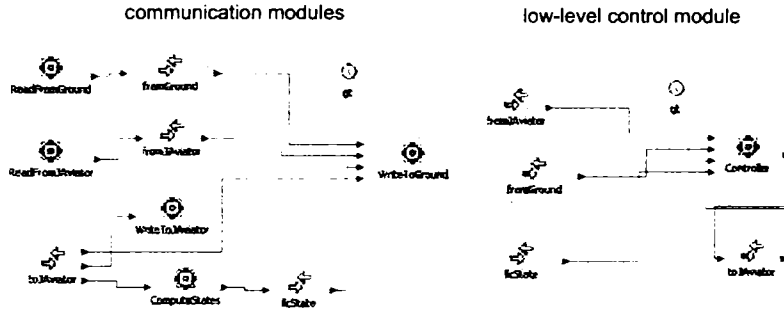communication modules   low-level control module

Fig. 6.13: Data-Flow View of the Top-Level HTL Program in Fig. 6.12

computes the state of the low-level controller based on the data received from the ground station. The $MLLControl$ module, contains two tasks, one task that sets all the command signals to zero, which it is invoked in mode $mShutdow$, and the other one $Controller$, which computes the actuate signals based on the values received from the sensors and from the ground station, and it is invoked in mode $mController$. The communication between the $Controller$ task and the tasks invoked in the two communication modules is done through communicators.
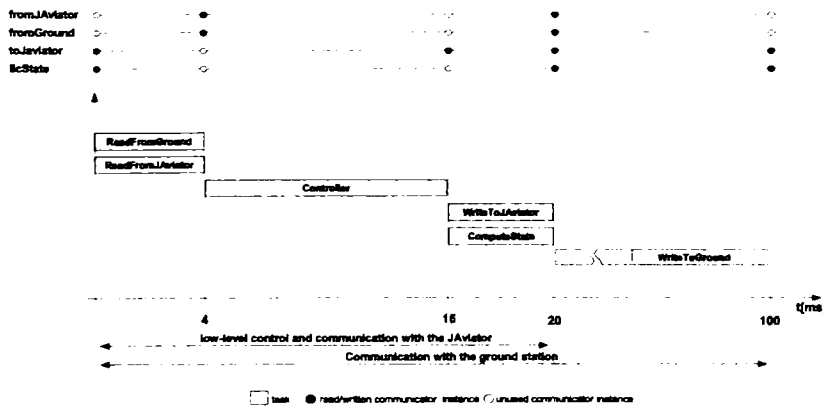
Fig. 6.14: Timing View of the HTL Program in Fig. 6.12

Figure 6.14, depicts, the timing of the HTL program.   Tasks invoked in the $MLLControl$ and $MJAviatorComm$ modules are executed once every 20ms, while tasks in the $MGroundComm$ module are executed once every 100ms.

All the communicators have a period of 4ms. Task $ReadFromGround$ and $ReadFromJAviator$ have a LET of 4ms, they are released for execution at the beginning of the period of the mode in which they are invoked, and they have to finish within 4ms when they update the second instance of the communicators $fromGround$ and $fromJAviator$, respectively. Task $Controller$ reads the second instance of both $fromGround$

and *fromJAviator* communicators, and writes to the fifth instance of the *toJAviator* communicator, thus it has an LET of 12ms. Task *WriteToJAviator* and *ComputeState* read the fifth instance of communicator *toJAviator*, and task *ComputeState* writes to the sixth instance of communicator *llcState*, while task *WriteToJAviator* does not write to any communicator it only sends the new commands to the motors; both tasks have an LET of 4ms. Task *WriteToGround* read the sixth instance of all the communicators and sends a report to the ground station, it has an LET of 80ms.

6.3.1.1. Results

Two experiments have been conducted with the Exotask-HTL implementation of the JAviator LLC. In both experiments an AMD64 four-way 2.4GHz machine was used. Although using such a powerful machine for embedded systems might sound un-realistic, there are embedded systems which requires powerful processors and for which space and energy consumption is not a problem, i.e., next generation battleships [35]. Nevertheless in the future I hope to optimize the Exotask-HTL implementation so that it can run on a Gumstix; Exotask programs using TT grammar have already been shown to run on Gumstix [9].

In the first experiment the Exotask-HTL JAviator LLC has been run on the AMD64 machine, at it was connected to a simulated JAviator plant. In Fig. 6.15 it is plotted the time interval between two successive runs of the *ReadFromJAviator* tasks.

min	19955.1 μsec
max	20254.0 μsec
mean	20070.4 μsec
median	20066.0 μsec
stddev	22.7 μsec
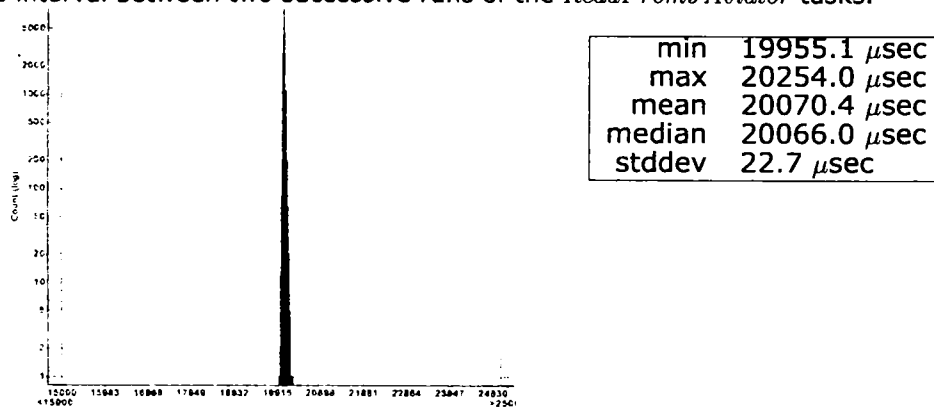
Fig. 6.15: Interarrival times of the ReadFromJAviator task, when no concurrent allocation is done

In the second experiment the Exotask-HTL JAviator LLC has been run on the AMD64 machine and an additional task, which allocates memory at a rate of 2MB/s, was run on the same machine in the same JVM as the control application. Again control plant was a simulated JAviator plant. In Fig. 6.16 it is plotted the time interval between two successive runs of the *ReadFromJAviator* tasks.

The two experiments show that the timing of the program is accurate enough, e.g., the time interval between two successive execution of a task varies with less then 300μsec. The second experiment shows that if there is enough computation power and if there are enough resources, the timing behavior of the application is not influenced by other tasks, not even by the GC task.
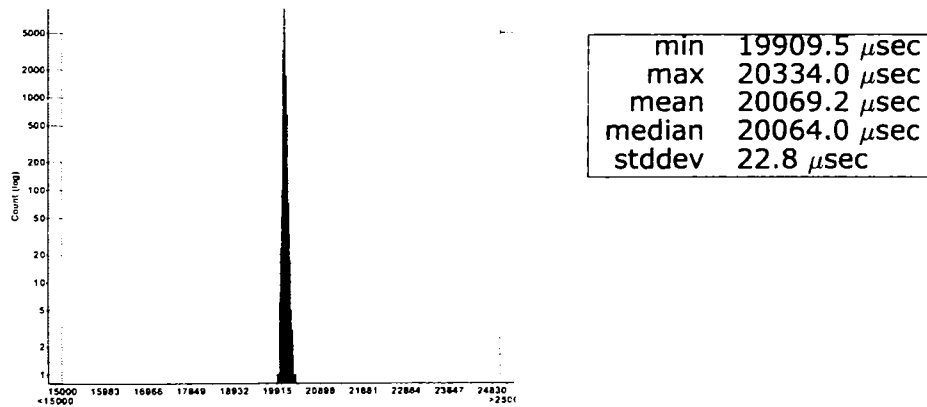
min	19909.5 μsec
max	20334.0 μsec
mean	20069.2 μsec
median	20064.0 μsec
stddev	22.8 μsec

Fig. 6.16: Interarrival times of the ReadFromJAviator task, when concurrently allocating 2MB per second

7. Micro HTL

Implementation of embedded applications (i.e., helicopter hover control [11]) in many cases has to be done on a hardware platform that is limited both in terms of the speed of the processor and in terms of the available amount of memory (e.g.: microcontroller). Thus in this chapter it is presented a *micro* implementation of HTL (micro HTL), which consists of an optimized version of the HTL compiler, *micro HTL compiler*, and an optimized version of the E machine, *micro E machine*.

The micro HTL compiler and the micro E machine support most of the features of HTL that have been presented in Chapter 2. The only feature that is currently not supported at all by the micro HTL is communication between tasks that are in the same mode through local ports, nevertheless it is possible to communicate between tasks that are in the same mode through communicators. The reason for not supporting this feature is that communicating between tasks through local ports introduces dependency constrains between those tasks, which requires invocation of the E machine after each task has completed execution in order to update the list of events for each trigger in readQ, which may introduce significant runtime overhead, and high memory cost. Thus in this case expressiveness has been trade off for runtime performance.

The target platform for the micro HTL implementation is reppresented by the Robostix [44], which is based on ATmega128 [47] microcontroller (clock frequency 16MHz, flash memory 128KB, EEPROM data memory 4096B, SRAM data memory 4096B, two timers/counters on 8bit and two on 16bit, two full duplex USART, I2C interface, support for in-system programming, etc.) from Atmel AVR.

7.1. Micro Embedded Machine

The micro E machine is the central part of the micro HTL runtime (Fig. 7.1). Micro HTL runtime also contains the *1ms timer* and the *micro EDF scheduler*. The 1ms timer is a timer that has a resolution of 1ms that is used as a time event generator. The micro EDF scheduler is an EDF scheduler [15] that is used to schedule tasks released by the micro E machine. The micro E machine runs as a task with deadline zero. When there is no instruction to be interpreted and no active trigger in any of the three trigger queues (i.e., *writeQ*, *switchQ*, and *readQ*) the task in which the micro E machine runs, terminates. Before the task in which the micro E machine runs, terminates, the 1ms timer will be set to release the micro E machine again after a time interval equal to the earliest time event on which one of the triggers in any of the three queues depends.

7.1.1. Micro EDF Scheduler

The micro EDF scheduler has been developed based on the *Super Simple Tasker* or SST [48]. SST is a priorities based scheduler that has been designed to work on a microcontroller. The main advantage of SST is that context switching is very cheap since it
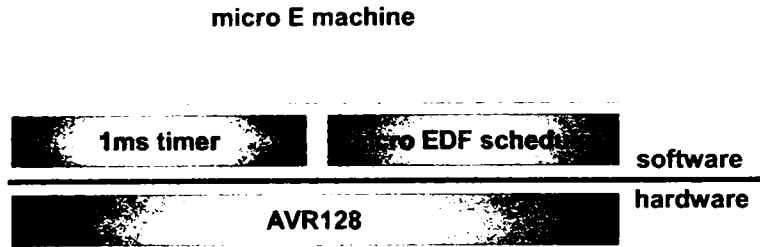
micro E machine

Fig. 7.1: Micro HTL runtime

uses a single stack to store context for all tasks. There are certain limitations regarding the tasks that can be scheduled by SST, thus a task has to be finite, it should contain no synchronization points, and it should not change dynamically its priority. Since HTL tasks are pure functional tasks, that are finite, periodic, and have a fix deadline, they meet the above limitations.

The micro EDF scheduler, schedules tasks based on their deadline, i.e., task with the earliest deadline will be run first. The micro EDF scheduler assumes that tasks can be released either be another task, or from outside (e.g., from an interruption event handler). When a task t releases for execution another task $t`$, task $t`$ must have a deadline greater or equal with the deadline of task t. The scheduler maintains an ordered list of tasks that have been released for execution, i.e., whenever a task is released it will be inserted into this list ordered ascending by its deadline. When a task finishes execution the first task in the list will be executed. When a task is released from outside, the schedule function has to be invoked manually. However, the schedule function does not have to be invoked manually if the task is released from another task, since it will be invoked automatically when task finishes execution and since the task that is released can not have a deadline smaller than the deadline of the task that does the release, it is obvious that the currently running tasks is still the task with the earliest deadline.

In the case of micro E machine implementation, the micro E machine runs as a task with a deadline of zero, while all other tasks have a deadline greater than zero. The only task that can release other tasks is the task in which the micro E machine runs. After the micro E machine task has finished execution it will be released for execution again after a period of time by the 1ms timer.
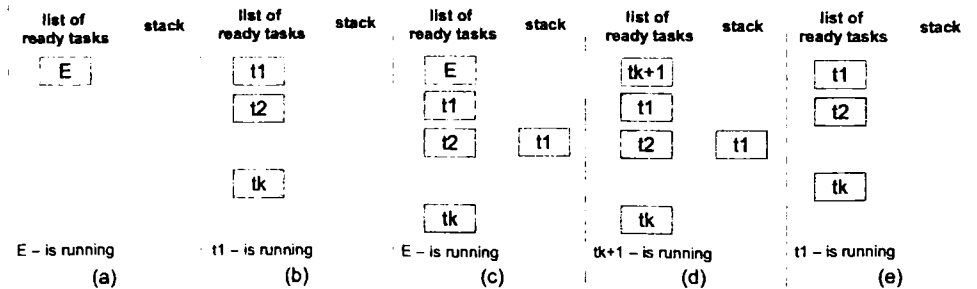
Fig. 7.2: Micro EDF scheduling example

In Fig. 7.2 it is presented an example of how the micro EDF scheduler works. Initially there is no task running except the micro E machine task (Fig. 7.2-a); during its execution the micro E machine releases for execution k tasks (e.g., t_1, t_2, ..., t_k). After the micro E machine task finishes execution, task t_1, which has the earliest deadline of all k tasks, is executed (Fig. 7.2-b). Next (Fig. 7.2-c) the micro E machine is released again from the event handler of the 1ms timer, this causes task t_1 to be suspended and its context to be saved onto the stack, and the micro E machine task is executed again. During the second execution of the micro E machine task, a new task (t_{k+1}) is released for execution; since this task has a deadline that is earlier than the deadline of t_1 it will be executed after the micro E machine task completes its execution (Fig. 7.2-d). Finally, after task t_{k+1} finishes, task t_1 will be resumed.
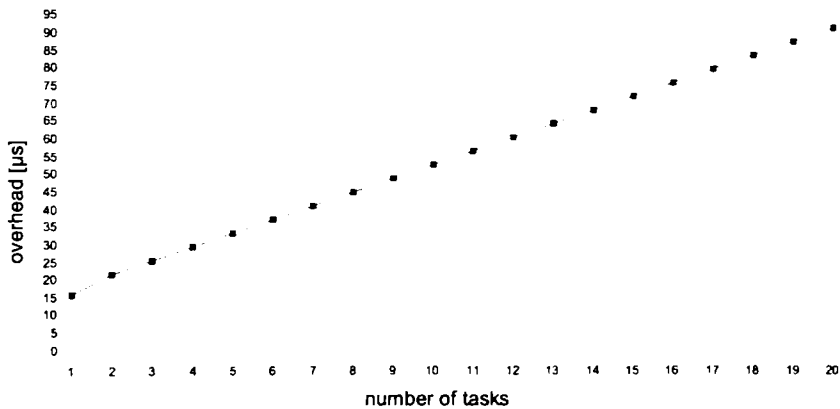
Fig. 7.3: Runtime overhead introduce by the release task operation in the worst case

Since the list of tasks that has to be executed is a list ordered by task deadline, it means that the time consuming operation in case of the micro EDF is the release task operation. Thus, release task operation has been benchmarked; in Fig. 7.3 it is presented the evolution of the runtime overhead introduced by the release task operation in the worst case. The worst case for the release task operation is represented by the case when the task that is released has the latest deadline of all already released tasks. In case of a single task the worst case runtime overhead is around $15\mu s$. The worst case runtime overhead increases with $7\mu s$ for each task. Thus, the runtime overhead introduced by the release task operation in the worst case is linear in terms of number of tasks.

7.1.2. Micro Embedded Machine Implementation

Implementation of the micro E Machine has to consider both the limited amount of memory that is available on a microcontroller and the low speed processing unit of a microcontroller. Thus, in order to minimize the size of memory used for data, each data structured has to be carefully analyzed so that the memory is not wasted. On the other hand, knowing that the processing power of a microcontroller is limited the runtime overhead introduced by the micro E machine has to be as low as possible. The micro E machine presented in this section has been developed starting from the HE machine that has implemented in C for Unix (Chapter 3).

In Fig. 7.4 it is presented the structure of the micro E machine. The micro E machine consists of an HE code interpreter and a set of lists, queues, and tables. The HE code

interpreter either interprets HE code or checks the three trigger queues for enabled triggers; when there is no instruction to be executed and no active trigger in any of the three queues, the HE code interpreter computes the smallest time interval after which at least one of the triggers in any of the three queues will get enabled, and sets the 1ms timer to release the HE code interpreter after that period of time. The micro E machine uses no dynamic memory allocation; everything is statically allocated at compile time.
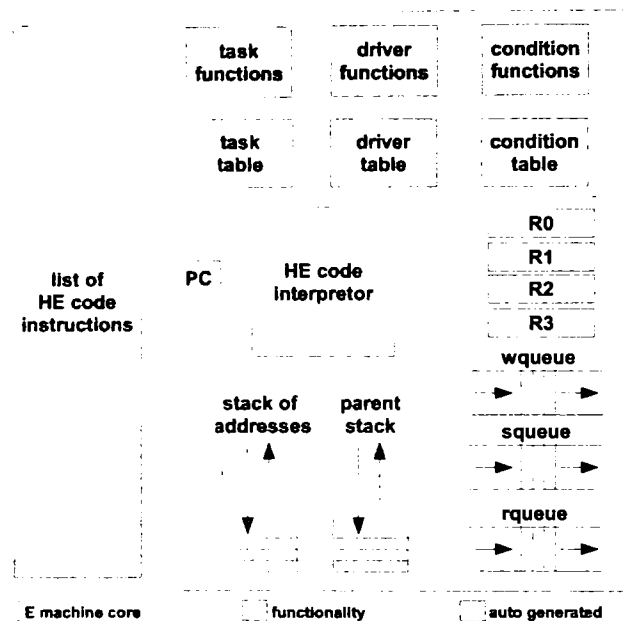
Fig. 7.4: Micro E machine

The C code from which the micro E machine is compiled, consists of three groups of code: C code that implements micro E machine functionality (i.e., HE code interpretor), C code that implements functionality of the HTL description to be run, and C code that is generated by the HTL compiler when compiling the HTL description. Thus every time the HTL description changes the entire micro E machine has to be recompiled.

The micro E machine contains three statically allocated tables: task table, driver table, and condition table. All the three tables are generated by the micro HTL compiler. Task table associates an index with a task function (i.e., a pointer to a C function that implements the functionality of a task); when a task has to be released for execution the *release*(t) instruction is used, where t is the index of the task function of the task that has to be released. Driver table associates an index with a driver function (e.g., a pointer to a C function that implements the functionality of a driver); when a driver has to be invoked the *call*(d) instruction will be used, where d is the index of the driver function of the driver that has to be invoked. Condition table associates a condition function (i.e., a pointer to a C function that implements the verification of a condition and returns one if condition is met and zero if condition is not met); when a conditional jump has to be made a *jumpIf*(cnd,a) instruction is used, where cnd is the index of the condition function that has to be evaluated and a is the address where the execution will jump if the condition is true. The micro E machine contains also four registers (e.g., R0, R1, R2, and R3), which in the micro E machine are represented as pointers to triggers.

One of the optimizations that has been done for the micro E machine, in order to

save data memory, was the encoding of the HE code instructions. In Fig. 7.5 it is presented the encoding of an HE code instruction. Each HE code instruction is encoded on 24 bits (3 bytes). The first five bits are used to encode instruction code; next eleven bits are used to encode first argument (in many cases this is an HE code address); last eight bytes are used to encode second and third arguments. Since the address parameter of an HE code instruction is limited to 11 bits this will limit the maximum size of an HE code program to 2048 instructions, nevertheless this is more then enough for a program that is intended to be run on a microcontroller.
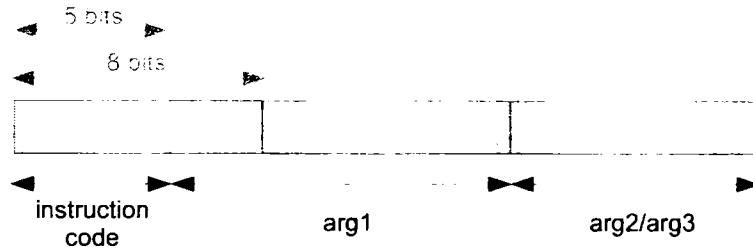
Fig. 7.5: Instruction encoding

All the HE code instructions have a maximum number of two arguments, except the three future instructions (e.g., *writeFuture*, *switchFuture*, and *readFuture*), and the *updateChildren* instruction. Since for the micro HTL there is no dependency relation between tasks, the third argument for the three future instructions (i.e., the lists of task completion events) is never used, thus the future instructions can be seen as a two arguments instruction also. For the *updateChildren* instruction arguments two and three can be encoded in the last eight bits of the instruction; the two arguments can take relatively small values, i.e., between 0 and 3. In Appendix C is presented encoding for each instruction.

The stack of addresses was build around a statically allocated array of integers on 16 bits in order to optimize each operation with the stack. The maximum size of the stack is computed by the HTL compiler.

The trigger queue is based on a double linked list, which has triggers as nodes. A trigger is represented as a structure that uses an unsigned 16 bits integer for the address, an unsigned 32 bits integer for the time event on which the trigger gets enabled, a double linked list that contains child triggers of the trigger, and a pointer to the parent trigger. Since a trigger can be a node in more then one double linked list at the same moment, i.e., trigger queue, parent stack, and children list of another trigger, it contains two arrays of pointers to triggers that are used to build the double linked list connections. In order to optimize runtime overhead triggers in all the trigger queues are ordered after the time event on which they have to activate. Triggers are statically allocated; the maximum number of triggers is computed by the compiler.

7.1.3. Micro E Machine Performance

Performance of the micro E machine implementation has been evaluated based on four experiments. In the first experiment the time interval between two consecutive releases of the same task has been measured. The HTL description that has been used in the first experiment implements the altitude and attitude control of a helicopter [11] (Subsection 6.3.1). The program has a period of 20ms. In Fig. 7.6 it is presented the time interval between two successive releases of the *groundConnect* task; the results have been

collected for over 20min. As shown in the figure, in one case the time interval between two successive releases of task *groundConnect* is off by 0.5ms, while in the rest of the cases it is off by less than 4μs. The second experiment is similar to the first one, just that another task has been added to the set of running tasks, in order to simulate heavy load. With the new task added, the load of the period was around 95%. As shown in Fig. 7.7 the performance of the micro E machine is preserved, as long as there are enough resources, even in heavy load conditions.
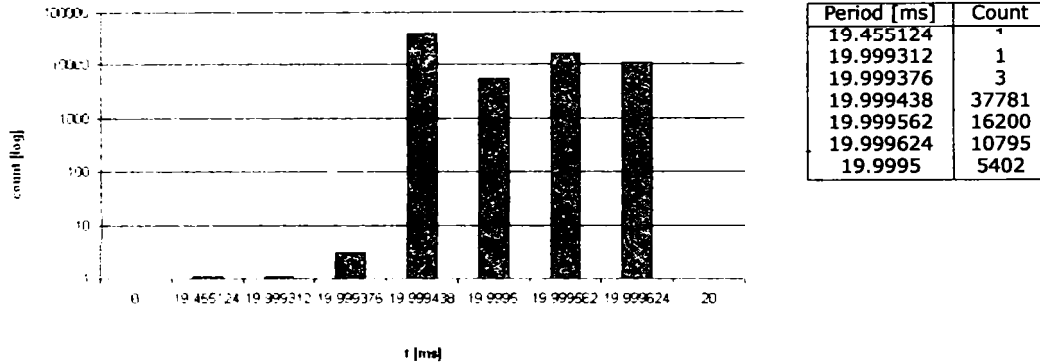
Period [ms]	Count
19.455124	1
19.999312	1
19.999376	3
19.999438	37781
19.999562	16200
19.999624	10795
19.9995	5402

Fig. 7.6: Time interval between two consecutive releases of task *groundConnect*

Period [ms]	Count
19.464938	1
19.999376	18576
19.999438	2
19.9995	83609
19.999562	1
19.999624	18576
19.999688	1
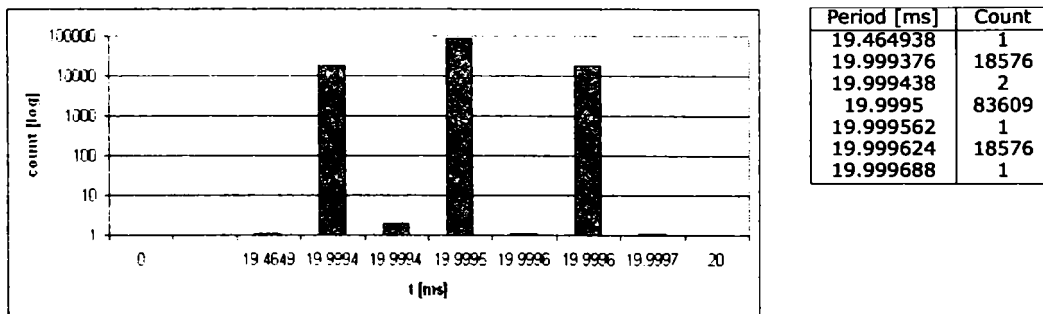
Fig. 7.7: Time interval between two consecutive releases of task *groundConnect* for a period load of 95%

In the third experiment the memory usage has been measured. In order to measure the amount of memory that is occupied by the micro E machine independent of the HTL description for which the micro E machine was compiled, the micro E machine has been compiled for an HTL description that has one program, one module, one mode, and one task. Fig. 7.8 shows that the micro E mchine needs at least 16Kb of program memory, which represents 12.2% of the entire program memory available on an ATmega128 microcontroller, and at least 710 bytes of data memory, which represents 17.3% of the entire data memory available on an ATmega128 microcontroller. Of course the amount of program and data memory used by the micro E machine is directly proportional with

the complexity of the HTL description for which the virtual machine has been compiled, i.e., for the JAviator control program about 25% of the program memory is occupied and about 65% of the data memory is occupied.
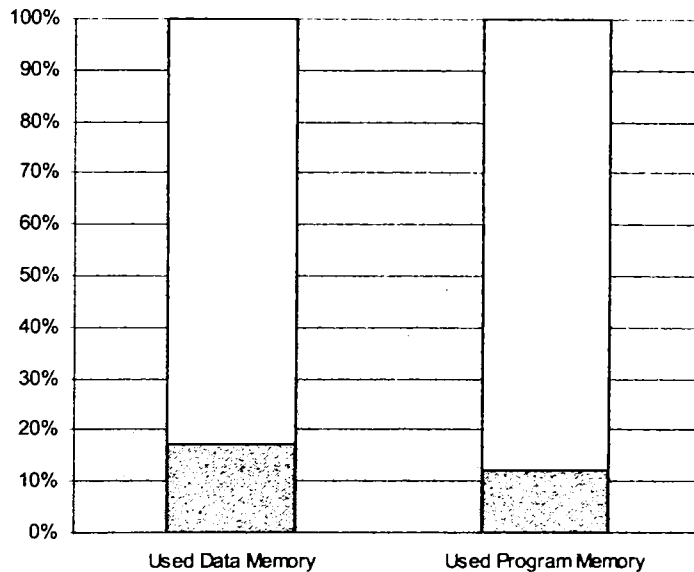
Fig. 7.8: Memory usage

In the last experiment the runtime overhead introduced by the micro E machine has been measured. For measuring the runtime overhead a high precision timer (62.5ns) has been used. The overhead was measured as follows: whenever the micro E machine has to be invoked the timer is started and before the micro E machine finishes execution the timer is stopped and the elapsed time, which represents the micro E machine runtime overhead, is computed. The experiment has been performed for the JAviator controller (Subsection 6.3.1). In Fig. 7.9 it is presented the overhead introduced by the micro E machine for several time periods. In the figure are presented time periods, in which different modes are executed, for the same set of modes the runtime overhead is always the same or there is very little variation. As presented in the figure the highest runtime overhead is 3.17ms. Higher runtime overhead is always at the beginning of the period, this is because at the beginning of the period the mode switches have to be checked, and the HE code that preserves the hierarchy has to be interpreted.

In Fig. 7.10 it is presented the total overhead introduced in a period by the execution of the micro E machine. In the figure are presented total overheads for several periods in which different sets of modes are executed, for the same set of modes the overhead is always the same or there is very little variation. The highest overhead is 4.93ms, which represents 24.65% of the entire 20ms period.

In order to measure the efficiency of micro E machine, the efficacy indicator (E) [1] will be used. Efficacy is defined as the ratio between the time the processor is available for executing application tasks and application period of execution. Thus for the overheads presented in Fig. 7.10 there is a maximum efficacy of $E_{min} = 0.81$ and a minimum efficacy of $E_{max} = 0.75$.
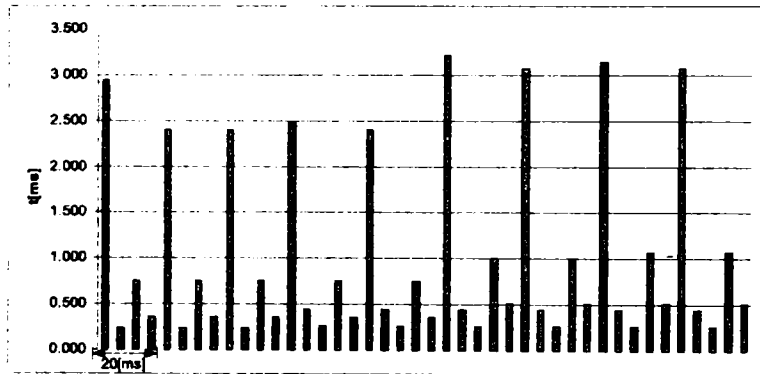
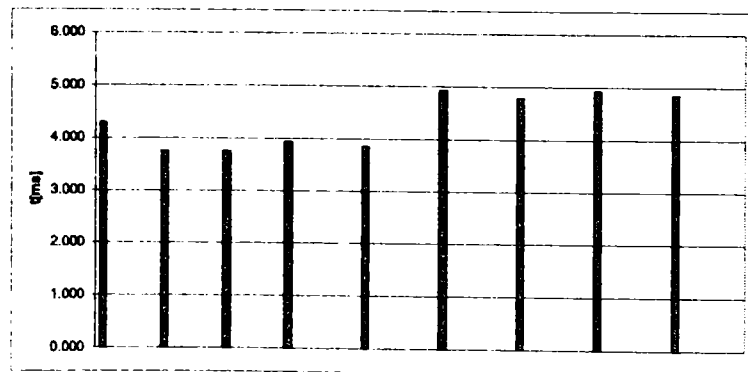Fig. 7.9: Micro E machine runtime overhead for JAviator control application, , for different modes combinations

Fig. 7.10: Micro E machine total runtime overhead over a period for JAviator control application, for different modes combinations

7.2. Micro HTL Compiler

The micro HTL compiler was implemented starting from the hierarchy-preserving HTL compiler presented in Section 4.2. The micro HTL compiler takes as input an HTL description and generates the HE code for it; the compiler also generates the task table, diver table, and condition table (Fig. 7.11), which are tables that associate a task, driver, and a condition, respectively, with an index, which is used in the HE code generation process to refer to a task, a driver, and a condition, respectively.

HTL
description

micro HTL compiler

| HE | task | driver | condition |
| code | table | table | table |
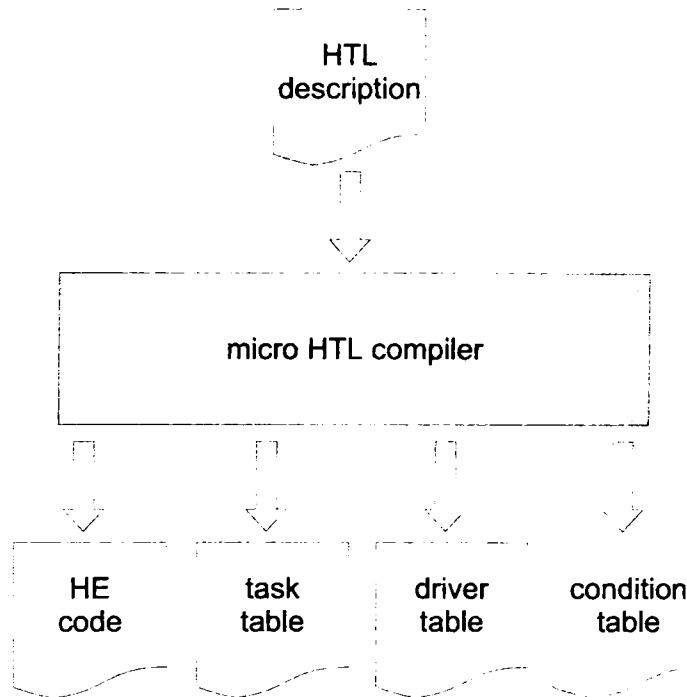
Fig. 7.11: Micro HTL compiler overview

Before generating HE code, the micro HTL compiler checks that the HTL description is well-formed and time-safe (Fig. 7.12). The *Type Checker*, verifies that formal parameters of a task declaration and actual parameters of a task invocation match, and that there are no references to program elements that are not declared. The *Frequency Checker*, checks that the frequency of a communicator matches the frequency of all the modes in which the communicator is accessed (i.e., read or written), and that hierarchy constraints are met. The *Schedulabilty Checker*, verifies that the HTL description is schedulable for a given architecture, which is specified though the WCET of each task. The *Tables Generator*, generates the task table, driver table, and condition table. The emph HE Code Generator, generates the HE code for the program.

The compiler algorithm implemented by the micro HTL compiler is the hierarchy-preserving HTL compiler algorithm (Section 4.2), which was optimized not to generate HE code for empty units, i.e., units in which nothing happens (e.g., no task is released and no communicator is read or written). The micro HTL compiler was extended to compute the maximum size of address stack, the maximum size of parent stack, and the maximum number of triggers that are needed to run the compiled HTL description. All
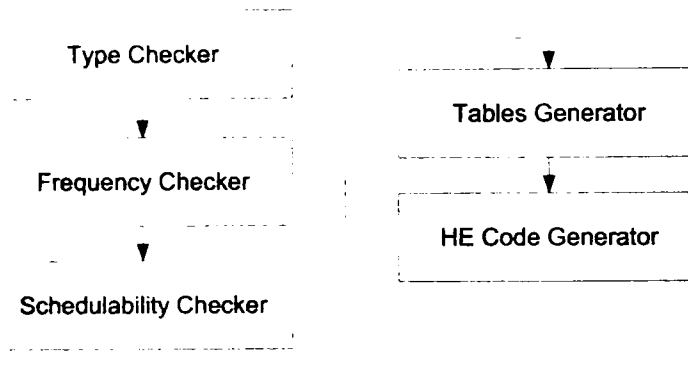
Fig. 7.12: Micro HTL compiler

this information is used to statically allocate address stack, parent stack, and the triggers that will be used at runtime.

7.3. Case Study

In this section are presented two examples of real-time control applications that have been implemented using micro HTL. The first example represents a controller for the 3TS plant. The second example implements the JAviator low-level controller.

7.3.1. 3TS Controller

The first example of real-time control application implemented using micro HTL, represents a controller for the 3TS plant. The hierarchical structure of the HTL description is similar to the one presented in Section 5.2, the only difference is represented by the fact that the functionality of the filter tasks has been moved into t_read task (Fig. 7.13). The entire HTL description can be found in Section E.3.

Although the structure of the HTL description is similar to the one presented in Section 5.2, the timing has been changed (Fig. 7.14). The program consists of six tasks that are executed every 250ms. Task t_read has an LET of 50ms, e.g., it reads no communicator and updates the second instance of communicators $h1$ and $h2$. Tasks t_T1 and t_T2 read the second instance of communicators $h1$ and $h2$, respectively, and update the fourth instance of communicators $u1$ and $u2$, respectively; both t_T1 and t_T2 have an LET of 100ms. Tasks $t\_estimateV1$ and $t\_estimateV2$ read the fourth instance of communicators $h1$ and $h2$, respectively, and update the sixth/first instance of communicators $v1$ and PI_SF1, and $v2$ and PI_SF2, respectively; tasks $t\_estimateV1$ and $t\_estimateV2$ have an LET of 100ms. Finally, task t_write, reads the fourth instance of communicators $u1$ and $u2$, and updates no communicator, thus it has an LET of 100ms.

Data-flow between tasks is similar to the data-flow presented in Section 5.2, with the observation that there are no filter tasks, thus tasks $t\_estimateV1$ and $t\_estimateV2$ communicate with task t_read through communicators $h1$ and $h2$, respectively.

7.3.1.1. Timing Analysis

For timing analysis, the WCET of each task has been measured using a high precision timer. Given the limited computation power of the microcontroller, the runtime overhead
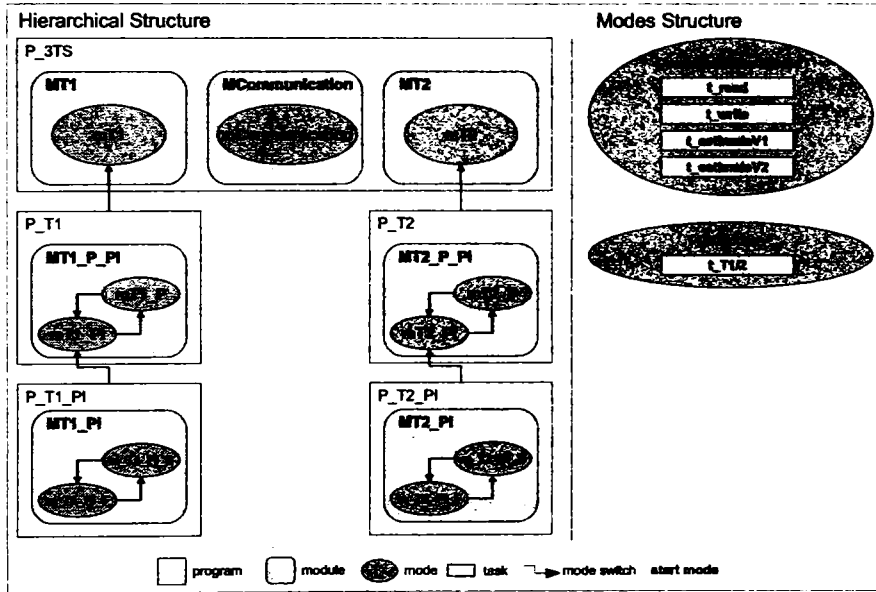
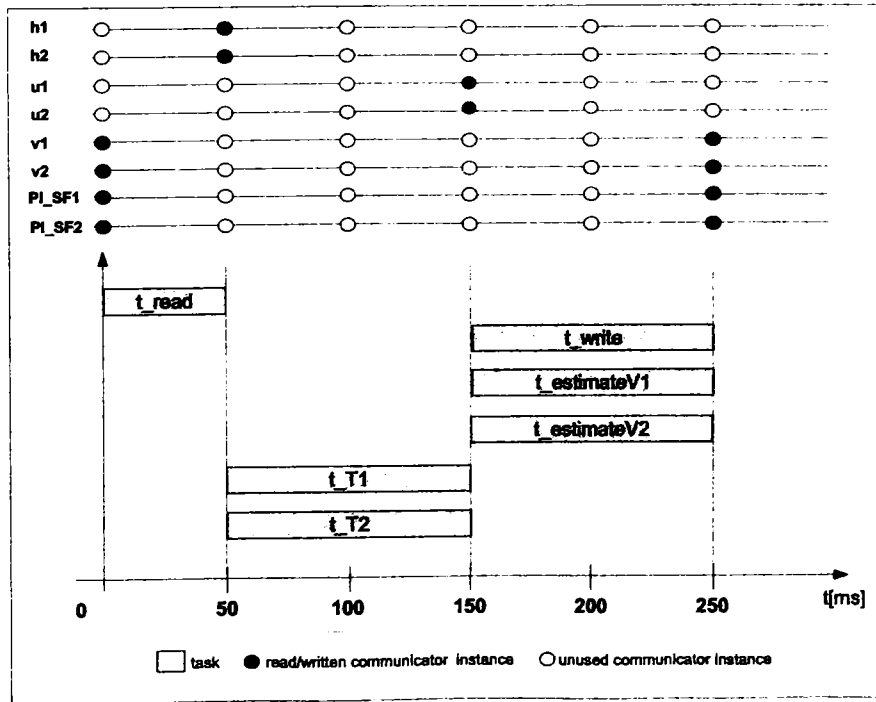Fig. 7.13: 3TS Controller: Hierarchical Structure

Fig. 7.14: 3TS Controller: Timing

introduced by each E machine invocation has been measured and has been considered in the analysis. Timing analysis for the 3TS controller is depicted in Fig. 7.15.With black arrow it is represented the LET of a task and in parenthesis it is noted the WCET of the task. With read arrows it is represented the time spent in interpreting E code for each E machine invocation.
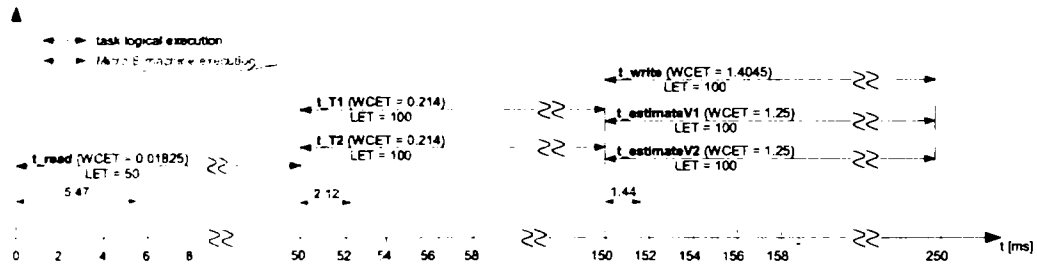

Fig. 7.15: 3TS Controller: Timing analysis

For the schedulability analysis the time overhead introduced by each E machine invocation has been added to the WCET of the task that has the earliest deadline relative to the moment when the E machine is invocated, the new WCET is further referred as extended worst case execution time (EWCET). For instance task t_read has an WCET of 0.01825ms, the overhead introduced by the E machine before executing this task is 5.47ms, thus the EWCET for task t_read is 5.48825ms which is less then the LET of the task, thus task t_read is schedulable. Similar it can be shown that all tasks in the program are schedulable. The efficacy of the E machine for this control application is around 95%.

7.3.1.2. Results

Fig. 7.16 presents the experimental results obtain by using the 3TS control application to control a simulated 3TS plant. The water level target is set to $40cm$ for tank T_1 and to $30cm$ for tank T_2. At moment $t = 0s$ the evacuation tap for tank T_1 is open; the rest of the taps are closed. Since in tank T_1 is perturbation the PI controller is used, while for tank T_2 a P controller is used (there is no perturbation). At moment $t = 70s$ evacuation tap for tank T_2 is open, thus the controller for the second tank switches to a PI controller, which succeeds to compensate for the perturbation. At moment $t = 367s$ the evacuation tap for tank T_1 is shutdown, thus the controller for T_1 switches back to P.

7.3.2. JAviator Low-Level Controller

The real-time control application presented in this subsection, implements the JAviator low-level controller, which has been presented in Section 6.3), using micro HTL. The HTL program that specifies the timing of the low-level controller is presented in visual syntax in Fig. 7.17, the entire HTL program can be found in Section E.6. The program consists of a root program, e.g., $MicroJAviator$, and two refining programs, i.e., $PControl$ and $PAuto$. The root program contains three modules: $Control$, $GroundCommunication$, and $JAviatorCommunication$.

The $GroundCommunication$ module contains two modes that can switch between each other, e.g., $mGroundConnect$ and $mGroundCommunication$. The start mode is $mGroundConnect$ mode, in which a single task is invoked, which waits for the ground station to connect. When the ground station connects the $mGroundConnect$ mode switches to
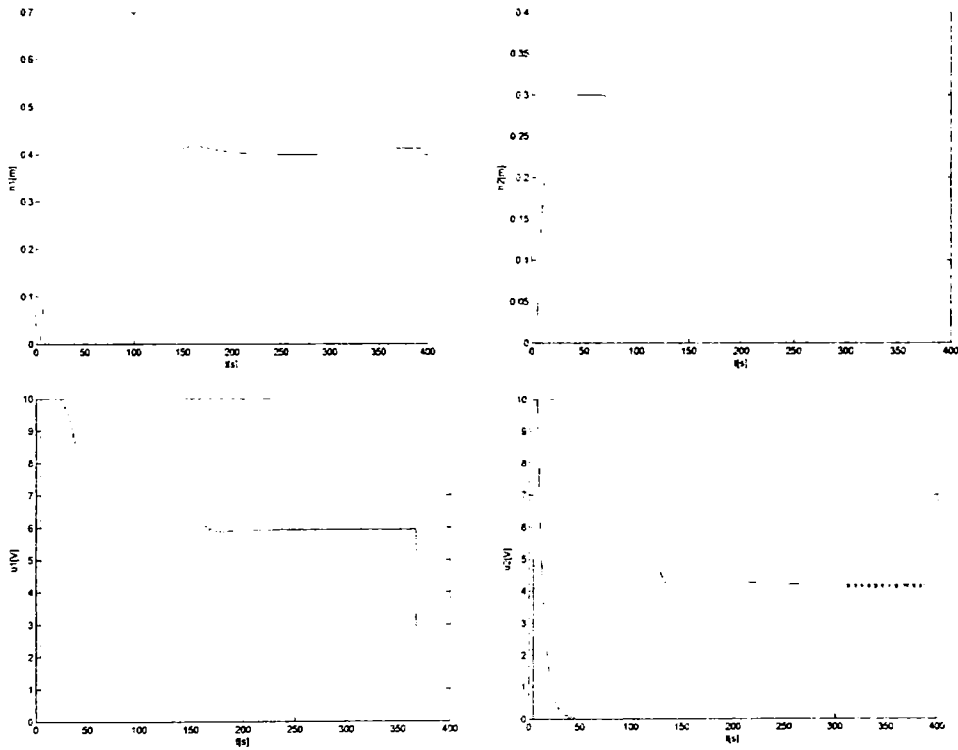
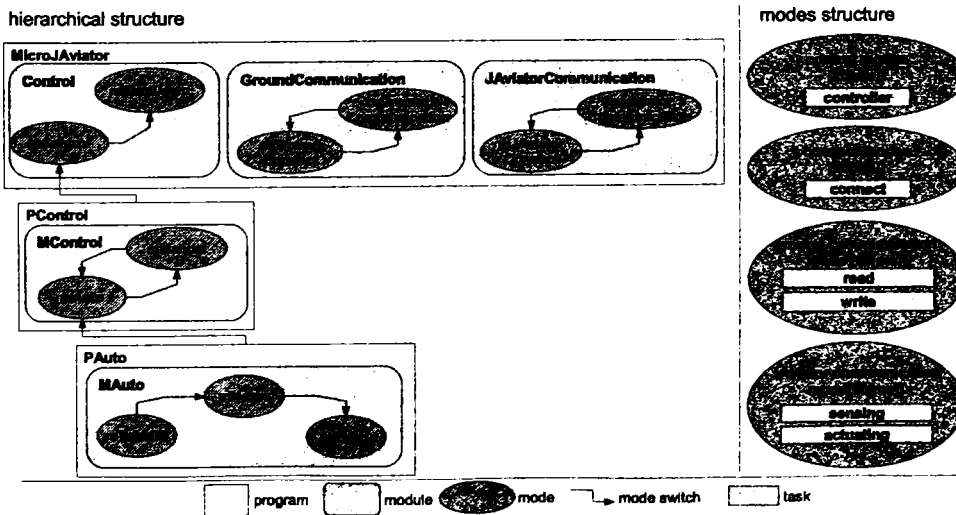Fig. 7.16: 3TS Controller: Experimental results

Fig. 7.17: The Structure of the Low-Leve Controller Implemented in Micro JAviator

mGroundCommunication, which invokes two tasks, i.e., *read* task, which reads information sent by the ground station (i.e., target values), and *write* task, which sends to the ground station reports about the altitude and the attitude of the JAviator, and about the commands that were sent to the rotors. In order to limit the traffic, the *write* tasks can not send more than 8 bytes of data in a single period, thus it needs four periods to send all the data to the ground station. In the period when the first packet is sent all the information that has to be sent (e.g., rotors commands, altitude, roll, pitch, yaw, etc.) is stored in the state of the task, and in the next three periods, the task will sent the information stored in its state and not the latest information available. When the ground station disconnects, the mode *mGroundCommunication*, switches back to mode *mGroundConnect*.

The *JAviatorCommunication* module contains two modes that can switch between each other, i.e., *mJAviatorConnect* and *mJAviatorCommunication*. The start mode is *mJAviatorConnect*, which invokes one task, which waits for the JAviator simulator to connect, when the simulator connects the mode *mJAviatorConnect* switches to mode *mJAviatorCommunication*. Mode *mJAviatorCommunication* invokes two tasks: *sensisng*, which reads sensor data sent by the simulator, and *actuating*, which sends rotors commands to the simulator. When the simulator disconnects, the *mJAviatorCommunication* mode switches back to the *mJAviatorConnect* mode.

The *Control* module contains two modes, i.e., *mControl*, which is also the start · mode, and *mShutDown*. Mode *mShutDown* can not switch to any mode, it implements the emergency shutdown procedure. Mode *mControl* can switch to mode *mShutDown*, it invokes *control* task, which is abstract (i.e., a placeholder for the task that implements the control). Mode *mControl* is refined by program *PControl*. Program *PControl* refines the *Control* mode into auto and manual mode. The auto mode is further refined by program *PAuto* into take off, hover, and land modes. In the manual mode, the commands for the four rotors are sent directly from the ground station. In the take off mode, the thrusts produce by the rotors are increased with a constant value until the gravity is compensated, when the helicopter can go into hover mode. In the hover mode a PID controller is used for the altitude and a PD controller is used for roll, pitch, and yaw, respectively. In the land mode the thrusts produced by the four rotors are decremented with a constant value until they are zero.
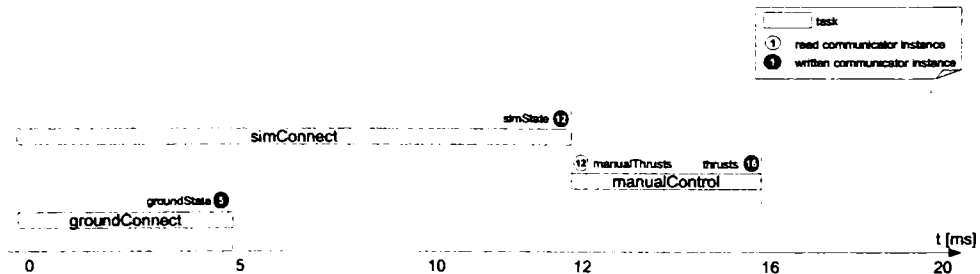

Fig. 7.18: Timing and data-flow before connect

Timing of the program can be divided in two: timing before the ground station and the simulator have connected, and timing after the ground station and simulator have connected. In Fig. 7.18 it is presented the timing of the program before the ground station and the simulator have connected. In this case there are only three tasks invoked. Task *groundConnect* is released at the beginning of the period, has an LET of 5ms, and updates the sixth instance of communicator *groundState*. Task *simConnect* is released at

the beginning of the period, has an LET of 12ms, and updates the thirteenth instance of communicator *simState*. Task *manualControl* reads the thirteenth instance of communicator *manualThrusts* and updates the seventeen instance of communicator *thrusts*, thus it has an LET of 4ms.
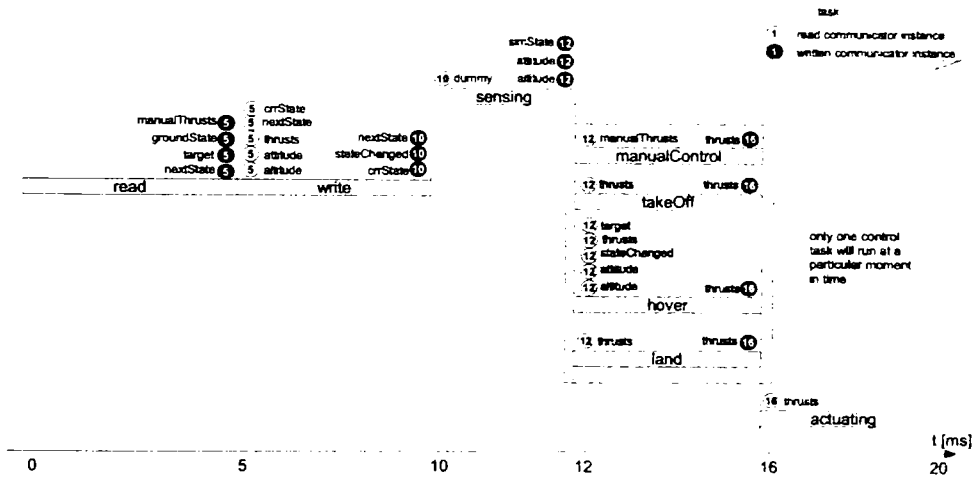

Fig. 7.19: Timing and data-flow after connect

Fig. 7.20, depicts the timing of the program after the ground station and simulator have connected, in this case there are invoked five tasks. Task *read* has an LET of 5ms and updates sixth instance of communicators: *manulThrusts, groundState, target,* and *nextState*. Task *write* has an LET of 5ms , reads the sixth instance of communicators: *crrState, nextState, thrusts, altitude,* and *attitude,* and updates eleventh instance of communicators: *nextState, stateChanged,* and *crrState*. Task *sensing* has an LET of 2ms, it reads the eleventh instance of communicator *dummy* (this is a communicator used only to set the desired LET for task *sensing,* it contains no information), and updates thirteenth instance of communicators: *simState, altitude,* and *attitude*. Task *actuating* has an LET of 4ms, it reads the seventh instance of communicator *thrusts*. The control task has an LET of 4ms, depending on the active control mode, the control task can be one of the following tasks: *manulControl, takeOff, hover,* or *land*. The control task updates the seventh instance of communicator *thrusts*.

7.3.2.1. Timing Analysis

For timing analysis, the WCET of each task has been measured using a high precision timer. Given the limited computation power of the microcontroller, the runtime overhead introduced by each E machine invocation has been measured and has been considered in the analysis. Here only two possible combinations of active modes are analyzed: before the simulator and ground station have connected and after they have connected; the rest of the combinations are similar to this two. In Fig. 7.20 it is presented the timing analysis for the HTL program before the ground station and the simulator have connected. In Fig. 7.21 it is presented the timing analysis for the HTL program after the ground station and the simulator have connected. With black arrow it is represented the LET of a task and in parenthesis it is noted the WCET of the task. With read arrows it is represented the

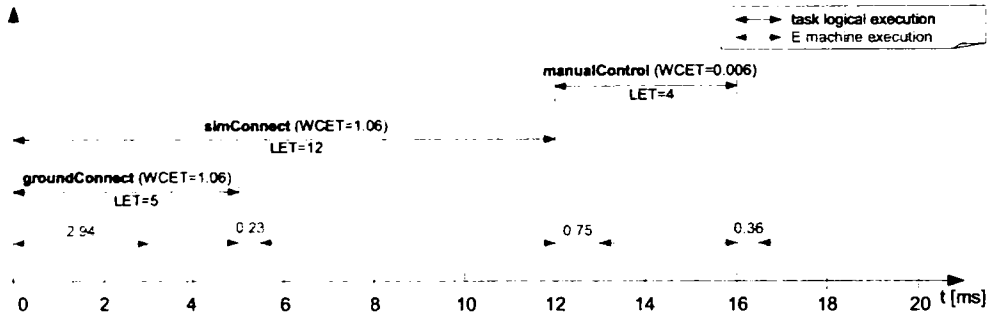time spent in interpreting E code for each E machine invocation.
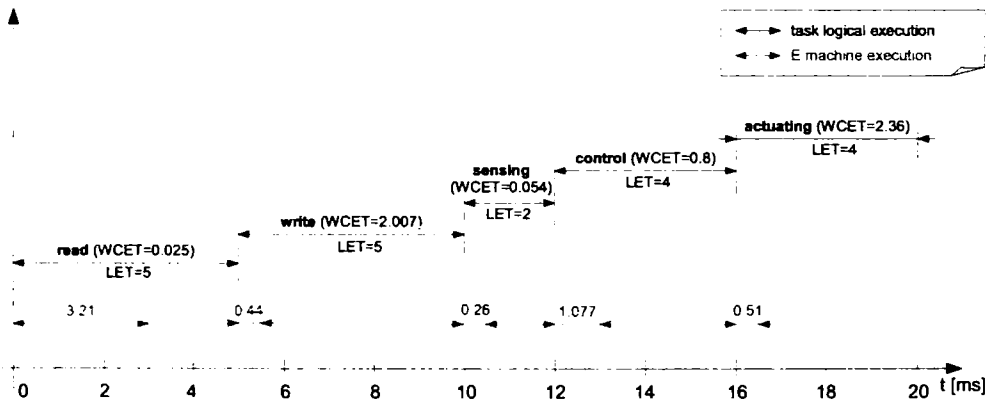

Fig 7.20: Timing analysis before connect

Fig. 7.21: Timing analysis after connect

For the schedulability analysis the time overhead introduced by each E machine invocation has been added to the WCET of the task that has the earliest deadline relative to the moment when the E machine is invocated, the new WCET is further referred as extended worst case execution time (EWCET). The schedulability analysis has been conducted using the EWCET of each task. Thus, for instance in case of Fig. 7.20 the EWCET for tasks $groundConnect$ and $simConnect$, which are executed in parallel, is $4ms$ and $1.29ms$, which means that for an LET of $5ms$ for $groundConnect$ task and an LET of $12ms$ for $simConnect$, the two tasks are schedulable. Similar it can be shown that all the task are schedulable for both scenarios that have been considered (Fig. 7.20 and Fig. 7.21). The efficacy for the two timing analyses that were presented is: 80% for the case when the ground station and simulator are not connected and 75% after the ground station and simulator have connected.

7.3.2.2. Results

In order to test the implementation of the low-level controller for the JAviator, the control application has been tested using a simulated version of the JAviator plant. In Fig. 7.22

are presented the evolution of the altitude, roll, pitch, and yaw, while in Fig. 7.23 are presented the thrusts prescribed by the controller. Although the period of the controller is $20ms$, data collection is done once every five period, thus data collection happens every $100ms$; this is way the charts do not look smooth.
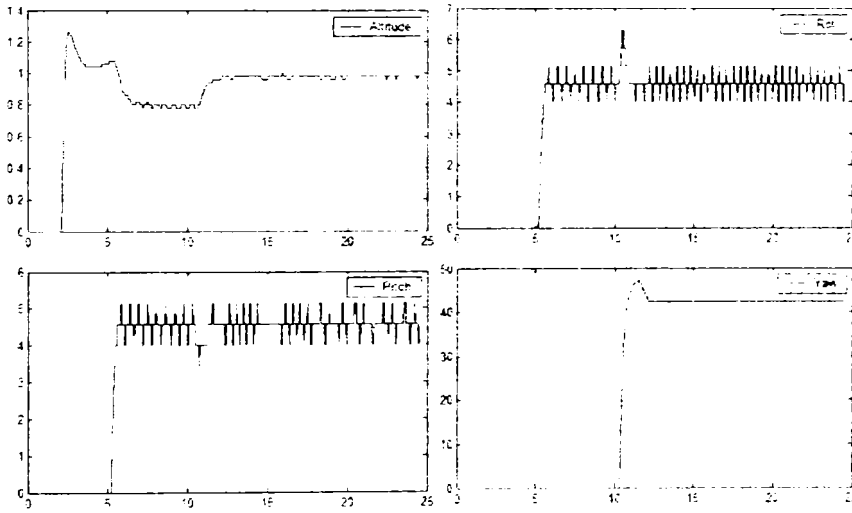

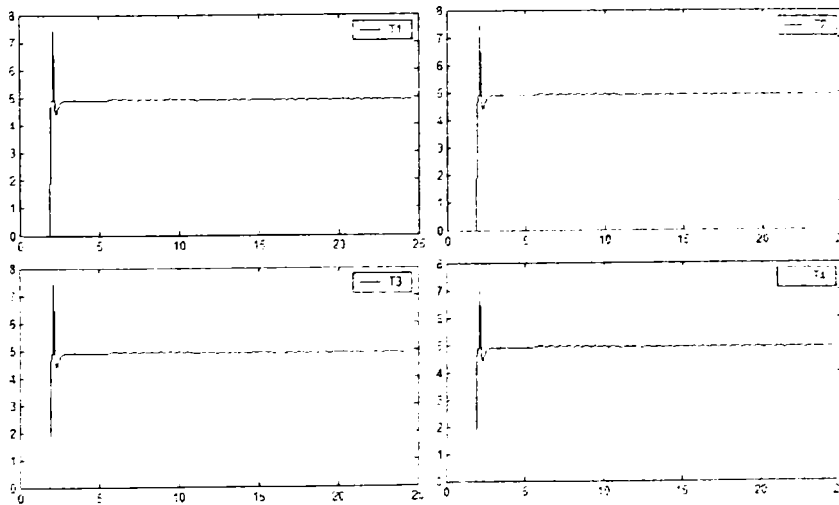Fig. 7.22: Sensors values when using low-level controller implemented in micro HTL

Fig. 7.23: Actuators values when using low-level controller implemented in micro HTL

At time zero both the ground station and the simulator connect to the controller implemented in micro HTL. The altitude target is set to $1m$ and at time $1.86s$ the helicopter

is commanded to take off. In the take off mode the thrusts for each rotor are incremented with a constant value until the gravity is compensated. At this moment the controller waits for a confirmation from the user in order to go into hover mode. As seen in Fig. 7.22 the target value is reached in about $2s$, and the overshooting is around 25% of the target value. Next, at moment $5s$ the target for roll and pitch are both set to 5 degrees; for both roll and pitch the control time is around 1s and the overshooting is zero. There is a small chattering ($+/-0.6$ degrees) for both roll and pitch; it is due to the rounding caused by the use of fixed point arithmetic operations for the implementation of the control laws. Finally at time $11s$ the target for the yaw is set to 45 degrees; the control time is around $3s$ and the overshooting is 11%.

Also there is some inter-influence between the attitude controller and altitude controller, in the sense that when there is a non-zero roll or pitch target angle, the altitude can not compensate for the variation in the thrusts, and the controlled level is lower then the target level. Nevertheless in normal flight conditions the target values for roll and pitch will be set to zero, and only when a movement in the horizontal plane is needed the targets will be set to a different value for a short time interval.

8. HTL to Simulink

In the process of development of real-time control application offline testing plays an important role. Simulink [10], which is a tool for modeling, simulating, and analyzing multidomain dynamic systems, is widely used for designing control algorithm. Simulink offers the possibility to model both plant and controller dynamics, thus allowing testing of a control algorithm before it is used on the real plant. Nevertheless even if the model of the plant is very close to the real plant, simulating the control algorithm in Simulink it is not enough to ensure that the implementation of the control algorithm for a given platform will work. This is because when the control algorithm is simulated in Simulink, it does not consider the timing of the application that will implement the controller, thus the implementation could introduce some unknown delays, which have not been accounted in the design of the control algorithm, and which in the end could make the control application not to work properly.

In this chapter, a way of modeling HTL descriptions in Simulink will be presented. The HTL compiler (Chapter 4) has been extended in order to compile an HTL description into a Simulink model. Using this new feature of the HTL compiler, it is possible to simulate not only the control algorithm, but also the timing of the real-time application that implements the controller, which should improved development of real-time control application using HTL. Beside of being able to simulate the control algorithm and the timing of the application that will implement the algorithm, modeling an HTL description in Simulink has another important advantage: the ability to generate C code for tasks directly from the Simulink schema, using Real-Time Workshop [49]. Thus, once the tasks have been modeled in Simulink functional C code for them can be generated automatically. Generated C code can be used as tasks implementation.

Modeling timing of a real-time application in Simulink is not a new idea, it has been done before for Giotto [50]. Nevertheless modeling a Giotto program is different than modeling an HTL description, since Gitto has no hierarchical structure and no communicators. Generating code from a Simulink schema has been done for Gitto and for other languages. For Lustre it has been developed a tool chain [51] that can generate a Lustre program out of a Simulink model, which is the opposite of what it is presented in this chapter for HTL, since from an HTL description it is generated the Simulink model, while for Lustre the Simulink model is created first and then from it the Lustre program is generated.

8.1. Increment/Decrement Counter

In order to facilitate the description of the method that maps an HTL description to a Simulink model, an HTL description that increments every second with a variable step (i.e., initial step is 1, then 5, and the last step is 10), a counter until it reaches the value of 50, then the counter is decrement every second with a constant step (i.e., 1) until the value of the counter reaches 0, then the entire process repeats. The hierarchical structure

of the description is presented in Fig. 8.1, while the timing and data-flow are presented in Fig. 8.2.
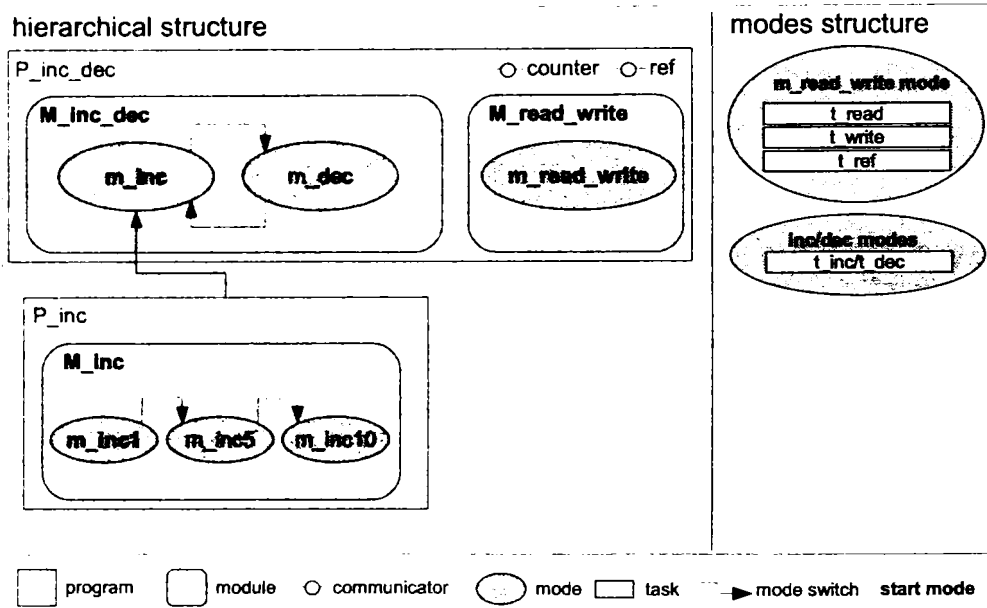

Fig. 8.1: Increment/decrement counter: structure

The description consists of two HTL programs: *P_inc_dec* and *P_inc*. The root program (i.e., *P_inc_dec*) declares two communicators, e.g., *counter* and *ref*. The *counter* communicator has a period of 100ms, it represents the counter which is incremented and decremented, and *ref* communicator has a period of 100ms and it is an auxiliary communicator, its role will be explained in Section 8.2. The root program contains two modules: *M_inc_dec* and *M_read_write*. Module *M_inc_dec* specifies the timing of the increment and decrement operations; it consists of two modes, *m_inc* and *m_dec*, that can switch between one another. Mode *m_inc* invokes every second task *t_inc*, which is a place holder (e.g., it is an abstract task) for the tasks that implement the three possible increment operations. Mode *m_inc* is refined by program *P_inc*, and it can switch to mode *P_dec*; the switch becomes enabled when the counter value reaches the value of 50. Mode *m_dec* invokes every second task *t_dec*, which decrements with one unit the second instance of the counter communicator and writes back the result to the third instance of the same communicator. Mode *m_dec* can switch to mode *m_inc*; the switch gets enabled when the counter reaches zero. The module *M_read_write* specifies the timing for the communication with the environment (i.e., counter sensing and actuating), it contains one mode, *m_read_write*. Mode *m_read_write* has a period of one second and invokes three tasks: task *t_read* reads the counter from the environment and updates the second instance of communicator *counter*, task *t_write* reads the third instance of communicator *counter* and writes its value to the environment, and task *t_ref* whose role will be explained in Section 8.2.

The program *P_inc* refines the increment operation into three operations: increment with 1, increment with 5, and increment with 10. It contains one mode for each operation, i.e., mode *m_inc1* invokes a task that increments the counter with 1, mode *m_inc5* invokes a task that increments the counter with 5, and mode *m_inc10* invokes a task that increments the counter with 10. All the three modes have exactly the same
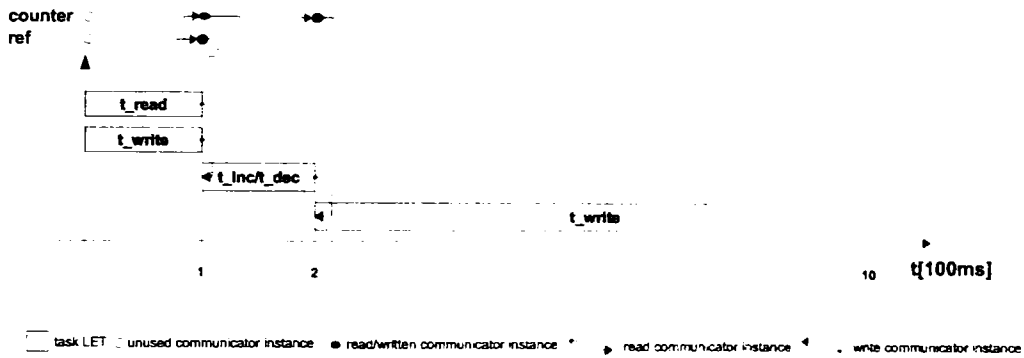
Fig. 8.2: Increment/decrement counter: timing

timing as the parent mode *m_inc*. Mode *m_inc1* switches to mode *m_inc5* when counter reaches the value of 5, and mode *m_inc5* switches to mode *m_inc10* when counter reaches the value of 20. Mode *m_inc10* does not switch to any mode.

8.2. Mapping HTL Programming Elements to Simulink Bocks

A control application monitors sensor values of a plant based on which it will compute a command, which will be sent to the plant, throughout the actuators, in order to ensure that the plant will achieve a desired state, which is defined through the so called reference values. Thus the HTL descriptions that are to be translated into Simulink models, consists of one and only one communication module with the plant and with the environment, which does the sensing and actuating, and which reads the reference values. Otherwise the program can contain as many modes, modules, and programs as they are needed, but they will communicate with the environment and with the plant indirectly through the communication module. The Simulink blocks that will be used in this section are explained in Appendix F.
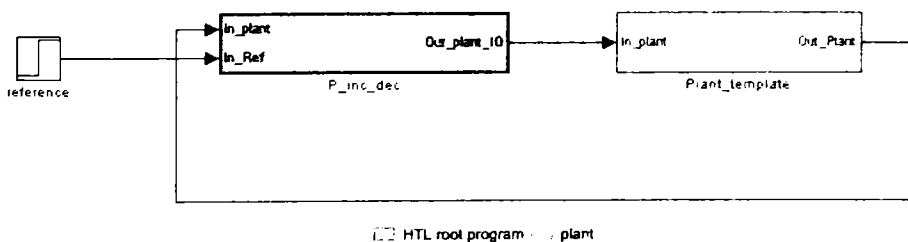
Fig. 8.3: Increment/decrement model first level

Having the idea of control application in mind the Simulink model that is generated from an HTL description will have on the top level two subsystems that communicate to each other. One subsystem will represent the HTL description (i.e., the control application), this subsystem will be referred as *controller subsystem*, while the other subsystem will represent the plant, and it will be referred as *plant subsystem*. The controller subsystem reads the sensor values from the plant subsystem and the reference values, and writes

the commands back to the plant subsystem. In Fig. 8.3 it is presented the first level of the Simulink model that has been generated for the increment/decrement HTL description.

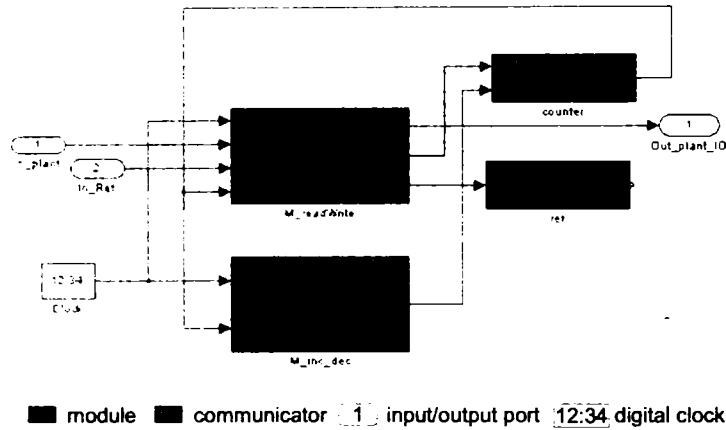■ module ■ communicator ⟨1⟩ input/output port ⟨12:34⟩ digital clock

Fig. 8.4: Simulink model increment/decrement root program

For any HTL program an atomic subsystem block will be generated, which will contain an atomic subsystem block for each HTL module in the HTL program and a subsystem block for each communicator. The difference between the root HTL program and any child HTL program consists in the fact that for the root program a *digital clock* is generated, while a child program receives the clock from the parent program. Also the inputs and outputs of the root program communicate with the plat, while for a child HTL program they are connected to communicators in the parent program. The digital clock that is generated for the root program is used to simulate time events, it should have a period that is at least an order of magnitude smaller then the smallest period in the HTL description; since the smallest period in the current implementation of HTL is 1ms, for the digital clock should be enough to have a period of 100us. In Fig. 8.4 it is presented the content of the atomic subsystem that represents the root program of the increment/decrement HTL description.

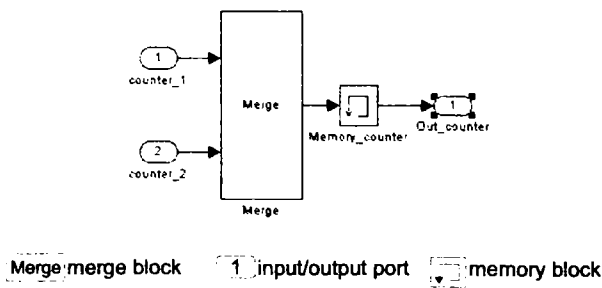
Merge⟩merge block ⟨1⟩input/output port ⟨▼⟩memory block

Fig. 8.5: Simulink model for counter communicator

An HTL communicator is modeled in Simulink as a subsystem that contains a merge block that is connected to a memory block. The subsystem always has one input,

but it can have as many inputs as modules, since for each module that writes to a certain instance of the communicator there has to be an input. At most one input can carry a value at a particular moment in time; this is ensured by HTL constraints since no communicator instance can be written from two different modes. In Fig. 8.5 it is presented the model of the counter communicator from the increment/decrement HTL description. The communicator can be written from both modules in the root program, thus it has two inputs.

■ mode   Merge merge block   1   input/output port   mode selector
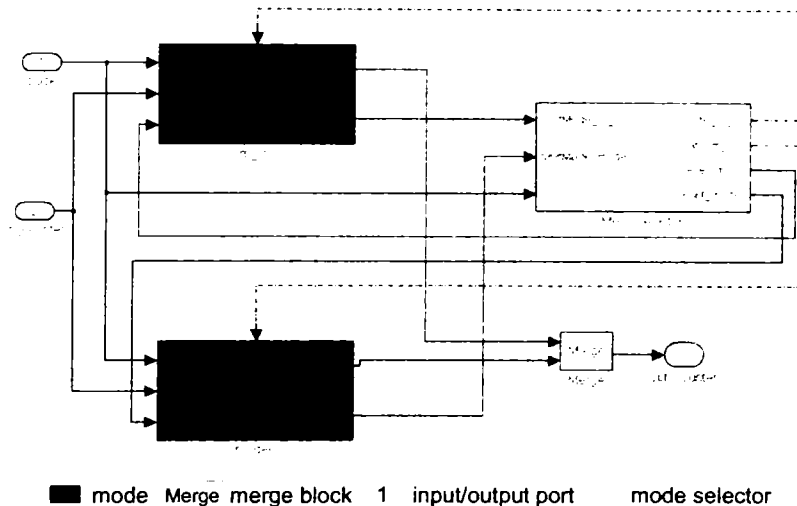
Fig. 8.6: Simulink model for M_inc_dec module

The Simulink model of a module consists of a set of action subsystems, which represent the modes that are in the module, a mode selector, which determines which mode should be invoked, and for each communicator that is written in more than one mode there has to be a merge block in order to merge together the signals from different modes and to produce a single signal that will be connected to the input of the communicator. The Simulink model of a module receives as an input the clock signal from the containing program and is broadcasted to each Simulink model of a mode. Except for the clock input, a Simulink model of a module has as inputs all the communicators that are read in a mode in the module being modeled. The outputs of the Simulink model of a module are represented by all the communicators written in any of the modes in the module. In order to implement mode switching each mode is associated with a unique integer number and each mode generates a signal that specifies which is the next mode. The mode selector block reads the next mode signals from each mode and activates the action subsystem that corresponds to the mode that has to be executed. For each mode in a module, the mode selector block has an output, which is connected to the action port of the action subsystem which models the mode. Also for modes that can switch to other modes, the mode selector block generates a reset signal, which ensures that a mode does not switch in the first period when the mode is executed. In Fig. 8.6 it is presented the Simulink model of the M_inc_dec module.

A mode selector consists of a merge block which merges together the next mode signals from all the modes in the containing module, a memory block which stores the last enabled mode (this block is needed in order to avoid algebric loops), and a switch case block which does the mode selection based on the signal that comes from the memory
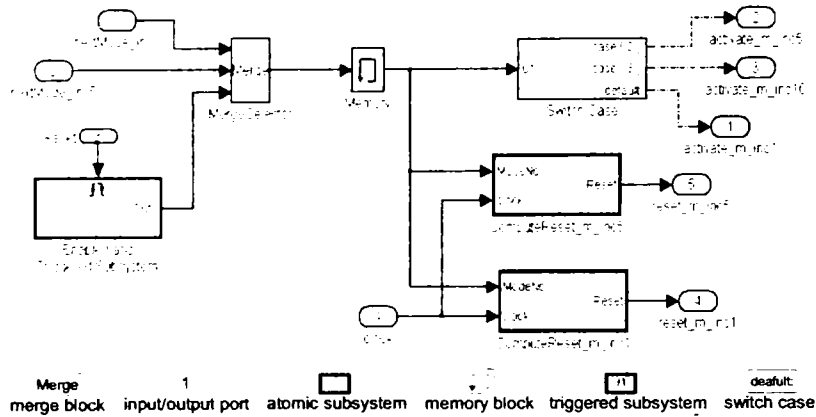
Fig. 8.7: Simulink model for mode selector in module M_inc

block. Beside the role of selecting the active mode, the mode selector block has a second role, which is the generation of reset signals. The reset signals are generated for all modes that can switch to any other mode; thus for each such mode there will be a reset mode generator. The reset signal is active only in the first period a mode is active; in order to do this, the value of the merged next mode signal at the beginning of the active mode period is stored until next mode period, the stored value is then compared with the current value of the same signal and if they are different, then one of the reset signal is activated based on the value of the merged next mode signal (which actually specifies which mode activated), this will ensure that the reset signal for a mode is active only in the first period of the mode that got active. The mode selectors that are in the modules from other programs than the root program, have a third role: to reset the active mode to the default mode whenever one of the parent modes switches. In order to do this a reset signal generated by the parent mode that switches is used in order to force the current active mode to the start mode of the module. In Fig. 8.7 it is presented the mode selector for module M_inc, which is in a child program, that is why the mode selector has a reset input.

The Simulink model of a mode consists of a set of blocks that implement mode clock generator, a set of blocks that implement tasks invoked in a mode, and set of blocks that implement mode switches. The mode clock generator computes modulo function between the system clock scaled with 10000 and the period of the mode expressed in ms. The mode clock has to be scaled by 10000 in order to achieve high precision. For modes that are not in the root program, the mode clock is generated by the parent mode (i.e., the parent mode and the child mode have the same period in a well-formed HTL program). For each task there is an atomic subsystem, which reads the mode clock signal and the communicators read by the task; the task block has an output port for each output port of the task being modeled. If two task invocations write to two different instances of the same communicator, then the two outputs will be merged by a merge block. For each mode switch in a mode a trigger subsystem is generated which implements the switch condition. The switch block has an input for each communicator read by a mode switch and one single output, which can take two values: 0 when the switch is not enabled and 1 when the switch is enabled. The switch block executes only at the beginning of the period; this is ensured by a block that reads the mode clock and if the clock is between 0 and 10, then the switch block execution will be triggered. The output of each switch block is read
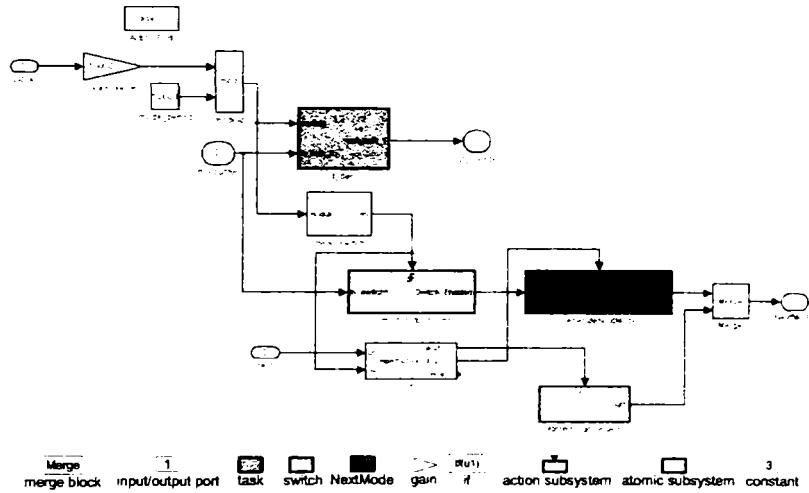
Fig. 8.8: Simulink model for mode m_dec

by an action subsystem which computes the next mode based on all the switch blocks outputs. If a mode contains at least one switch, then it also has to read a reset signal, which is active in the first period the mode is executed and which is used to disable mode switch logic and to force the next mode signal to the integer value that is associated with the mode modeled by the block.
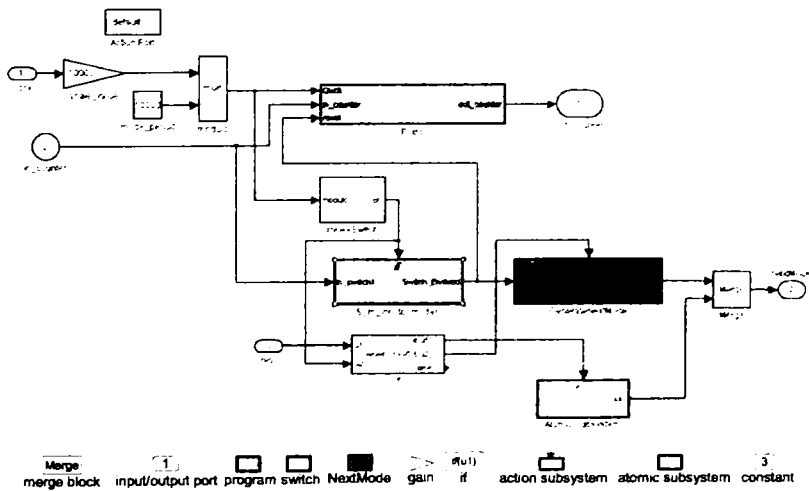

Fig. 8.9: Simulink model for mode m_inc

In Fig. 8.8 it is presented the Simulink model for the m_dec mode and in Fig. 8.9 it is presented the Simulink model for the m_inc mode. The main difference between the two models consists in the fact that mode m_inc is refined by program P_inc; it can be seen that the mode clock computed in mode m_inc is transmitted to the program P_inc. Also based on the switch block in mode m_inc the reset signal for all the modes in program

P_inc is computed, if there are multiple switches, then the reset signal has to be active whenever a switch is active. If the mode is not in the root program, then the reset signal has to be active not only when a switch is active, but also when the reset that comes from the parent mode is active.
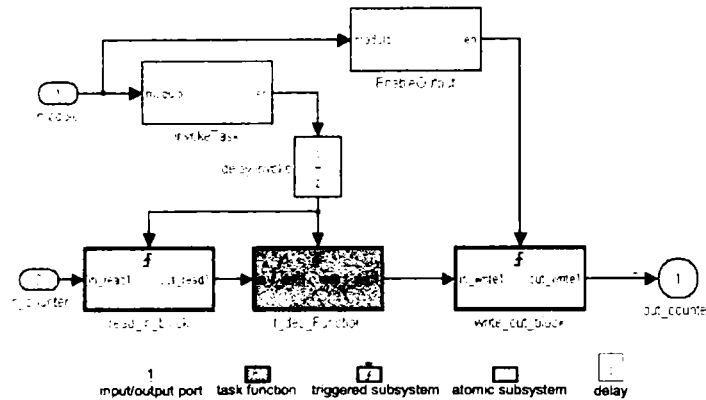

Fig. 8.10: Simulink model for task t_dec

An HTL task invocation is modeled as an atomic subsystem, which contains a triggered subsystem fro each task input and output, the implementation of the task functionality is also done in a triggered subsystem. For each input/output, logic is generated to activate the corresponding triggered subsystem based on the mode clock and on the communicator instance that is read/written. For the activation of the inputs a delay block is used in order to allow a communicator to be written before it is read. The delay is very small as compare to the entire period of a mode, thus the timing is not affected. The triggered subsystem that represents the task functionality has to be activated when the latest communicator is read, for tasks with dependencies activation of the triggered subsystem that implements the functionality has to consider the latest communicator read by any task in the dependency list. In Fig. 8.10 it is presented the Simulink model for the invocation of task t_dec, which reads instance two of communicator counter and writes back to instance three of the same communicator.

An HTL mode switch is modeled as a triggered subsystem, which is activated at the beginning of the period of a mode. A mode switch block consists of an if block which implements the switch condition and which activates one of two blocks. If the condition is true, then a block that has always one at output is activated, otherwise a block that has always zero at output is activated. The output of the two blocks is merged and written to a single output. In Fig. 8.11 it is presented the Simulink model of the mode switch that switches from mode m_dec to m_inc.

The result of simulating the Simulink model that has been generated for the increment/decrement HTL description is presented in Fig. 8.12. In the figure one can see that initially the counter is incremented with one every second until the counter value reaches the value of five, when the mode m_inc1 switches to mode m_inc5. Next the counter is incremented with five every second until it reaches the value of twenty, when the mode m_inc5 switches to mode m_inc10. In mode m_inc10 the counter is incremented with ten every second until it reaches fifty, the parent mode of m_inc10 switches to mode m_dec. Next the counter is decremented with one every second until it reaches zero, when mode m_dec switches back to m_inc and the entire process is repeated.

The HTL compiler has been extended to support generation of a Simulink model
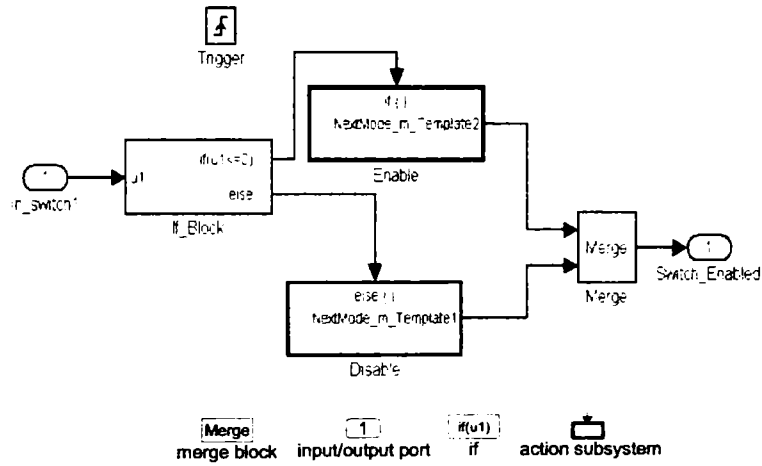
Fig. 8.11: Simulink model for mode switch m_dec to m_inc
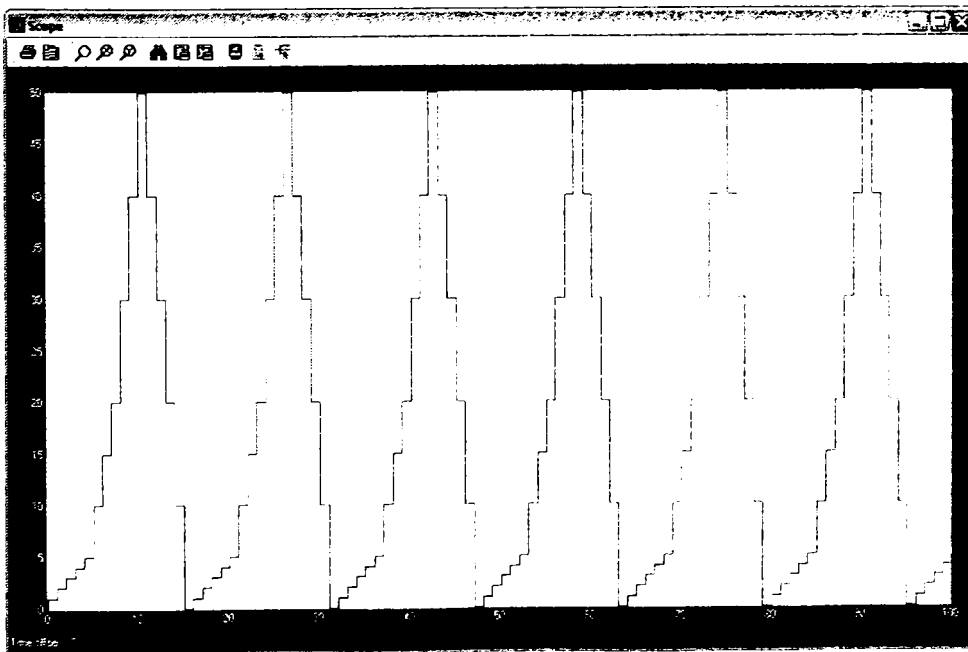

Fig. 8.12: Counter evolution

out of an HTL program. So far the only HTL programs with no task dependencies and with no more than one mode switch per mode can be converted into Simulink model with the current implementation.

8.3. Developing Real-Time Control Applications with HTL and Simulink

In Fig. 8.13 it is presented an overview of the methodology of developing real-time control application using HTL and Simulink. Based on the initial specifications for the control application, the timing of the application can be extracted and implemented as an HTL description. From the HTL description using the HTL compiler it can be generated both the HE code (represented as C code) and a Simulink model. The Simulink model of the HTL description will be used to develop the control algorithm, and it can also be used to simulate and test timing of the application; in case of errors the HTL description can be reviewed. After the control solution has been developed and tested in Simulink, the Real-Time Embedded Coder can be used to generate C code for those blocks that implement tasks, which represent the functionality of the application. In the last step the HE code and the functionality are compiled together with the C implementation of the E machine , and it results the real-time control application which can be tested on the real plant. If this test fails then either the functionality or the timing of the application can be reviewed. The process is repeated until a stable application results.
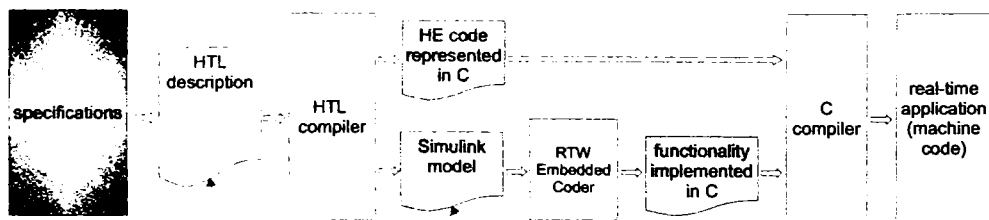

Fig. 8.13: HTL-Simulink tool chain

8.4. Case Study

In order to test and validate the HTL2simulink conversion tool, a controller for the three tanks system plant has been implemented using this tool. The structure of the HTL description that specifies the timing of the controller is presented in visual syntax in Fig. 8.14, the entire HTL description is presented in Section E.4.

The description consists of a root program that contains three modules: two of them (i.e., $T1$ and $T2$) specify the timing for tank $T1$ controller and tank $T2$ controller and the third module specifies the timing for the communication module (i.e., IO). Each of the controller modules contains one mode, which invokes one task (e.g., task that implements the control law), which is refined by a program (i.e., P_T1 or P_T2) into a P or a PI controller. The switch between the P and PI depends on the presence or absence of the perturbation, for this example it has been considered that there are sensors not only for measuring the height but also for determining if there is perturbation in a tank or not, thus no estimation is needed. The communication module, IO, contains one mode, $readWrite$, which invokes three tasks: t_read, which reads sensor values and updates communicators $h1$, $h2$, $v1$, and

hierarchical structure   **modes structure**
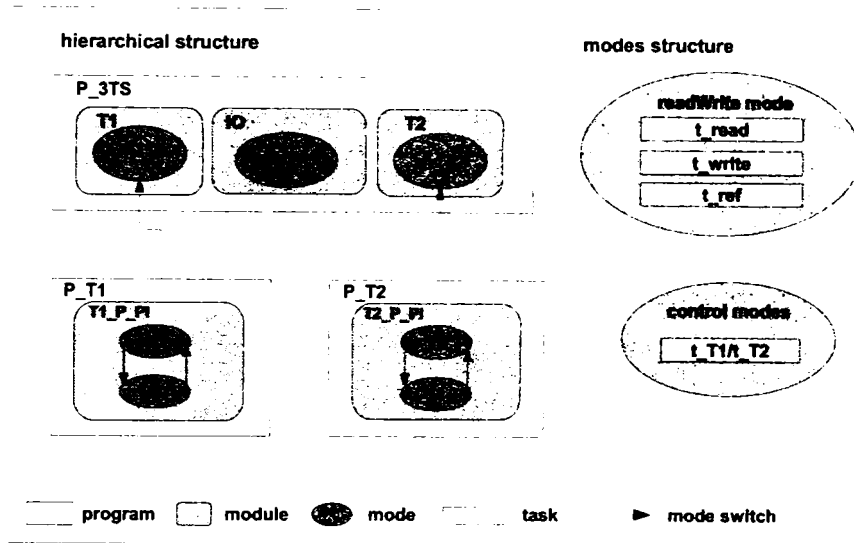

Fig. 8.14: 3TS controller: hierarchical structure

$v2$, t_write, which reads communicators $u1$ and $u2$ and sends the commands to the pumps, and t_ref, which reads the target values and updates communicators $h1\_ref$ and $h2\_ref$. The HTL description presented above is different from the one presented in Subsection 5.2 in the sense that here there is only one level of refinement and it has been considered that the presence or the absence of the perturbation can be sensed.
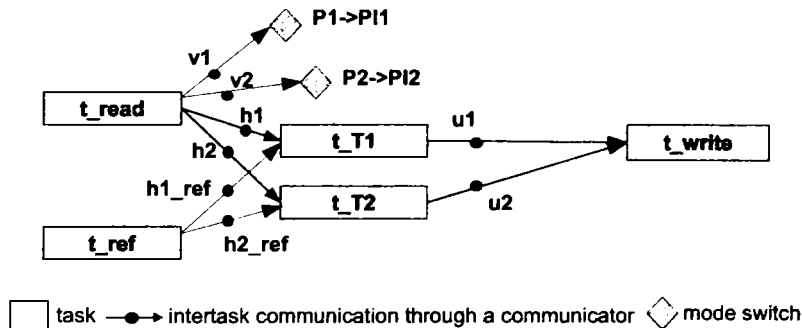
Fig. 8.15: 3TS controller: data flow

In Fig. 8.15 it is presented the data-flow between the tasks in the HTL description. Thus task t_read reads the sensor values and updates communicators $h1$, $h2$, $v1$, and $v2$. Communicators $v1$ and $v2$ are read by mode switches in the control modes. Task t_ref reads target values and updates communicators $h1\_ref$ and $h2\_ref$.The controller task for $T1$ and the controller task for $T2$, read communicators $h1$ and $h1\_ref$, and $h2$ and $h2\_ref$, respectively, and updates communicator $u1$ and $u2$, respectively. Finally task t_write reads the values of communicators $u1$ and $u2$ and sends them to the plant.

The timing of the HTL description is depicted in Fig. 8.16. The program consists of running every $500ms$ five tasks. Task t_read has an LET of $300ms$, it updates forth instance
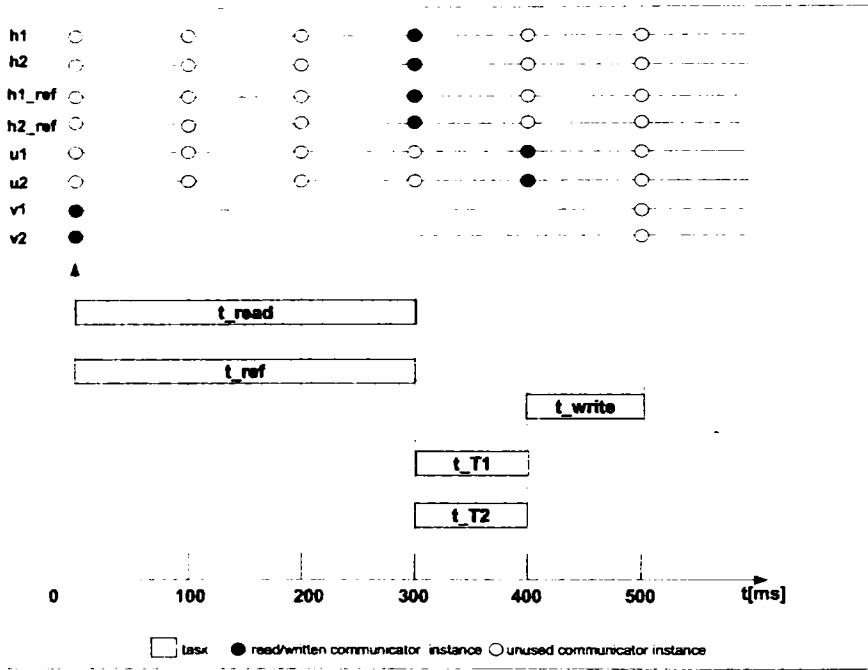
Fig. 8.16: 3TS controller: timing

of communicators: $h1$, $h2$, $v1$, and $v2$. The controller tasks for $T1$ and $T2$, read the forth instance of communicators $h1$ and $h1\_ref$, and $h2$ and $h2\_ref$, respectively, and updates the fifth instance of communicator $u1$ and $u2$, respectively; they have an LET of $100ms$. Task t_write reads the fifth instance of communicators $u1$ and $u2$; it has an LET of $100ms$.

8.4.1. Simulink Model for the 3TS Controller

The Simulink model of the 3TS controller has been generated using the Simulink2HTL tool. On the top level (Fig. 8.17) the model contains two subsystems: the plant subsystem and the controller subsystem. In the plant subsystem the model of the plant has to be implemented manually, while in the controller subsystem has been generated the Simulink model of the timing specified by the HTL description.
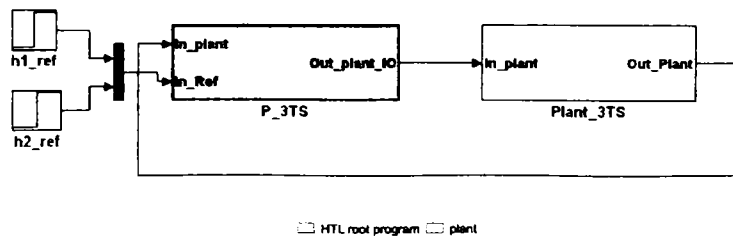

Fig. 8.17: Top level of the 3TS controller Simulink model

Fig. 8.18 presents the Simulink model which was generated for the top level program; it contains a subsystem for each module and communicator in the program. The model defines input and output ports for communicating with the plant and a digital clock, which generates the clock that is used to implement the timing.
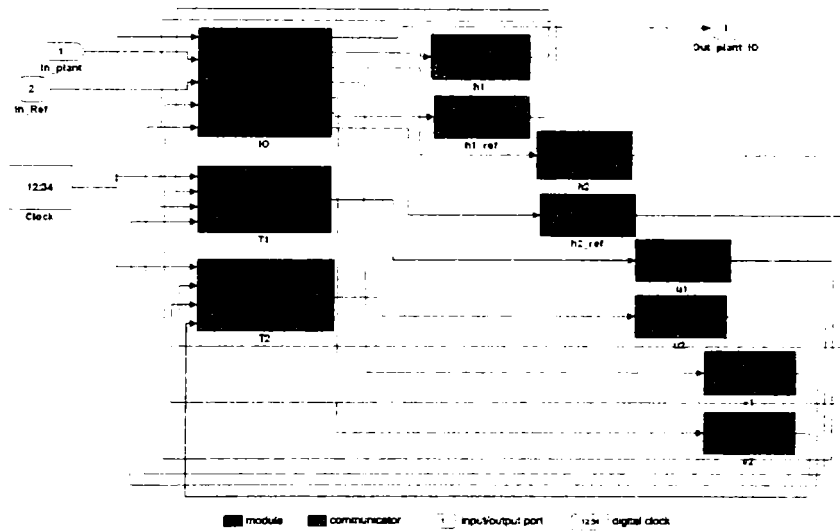

Fig. 8.18: Model Simulink for the root program of 3TS controller

After the tasks, mode switches and plant Simulink models have been implemented, the entire model has been simulated using a target value of $0.5m$ for tank $T1$ and $0.4m$ for tank $T2$. In Fig. 8.19 are plotted the levels of the water in the two tanks. Initially there was no perturbation in any of the tanks, thus the P controller was used for both tanks. At time moment $t = 180s$, perturbation is introduced in tank $T2$, thus the controller for tank $T2$ switches to PI and compensates perturbation. Similar for tank $T1$ at time moment $t = 210s$ it is introduced perturbation, thus the controller for $T1$ switches to PI also.
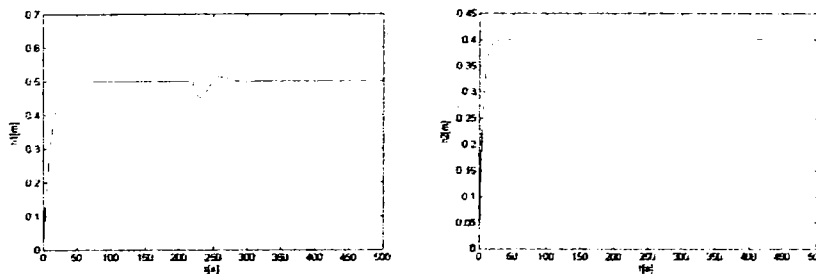

Fig. 8.19: Evolution of the level of the water in $T1$ and $T2$ ($h10 = 50$, $h20 = 40$ for the simulated controller

8.4.2. Implementation

For the controller tasks, C code has been generated using the Real-Time Embedded Workshop, which was then used as the implementation of the controller tasks and was compiled

together with the Unix implementation of the E machine. The resulted application was used to control a simulated 3TS plant, which is implemented in Java. Fig. 8.20, depicts the evolution of the level of the water in the two tanks; the target value for tank $T1$ was set to $0.5m$ and for tank $T2$ it was set to $0.4m$. Initially there was no perturbation in any of the two tanks. At time moment $t = 180s$, perturbation was introduced in tank $T2$, and at time moment $t = 200s$, perturbation was introduced in tank $T1$. As seen in Fig. 8.20, the results obtained in the case of the real application are close to the results obtained by simulating the Simulink model of the application.
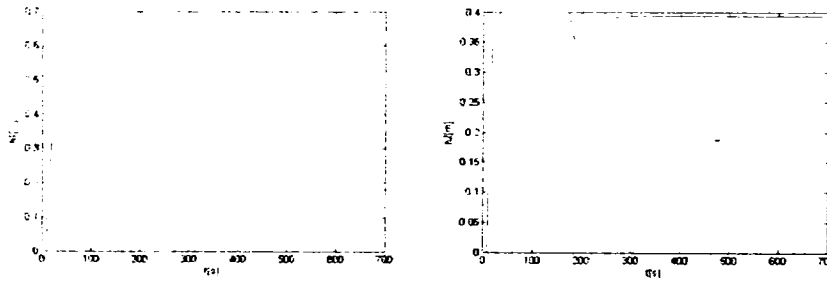
Fig. 8.20: Evolution of the level of the water in $T1$ and $T2$ ($h10 = 50$, $h20 = 40$ for the controller implemented in C

9. Conclusion

In the last two decades there has been an accelerated growth in the complexity of embedded and real-time systems. This growth lead to the evolution of the programming models for real-time application from the physical execution model to logical execution model, and to the evolution of the real-time programming languages from assembly programming languages to object oriented programming languages (i.e., Java). The focus in this thesis was on high-level programming constructs for specifying timing behavior of a real-time application.

HTL [2] is one of the most recent innovations in the field of real-time programming languages; it is a programming language that can be used to specify timing behavior of tasks in a real-time application, and interaction between them. HTL is considered to be the successor of Giotto [5]. An informal description of the HTL syntax and the most important features of HTL were presented in Chapter 2.

As its predecessor, Giotto, HTL it is not compiled directly into machine code, but into the so called *E code*, which is code interpreted by a virtual machine, namely, the E machine [6]. Thus, real-time applications developed with HTL can be executed on any platform for which an E machine implementation exists, and if there are enough resources to run the application, its timing behavior will be the same independent of the platform. In Chapter 3 it was presented the original E machine and how parallel composition of sets of periodic tasks can be expressed in E code. Nevertheless, when hierarchy had to be expressed in original E code it turned out to be impossible. Thus, the original E machine has been extended in order to allow handling of hierarchical structure at runtime. The extended E machine is named HE machine and the extended E code interpreted by the HE machine is called HE code. In Chapter 3 is also presented the HE machine and how hierarchical structure can be expressed in HE code.

Two compile algorithms have been designed to translate an HTL description into E code or HE code, respectively, both algorithms were presented in Chapter 4. The flattening HTL compiler translates an HTL description into an E code program; since E code can not express hierarchical structure, the flattening HTL compiler has to flatten the HTL description before it generates the E code program for it. The hierarchy-preserving HTL compiler translates an HTL description into an HE code program; it does not have to alter the structure of the HTL description being compiled. In the last part of Chapter 4 the two compilers have been compared both analytically and experimentally. Both analytical results and experimental results show that the flattening HTL compiler generates exponentially many E code instruction in terms of the degree of parallelism in the refinement, while in the case of hierarchy-preserving HTL compiler the number of generated HE code instructions grows linearly with the degree of parallelism in the refinement. Nevertheless, the flattening HTL compiler turns out to generate more efficient E code than the hierarchy-preserving HTL compiler, when there is no parallelism in the refinement.

All the ideas presented in the thesis have been tested on real-time control applications. Two plants have been used as case studies: the Three Tanks System (3TS), which exists at University "Politehnica" of Timisoara, it was presented in Chapter 5, and

the JAviator, which is a quad rotor helicopter developed at University of Salzburg in order to be used as a test platform for new real-time programming techniques, it was presented in Chapter 6. In Chapter 5 has been presented a real-time control application developed using HTL for controlling 3TS plant. The control application was intended to show the capabilities of HTL, i.e., refinement, parallel composition of sets of tasks, distributions, etc. All the tests presented in this chapter have been run on the real 3TS plant. The rest of the HTL control applications presented in the thesis have been tested only on simulated 3TS or JAviator plants, nevertheless this does reduce the relevance of the results since the timing of the application is the same regardless if the controlled plant is a real one or a simulated one.

In the last decade there has been a growing interest in making Java a programming language for real-time application. One of the latest innovations in the field of real-time programming constructs that use Java is Exotask [8; 9]. An Exotask program consists of a graph in which nodes represent tasks and edges represent connections between tasks. The main elements of Exotask have been presented in Chapter 6. One important advantage of Exotask is the support for pluggable timing grammar and scheduler, thus in Chapter 6 has been presented a timing grammar for Exotask that makes possible the use of HTL syntax into an Exotask graph; the new programming construct was named Exotask-HTL. A scheduler that understands the new grammar has been developed also; it was presented in the same chapter. The Exotask-HTL programming construct has been used to develop a control application for the JAviator plant. The application has been tested on an AMD64 four-way 2.4GHz machine and it was observed that the variation in the period of the application (20ms) was bellow 0.5ms (1% of the period). Although the hardware might be considered unrealistic for an embedded application, there are embedded applications that use powerful computers, i.e., next generation battleships [35].

Many of the embedded systems that exists today are limited in terms of resources and power of computation, thus in this thesis has been presented an implementation of HTL that targets a microcontroller (Chapter 7), the new software platform was called *micro HTL*. The platform consists of an optimized version of HE machine that can run on a microcontroller, namely, *micro E machine*, and an HTL compiler based on the hierarchy-preserving HTL compiler, which generates optimized HE code, i.e., no HE code is generated for empty units. Since there was no operating system on the microcontroller, a small real-time executive, which can schedule tasks based on the EDF scheduling algorithm, had to be developed. Micro HTL has been tested by implementing two control application: one for 3TS plant and the other one for JAviator plant. For the JAviator control application, it has been achieved an efficacy of 75%-80% which is closed to the efficacy of regular real-time application 90% [1] and for the 3TS control application the efficacy was around 95%.

Automatic control system represents an important category of embedded systems. It is a common practice among the control engineers to use modeling tools, e.g., Simulink, in order to design and test control algorithms. However, it is not that common to also model and test the timing of the application that will implement the control algorithm, thus problems my arise after the application has been implemented due to delays that have not been considered in the design of the control algorithm. In Chapter 8 has been presented a way of modeling an HTL description in Simulink. The HTL compiler has been extended in order to be able to convert an HTL description into a Simulink model. The advantage of converting an HTL description into a Simulink model consists in the fact that the timing of the final application can be simulated and tested in Simulink before it is implemented. Another important advantage is the possibility to generate C code for a Simulink model, which can be used as functional code for tasks in the final application. The HTL-to-Simulink tool chain has been tested by implementing a control application for the 3TS plant. It has been shown that the results obtain by simulating the Simulink model

of the control application are very close to does obtain by executing the application for which functional code has been generated directly from the Simulink model.

9.1. Personal Contributions

As a result of my scientific research I have published as a coauthor a total of seven papers. Five conference papers: three ISI papers [2; 8; 37], two papers at international conferences in Romania [24; 38]. One technical report at University of California at Berkeley [4]. One paper that has been accepted to be published in the ACM Transactions on Embedded Computing Systems journal [9]. Another paper has been submitted at Elsevier Science of Computer Programming journal, but I have received no response from the editors yet; this journal paper is based on the paper that has been presented at the APGES workshop [7].

Contributions summary:

- I have designed and implemented the compiler for Timing Specification Language, which is an intermediate language between HTL and Giotto [24];
- I have contributed to the development of HTL [2; 4; 37]:
  - design and implementation of the flattening HTL compiler
  - implementation of the separation of concerns concept for reliability into the HTL compiler and E machine
- I have contributed to the design of the HE machine and to the design of the HTL compiler that generates code for this new E machine, I have implemented both the HE machine and the HTL compiler, and I have compared both analytical and experimental the flattening HTL compiler and the hierarchy-preserving HTL compiler [7; 52];
- I have designed and implemented the HTL grammar for Exotask system [8; 9];
- I have designed and I have implemented the micro HTL;
- I have defined the mapping of an HTL description to a Simulink model and I have implemented the HTL2Simulink module in the HTL compiler;
- I have designed and implemented HTL and Exotask-HTL controllers for the 3TS and the Javiator plants.

9.2. Future Work

Current HTL syntax allows only specification of timing behavior for periodic tasks; nevertheless many real-time applications contain also aperiodic and sporadic tasks. Thus one possible research direction would be to extend HTL to support timing specification for aperiodic and sporadic tasks.

Applications developed with Exotask-HTL has been shown to be working only on powerful hardware; still it should be possible to run it on less powerful hardware, i.e., Gumstix, e.g., Exotask has already been tested with success on such a hardware. In order to improve performance of Exotask-HTL there are two changes that can be made: use of more efficient HE code (e.g., define new instructions or remove the existing ones), and improve the Exotask-HTL scheduler.

So far all the timing analysis that have been done for Giotto and HTL descriptions assumed that overhead introduce by E machine is zero; this might be true for powerful

hardware, nevertheless when it comes to running HTL real-time applications on micro-controllers this assumption does not hold anymore. In Chapter 7 it has been presented a timing analysis that considers not only tasks WCET but also overhead introduced by micro E machine; it was shown that in the worst case the overhead could go up to 25% of the application period, which is a significant overhead and has to be consider in the timing analysis. The overhead that used in the analysis was measured experimentally, thus another research direction would be to find a way of computing the overhead based on the generated HE code and on the target hardware platform.

Micro E machine presented in Chapter 7 does not support distribution of HTL descriptions. Developing a micro E machine that can support distribution represents another research direction; for communication between different E machines, on which parts of an HTL application are executed, it could be used either the RS232 interface, or the I2C interface, or maybe both. An even more ambition plan would be to allow communication between E machines that run on heterogeneous hardware (i.e., PC and microcontroller) that use different communication interface (i.e., RS232, I2C, and Ethernet). Timing analysis for such a distributed system should consider worst case transmission times for all the communication channels, worst case overhead introduced by E machines running on different hardware, and worst case execution time of each task.

The JAviator control applications that have been implemented using micro HTL and Exotask-HTL have only been tested on simulated JAviator plants, in the future I plan to use both this application in order to control the real JAviator plant.

Finally, I want to create a software development tool for HTL that can be used to developed HTL real-time application for PCs, microcontrollers and for Exotask. The new development environment will contain all the tools that have been developed for HTL so far, a graphical editor for HTL, with possibility to edit either an HTL description or an Exotask graph that uses HTL grammar and a graphical timing analyzer that can be used to visualize the timing behavior of a real-time application.

A. Three Tank System Mathematical Model

Three Tanks System (3TS) plant is made up of three identical cylindrical tanks (T_1, T_2, T_3), which have the same transversal section A. The three tanks are interconnected through pipes, which have the same section S ($S << A(m^2)$). Each thank has a tap through which the fluid drains. Tank T_2 has a supplementary tap. There are also two pumps P_1 and P_2, which are connected to T_1 and T_2, respectively. The pumps are powered by two DC-motors. In order to be able to simulate perturbations in the system, the interconnection pipes as well as the draining pipes are equipped with a tap α_i where $i \in \{S_1, S_2, g_2, e_1, e_2, e_3\}$. In Fig. A.1 it is presented the block schema for 3TS plant.
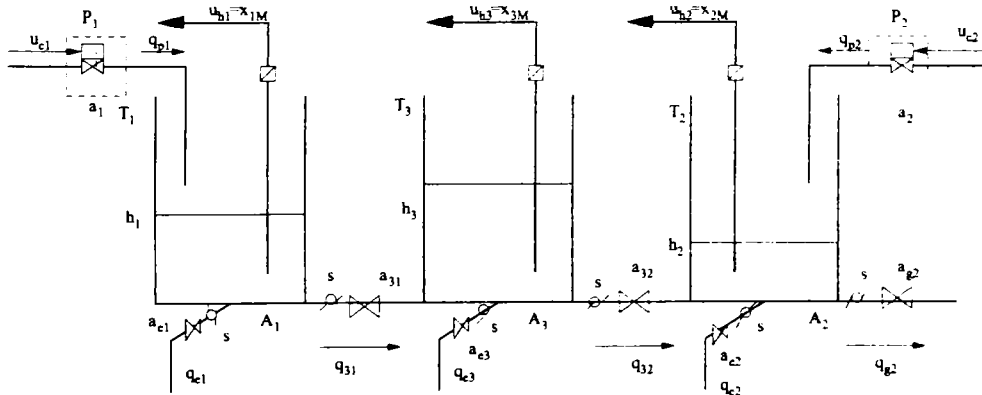
Fig. A.1: Three Tanks System

The level of the fluid in the three tanks depends on:
- the filling flow capacities of T_1 and T_2 (i.e., q_{p1} and q_{p2}, respectively);
- the draining flow capacities of the six taps:
  - q_{e1}, q_{e2}, q_{e3}, and q_{g2} - emptying flow capacities (these represent the perturbations);
  - q_{13} and q_{32} - the interconnection flow capacities;

The interconnection flow capacities are considered to be oriented:
$$q_{13} > 0, \ if \ h_1 > h_3 \ (T_1 \longrightarrow T_3);$$
$$q_{13} < 0, \ if \ h_1 < h_3 \ (T_3 \longrightarrow T_1);$$
respectively:
$$q_{32} > 0, \ if \ h_3 > h_2 \ (T_3 \longrightarrow T_2);$$
$$q_{32} < 0, \ if \ h_3 < h_2 \ (T_2 \longrightarrow T_3);$$

In order to be able to mathematically model the plant, the physical phenomena that takes place must be known. The main equation for 3TS system is Bernoulli's equation. The equation relates the speed and the pressure off moving fluid.

$$p + \frac{\delta v^2}{2} + \delta g h = const. \qquad . \qquad (A.1)$$

Considering the "homogeneous environment" and $S << A$, then the speed of the draining fluid could be approximated by the following relation:

$$v \approx \sqrt{2g\Delta h} \qquad (A.2)$$

where Δh represents the fluid level deference between interconnected tanks.

Nonlinear model for the Three Tanks System (3TS) plant is:

$$\begin{cases} \dot{h_1} = \frac{1}{A}(q_{p1} - q_{13} - q_{e1}) \\ \dot{h_2} = \frac{1}{A}(q_{p2} + q_{32} - q_{e2} - q_{g2}) \\ \dot{h_3} = \frac{1}{A}q_{13} - q_{32} - q_{e3} \end{cases} \qquad (A.3)$$

where

$$\begin{cases} q_{13} = \mu_{S1} \cdot S \cdot sgn(h_1 - h_3)\sqrt{2g|h_1 - h_3|} \\ q_{32} = \mu_{S2} \cdot S \cdot sgn(h_3 - h_2)\sqrt{2g|h_3 - h_2|} \\ q_{20} = \mu_{g2} \cdot S \cdot \sqrt{2gh_2} \\ q_{ei} = \mu_{ei} \cdot S \cdot \sqrt{2gh_i} \\ q_{pi} = c_i u_{ci} \end{cases} \qquad (A.4)$$

Linearized model of the 3TS plant in the neighborhood of (h_{10}, h_{20}, h_{30}) :

$$
\begin{cases}
\Delta \dot{h}_1 = \\
\quad -\frac{\sqrt{2g}}{A} sgn(h_{10} - h_{30}) \cdot \sqrt{|h_{10} - h_{30}|} \cdot \Delta u_{s1} \\
\quad -\frac{\sqrt{2g}}{A} u_{s10} sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_1 \\
\quad +\frac{\sqrt{2g}}{A} u_{s10} sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_3 \\
\quad -\frac{\sqrt{2g}}{A} \cdot \sqrt{h_{10}} \cdot \Delta u_{e1} - \frac{\sqrt{2g}}{A} u_{e10} \cdot \frac{1}{2\sqrt{h_{10}}} \Delta h_1 + \frac{c_1}{A} \Delta u_{c1} \\[2mm]
\Delta \dot{h}_2 = \\
\quad \frac{\sqrt{2g}}{A} sgn(h_{30} - h_{20}) \cdot \sqrt{|h_{30} - h_{20}|} \cdot \Delta u_{s2} \\
\quad +\frac{\sqrt{2g}}{A} u_{s20} sgn(h_{30} - h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_3 \\
\quad -\frac{\sqrt{2g}}{A} u_{s20} sgn(h_{30} - h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_2 \\
\quad -\frac{\sqrt{2g}}{A} \cdot \sqrt{h_{20}} \cdot \Delta u_{e2} - \frac{\sqrt{2g}}{A} u_{e20} \cdot \frac{1}{2\sqrt{h_{20}}} \cdot \Delta h_2 \\
\quad -\frac{\sqrt{2g}}{A} \sqrt{h_{20}} \Delta u_{g2} - \frac{\sqrt{2g}}{A} u_{g20} \frac{1}{2\sqrt{h_{20}}} \Delta h_2 + \frac{c_2}{A} \Delta u_{c2} \\[2mm]
\Delta \dot{h}_3 = \\
\quad \frac{\sqrt{2g}}{A} sgn(h_{10} - h_{30}) \cdot \sqrt{|h_{10} - h_{30}|} \cdot \Delta u_{s1} \\
\quad +\frac{\sqrt{2g}}{A} u_{s10} \cdot sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_1 \\
\quad -\frac{\sqrt{2g}}{A} u_{s10} \cdot sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_3 \\
\quad -\frac{\sqrt{2g}}{A} sgn(h_{30} - h_{20}) \cdot \sqrt{|h_{30} - h_{20}|} \cdot \Delta u_{s2} \\
\quad -\frac{\sqrt{2g}}{A} u_{s20} \cdot sgn(h_{30} - h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_3 \\
\quad +\frac{\sqrt{2g}}{A} u_{s20} \cdot sgn(h_{30} - h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_2 \\
\quad -\frac{\sqrt{2g}}{A} \cdot \sqrt{h_{30}} \cdot \Delta u_{e3} - \frac{\sqrt{2g}}{A} u_{e30} \cdot \frac{1}{2\sqrt{h_{30}}} \Delta h_3
\end{cases}
\tag{A.5}
$$

In Fig. A.2 it is presented the Simulink model of the 3TS plant. The Simulink model was used to test control algorithms before they were used for the real plant. In order to be able to test the final HTL program a simulator of the 3TS plant has been implemented in Java (Fig. A.3). The program works like a TCP server to which TCP clients that implement different control strategy can connect in order to control the plant. Although the simulator accepts multiple connections, only one client can control the plant.
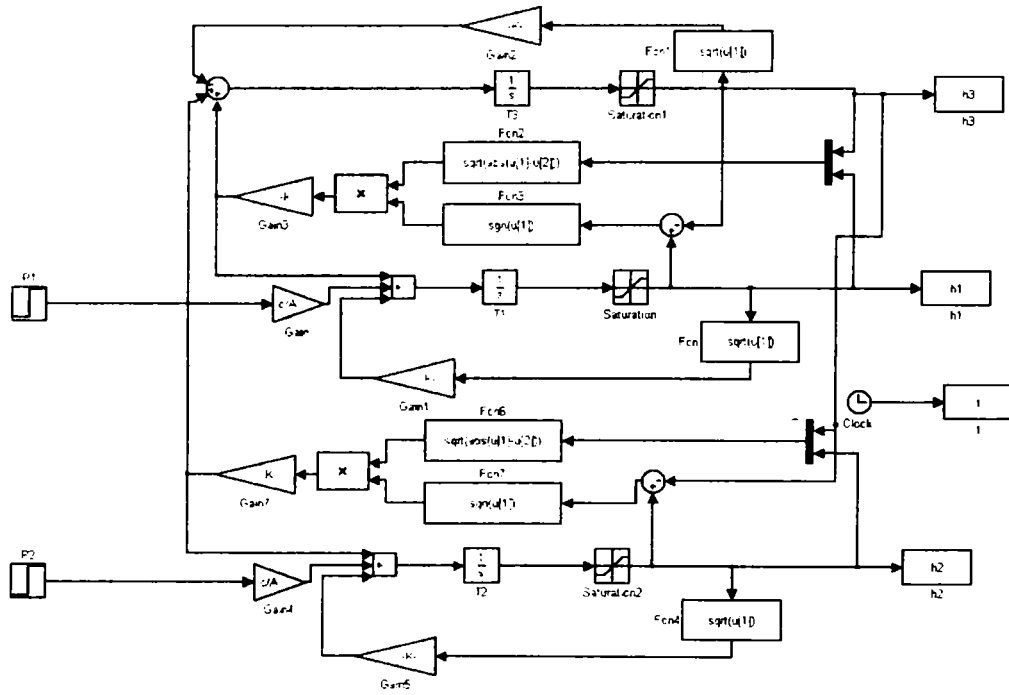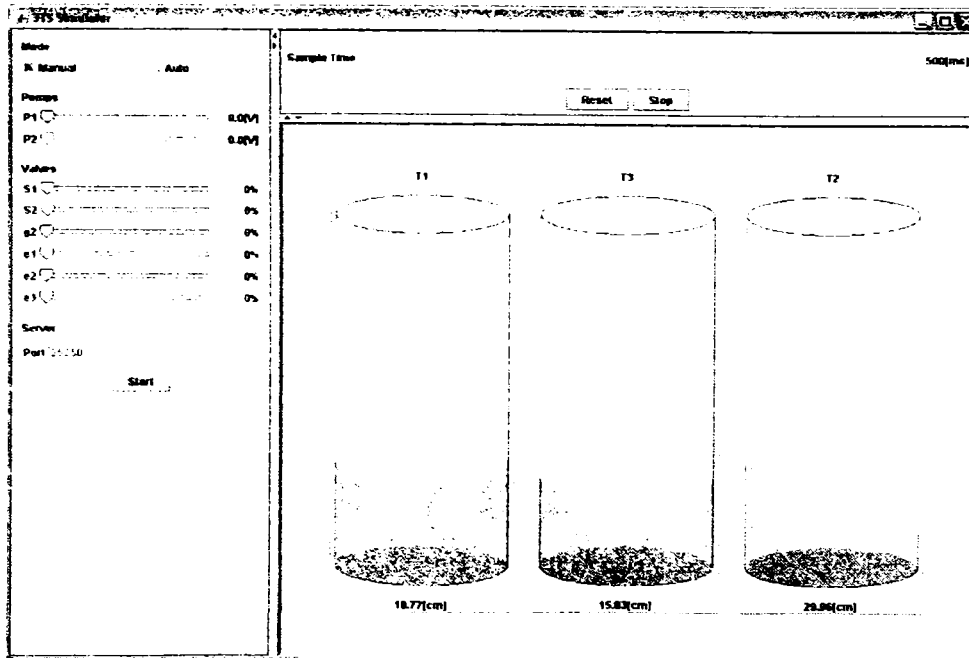
Fig. A.2: Simulink model of the 3TS plant

Fig. A.3: Java simulator of the 3TS plant

B. JAviator Mathematical Model

In the literature there are several simplified mathematical models for the quadrotor heli-copter ([53], [54], and [55], are just a few papers in which such a model is presented). The mathematical model presented here is the mathematical model described in [55].
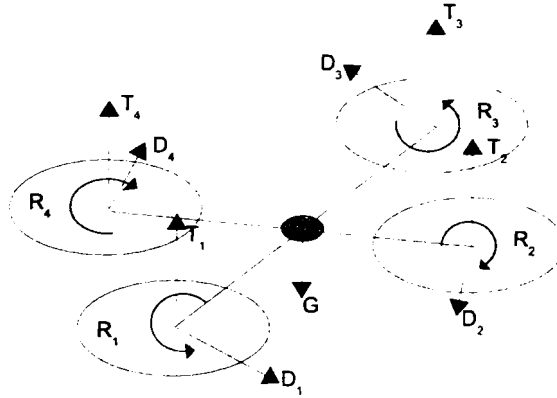

Fig. B.1: Quadrotor block diagram.

In Fig. B.1 it is presented the block diagram of a quadrotor helicopter. As shown in the figure a quadrotor helicopter consists of four rotors: two of them are spinning clockwise (R_2 and R_4), while the other two are spinning counterclockwise (R_1 and R_3). The effect of each spinning rotor is represented by a force perpendicular on the rotation plane (T_i, $i = 1,2,3,4$) and one force in the rotation plane (D_i, $i = 1,2,3,4$). The resultant of T_i forces will be noted with T, while the resultant of D_i will be noted with D.

The simplified input-output mathematical model is presented in (**B.1**).

$$\begin{cases} \ddot{\Phi} = \frac{l}{I}(T_2 - T_4) \\ \ddot{\Theta} = \frac{l}{I}(T_1 - T_3) \\ \ddot{\Psi} = \frac{K_r}{2I}(T_1 - T_2 + T_3 - T_4) \\ \ddot{z} = \frac{1}{m}(T_1 + T_2 + T_3 + T_4 - mg) \end{cases} \tag{B.1}$$

From the input-output mathematical model the following input-state-output mathematical model was driven:

$$
\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \\ \dot{x}_7 \\ \dot{x}_8 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{m} & \frac{1}{m} & \frac{1}{m} & \frac{1}{m} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{l}{I} & 0 & -\frac{l}{I} \\ 0 & 0 & 0 & 0 \\ \frac{l}{I} & 0 & -\frac{l}{I} & 0 \\ 0 & 0 & 0 & 0 \\ \frac{K_r}{2I} & -\frac{K_r}{2I} & \frac{K_r}{2I} & -\frac{K_r}{2I} \end{pmatrix} \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{pmatrix} + \begin{pmatrix} 0 \\ -mg \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}
$$

$$\text{(B.2)}$$

where

$$
\begin{aligned}
x_1 &= z \\
x_2 &= \dot{z} \\
x_3 &= \Phi \\
x_4 &= \dot{\Phi} \\
x_5 &= \Theta \\
x_6 &= \dot{\Theta} \\
x_7 &= \Psi \\
x_8 &= \dot{\Psi}
\end{aligned}
\qquad \text{(B.3)}
$$

$m = 2kg$ - the weight of the flying object, $l = 0.34m$ - the length of the arm measured from the center of the flying object, $I = 0.017kg * m^2$ - the moment of inertia of the flying object ($I_{xx} = I_{yy} = I$, $I_{zz} = 2I$), $g = 9.8m/s^2$ - gravitational acceleration, and $K_r = 4$ - the proportionality factor between T_i and D_i ($D_i = K_r T_i$).

The mathematical model presented above was used only for the design of the controllers, its may advantage being the fact that it can be split into four independent processes, i.e., altitude process, roll process, pitch process, and yaw process, thus the controllers for each of the four processes can be designed separately. Nevertheless for the real plant the four processes are not decoupled, thus the control solution should be tested on a more detailed mathematical model which also considers the interdependence between the four processes. The mathematical model of the JAviator plant that has been used for testing the control solution is presented in equations (**B.4**).

$$
\begin{cases}
\ddot{\Phi} = \frac{l}{I} f (T_2 - T_4) \\
\ddot{\Theta} = \frac{l}{I} f (T_1 - T_3) \\
\ddot{\Psi} = \frac{K_r}{2I} f (T_1 - T_2 + T_3 - T_4) \\
\ddot{z} = \frac{1}{m} (f(T_1 + T_2 + T_3 + T_4) - mg) \\
\ddot{x} = (sin(\Phi)sin(\Psi) + cos(\Phi)sin(\Theta)cos(\Psi))(T_1 + T_2 + T_3 + T_4)\frac{1}{m} \\
\ddot{y} = (sin(\Phi)cos(\Psi) + cos(\Phi)sin(\Theta)sin(\Psi))(T_1 + T_2 + T_3 + T_4)\frac{1}{m} \\
f = cos(\Phi)cos(\Theta)
\end{cases}
\qquad \text{(B.4)}
$$

The detailed mathematical model of the JAviator also describes how the x and y position evolve based on the four forces, e.g., T_1, T_2, T_3, and T_4, and on the roll, pitch, and yaw angles. This detailed model has been implemented both in Simulink and in a Java program. The Simulink model is presented in Fig. B.2. In Fig. B.3 it is presented the Simulink model for the roll process. In Fig. B.4 it is presented the Simulink model for the pitch process. In Fig. B.5 it is presented the Simulink model for the yaw process. In Fig. B.6 it is presented the Simulink model for the z process. In Fig. B.7 it is presented the Simulink model for the x process. In Fig. B.8 it is presented the Simulink model for the y process.
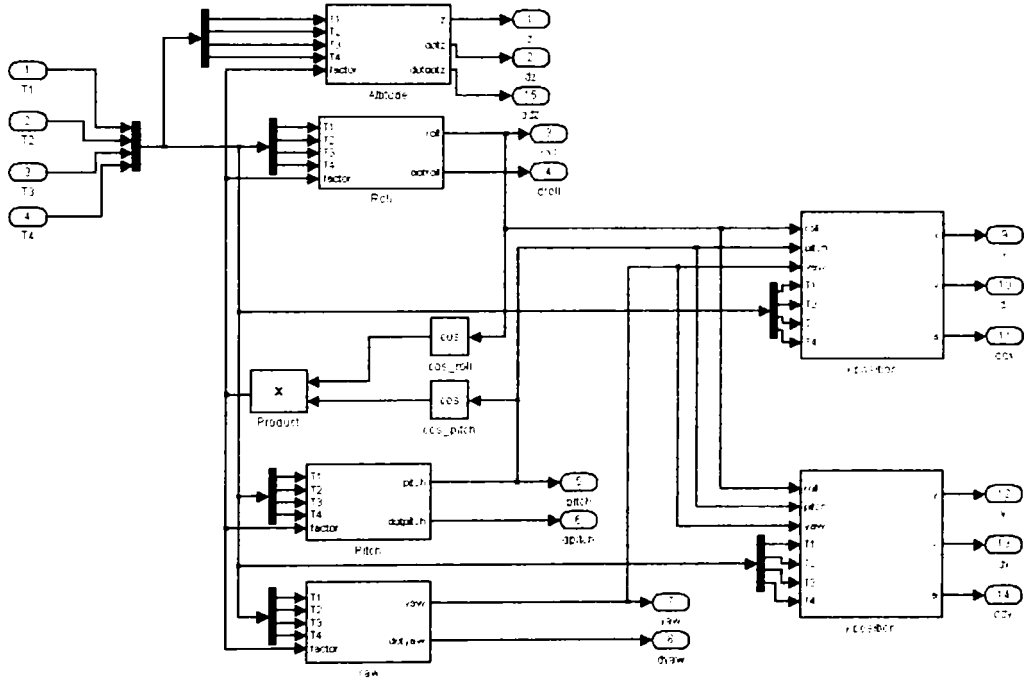
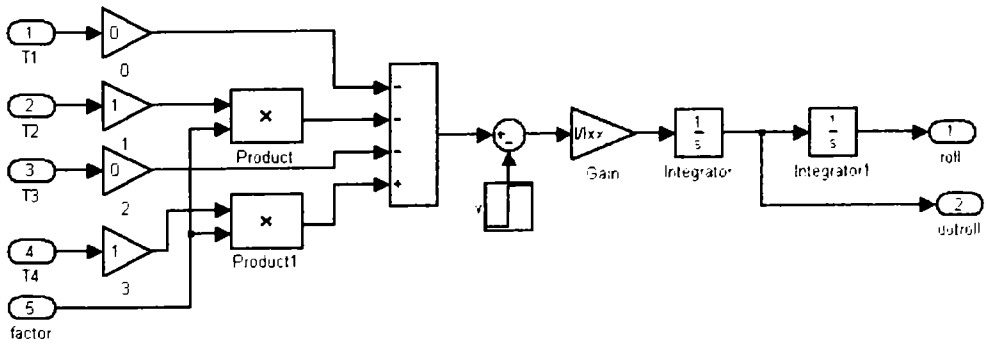Fig. B.2: JAviator: detailed Simulink model.
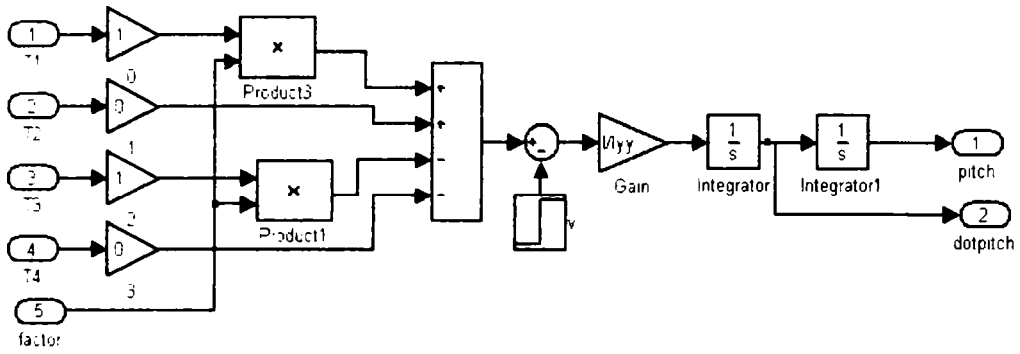

Fig. B.3: JAviator: roll Simulink model.

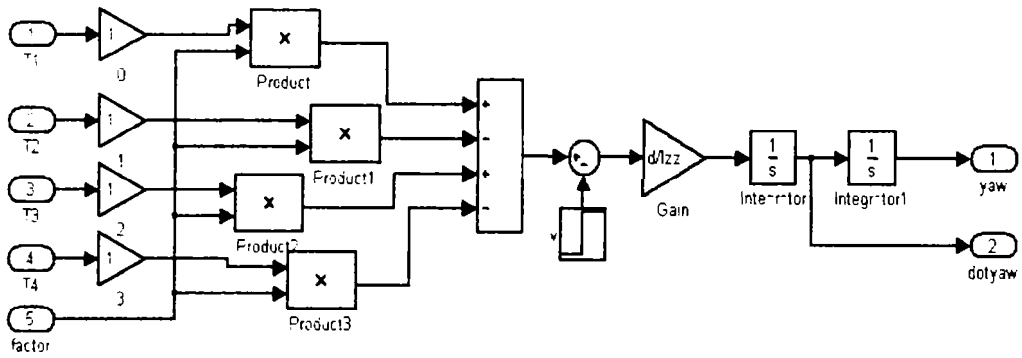Fig. B.4: JAviator: pitch Simulink model.

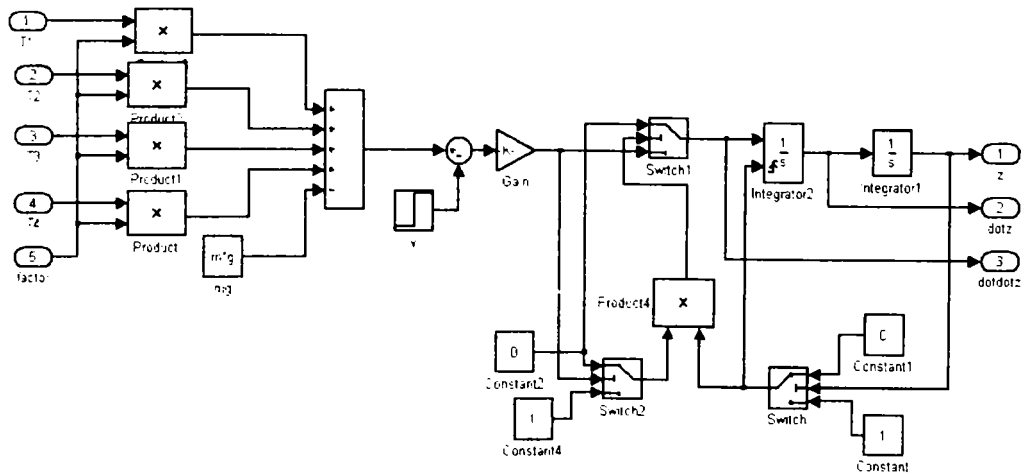Fig. B.5: JAviator: yaw Simulink model.
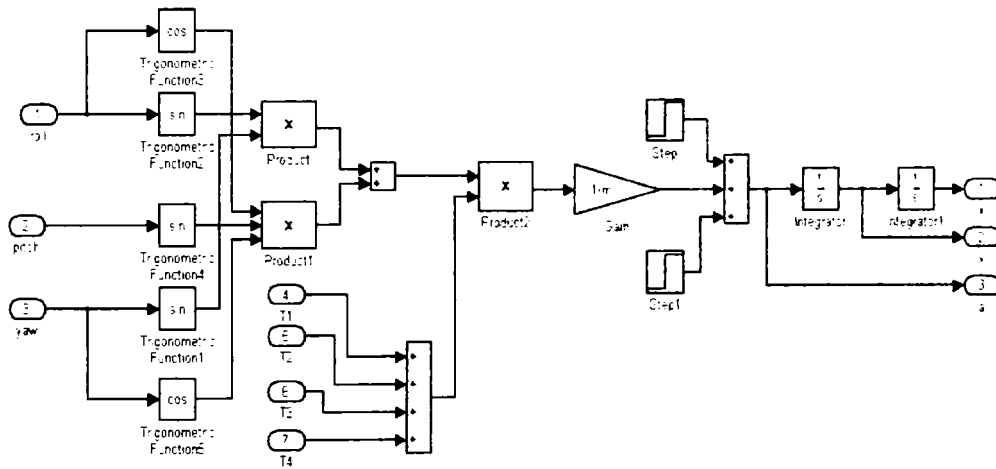

Fig. B.6: JAviator: z Simulink model.

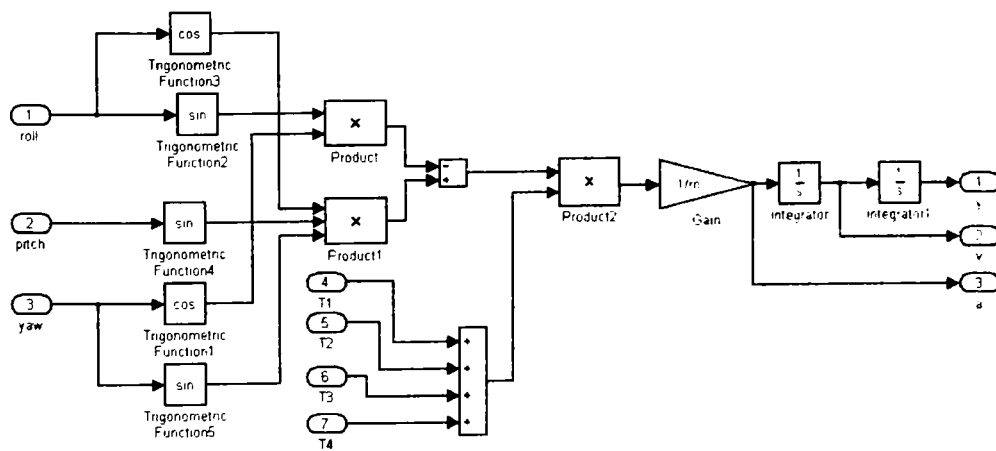Fig. B.7: JAviator: x Simulink model.

Fig. B.8: JAviator: y Simulink model.
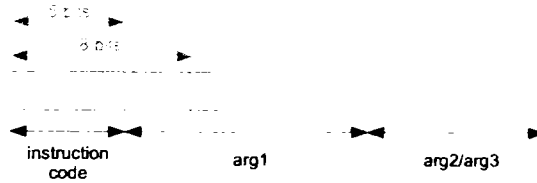
C. Encoding of HE code Instructions for Micro HTL

Fig. C.1: Instruction encoding

Encoding of each HE code instructions for the micro E machine is as follows:
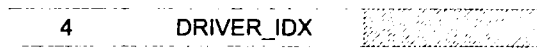
- *call*(*d*) Fig. C.2

Fig. C.2: *Call* instruction encoding

arg1 = DRIVER_IDX: the index of the driver to be executed;

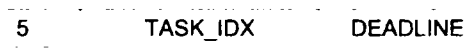- *release*(*t*, *dl*) Fig. C.3

Fig. C.3: *Release* instruction encoding

arg1 = TASK_IDX: the index of the task that has to be released;
arg2 = DEADLINE: relative time when the task should have complete execution;
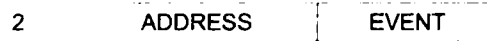
- *writeFuture*(*e*, *a*) Fig. C.4

Fig. C.4: *WriteFuture* instruction encoding

arg1 = ADDRESS: address of the block of HE code that has to be executed when the trigger created by the instruction gets enabled;
arg2 = EVENT: time event on which created trigger gets enabled;

| 2 | ADDRESS | EVENT |

Fig. C.5: *SwitchFuture* instruction encoding

- *switchFuture(e.a)* Fig. C.5
  arg1 = ADDRESS: address of the block of HE code that has to be executed when the trigger created by the instruction gets enabled;
  arg2 = EVENT: time event on which created trigger gets enabled;
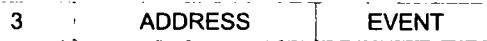- *readFuture(e.a)* Fig. C.6

| 3 | ADDRESS | EVENT |

Fig. C.6: *ReadFuture* instruction encoding

arg1 = ADDRESS: address of the block of HE code that has to be executed when the trigger created by the instruction gets enabled;
arg2 = EVENT: time event on which created trigger gets enabled;

- *jumpIf(cnd.a)* Fig. C.7

| 6 | ADDRESS | COND_IDX |

Fig. C.7: *JumpIf* instruction encoding

arg1 = ADDRESS: address of the block of HE code that has to be executed when condition is true;
arg2 = COND_IDX: index of the condition to be evaluated;

- *jumpAbsolute(a')* Fig. C.8
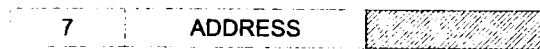
| 7 | ADDRESS | |

Fig. C.8: *JumpAbsolute* instruction encoding

arg1 = ADDRESS: address of the block of HE code where the execution has to jump;

- *jumpSubroutine(a')* Fig. C.9

| 9 | ADDRESS | |

Fig. C.9: *JumpSubroutine* instruction encoding

arg1 = ADDRESS: address of the block of HE code that has to be invoked;

- *copyRegister(Rx,Ry)* Fig. C.10
  arg1 = Rx: integer value between 0 and 3 that identifies the source register;
  arg2 = Ry: integer value between 0 and 3 that identifies the destination register;
- *pushRegister(Rx)* Fig. C.11
  arg1 = Rx: integer value between 0 and 3 that identifies the register that has to be pushed onto the parent stack;

10	Rx	Ry

Fig. C.10: *CopyRegister* instruction encoding

11	Rx	

Fig. C.11: *PushRegister* instruction encoding

12	Rx	

Fig. C.12: *PopRegister* instruction encoding

- *popRegister(Rx)* Fig. C.12
  arg1 = Rx: integer value between 0 and 3 that identifies the register that will be loaded with the value popped from the parent stack;
- *getParent(Rx, Ry)* Fig. C.13

13	Rx	Ry

Fig. C.13: *GetParent* instruction encoding

arg1 = Rx: integer value between 0 and 3 that identifies the register whose parent will be copied in register identified by parameter Ry;
arg2 = Ry: integer value between 0 and 3 that identifies the register in which the parent of the register identified by parameter Rx, will be copied;

- *setParent(Rx, Ry)* Fig. C.14

14	Rx	Ry

Fig. C.14: *SetParent* instruction encoding

arg1 = Rx: integer value between 0 and 3 that identifies the register whose parent will be copied from register identified by parameter Ry;
arg2= Ry: integer value between 0 and 3 that identifies the register from which the parent of the register identified by parameter Rx, will be copied;

- *copyChildren(Rx, Ry)* Fig. C.15

19	Rx	Ry

Fig. C.15: *CopyChildren* instruction encoding

arg1 = Rx: integer value between 0 and 3 that identifies the register to which children list will be copied;
arg2 = Ry: integer value between 0 and 3 that identifies the register from which the children list will be copied;

15	Rx	Ry

Fig. C.16: *UpdateChildren* instruction encoding

- *updateChildren(Rx, Ry)* Fig. C.16
  arg1 = Rx: integer value between 0 and 3 that identifies the register whose children list will be updated;
  arg2 = Ry: integer value between 0 and 3 that identifies the register that contains the parent;
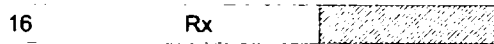- *deleteChildren(Rx)* Fig. C.17

16	Rx	

Fig. C.17: *DeleteChildren* instruction encoding

arg1 = Rx: integer value between 0 and 3 that identifies the register whose children tree has to be deleted;

- *replaceChild(Rx, Ry, Rz)* Fig. C.18

8	Rx	Ry:Rz

Fig. C.18: *replaceChild* instruction encoding

arg1 = Rx: integer value between 0 and 3 that identifies the register whose children will be updated;
arg2 = Ry: integer value between 0 and 3 that identifies the register whose value will be replaced;
arg3 = Rz: integer value between 0 and 3 that identifies the register whose value will replace the value identified by Ry;

- *cleanChildren(Rx)* Fig. C.19

18	Rx	

Fig. C.19: *CleanChildren* instruction encoding

arg1 = Rx: integer value between 0 and 3 that identifies the register whose children list has to be cleaned;

- *return()* Fig. C.20

8		

Fig. C.20: *Return* instruction encoding

D. HTL Grammar

```
Package htlc;

Helpers

all = [0 .. 0xFFFF];
lowercase = ['a' .. 'z'];
uppercase = ['A' .. 'Z'];
digit = ['0' .. '9'];
hex_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];

tab = 9;
cr = 13;
lf = 10;
eol = cr lf | cr | lf; // This takes care of different platforms

not_cr_lf = [all -[cr + lf]];
not_star = [all -'*'];
not_star_slash = [not_star -'/'];

blank = (' ' | tab | eol)+;
short_comment = '//' not_cr_lf* eol;
long_comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
comment = short_comment | long_comment;

letter = lowercase | uppercase | '_';
name = letter (letter | digit)*;
ident = name ('.' name)*;
number = digit+;

Tokens

program = 'program';
communicator = 'communicator';
sensor = 'sensor';
actuator = 'actuator';
general = 'general';
period = 'period';
uses='uses';
module = 'module';
start = 'start';
import = 'import';
export = 'export';
task = 'task';
output = 'output';
input = 'input';
state = 'state';
```

```
parent = 'parent';
function = 'function';
update = 'update';
port = 'port';
mode = 'mode';
invoke = 'invoke';
switch = 'switch';
wcet = 'wcet';
init = 'init';
host = 'host';
lrc = 'LRC';
srg = 'SRG';
model = 'model';

ident = ident;
number = number;

semicolon = ';';
comma = ',';
dot = '.';
zero = '0';
colon = ':';

greater_than = '>';
less_or_equal = '<=';
assign = ':=';

l_par = '(';
r_par = ')';
l_brace = '{';
r_brace = '}';
l_bracket = '[';
r_bracket = ']';

blank = blank;
comment = comment;

Ignored Tokens

blank, comment;

Productions

program_declaration_list = program_declaration*;

program_declaration = program [program_name]:ident l_brace
  communicator_declaration_list?
  module_declaration_list
      r_brace;

communicator_declaration_list = communicator communicator_declaration* ;
communicator_declaration = [type_name]:ident [communicator_name]:ident
                           period [communicator_period]:number
                           init [init_driver]:ident
                           lrc_specification?
                           semicolon;
lrc_specification = lrc lrc_value;
lrc_value = float;
float = [int]:number dot [frac]:number;
```

```
module_declaration_list = module_declaration*;
module_declaration = module [module_name]:ident host_declaration_list?
  start [start_mode]:ident l_brace
  port_declaration_list?
  task_declaration_list
  mode_declaration_list
      r_brace;

host_declaration_list = l_bracket hosts_list? r_bracket;
hosts_list = {concrete} host_declaration host_declaration_tail* |
  (host_declaration+) ;
host_declaration_tail = comma host_declaration ;
host_declaration = [host_name]:ident [host_ip]:ip_declaration colon
  [host_port]:number srg_specification?;
srg_specification = srg srg_value;
srg_value = float;
ip_declaration = [a]:number [d1]:dot [b]:number [d2]:dot [c]:number
  [d3]:dot [d]:number;

port_declaration_list = port port_declaration*;
port_declaration = [port_type]:ident [port_name]:ident assign
  [init_driver]:ident semicolon;

task_declaration_list = task_declaration*;
task_declaration = task [task_name]:ident
    input [input_formal_ports]:formal_ports
    state [state_formal_ports]:state_ports
    output [output_formal_ports]:formal_ports
    task_function?
    task_wcet?
    reliability_model?
    semicolon;

task_function = function [function_name]:ident;
task_wcet = wcet [wcet_map]:number;
reliability_model = model [model_type]:number;

formal_ports = l_par formal_port_list? r_par ;
formal_port_list = {concrete} formal_port formal_port_tail* | (formal_port+) ;
formal_port_tail = comma formal_port ;
formal_port = [type_name]:ident [port_name]:ident default_value?;
default_value = assign [default_driver]:ident;

state_ports = l_par state_port_list? r_par ;
state_port_list = {concrete} state_port state_port_tail* | (state_port+) ;
state_port_tail = comma state_port ;
state_port = [type_name]:ident [state_name]:ident assign [init_driver]:ident ;

mode_declaration_list = mode_declaration*;
mode_declaration = mode [mode_name]:ident period [mode_period]:number
  refine_program?
    l_brace
    sensor_device_driver_list
    actuator_device_driver_list
    task_invocation_list
    mode_switch_list
    r_brace;

refine_program = program [program_name]:ident;
```

```
sensor_device_driver_list = sensor_device_driver*;
sensor_device_driver = sensor update [driver_name]:ident l_par
  [communicator_name]:ident comma [communicator_instance]:number r_par semicolon;

actuator_device_driver_list = actuator_device_driver*;
actuator_device_driver = actuator update [driver_name]:ident l_par [communicator_name]:ident
  comma [communicator_instance]:number r_par semicolon;

task_invocation_list = task_invocation*;
task_invocation = invoke [task_name]:ident input [input_actual_ports]:actual_ports output
  [output_actual_ports]:actual_ports parent_task? semicolon;
parent_task = parent [task_name]:ident;

actual_ports = l_par actual_port_list? r_par ;
actual_port_list = {concrete} actual_port actual_port_tail* | (actual_port+) ;
actual_port_tail = comma actual_port ;
actual_port = {concrete} [port_name]:ident | communicator_instance ;
communicator_instance = l_par [communicator_port_name]:ident comma
  [communicator_instance_number]:number r_par ;

mode_switch_list = mode_switch*;
mode_switch = switch
  l_par [condition_function]:ident switch_ports r_par
  [destination_mode]:ident
  semicolon;

switch_ports = l_par switch_port_list? r_par ;
switch_port_list = {concrete} switch_port switch_port_tail* | (switch_port+) ;
switch_port_tail = comma switch_port ;
switch_port = [port_name]:ident;
```

E. HTL Descriptions

E.1. Increment/Decrement Counter

```
program P_inc_dec{
  communicator
    c_int counter period 100 init c_zero;
    c_int ref period 100 init c_zero;

  module M_read_write start m_read_write{

    task t_read input() state() output(c_int p_counter) function f_read;
    task t_ref input() state() output(c_int p_ref) function f_ref;
    task t_write input(c_int p_counter) state() output() function f_write;

    mode m_read_write period 1000{
      invoke t_read input() output((counter,1));
      invoke t_ref input() output((ref,1));
      invoke t_write input((counter,2)) output();
    }
  }

  module M_inc_dec start m_inc{

    task t_inc input(c_int p_counter_in) state() output(c_int p_counter_out);
    task t_dec input(c_int p_counter_in) state() output(c_int p_counter_out) function f_dec;

    mode m_inc period 1000 program P_INC{
      invoke t_inc input((counter,1)) output((counter,2));
      switch(inc_to_dec(counter)) m_dec;
    }

    mode m_dec period 1000{
      invoke t_dec input((counter,1)) output((counter,2));
      switch(inc_to_dec(counter)) m_inc;
    }
  }
}

program P_inc{

  module M_inc start m_inc1{

    task t_inc1 input(c_int p_counter_in) state() output(c_int p_counter_out) function f_inc1;
    task t_inc5 input(c_int p_counter_in) state() output(c_int p_counter_out) function f_inc5;
    task t_inc10 input(c_int p_counter_in) state() output(c_int p_counter_out) function f_inc10;

    mode m_inc1 period 1000{
      invoke t_inc1 input((counter,1)) output((counter,2)) parent t_inc;
      switch(inc1_to_inc5(counter)) m_inc5;
    }

    mode m_inc5 period 1000{
      invoke t_inc5 input((counter,1)) output((counter,2)) parent t_inc;
      switch(inc5_to_inc10(counter)) m_inc10;
    }
```

```
    mode m_inc10 period 1000{
      invoke t_inc10 input((counter,1)) output((counter,2)) parent t_inc;
    }
  }
}
```

E.2. Three Tanks System Controller Distributed HTL Implementation

```
program P_3TS{
  communicator
    c_double h1 period 100 init c_zero;
    c_double h2 period 100 init c_zero;
    c_double u1 period 100 init c_zero;
    c_double u2 period 100 init c_zero;
    c_bool v1 period 500 init c_false;
    c_bool v2 period 500 init c_false;
    c_double h1f period 500 init c_zero;
    c_double h2f period 500 init c_zero;
    c_double h1c period 500 init c_zero;
    c_double h2c period 500 init c_zero;
    c_bool PI_LR1 period 500 init c_false;
    c_bool PI_LR2 period 500 init c_false;

  module IO start readWrite{
    port
      c_double local_h1 := c_zero;
      c_double local_h2 := c_zero;

      task t_read input() state() output(c_double p_h1, c_double p_h2) function f_read;
      task t_write input(c_double p_u1, c_double p_u2, c_double p_h1f, c_double p_h2f,
        c_double p_h1c, c_double p_h2c) state() output() function f_write;
      task t_estimateV1 input(c_double p_h1, c_double p_u1) state() output(c_bool p_v1,
        c_double p_h1c, c_bool p_PI_LR1) function f_estimateH1;
      task t_estimateV2 input(c_double p_h2, c_double p_u2) state() output(c_bool p_v2,
        c_double p_h2c, c_bool p_PI_LR2) function f_estimateH2;
      task t_filterH1 input(c_double p_h1) state() output(c_double p_h1F, c_double p_h1f)
        function f_filterH1;
      task t_filterH2 input(c_double p_h2) state() output(c_double p_h2F, c_double p_h2f)
        function f_filterH2;

      mode readWrite period 500{
        invoke t_read input() output((h1,3), (h2,3));
        invoke t_write input((u1,4), (u2,4), (h1f, 0), (h2f, 0), (h1c, 0), (h2c, 0)) output();
        invoke t_filterH1 input((h1,3)) output(local_h1, (h1f, 1));
        invoke t_filterH2 input((h2,3)) output(local_h2, (h2f, 1));
        invoke t_estimateV1 input(local_h1, (u1,4)) output((v1,1), (h1c, 1), (PI_LR1, 1));
        invoke t_estimateV2 input(local_h2, (u2,4)) output((v2,1), (h2c, 1), (PI_LR2, 1));
      }
  }

  module T1 start m_T1{
    task t_T1 input(c_double p_h1) state() output(c_double p_u1);
    mode m_T1 period 500 program P_T1{
      invoke t_T1 input((h1,3)) output((u1,4));
    }
  }

  module T2 start m_T2{
    task t_T2 input(c_double v_h2) state() output(c_double v_u2);
    mode m_T2 period 500 program P_T2{
      invoke t_T2 input((h2,3)) output((u2,4));
    }
  }
}

program P_T1{
```

```
  module T1_P_PI start m_T1_P{
    task t_T1_P input(c_double v_h1) state() output(c_double v_u1) function f_T1_P;
    task t_T1_PI input(c_double v_h1) state() output(c_double v_u1);
    mode m_T1_P period 500{
      invoke t_T1_P input((h1,3)) output((u1,4)) parent t_T1;
      switch(withPerturbation(v1)) m_T1_PI;
    }

    mode m_T1_PI period 500 program P_T1_2PI{
      invoke t_T1_PI input((h1,3)) output((u1,4)) parent t_T1;
      switch(withoutPerturbation(v1)) m_T1_P;
    }
  }
}

program P_T2{

  module T2_P_PI start m_T2_P{

    task t_T2_P input(c_double v_h2) state() output(c_double v_u2) function f_T2_P;
    task t_T2_PI input(c_double v_h2) state() output(c_double v_u2);

    mode m_T2_P period 500{
      invoke t_T2_P input((h2,3)) output((u2,4)) parent t_T2;
      switch(withPerturbation(v2)) m_T2_PI;
    }

    mode m_T2_PI period 500 program P_T2_2PI{
      invoke t_T2_PI input((h2,3)) output((u2,4)) parent t_T2;
      switch(withoutPerturbation(v2)) m_T2_P;
    }
  }
}

program P_T1_2PI
{
  module T1_2PI start m_T1_PI_R{
    task t_T1_PI_L input(c_double v_h1) state() output(c_double v_u1) function f_T1_PI_L;
    task t_T1_PI_R input(c_double v_h1) state() output(c_double v_u1) function f_T1_PI_R;

    mode m_T1_PI_L period 500{
      invoke t_T1_PI_L input((h1, 3)) output((u1, 4)) parent t_T1_PI;
      switch(PIRapid(PI_LR1)) m_T1_PI_R;
    }

    mode m_T1_PI_R period 500{
      invoke t_T1_PI_R input((h1, 3)) output((u1, 4)) parent t_T1_PI;
      switch(PILent(PI_LR1)) m_T1_PI_L;
    }
  }
}

program P_T2_2PI
{
  module T2_2PI start m_T2_PI_R{
    task t_T2_PI_L input(c_double v_h2) state() output(c_double v_u2) function f_T2_PI_L;
    task t_T2_PI_R input(c_double v_h2) state() output(c_double v_u2) function f_T2_PI_R;

    mode m_T2_PI_L period 500{
      invoke t_T2_PI_L input((h2, 3)) output((u2, 4)) parent t_T2_PI;
      switch(PIRapid(PI_LR2)) m_T2_PI_R;
    }

    mode m_T2_PI_R period 500{
      invoke t_T2_PI_R input((h2, 3)) output((u2, 4)) parent t_T2_PI;
      switch(PILent(PI_LR2)) m_T2_PI_L;
    }
  }
}
```

E.3. Three Tanks System Controller Micro HTL Implementation

```
program P_3TS{
  communicator
    c_int h1 period 50 init c_zero;
    c_int h2 period 50 init c_zero;
    c_int u1 period 50 init c_zero;
    c_int u2 period 50 init c_zero;
    c_bool v1 period 50 init c_false;
    c_bool v2 period 50 init c_false;
    c_bool PI_SF1 period 50 init c_false;
    c_bool PI_SF2 period 50 init c_false;
    c_controller_type prevT1Controller period 50 init P_controller;
    c_controller_type prevT2Controller period 50 init P_controller;

    module IO start readWrite{

      task t_read input() state() output(c_int p_h1, c_int p_h2) function f_read;
      task t_write input(c_int p_u1, c_int p_u2) state() output() function f_write;
      task t_estimateV1 input(c_int p_h1, c_int p_u1) state(c_history history:=zero_history)
        output(c_bool p_v1, c_bool p_PI_SF1) function f_estimateH1;
      task t_estimateV2 input(c_int p_h2, c_int p_u2) state(c_history history:=zero_history)
        output(c_bool p_v2, c_bool p_PI_SF2) function f_estimateH2;

      mode readWrite period 250{
        nvoke t_read input() output((h1,1), (h2,1));
      · invoke t_write input((u1,3), (u2,3)) output();
        invoke t_estimateV1 input((h1,1), (u1,3)) output((v1,5), (PI_SF1, 5));
        invoke t_estimateV2 input((h2,1), (u2,3)) output((v2,5), (PI_SF2, 5));
      }
    }

    module T1 start m_T1{
      task t_T1 input(c_int p_h1) state() output(c_int p_u1);
      mode m_T1 period 250 program P_T1{
        invoke t_T1 input((h1,1)) output((u1,3));
      }
    }

    module T2 start m_T2{
      task t_T2 input(c_int v_h2) state() output(c_int v_u2);
      mode m_T2 period 250 program P_T2{
        invoke t_T2 input((h2,1)) output((u2,3));
      }
    }
}

program P_T1{

    module T1_P_PI start m_T1_P{

      task t_T1_P input(c_int v_h1) state() output(c_int v_u1, c_controller_type prevController)
        function f_T1_P;
      task t_T1_PI input(c_int v_h1) state() output(c_int v_u1);

      mode m_T1_P period 250{
        invoke t_T1_P input((h1,1)) output((u1,3), (prevT1Controller,3)) parent t_T1;
        switch(withPerturbation(v1)) m_T1_PI;
      }

      mode m_T1_PI period 250 program P_T1_2PI{
        invoke t_T1_PI input((h1,1)) output((u1,3)) parent t_T1;
        switch(withoutPerturbation(v1)) m_T1_P;
      }
    }
}

program P_T2{
```

```
module T2_P_PI start m_T2_P{

    task t_T2_P input(c_int v_h2) state() output(c_int v_u2, c_controller_type prevController)
      function f_T2_P;
    task t_T2_PI input(c_int v_h2) state() output(c_int v_u2);

    mode m_T2_P period 250{
      invoke t_T2_P input((h2,1)) output((u2,3), (prevT2Controller,3)) parent t_T2;
      switch(withPerturbation(v2)) m_T2_PI;
    }

    mode m_T2_PI period 250 program P_T2_2PI{
      invoke t_T2_PI input((h2,1)) output((u2,3)) parent t_T2;
      switch(withoutPerturbation(v2)) m_T2_P;
    }
  }
}

program P_T1_2PI
{
    module T1_2PI start m_T1_PI_F{
      task t_T1_PI_S input(c_int v_h1, c_controller_type prevControllerOld, c_int u1old)
        state(c_PI_state s:=init_state) output(c_int v_u1, c_controller_type prevControllerNew)
function f_T1_PI_S;
      task t_T1_PI_F input(c_int v_h1, c_controller_type prevControllerOld, c_int u1old)
        state(c_PI_state s:=init_state) output(c_int v_u1, c_controller_type prevControllerNew)
function f_T1_PI_F;

      mode m_T1_PI_S period 250{
        invoke t_T1_PI_S input((h1, 1), (prevT1Controller,1), (u1, 1)) output((u1, 3),
  (prevT1Controller,3)) parent t_T1_PI;
          switch(PIFast(PI_SF1)) m_T1_PI_F;
      }

      mode m_T1_PI_F period 250{
        invoke t_T1_PI_F input((h1, 1), (prevT1Controller,1), (u1, 1)) output((u1, 3),
  (prevT1Controller,3)) parent t_T1_PI;
          switch(PISlow(PI_SF1)) m_T1_PI_S;
      }
    }
}

program P_T2_2PI
{
    module T2_2PI start m_T2_PI_F{
      task t_T2_PI_S input(c_int v_h2, c_controller_type prevControllerOld, c_int u2old)
        state(c_PI_state s:=init_state) output(c_int v_u2, c_controller_type prevControllerNew)
function f_T2_PI_S;
      task t_T2_PI_F input(c_int v_h2, c_controller_type prevControllerOld, c_int u2old)
        state(c_PI_state s:=init_state) output(c_int v_u2, c_controller_type prevControllerNew)
function f_T2_PI_F;

      mode m_T2_PI_S period 250{
        invoke t_T2_PI_S input((h2, 1), (prevT2Controller,1), (u2, 1)) output((u2, 3),
  (prevT2Controller,3)) parent t_T2_PI;
          switch(PIFast(PI_SF2)) m_T2_PI_F;
      }

      mode m_T2_PI_F period 250{
        invoke t_T2_PI_F input((h2, 1), (prevT2Controller,1), (u2, 1)) output((u2, 3),
  (prevT2Controller,3)) parent t_T2_PI;
          switch(PISlow(PI_SF2)) m_T2_PI_S;
      }
    }
}
```

E.4. Three Tanks System Controller HTL-Simulink Implementation

```
program P_3TS{
   communicator
      c_double h1 period 100 init c_zero;
      c_double h1_ref period 100 init c_zero;
      c_double h2 period 100 init c_zero;
      c_double h2_ref period 100 init c_zero;
      c_double u1 period 100 init c_zero;
      c_double u2 period 100 init c_zero;
      c_bool v1 period 500 init c_false;
      c_bool v2 period 500 init c_false;

   module IO start readWrite{
      task t_read input() state() output(c_double p_h1, c_double p_h2,c_bool p_V1, c_bool p_V2)
               function f_read;
      task t_write input(c_double p_u1:=def_double,c_double p_u2:=def_double) state() output()
               function f_write;
      task t_ref input() state() output(c_double p_h1_ref, c_double p_h2_ref) function f_ref;

      mode readWrite period 500{
         invoke t_read input() output((h1,3), (h2,3), (v1,1), (v2,1));
         invoke t_write input((u1,4), (u2,4)) output();
         invoke t_ref input() output((h1_ref,3), (h2_ref,3));
      }
   }

   module T1 start m_T1{
      task t_T1 input(c_double p_h1,c_double p_h1_ref) state() output(c_double p_u1);

      mode m_T1 period 500 program P_T1{
         invoke t_T1 input((h1,3),(h1_ref,3)) output((u1,4));
      }
    }

   module T2 start m_T2{
      task t_T2 input(c_double v_h2,c_double v_h2_ref) state() output(c_double v_u2);

      mode m_T2 period 500 program P_T2{
         invoke t_T2 input((h2,3),(h2_ref,3)) output((u2,4));
      }
   }
}

program P_T1{

   module T1_P_PI start m_T1_P{
      task t_T1_P input(c_double v_h1,c_double v_h1_ref) state() output(c_double v_u1)
               function f_T1_P;
      task t_T1_PI input(c_double v_h1,c_double v_h1_ref) state() output(c_double v_u1)
                function f_T1_PI;

      mode m_T1_P period 500{
         invoke t_T1_P input((h1,3),(h1_ref,3)) output((u1,4)) parent t_T1;
         switch(withPerturbation(v1)) m_T1_PI;
      }

      mode m_T1_PI period 500{
         invoke t_T1_PI input((h1,3),(h1_ref,3)) output((u1,4)) parent t_T1;
         switch(withoutPerturbation(v1)) m_T1_P;
      }
   }
}

program P_T2{

   module T2_P_PI start m_T2_P{
     task t_T2_P input(c_double v_h2,c_double v_h2_ref) state() output(c_double v_u2)
```

```
                            function f_T2_P;
        task t_T2_PI input(c_double v_h2,c_double v_h2_ref) state() output(c_double v_u2)
                            function f_T2_PI;


        mode m_T2_P period 500{
            invoke t_T2_P input((h2,3),(h2_ref,3)) output((u2,4)) parent t_T2;
            switch(withPerturbation(v2)) m_T2_PI;
        }

        mode m_T2_PI period 500{
            invoke t_T2_PI input((h2,3),(h2_ref,3)) output((u2,4)) parent t_T2;
            switch(withoutPerturbation(v2)) m_T2_P;
        }
    }
}
```

E.5. Exotask-HTL Graphs

E.5.1. Exotask Graph for mController Mode

```
<ExotaskGraph>
 <TimingProvider kind = 'htl' parser = 'at.uni_salzburg.cs.exotasks.timing.htl.HTLTimingDataParser'
   slowdownFactor = '1' graphics='60 60 15 99'>
  <ProgramList>
    <Program name = 'JAviatorControl'/>
    <Program name = 'PAttitudeControl'/>
  </ProgramList>
  <ModuleList>
    <Module name = 'MAttitudeControl' start = 'mAttitudeControl' program = 'JAviatorControl'/>
  </ModuleList>
  <ModeList>
    <Mode name = 'mAttitudeControl' period = '20' module = 'MAttitudeControl'
      refine = 'PAttitudeControl'/>
  </ModeList>
</TimingProvider>
 <Communicator id='fromGround' type='javiator.util.NavigationData' initialValue='()'
    graphics='91 60 230 93'>
  <Timing period = '1' program = 'JAviatorControl'/>
 </Communicator>
 <Communicator id='llcState' type='javiator.hierarchical.control.util.ControllerState'
    initialValue='()' graphics='85 60 220 182'>
  <Timing period = '1' program = 'JAviatorControl'/>
 </Communicator>
 <Task id='controlAttitude' implementation='javiator.hierarchical.control.attitude.
    AbstractAttitudeController' isolation='strong' graphics='72 60 464 70'>
  <Input id='sensors' type='javiator.util.SensorData'/>
  <Input id='targets' type='javiator.util.NavigationData'/>
  <Input id='oldActuators' type='javiator.util.ActuatorData'/>
  <Input id='isNewState' type='javiator.hierarchical.control.util.ControllerState'/>
  <Output id='actuators' type='javiator.util.ActuatorData'/>
  <Timing isAbstract = 'true' parent = ''>
    <ModeAssignment mode = 'mAttitudeControl'/>
  </Timing>
 </Task>
 <Communicator id='toJAviator' type='javiator.util.ActuatorData' initialValue='()'
    graphics='60 60 474 179'>
  <Timing period = '1' program = 'JAviatorControl'/>
 </Communicator>
 <Communicator id='fromJAviator' type='javiator.util.SensorData' initialValue='()'
    graphics='62 60 225 21'>
  <Timing period = '1' program = 'JAviatorControl'/>
 </Communicator>
 <Connection id='fromGround_controlAttitude' source='fromGround' target='controlAttitude'
    targetPort='targets'>
  <Timing instance='4' writesCommunicator='false'>
    <ModeAssignment mode = 'mAttitudeControl'/>
  </Timing>
 </Connection>
 <Connection id='fromJAviator_controlAttitude' source='fromJAviator' target='controlAttitude'>
```

```
   <Timing instance='4' writesCommunicator='false'>
     <ModeAssignment mode = 'mAttitudeControl'/>
   </Timing>
 </Connection>
 <Connection id='controlAttitude_toJAviator' source='controlAttitude' target='toJAviator'>
  <Timing instance='16' writesCommunicator='true'>
     <ModeAssignment mode = 'mAttitudeControl'/>
  </Timing>
 </Connection>
 <Connection id='llcState_controlAttitude' source='llcState' target='controlAttitude'
     targetPort='isNewState'>
  <Timing instance='0' writesCommunicator='false'>
     <ModeAssignment mode = 'mAttitudeControl'/>
  </Timing>
 </Connection>
 <Connection id='toJAviator_controlAttitude' source='toJAviator' target='controlAttitude'
     targetPort='oldActuators'>
  <Timing instance='0' writesCommunicator='false'>
     <ModeAssignment mode = 'mAttitudeControl'/>
  </Timing>
 </Connection>
</ExotaskGraph>
```

E.5.2. Exotask Graph for Communication Modules

```
<ExotaskGraph>
 <TimingProvider kind = 'htl' parser = 'at.uni_salzburg.cs.exotasks.timing.htl.HTLTimingDataParser'
    slowdownFactor = '1' graphics='60 60 446 19'>
  <ProgramList>
    <Program name = 'JAviatorControl'/>
  </ProgramList>
  <ModuleList>
    <Module name = 'MGroundComm' start = 'mGroundComm' program = 'JAviatorControl'/>
    <Module name = 'MState' start = 'mState' program = 'JAviatorControl'/>
  </ModuleList>
  <ModeList>
    <Mode name = 'mGroundComm' period = '100' module = 'MGroundComm' refine = ''/>
    <Mode name = 'mState' period = '20' module = 'MState' refine = ''/>
  </ModeList>
</TimingProvider>
 <Communicator id='fromGround' type='javiator.util.NavigationData' initialValue='()'
    graphics='91 60 245 12'>
  <Timing period = '1' program = 'JAviatorControl'/>
 </Communicator>
 <Task id='WriteToGround' implementation='javiator.hierarchical.control.communication.ProcessGroundReport'
    isolation='strong' graphics='105 60 492 104'>
  <Input id='fromGroundPort' type='javiator.util.NavigationData'/>
  <Input id='fromJaviatorPort' type='javiator.util.SensorData'/>
  <Input id='toJaviatorPort' type='javiator.util.ActuatorData'/>
  <Input id='statePort' type='javiator.hierarchical.control.util.ControllerState'/>
  <Timing isAbstract = 'false' parent = ''>
    <ModeAssignment mode = 'mGroundComm'/>
  </Timing>
 </Task>
 <Communicator id='llcState' type='javiator.hierarchical.control.util.ControllerState' initialValue='()'
    graphics='64 60 394 239'>
  <Timing period = '1' program = 'JAviatorControl'/>
 </Communicator>
 <Task id='WriteToJAviator' implementation='javiator.JControl.exotasks.MotorActuator'
    isolation='strong' graphics='78 60 254 162'>
  <Input id='inToJAviator' type='javiator.util.ActuatorData'/>
  <Timing isAbstract = 'false' parent = ''>
    <ModeAssignment mode = 'mState'/>
  </Timing>
 </Task>
 <Task id='ReadFromGround' implementation='javiator.JControl.exotasks.CommandSensor'
    isolation='weak' graphics='84 60 124 15'>
  <Output id='outNavigationData' type='javiator.util.NavigationData'/>
  <Timing isAbstract = 'false' parent = ''>
```

```
    <ModeAssignment mode = 'mGroundComm'/>
  </Timing>
</Task>
<Task id='ComputeStates' implementation='javiator.hierarchical.control.communication.ComputeStates'
    isolation='weak' graphics='74 60 270 236'>
 <Input id='thrustsPort' type='javiator.util.ActuatorData'/>
 <Output id='statePort' type='javiator.hierarchical.control.util.ControllerState'/>
 <Timing isAbstract = 'false' parent = ''>
    <ModeAssignment mode = 'mState'/>
 </Timing>
</Task>
<Communicator id='toJAviator' type='javiator.util.ActuatorData' initialValue='()'
    graphics='60 60 138 214'>
 <Timing period = '1' program = 'JAviatorControl'/>
</Communicator>
<Communicator id='fromJAviator' type='javiator.util.SensorData' initialValue='()'
    graphics='62 60 261 80'>
 <Timing period = '1' program = 'JAviatorControl'/>
</Communicator>
<Task id='ReadFromJAviator' implementation='javiator.JControl.exotasks.FlightSensor'
    isolation='strong' graphics='89 60 122 102'>
 <Output id='outFromJaviator' type='javiator.util.SensorData'/>
 <Timing isAbstract = 'false' parent = ''>
    <ModeAssignment mode = 'mState'/>
 </Timing>
</Task>
<Connection id='ComputeStates_llcState' source='ComputeStates' target='llcState'>
 <Timing instance='20' writesCommunicator='true'>
    <ModeAssignment mode = 'mState'/>
 </Timing>
</Connection>
<Connection id='fromJAviator_WriteToGround' source='fromJAviator' target='WriteToGround'
    targetPort='fromJaviatorPort'>
 <Timing instance='4' writesCommunicator='false'>
    <ModeAssignment mode = 'mGroundComm'/>
 </Timing>
</Connection>
<Connection id='toJAviator_WriteToJAviator' source='toJAviator' target='WriteToJAviator'>
 <Timing instance='16' writesCommunicator='false'>
    <ModeAssignment mode = 'mState'/>
 </Timing>
</Connection>
<Connection id='ReadFromGround_fromGround' source='ReadFromGround' target='fromGround'>
 <Timing instance='4' writesCommunicator='true'>
    <ModeAssignment mode = 'mGroundComm'/>
 </Timing>
</Connection>
<Connection id='toJAviator_ComputeStates' source='toJAviator' target='ComputeStates'>
 <Timing instance='16' writesCommunicator='false'>
    <ModeAssignment mode = 'mState'/>
 </Timing>
</Connection>
<Connection id='llcState_WriteToGround' source='llcState' target='WriteToGround'
    targetPort='statePort'>
 <Timing instance='20' writesCommunicator='false'>
    <ModeAssignment mode = 'mGroundComm'/>
 </Timing>
</Connection>
<Connection id='fromGround_WriteToGround' source='fromGround' target='WriteToGround'>
 <Timing instance='4' writesCommunicator='false'>
    <ModeAssignment mode = 'mGroundComm'/>
 </Timing>
</Connection>
<Connection id='toJAviator_WriteToGround' source='toJAviator' target='WriteToGround'
    targetPort='toJaviatorPort'>
 <Timing instance='16' writesCommunicator='false'>
    <ModeAssignment mode = 'mGroundComm'/>
 </Timing>
</Connection>
<Connection id='ReadFromJAviator_fromJAviator' source='ReadFromJAviator' target='fromJAviator'>
 <Timing instance='4' writesCommunicator='true'>
```

```
    <ModeAssignment mode = 'mState'/>
  </Timing>
 </Connection>
</ExotaskGraph>
```

E.6. Micro JAviator JAviator Low-Level Control

```
program MicroJAviator{
  communicator
    t_attitude   attitude      period 1 init attitude_init;   //t_atitude{roll, pitch, yaw, droll,
      // dpitch, dyaw, ddx, ddy, ddz}
    t_target   target        period 1 init target_init;    //t_target{roll_target, pitch_target,
      // yaw_target, z_target}
    t_altitude   altitude      period 1 init altitude_init;  //t_altitude{z, dz}
    t_thrusts   thrusts      period 1 init thrusts_init;    //t_thrusts{T1, T2, T3, T4}
    t_thrusts   manualThrusts  period 1 init thrusts_init;    //t_thrusts{T1, T2, T3, T4}
    t_position   position      period 1 init position_init;  //t_position(x, dx, y, dy)
    t_state   crrState      period 1 init state_init;     //t_state = unsigned char
    t_state   nextState      period 1 init state_init;    //t_state = unsigned char
    t_state    groundState     period 1 init state_init;    //t_state = unsigned char
    t_state    simState     period 1 init state_init;    //t_state = unsigned char
    t_int    stateChanged  period 1 init state_init;    //t_int = int
    t_int    dummy        period 1 init zero_init;    //t_int = int

  module JAviatorCommunication start mJAviatorConnect{
    // authenticate with JAvitor.
    task simConnect input() state() output(t_state simState) function sim_connect;

    //read sensor data and filter it.
    //use dummy communicator to delay task execution.
    task sensing input(t_int dummy) state() output(t_attitude attitude, t_altitude altitude,
      t_state simState) function sensing;

    //send new command to the JAviator
    task actuating input(t_thrusts thrusts) state() output() function actuating;

    mode mJAviatorCommunication period 20{
      invoke sensing input((dummy, 10)) output((attitude, 12), (altitude, 12), (simState, 12));
      invoke actuating input((thrusts, 16)) output();
      switch(isNotConnected(simState)) mJAviatorConnect;
    }

    mode mJAviatorConnect period 20{
      invoke simConnect input() output((simState, 12));
      switch(isConnected(simState)) mJAviatorCommunication;
    }
  }

  module GroundCommunication start mGroundConnect{
    // authenticate with JAvitor.
    task groundConnect input() state() output(t_state groundState) function ground_connect;

    //read targets and next requested state
    //t_thrust should be set only in manual mode
    //we need some synchronization primitives, maybe the best is
    //to send data from ground only after log information has been recieved
    task read input() state() output(t_target target, t_state nextState, t_thrusts thrusts,
      t_state groundState) function read;

    //send log information to the Ground station:
    //       (1) roll, pitch, yaw, z
    //       (2) droll, dpitch, dyaw, dz
    //       (3) ddx, ddy, ddz
    //       (4) T1, T2
    //       (5) T3, T4
    //although this data is send in 4 periods, it will be from
    //the same period (i.e., period when first packet is sent)
    //this task also computes next state
    task write input(t_attitude attitude, t_altitude altitude, t_thrusts thrusts, t_state crrState,
```

```
          t_state nextState)
        state(t_attitude s_attitude:=empty_attitude, t_altitude s_altitude:=empty_altitude,
        t_thrusts s_thrusts:=empty_thrusts, t_int count:=zero_init, t_int prevStateChanged:=zero_init)
        output(t_state outCrrState, t_state outNextState, t_int outStateChanged) function write;

    mode mGroundCommunication period 20{
      invoke read input() output((target, 5), (nextState, 5), (manualThrusts, 5), (groundState, 5));
      invoke write input((attitude, 5), (altitude, 5), (thrusts, 5), (crrState, 5), (nextState, 5))
        output ((crrState, 10), (nextState, 10), (stateChanged, 10));
      switch(isNotConnected(groundState)) mGroundConnect;
    }

    mode mGroundConnect period 20{
      invoke groundConnect input() output((groundState, 5));
      switch(isConnected(groundState)) mGroundCommunication;
    }
  }

  module Control start mControl{

    //this task will be invoked when emergency shutdown is needed
    task shutDown input() state() output(t_thrusts thrusts) function shutDown;

    //this task is a place holder for the control task
    task abstractControl input(t_thrusts prevThrusts) state() output(t_thrusts thrusts);

    mode mControl period 20 program PControl{
      invoke abstractControl input ((thrusts, 12)) output((thrusts, 16));
      switch(isShutDown(crrState)) mShutDown;
    }

    mode mShutDown period 20{
      invoke shutDown input() output((thrusts, 16));
    }
  }
}

program PControl{

  module MControl start mManual{
    //manual control task
    task manualControl input(t_thrusts manualThrusts) state() output(t_thrusts thrusts) function manual;

    //this task is a place holder for the control task
    task autoControl input(t_thrusts prevThrusts) state() output(t_thrusts thrusts);

    mode mManual period 20{
      invoke manualControl input((manualThrusts, 12)) output((thrusts, 16)) parent abstractControl;
      switch(isAuto(crrState)) mAuto;
    }

    mode mAuto period 20 program PAuto{
      invoke autoControl input((thrusts, 12)) output((thrusts, 16)) parent abstractControl;
      switch(isManual(crrState)) mManual;
    }
  }
}

program PAuto{

  module MAuto start mTakeOff{
    //this task implements the takeoff procedure
    task takeOff input(t_thrusts prevThrusts) state() output(t_thrusts thrusts) function takeOff;

    //this task implements the hover procedure
    task hover input(t_int stateChanged, t_attitude attitude, t_altitude altitude, t_target target,
      t_thrusts prevThrusts) state() output(t_thrusts thrusts) function hover;

    //this task implements the land procedure
    task land input(t_thrusts prevThrusts) state() output(t_thrusts thrusts) function land;
```
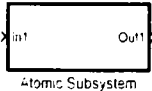
```
    mode mTakeOff period 20{
      invoke takeOff input((thrusts, 12)) output((thrusts, 16)) parent autoControl;
      switch(isHover(crrState)) mHover;
    }

    mode mHover period 20{
      invoke hover input((stateChanged, 12), (attitude, 12), (altitude, 12), (target, 12),
        (thrusts, 12)) output((thrusts, 16)) parent autoControl;
      switch(isLand(crrState)) mLand;
    }

    mode mLand period 20{
      invoke land input((thrusts, 12)) output((thrusts, 16)) parent autoControl;
    }
  }
}
```

F. Simulink Blocks

Block	Description
Memory	The output value of a *memory block* is represented by the input value from the previous step. A memory block can have a constant or an inherit sample time.
$\frac{1}{z}$ Unit Delay	The *unit delay* block delays the input signal with one step in time domain. The sample time of the block can either be constant or inherit.
In1 → Out1 Atomic Subsystem	A *subsystem* block can be used to modularize a Simulink model, a *subsystem* defines a set of inputs and outputs through which the block is interconnected with the rest of the Simulink model. The functionality of a *subsystem* block is implemented using Simulink block, it is possible to use *subsystem* blocks also. The *atomic subsystem* is a subsystem which is treated as a single entity when the Simulink schema has to be simulated. If a subsystem is not marked as atomic, then at simulation-time it is possible that blocks, which implement the functionality of the subsystem, to be intercalated with blocks from the rest of the Simulink model.

Block	Description
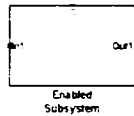
The *enabled subsystem* is a subsystem that has a control input, which determines when the subsystem has to be executed at simulation-time. The control input must have a positive value in order for the block to be executed. An *enabled subsystem* can contain both continuous and discrete block, and the sample time of discrete blocks does not have to be inherit.
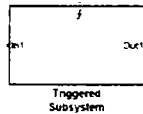
The *triggered subsystem* is a subsystem that has a control input, which determines when the block has to be executed at simulation-time. The execution of the block can be triggered: on the rising front of the control signal, on the falling front of the control signal, or on both fronts. In order to detect that there has been a rising or falling front, the signal has to stay on 1 or 0, respectively, for more than one sample time period. The subsystem can not contain continuous blocks and all the blocks must inherit the sample time.
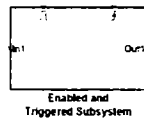
The *triggered and enabled subsystem* is a subsystem that combines the *triggered subsystem* and the *enabled subsystem*. It has two control inputs: a trigger control input and an enable control system. The subsystem executes when both control inputs activate, e.g., there is a rising or a falling front to the trigger control input and the enable control input is positive.
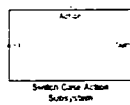
Block	Description
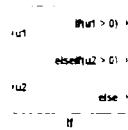
If block can be unused together with an *if action subsystem* in order to models a behavior similar to the *if* statement in a programming language. The block can have a variable number of inputs, and a variable number of outputs. Each output is associated with an expression, which depends on the inputs of the block and which is evaluated in order to determine if the corresponding output should be activated. Only one output can be active at a particular moment in time. When an output activates the execution of the action subsystem connected to that output, will be triggered.
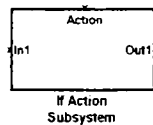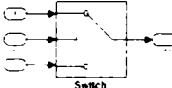
The *if action subsystem* is a subsystem which can be connected to the outputs of an *if* block. The subsystem executes only if the output to which it is connected is active.
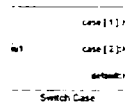
The *switch case* block together with an *switch case action subsystem* block models a behavior similar to the *switch case* statement in programming languages. The *switch case* block has one input and multiple outputs. Only one of the inputs can be active at a specific moment in time. The block also has a *default* output, which activates when none of the inputs activates. Activation of one of the outputs of the *switch case* block will trigger execution of the action subsystem that is connected to that output.

The *switch case action subsystem* is a subsystem which can be connected to the outputs of a *switch case* block. The subsystem executes only if the output to which it is connected is active.

Block	Description
	The *mux* block transforms the signals from its inputs into a single vectorial signal. The resulted signal at any moment contains an array of values, i.e., one value for each input. The block can have multiple inputs but only one output. The input signals can be scalar or vectorial.
	The *demux* block splits a vectorila signal into multiple scalar or vectorial signals, i.e., it is the opposite of the *mux* block. The block can have one input but multiple outputs. Usually the *mux* and *demux* blocks are used in pair.
	T e *switch* oc as t ree inputs and one output, based on the second input the block w p fi the third input.
	The *merge* blocks combines multiple input signals into one output single signal. The output value of this block is equal with the last output value of one of the blocks that send signals to the block. Usually the *merge* block is used to combine signals from blocks that execute alternatively.

References

[1] Nicolae Robu. **Programare Concurenta. Mecanisme Suport Orientate Timp Real**. Editura Politehnica, 2002. 19, 21, 24, 69, 89, 116

[2] A. Ghosal, T.A. Henzinger, **D. Iercan**, C.M. Kirsch, and A.L. Sangiovanni-Vincentelli. **A Hierarchical Coordination Language for Interacting Real-Time Tasks**. In *Proc. EMSOFT*, Seoul, South Korea, 2006. 19, 21, 22, 25, 48, 115, 117

[3] A. Ghosal. **A hierarchical coordination language for reliable real-time tasks**. Technical report, University of California, Berkeley, 2008. Ph.D. thesis. 19, 22, 25

[4] A. Ghosal, T.A. Henzinger, **D. Iercan**, C.M. Kirsch, and A.L. Sangiovanni-Vincentelli. **Hierarchical timing language**. Technical report, University of California, Berkeley, 2006. Technical report. 19, 25, 117

[5] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. **Giotto: A Time-triggered Language for Embedded Programming**. *Proceedings of the IEEE*, **91**(1):84--99, January 2003. 19, 21, 22, 24, 25, 69, 115

[6] T.A. Henzinger and C.M. Kirsch. **The Embedded Machine: Predictable, Portable Real-Time Code**. *ACM TOPLAS*, **29**(6):33--61, October 2007. 20, 22, 24, 35, 115

[7] A. Ghosal, **D. Iercan**, C.M. Kirsch, T.A. Henzinger, and A.L. Sangiovanni-Vincentelli. **Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code**. In *Workshop Proc. APGES 2007*, Salzburg, Austria, 2007. 20, 35, 48, 117

[8] J. Auerbach, D.F. Bacon, **D. Iercan**, C.M. Kirsch, V.T.Rajan, H. Rock, and R. Trummer. **Java takes flight: Time-portable real-time programming with exotasks**. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools, San Diego, California, USA*, 2007. 20, 23, 69, 70, 77, 116, 117

[9] J. Auerbach, D.F. Bacon, **D. Iercan**, C.M. Kirsch, V.T.Rajan, H. Rock, and R. Trummer. **Low-Latency Time-portable Real-time Programming with Exotasks**. *ACM Transactions on Embedded Computing Systems*, 2008. 20, 80, 116, 117

[10] **Simulink**. http://www.mathworks.com/products/simulink/. 20, 101

[11] S.S. Craciunas, C.M. Kirsch, H. Röck, and R. Trummer. **The JAviator: A High-Payload Quadrotor UAV with High-Level Programming Capabilities**. In *Proc. AIAA Guidance, Navigation and Control Conference and Exhibit (GNC)*, 2008. 21, 24, 61, 76, 83, 87

[12] **JAviator homepage**. http://javiator.cs.uni-salzburg.at/. 21, 76

[13] C.M. Kirsch and R. Sengupta. **The Evolution of Real-Time Programming**. In *Handbook of Real-Time and Embedded Systems*. Chapman and Hall/CRC, 2007. 21

[14] **Real-Time Linux**. http://www.rtlinuxfree.com/. 21

[15] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publisher, 1997. 21, 26, 83

[16] N. Halbwachs. **Synchronous programming and reactive systems**. 21

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. **The synchronous dataflow programming language Lustre**. 21, 22

[18] G. Berry and G. Gonthier. **The Esterel synchronous programming language: design, semantics, and implementation**, 1992. 21, 22

[19] N. Halbwachs. **A synchronous language at work: the story of Lustre**. 21

[20] S. Fischmeister and I. Lee. **Temporal Control in Real-Time Systems: Languages and Systems**. In *Handbook of Real-Time and Embedded Systems*. Chapman and Hall/CRC, 2007. 21

[21] E. Farcas, C. Farcas, W. Pree, and J. Templ. **Transparent distribution of real-time components based on logical execution time**. 22

[22] J. Liu and E. A. Lee. **Timed multitasking for real-time embedded software**. *IEEE Control Systems Magazine*, 2003. 22

[23] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. A. Sanvido. **Event-driven programming with logical execution times**. *Hybrid Systems Computation and Control, Lecture Notes in Computer Science 2993*, 2004. 22

[24] **D. Iercan** and A. Ghosal. **Timed Input/Output Determinacy for Tasks with Precedence Constraints**. In *Proc. Of the 7th International Conference On Tehnical Informatics - CONTI08*, **2**, pages 149--154, June 2006. 22, 117

[25] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. **Java operating system: Design and implementation**. Technical report, University of Utah, 1998. 23

[26] H. McGhan and M. O'Connor. **picoJava: A direct execution engine for Java bytecode**. In *IEEE Computer*, **31**, 1998. 23

[27] G. Bollella, J. Gosling, B.M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000. 23, 70

[28] A. Wellings and A. Burns. **Real-Time Java**. In *Handbook of Real-Time and Embedded Systems*. Chapman and Hall/CRC, 2007. 23

[29] D. F. Bacon, P. Cheng, and V. T. Rajan. **A Real-time Garbage Collector with Low Overhead and Consistent Utilization**. In *Proc. POPL*, pages 285--298, New Orleans, Louisiana, January 2003. 23, 70

[30] Fridtjof Siebert. **The Impact of Realtime Garbage Collection on Realtime Java Programming.** In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 33--40, 2004. 23

[31] Fergus Henderson. **Accurate garbage collection in an uncooperative environment.** pages 256--263, 2002. 23

[32] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. **Eventrons: a safe programming construct for high-frequency hard real-time applications**. In *Proc. PLDI*, pages 283--294, Ottawa, Ontario, Canada, 2006. 23, 69

[33] J. H. Spring, F. Pizlo, R. Guerraoui, and J. Vitek. **Programming Abstractions for Highly Responsive Systems**. In *Proc. VEE*, San Diego,, 2007. 23, 69

[34] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. **StreamFlex: High-throughput Stream Programming in Java**. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2007. 23, 69

[35] IBM. **DDG1000 Next Generation Navy Destroyers**, 2007. www.ibm.com/press/us/en/pressrelease/21033.wss. 23, 80, 116

[36] A. L. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi. **Benefits and Challenges for Platform-Based Design**. In *Proc. DAC*, **91**. ACM, 2004. 25

[37] K. Chatterjee, A. Ghosal, **D. Iercan**, C.M. Kirsch, T.A. Henzinger, C. Pinello, and A.L. Sangiovanni-Vincentelli. **Logical reliability of interacting real-time tasks**. In *Proc. DATE*, pages 909--914. IEEE, 2008. 25, 117

[38] **D. Iercan** and M. Mezin. **A Distributed Multimode Real-Time Controller for the Three Tanks System**. In *Proc. Of the 8th International Conference On Tehnical Informatics - CONTI08*, **3**, pages 67--70, June 2008. 61, 117

[39] Eclipse Foundation. **The Eclipse Open Development Platform**. www.eclipse.org, 2007. 70

[40] IBM Corp. *WebSphere Real-Time User's Guide*, first edition, 2006. 70

[41] Mike Fulton and Mark Stoodley. **Compilation Techniques for Real-Time Java Programs**. In *Proc. International Symposium on Code Generation and Optimization*, 2007. 70

[42] **Quadrotor wiki**. http://en.wikipedia.org/wiki/Quadrotor. 76

[43] **Gumstix**. http://www.gumstix.com/. 76

[44] **Robostix**. http://docwiki.gumstix.org/index.php/Robostix. 76, 83

[45] **Microstrain Gyro**. http://www.microstrain.com/3dm-gx1.aspx. 76

[46] **SFR10 Ultrasonic range finder**. http://www.robot-electronics.co.uk/htm/srf10tech.htm. 76

[47] **ATmega128**. http://www.atmel.com. 83

[48] Miro Samek and Robert Ward. **Build a Super Simple Tasker**. http://www.embedded.com/columns/ technicalinsights/190302110. 83

[49] **Real-Time Workshop**. http://www.mathworks.com/products/rtw/. 101

[50] T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, and W. Pree. **From Control Models to Real-Time Code using Giotto**. *IEEE Control Systems Magazine*, **23**(1):50--64, February 2003. 101

[51] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. **Translating Discrete-Time Simulink to Lustre**. *ACM Transactions on Embedded Computing Systems*, **4**(4):779--818, November 2005. 101

[52] A. Ghosal, **D. Iercan**, C.M. Kirsch, T.A. Henzinger, and A.L. Sangiovanni-Vincentelli. **Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code**. 2008. 117

[53] M.S. Kang, S. Park, H.G. Lee, D.H. Won, and T.J. Kim. **Development of a Hovering Robot System for Calamity Observation**. In *ICCAS2005*, June 2005. 123

[54] S. Bouabdallah, P. Murrieri, and R. Siegwart. **Design and Control of an Indoor Micro Quadrotor**. In *ICRA, New Orleans*, April 2004. 123

[55] Steven L. Waslander, Gabriel M. Hoffmann, Jung Soon Jang, and Claire J. Tomlin. **Multi-Agent Quadrotor Testbed Control Design: Integral Sliding Mode vs. Reinforcement Learning**. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, Edmonton, Alberta, Canada*, August 2-6. 123