Daniel T. Iercan

TSL Compiler

Lucrare de diploma,

realizata de Daniel T. Iercan

pentru obtinerea titlului de Diplomat Inginer.

Supervizori: Prof. Dr. Ing. Stefan Preitl

Prof. Dr. Ing. Christoph Kirsch

S.L. Ing. Florin Dragan

Pentru Ana.

Multumiri

Parintilor mei.

Multumesc!

Contents

Rezumat

Tema lucrarii de diploma consta in realizarea unui compilator pentru un limbaj de programare de nivel inalt, destinat in special aplicatiilor “hard real-time”, numit Timing Specification Language (*TSL*), precum si rezolvarea unei probleme de conducere utilizand acest limbaj.

Rezolvarea unei probleme de conducere consta din doua etape: (1)inginerul automatist prelucreaza ecuatiile diferentiale, care descriu din punct de vedere fizic procesul care urmeaza sa fie condus, utilizand programe ca Matlab si (2)inginerul programator implementeaza algoritmul de reglare pe o anumita platforma (prin platforma se intelege o anumita configuratie hard impreuna cu sistemul de operare in timp real). Ceea ce trebuie sa faca un inginer automatist este: modelarea procesului si a perturbatiilor, obtinerea si optimizarea legii de reglare si validarea functionalitatii si performatelor sistemului de reglare automat (SRA) prin analiza si simulare. Dupa ce regulatorul a fost proiectat, el trebuie implementat pe o anumita platforma de catre un inginer programator. In mod normal inginerul programator descompune activitatile computationale necesare in taskuri, seteaza prioritatilor pentru taskuri, astfel incat sa indeplineasca conditiile de timp-real pentru un algoritm de dispecerizare dat si o configuratie hardware data, dar si asigura un nivel de toleranta la erori, prin reproducerea si corectarea erorilor.

TSL asigura un nivel intermediar de abstractizare, care 1) permite inginerului programator sa comunice mai eficient cu inginerul automatist si 2) apropie mai mult implementarea de modelul matematic al regulatorului. *TSL* defineste o arhitectura soft care permite separarea functionalitatii de timing. Functionalitatea si timing-ul sunt suficiente pentru a asigura, ca implementarea este consistenta relativ la modelul matematic. Pe de alta parte, permite inginerului programator sa nu isi faca griji in ceea ce priveste performantele echipamentului hardware si algoritmului de dispecerizare, atunci cand discuta cu inginerul automatist. Dupa ce programul *TSL* a fost scris, ceea ce ii mai ramane de facut iniginerului programator este sa implementeze

programul pe o platforma data, in *TSL* acest pas este total decuplat de primul si se poate executa fara ca inginerul programator sa mai discute cu inginerul automatist.

TSL este de fapt o extensie a limbajului *Giotto*. *Giotto* se bazeaza pe un model de taskuri numit LET (Logical Execution Time); in acest model timpul logic de terminare este specificat in momentul in care taskul este eliberat. Iesirile taskului sunt disponibile numai dupa timpul de terminare, chiar daca executia taskului s-a incheiat mai devreme. *Giotto* are trei proprietati importante: 1)“time and value determinism”, 2)“switchability”, si 3) “schedulability”. “Time determinism” inseamna ca senzorii sunt cititi, iar elementele de executie sunt scrise la momente de timp predeterminate. “Value determinism” inseamna ca: dandu-se o secventa de valori pentru senzori, rezulta o secventa de valori pentru elementele de executie, care este in mod unic determinata de program si nu depinde de modul in care taskurile sunt dispecerizate. “Switchability” este posibilitatea de a trece de la un mod la altul. “Schedulability” inseamna ca toate taskurile care au fost eliberate isi vor termina executia inainte de expirarea intervalului de timp alocat. Totusi *Giotto* are si cateva limitari: prima ar fi aceea ca momentul de timp in care este eliberat un task si cel in care se termina sunt definite implicit prin perioada taskului si a doua limitare este ca toate taskurile sunt LET-uri si nu exista posibilitatea de a avea taskuri clasice cu constrangeri de precedenta specificate de dependente ale intrarilor si iesirilor.

Pentru a elimina cele doua neajunsuri ale lui *Giotto*, a fost definit *TSL*, care este o extensie a primului limbaj. In acest nou limbaj a fost generalizat conceptul de task LET introdus in *Giotto* si a fost combinat cu taskuri care au constrangeri de precedenta.

Un task LET generalizat este numit *anchored task*; momentul in care taskul este eliberat si cel in care el trebuie sa se termine sunt specificate prin, “release time” specificat ca si un offset relativ la perioada taskului, respectiv “termination time” este specificat ca fiind sfarsitul perioadei taskului. Intrarile taskului sunt citite in momentul in care taskul este eliberat, iar iesirile sunt scrise in momentul in care taskul se termina (“termination time”); intre cele doua momente de timp taskul poate fi executat oricum, singura constrangere fiind aceea ca el nu poate sa comunice cu alte taskuri. In afara de taskurile ancorate *TSL* are si taskuri mobile (“float tasks”). Constrangerile taskurilor mobile sunt specificate prin dependente. Fiecare dependenta introduce o constrangere de precedenta in ceea ce priveste executia unui task, astfel incat taskul nu va putea fi executat pana cand nu sunt indeplinite toate constrangerile de precedenta. Pentru a asigura determinismul, citirea sensorilor, respectiv scrierea elementelor de

executie sunt ancorate in timp.
Un program *TSL* consta din:

porturi – un port este utilizat pentru a comunica cu mediul exterior sau intre taskuri;

drivere – sunt utilizate pentru a transfera informatia de la un port la altul si pentru a o converti daca este cazul;

taskuri – sunt utilizate pentru a face calcule consumatoare de timp (ex., calculul unei legi de reglare), in mod normal un task citeste de la unul sau mai multe porturi de intrare si actualizeaza unul sau mai multe porturi de iesire

moduri – un mod are o perioada, prin care se precizeaza cu ce frecventa este executat modul respectiv; un mod consta din invocari de taskuri ancorate si mobile, actualizari de senzori, actualizari de elemente de executie si schimbari de mod, toate aceste elemente sunt caracterizate de o frecventa, iar in afara de taskurile mobile si schimbarile de mod, toate elementele mai au si un offset, care este specificat relativ la inceputul perioadei.

Compilatorul de *TSL*va avea ca intrare, un program *TSL*, iar rezultatul compilarii va fi un fisier care va contine *E code*-ul, care va fi interpretat de o masina virtuala, numita *E Machine*. Compilatorul este construit pornind de la un parser, care a fost generat automat cu *SableCC*, care este un tool specializat in generarea de parsere pornind de la un fisier in care este descrisa gramatica limbajului respectiv. Dupa ce un program a fost parsuit, se obtine un Abstract Syntax Tree (AST). Utilizand acest AST compilatorul de *TSL*va:

- crea o tabela de simboluri, care va contine cate o lista pentru fiecare tip de declaratie, in plus compilatorul va verifica unicitatea fiecarei declaratii;
- verifica tipurile, in acest pas compilatorul va verifica daca tipul si numarul parametrilor formali corespund cu tipul si numarul parametrilor actuali;
- crea tabela de dependente, aceasta tabela va contine informatii despre dependentele de intrare, respectiv iesire ale unui task mobil;
- verifica dependentele, in acest pas compilatorul va cauta bucle inchise intre dependente;

- verifica frecventele, in acest pas compilatorul va verifica faptul ca frecventa unui element dintr-un anumit mod este mai mare ca zero si divide perioada modului, de asemenea pentru taskurile mobile va verifica faptul ca dependentele sale au aceeasi frecventa ca si el;

- verifica schimbarile de mod, in acest pas compilatorul verifica faptul ca, prin trecerea de la un mod la altul nu apar supraincarcari;

- genera *E code*-ul, in acest pas compilatorul va genera *E code*-ul.

Procesul ales pentru a fi condus este *Sistemul Celor Trei Rezervoare* (3TS). Procesul consta din trei rezervoare (T_1, T_2, and T_3), T_1 si T_2 fiind interconectate cu T_3. Fiecare rezervor este prevazut si cu un robinet de scurgere in exterior, T_2 avand doi robineti de scurgere in exterior. Avand in vedere ca nu am avut disponibil procesul, am realizat un simulator (program scris in Java), care poate fi condus prin TCP/IP, in plus calculatorul la care este conectat procesul este unul destul de vechi (486), pentru ca foloseste o placa de achizitie pentru care nu mai exista slot compatibil in calculatoarele mai noi, acesta fiind inca un motiv pentru care nu am condus direct procesul. Regulatorul a fost implementat in doua variante: o prima varianta a fost implementata in Java, acest program putand fi folosit atat ca si regulator, dar si doar pentru a urmarii evolutia semnalelor din proces, a doua varianta este implementata in *TSL*. Asa cum am spus a fost nevoie de un simulator pentru ca nu am avut disponibil procesul, totusi trecerea de pe simulator pe procesul real se poate face destul de simplu fara a face modificari in regulator. Aceasta pentru ca, comenzile regulatorului se dau prin socketuri, ceea ce inseamna ca este suficienta scrierea uni program care sa ruleze pe calculatorul la care este conectat procesul si care sa implementeze partea de server a protocolului “Control Protocol”, acesta fiind protocolul utilizat in comunicarea dintre regulator si simulator.

Lucrarea este impartita in doua parti: (1) suportul teoretic si (2) implementare.

In prima parte sunt prezentate elementele de natura teoretica pe care se bazeaza lucrarea, astfel, in **Chapter 1** sunt prezentate conceptual Limbajul *Giotto*, Limbajul *TSL*, si *SableCC*, care este un utilitar pentru generarea automata a unui parser, tot in prima parte este inclus si **Chapter 2** in care sunt sunt prezentate conceptele din Ingineria Reglari utilizate in studiul de caz.

In a doua parte sunt prezentate modificarile aduse compilatorului *Giotto*, astfel incat sa ofere support pentru noul limbaj (*TSL*) (**Chapter 3**), precum si studiul de caz ales pentru a scoate in evidenta proprietatile limbajului *TSL*.

In finalul lucrarii sunt prezentate cateva idei pentru dezvoltarea ulterioara a limbajului *TSL*.

Introduction

The main goal of this thesis is to extend *Giotto Compiler*[4], which is a compiler for *Giotto* Language[2], in order to support the new features, introduced by Timing Specification Language (*TSL*), which is an extension of *Giotto*. Still in this thesis I will present a solution to a control problem, that uses *TSL* for the implementation.

Giotto is a high level language that provides an intermediate level of abstraction in the cycle of developing control application. In *Giotto* one can easily observe the separation between timing and functionality. In fact in *Giotto* can be implemented only the control application timing, while the functionality will be implemented in another language (e.g., C). In **Fig. 1** I present the new model introduced by *Giotto*. As it can be seen from the figure after compiling a *Giotto* program, results the *E code*, that will be interpreted by a virtual machine, called the *E Machine*. This approach makes *Giotto* platform independent, in the sense that, given a *Giotto* program, it could be run on any platform(the word "platform" represents a hardware architecture together with the operating system running on it), with the condition that there should be an *E Machine* for that platform. The basic elements *Giotto* deals with are: ports, drivers, and tasks. All of them are presented in **Section 1.1**, Here I will say a few words about tasks in *Giotto*. A tasks in *Giotto* is a piece of code, that needs a significant amount of time to be executed. An important property of a task, is task period, which defines how often the task will be executed, but also defines the time interval in which task should be executed, and even if the task is faster, and it doesn't need the hole period to execute, the results produced by task will be considered to be valid only at the end of the period.Hence the period of a task is also called Logical Execution Time (LET), and a *Giotto* tasks is also called a LET task.

Giotto has three main properties:

1. the first property is *time and value determinism*. Time determinism means that sensors

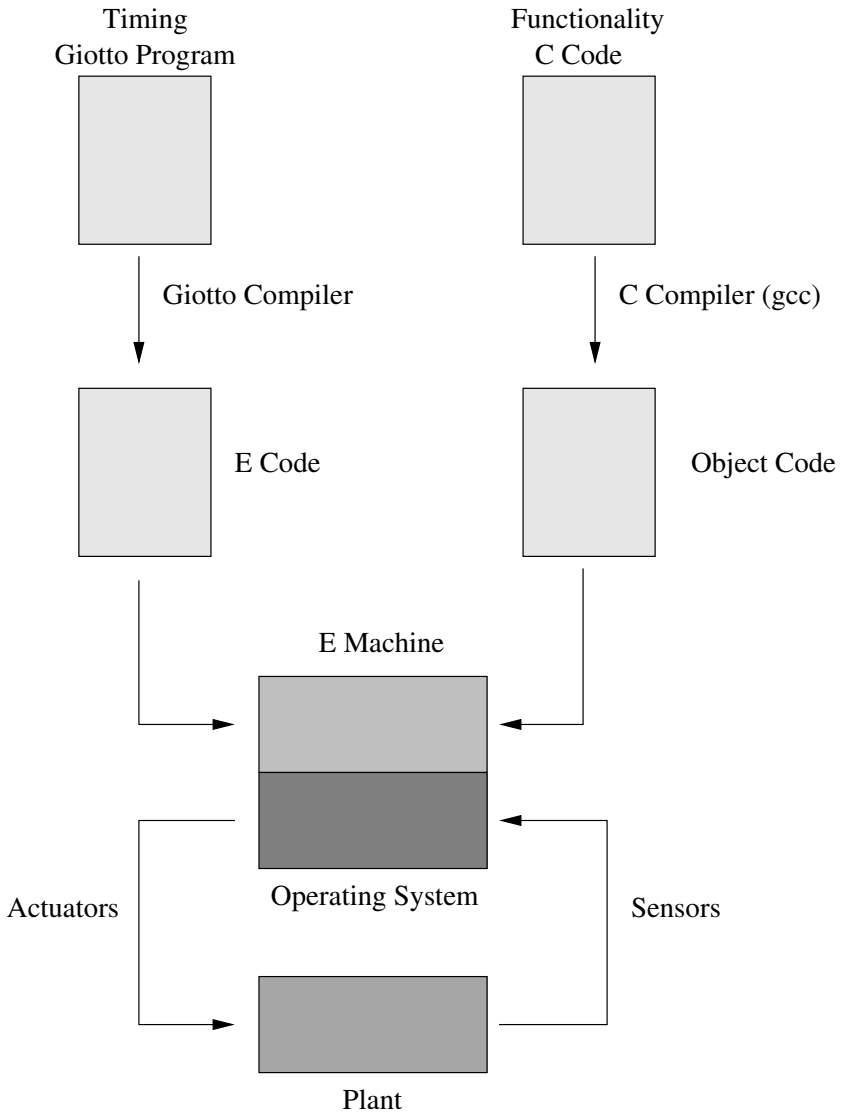


Figure 1: Giotto model.

are read and actuators are written at predetermined points in time. Value determinism means that given a sequence of sensor readings, the corresponding sequence of actuator writings is uniquely determined by the program; it does not depend on the scheduling of the tasks. This implies in particular, the absence of race conditions and priority inversion problems.

2. the second property of *Giotto* is *switchability*, the ability to switch modes in a nontrivial fashion, possibly preempting the LET of tasks, while maintaining determinism.

3. third, *Giotto* allows for a simple check of *schedulability*, that all released tasks are able to complete execution before termination.

However, control applications from the automotive industry have also uncovered several severe shortcomings of *Giotto*. The first limitation of *Giotto* is that both the release and termination times of tasks are defined implicitly through the task period: every task is released at the beginning of its period, and terminated at the end of the period. This is problematic in particular with short but infrequent tasks. The second limitation of *Giotto* is that all tasks are LET tasks, and no accommodation is provided for classical real-time tasks with precedence constraints specified by I/O dependencies.

In order to overcome this limitations a new language called Timing Specification Language (*TSL*) was created. *TSL* is an extension of *Giotto*, a language targeted towards hard real-time applications with multi-modal time-periodic behavior. In order to do this, the LET tasks of *Giotto* were generalized and combined with tasks that have precedence constraints.

A generalized LET task is called an *anchored task*; its scheduling constraints are specified by an arbitrary release time and an arbitrary termination time. The task inputs are read at the release time and the task outputs are written at the termination time; the task can be executed between these two times in any way, but no interaction with other tasks may happen between these two times. The release time is specified as an *offset* relative to the period of the task. In addition to anchored tasks, *TSL* has *floating tasks*. The scheduling constraints of a floating task are specified only through dependencies: task inputs may depend on sensor readings and the outputs of other (anchored and floating) tasks; task outputs may be depended on by drivers writing the inputs of other tasks and actuators. Each such dependency introduces a precedence constraint on the scheduler; as long as these constraints are met, floating tasks can

be executed at any time. To ensure determinism, all sensor readings and actuator writings are anchored in time; they have fixed periods and offsets. To ensure the efficient schedulability of mode switches, from one mode to the next, some (anchored and floating) tasks may be removed or some tasks may be added, but not both.
A *TSL* program consists of:

ports – a port is used for communication with the environment or for communication between tasks;

drivers – are used to transfer and convert the information between ports;

tasks – are used for computing time consuming functions that read from task input ports an state ports and updates task output ports and state ports;

modes – a mode has a period that specifies after how much time the mode will be executed again; a mode consists of task invocations (anchored or float), sensors updates, actuator updates, and mode switches, each of the presented elements has a frequency that specifies how many times it will be executed per mode, and except for float task invocations, and mode switch, all the elements can have an offset.

The input for the *TSL Compiler* will be of course a *TSL* program, and the result of the compilation will be the *E code*, as for a *Giotto* program, but the *E code* generated for *TSL* programs is an extension of the *E code* generated for a *Giotto* program. The resulting *E code* will be interpreted by a virtual machine, also called *E Machine*, which is an extension of the *E Machine* used for *Giotto E code*.

The *TSL Compiler* is based on a parser that was automatically generated using a compiler compiler tool called *SableCC*. After a *TSL* program was parsed, using the parser generated by *SableCC*, an Abstract Syntax Tree (AST) is obtain. Using this AST the *TSL Compiler* will:

- create a *Symbol Table*, the *Symbol Table* will contain a list for each type of declaration (e.g., a list for tasks declaration, one for drivers, etc.), at this step the compiler will perform some checks, so that there will not be two definitions with the same name ;
- perform type checking, on this step the compiler will verify that formal parameters type and number is the same with the actual parameters type and number;

- create a *Dependency Table*, this table will contain information about what elements a float task depends on, and what are the elements that depend on a task;
- perform the dependency check, this means that the compiler will check the program against closed loops;
- frequency check, on this step the compiler will check that the frequency of each element from a node is not zero and divides mode period, also for float tasks it will check that the elements that depends on the task as well as the elements the task depends on have the same frequency;
- mode switch check, on this step the compiler will check that each mode switch is possible without overloading the system;
- *E code* generation, on this step the compiler will generate the *E code*, for the *TSL* program got as input, the generation of the *E code* consist of two parts, first C code will be generated for the *Symbol Table*, this will be compiled and linked into the *E Machine*, and second will be generated the *E code*.

The plant I have chosen to control is *Three Tanks System* (3TS). The plant is made up of 3 tanks (T_1, T_2, T_3), and both T_1 and T_2 are connected with T_3, through pipes. Each tank has a draining pipe, that let the fluid go out of the tank, tank T_2 has 2 such pipes. There are also two pumps connected to T_1, respectively T_2, and through this pumps the controller will be able to control the level of the fluid in T_1, and T_2. The command to the pumps in given in voltage.

The 3TS plant is interesting because it is nonlinear, multi-variable, and in order to be able to controller it in all possible scenario you need more then one controller, which in terms of *TSL* means mode switch (which is an important feature of *TSL*).

Since I didn't have the process available, I had implemented a simulator (*3TS Simulator*) for the process (written in Java). The simulator has to functioning modes: (1) manual, in this mode the user can control the command given to pumps, and (2) auto, in this mode the user can only change the opening coefficient for taps, while the commands to pumps are given by a regulator. The simulator acts as a server, it waits for a regulator to connect via TCP/IP. The communication between the plant simulator server and the controller client is done using a protocol, defined by me, called *Control Protocol*.

The controller is implemented both in Java (3TS Conttroller) and in *TSL*. The Java version of the controller can be used in two modes: (1) view mode, in this mode it will be connected to the process server, but it will not control the plant, it will just read the main signal values and it will draw them, and (2) control mode, in this mode the controller will control the plant.

The thesis is split in two parts: (1) theoretical background, and (2) implementation. In the first part I present theoretical concepts on which is based my thesis, in this part I present conceptually the old *Giotto* Language, and the new *TSL* Language, and *SableCC*, all this three subjects are presented in **Chapter 1**, as well as the control engineering concepts I used for the case study (**Chapter 2**).

In the second part I present the changes I made to *Giotto* Compiler in order to add support for the new features introduced by *TSL* Language (**Chapter 3**). Still in this part I will present the case study I chose, in order to show some of the new features introduced in *TSL* (**Chapter 4**).

Thesis ends with some ideas we have about possible extensions of *TSL*.

Chapter 1

Timing Specification Language (TSL)

In this chapter I will present *Giotto* language, which is base for the *TSL* then I will preset the *TSL*. I will also present *SableCC*, which is a compiler compiler tool, that was used when implementing *Giotto Compiler* and of course *TSL Compiler*.

1.1 Giotto

Giotto[2] is a platform-independent language for specifying software for high-performance control applications. The *Giotto Compiler* generates code for a virtual machine, called the *E Machine*[3], which can be ported to different platforms. The *Giotto Compiler* also checks if the generated *E Code* is time safe for a given platform, that is, if the platform offers sufficient performance to ensure that *E Code* is executed in a timely fashion that conforms with the *Giotto* semantics. The most important benefit of the platform-independent approach is that it permits a clean separation of timing and function.

There are two kinds of software processes, which together make up the functional part of an *E program*. One that needs a significant amount of time, or in other words has a non-negligible WCET (worst-case execution time), called task (i.e., computation of a control law), and the other one, that has a negligible WCET, called driver(i.e., reading the value of a sensor). Both tasks and drivers are written in a conventional language, such as C.

The timing part of an *E program* consists of a set of E actions. Each E action is triggered by an event, and may call a driver, which is executed immediately, or release a task which is handed over to the schedule of the operating system. The *E Machine* is a virtual machine that

executes *E Code*, whose instructions can specify arbitrary sequences of E actions. *Giotto* is a structured language for specifying limited combinations of E actions that occur typical control applications. Next I will present the syntax for *Giotto* language and the *E Machine*, each in a different subsection.

1.1.1 Giotto syntax

The basic functional unit in *Giotto* is the task, which is a periodically executed piece of, say, C code. Several concurrent tasks make up a mode. Tasks can be added or removed by switching from one mode to another. Tasks communicate with each other, as well as with sensors and actuators, by so-called drivers, which is code that transports and converts values between ports. A *Giotto* program does not specify where, how and when tasks are scheduled. Further I will describe informally the syntax.

Ports – in *Giotto* all data is communicated through ports. A port is a typed variable located in a globally shared name space. There are three types of ports in *Giotto* program: sensors, actuators, and tasks ports. Sensors ports are updated by environment and read by the *Giotto* program. Actuators pots are updated by *Giotto* program and read by the environment. Tasks ports are both read and updated by *Giotto* program, and they are used to communicate between tasks and modes. An important particularity of tasks ports is the fact that they are double buffered, in order to avoid inconsistency.

Tasks – a task is a piece of code written in a sequential language, which contains no synchronization and can not be terminated prematurely. At an abstract level a task is nothing more then a function that reads a set of input ports and state ports, and updates a set of output ports and state ports. There are some constrains a task should respect: (1)input ports and state ports should be unique for each task, while (2) output ports must be unique only for tasks invoked in the same mode.

Drivers – a driver is a piece of code that can be executed in logical zero time. It is a function that reads from a set of input ports and updates a set of output ports. A driver is guarded by a condition and only if that condition is true the driver function will be executed.

Task Invocations – a *Giotto* task is a periodic tasks. A task invocation is characterized by the

task it invokes and the frequency the task is invoked. The frequency is a non-zero natural number (ω_{task}), which specifies how many times the task will be invoked per mode and it also specifies task period.

Mode – a *Giotto* program consists of a set of modes, but only one mode is active at a specified moment in time, there can be specified a possible transition from one mode to another through a mode switch. A mode consist of a set of task invocations, actuator updates and mode switches. A mode has a period, and a set of mode ports which are update when a mode switch takes place.

Mode Switch – a mode switch describes the transition form one mode to another. A mode switch has a frequency, a target mode and a driver that will be invoked when the mode switch takes place. The guard of the driver is called the exit condition, because only if it is evaluated to true the mode switch takes place. The exit condition is evaluated periodically, as specified by mode switch frequency.

1.1.2 E-Machine

The *E Machine* is a virtual machine that mediates between the physical processes and the software processes of an embedded system through a control program written in *E Code*. *E Code* controls the execution of software processes in relation to physical events, such as clock ticks, and software events, such as task completion. *E Code* is interpreted on the *E Machine* in real time.

Every time an event (timer or completion) occurs, the *E Machine* observes it and can initiate the execution of *E Code*. *E Code*, in turn, supervises the execution of both tasks and drivers.

The *E Code* has the following instructions:

Call driver – the **call** instruction initiates the execution of a driver. This instruction has only one parameter, represented by the driver to be called. Since the driver is considered to be synchronous system-level code, it will be executed directly by the *E Machine* before interpreting the next instruction.

Release task – the **release** instruction hands a task to the operating system. It has only one parameter, representing the task to be released.

Future E-Code – the **future** instruction marks a block of *E Code* for execution at some future time. It has two parameters: the address of a block of *E Code* where to jump, and the interval of time after which the jump is performed.

Jump – the **jump** instruction represent an unconditioned jump to a specified address. It has only one parameter, the address of the block of *E Code* where to jump.

If – the **if** instruction represents a conditioned jump to a specified address. It has three parameters: the address of block of *E Code* where to jump, and the condition to be evaluated.

Return – the **return** instruction, stops the *E Machine* from interpreting *E Code* until an event happens.

1.1.3 Giotto pros and cons

Giotto is based on the Logical Execution Time (LET) model of tasks; in this model a logical termination time is specified at the time of release of a task. The outputs of a task are only available at the termination time even if the execution is complete before the termination instance. LET sacrifices end-to-end delays in the execution of a set of tasks, and in return, secures three key properties for the language. The first property is *time and value determinism*. Time determinism means that sensors are read and actuators are written at predetermined points in time. Value determinism means that given a sequence of sensor readings, the corresponding sequence of actuator writings is uniquely determined by the program; it does not depend on the scheduling of the tasks. This implies in particular, the absence of race conditions and priority inversion problems. The second property of *Giotto* is *switchability*, the ability to switch modes in a nontrivial fashion, possibly preempting the LET of tasks, while maintaining determinism. Third, *Giotto* allows for a simple check of *schedulability*, that all released tasks are able to complete execution before termination.

However, control applications from the automotive industry have also uncovered several severe shortcomings of *Giotto*. The first limitation of *Giotto* is that both the release and termination times of tasks are defined implicitly through the task period: every task is released

at the beginning of its period, and terminated at the end of the period. This is problematic in particular with short but infrequent tasks. The second limitation of *Giotto* is that all tasks are LET tasks, and no accommodation is provided for classical real-time tasks with precedence constraints specified by I/O dependencies.

1.2 TSL

In order to relax the limitations of *Giotto*, presented in the previous section, while still maintaining the three key properties of determinism, switchability, and schedulability. A new language called Timing Specification Language (*TSL*) was created. *TSL* is an extension of *Giotto* [2], a language targeted towards hard real-time applications with multi-modal time-periodic behavior. In order to do this, the LET tasks of *Giotto* were generalized and combined with tasks that have precedence constraints.

A generalized LET task is called an *anchored task*; its scheduling constraints are specified by an arbitrary release time and an arbitrary termination time. The task inputs are read at the release time and the task outputs are written at the termination time; the task can be executed between these two times in any way, but no interaction with other tasks may happen between these two times. The release time is specified as an *offset* relative to the period of the task. In addition to anchored tasks, *TSL* has *floating tasks*. The scheduling constraints of a floating task are specified only through dependencies: task inputs may depend on sensor readings and the outputs of other (anchored and floating) tasks; task outputs may be depended on by drivers writing the inputs of other tasks and actuators. Each such dependency introduces a precedence constraint on the scheduler; as long as these constraints are met, floating tasks can be executed at any time. To ensure determinism, all sensor readings and actuator writings are anchored in time; they have fixed periods and offsets. To ensure the efficient schedulability of mode switches, from one mode to the next, some (anchored and floating) tasks may be removed or some tasks may be added, but not both.

Figure 1.1 illustrates the extension of *Giotto* to *TSL*. In *Giotto* each mode consists of LET tasks with a specified frequency; the mode period and the frequency determines the LET (which is equal to the period) of the task. The task is logically released at the start of its period and logically terminated at the end of the corresponding period (Figure 1.1a). In *TSL*,

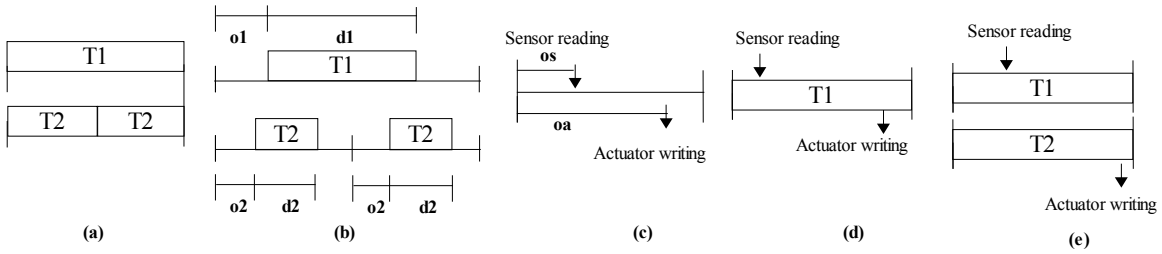


Figure 1.1: The new properties of the extended language

an invocation of an anchored task will be specified by three parameters: frequency, offset, and duration. The frequency denotes the number of times the task is invoked within a mode period. The offset denotes the time when the task is logically released after the start of the period, and the duration denotes the LET of the task. The sum of offset and duration is less than the period of invocation. Figure 1.1b shows an example of two tasks *T1* and *T2* with corresponding offsets (*o1* and *o2*) and durations (*d1* and *d2*). The *Giotto* style of task invocation is the special case with offset being equal to zero and duration being equal to the period of the invocation. In *TSL* the sensor and actuator updates of *Giotto* are also extended to have offsets. The duration is not relevant here as sensor and actuator updates use logical zero time to execute. Figure 1.1c shows an instance of a sensor and an actuator update. The invocation periods are the same; the sensor is updated at an offset *os* while the actuator is updated at an offset *oa*.

The second type of *TSL* tasks are the floating tasks. The execution window of a floating task is specified by the frequency of invocation of the task and by dependencies between the task and sensor reads, actuator writes, and other tasks. Figure 1.1d shows an example of a floating task *T1*; the offset of the sensor *s* it reads and the offset of the actuator *a* it writes determines the actual span of time for execution. Figure 1.1e shows an instance where the floating task *T2* reads the output of *T1*. The task *T1* reads from sensor *s* and writes to *T2*, which writes to actuator *a*. The time duration between the sensor read and actuator update is the time available for executing the tasks (time constraints) and the fact that *T2* reads the output of *T1* forces the execution of *T1* to precede that of *T2* (order constraint). Sensor and actuator updates cannot be floating.

The anchored tasks read input ports and update output ports at the specified LET boundaries. A floating task communicates in as-soon-as-possible fashion; the task reads the

input ports as soon as the source ports of the drivers communicating to the input port are updated in the period of invocation; the task updates the output ports as soon as the execution is completed. If the inputs of a floating task depends on several anchored tasks, floating tasks, and sensor updates, then the release time of the floating task is the earliest time when all anchored tasks have terminated, all floating tasks have completed execution, and all sensor updates have happened. If the outputs of a floating task are depended on by several anchored tasks and actuator updates, then the deadline of the floating task is the earliest of the release times of the anchored tasks and actuator updates.

1.2.1 TSL Syntax

I will discuss the main features of the language on the basis of a control application. In order to do this I have implemented a *TSL* program that controls a plant, which is simulated by a *Java program*. The process consist of three interconnected tanks and the main target for the controller is to keep the level of the fluid in two of the tanks constant by commanding two pumps, each pump is connected to one of the two tanks. The plant is detailed in **Section 4.1**. The plant I chose is interesting, because it is nonlinear and one can not obtain good control results in all possible scenario without using more then one regulator, which in terms of *TSL* means mode switch (which is an important feature of *TSL* I want to illustrate). There can be distinguished two different situations for the control problem:(1)a tank dose not lose any fluid and (2)a tank loses fluid. In the first case it is needed a P Controller, while in the second case a PI Controller. Since there are two pumps each connected to a tank, thus results that are needed four modes in order to cover all possible combinations. The program consists of four modes of operation, one for each possible combination as described before. Each of the four modes contains two tasks (each computing the command for one of two pumps), from one mode the program can switch to any of other three modes depending on what taps are opened. The full *TSL* program can be found in **Appendix A**.

Port. A port is a program variable associated with a type and carries a value consistent with the type. Ports are classified as sensor ports (e.g. port *sen_h1*), actuator ports (e.g. port *act_pumps*), output ports (e.g. port *out_u1*), input ports, and private ports.

Task. A task communicates through an interface consisting of a set of input ports and a set

of output ports. For example, task *rg1_P* computes the command for the first pump, it has one input port and one output port. A set of private ports is used to store the internal state of a task. A task computes a function, implemented by a sequential block of code, from its input and private ports to its output and private ports. At its release, the task reads the latest values of the input ports, and, at its termination, the task updates the output ports.

Driver. Drivers transport values from ports to ports. A driver connects sensor ports and/or output ports of tasks (source ports) to input ports of tasks (destination ports), or output ports (source ports) to actuator ports (destination ports). For example, driver *update_rg1* connects the sensor ports *sen_h1* to the input port of task *rg1_P*. The communication scheme is as follows: a driver consists of a function and a guard; if the guard is evaluated to *true* the function is executed. While tasks require logically non-zero time for execution and can be preempted, drivers always execute in logical zero time and are atomic.

Modes. A mode consists of periodic task invocations, sensor and actuator updates, and has a mode period (e.g., mode *P_P* has a period of 500ms) and communicates with other modes through mode ports. The mode ports are a subset of the output ports of the tasks invoked in the mode. A sensor update specifies the sensor port being updated, a frequency, and an offset. For example, in the mode *P_P* the sensor port *sen_h1* is updated every 500ms with an offset of 100ms. An actuator update specifies the actuator port being updated, a frequency, a driver, and an offset. The driver connects a subset of the mode ports to the actuator port and updates the actuator port at the specified offset from the start of the period. For example, in the mode *P_P* the actuator port *act_pumps* is updated with frequency 1 and offset 500ms. To avoid races no two sensor updates should write to the same sensor port and no two actuator updates should write to the same actuator port.

There are two ways of task invocations: anchored and floating. An anchored task invocation is specified with a frequency, task name, offset, duration, and input driver. The task is (logically) released at the specified offset from the period and the task is (logically) terminated at the sum of offset and duration from the start of the period; in other words, the offset denotes the release time whereas the duration denotes the task's LET. The sum of the offset and the duration is less or equal to the period of the invocation. The input driver connects sensor ports and mode ports to input ports of the task. A floating task invocation is specified by a frequency, a task name, and an input driver; the task *rg1_P* is invoked in floating style. The precedence

relation for a floating task will be discussed later. In *TSL* mode, a task can either be invoked in anchored style or in floating style but not both. To avoid races no two task invocations (anchored and/or floating) can update the same output ports. *TSL* also distinguishes between *Giotto* modes (all anchored style invocation has zero offset and duration equal to the period of invocation and no floating style task invocation) and non-*Giotto* modes (any other mode).

Mode Switches. A mode switch is specified by a frequency, a destination mode, and a driver. At specified intervals the associated driver guard is evaluated. If the evaluation returned true control switches to the target mode. Mode *P_P* switches to mode *P_PI* and the mode switch is checked with frequency one.

Precedences The execution of a floating task is dependent on the frequency of invocation (the execution must be completed within the period of invocation) and the relation with invocation/ termination of LET tasks, release/ completion of other floating tasks and sensor/ actuator updates. The precedence is imposed by the input drivers of the floating task invocations and are of the following types:

- Input driver of floating task reads from a sensor port/ output port of an anchored task → the floating task must be released after sensor is updated/ the logical termination of the LET task. The constrain imposes a time restriction on when the floating task can be released.

- Input driver reads from the output port of another floating task → the second floating task must be released after the first floating task has completed execution. The constraint imposes a restriction on the execution order of the floating tasks.

- A driver for an actuator update or an input driver of an anchored task reads from the output port of a floating task → the execution of the floating task should be completed before the actuator is updated/ the LET task is logically released. The constraint imposes a time restriction on when the execution of a floating task should be complete.

In mode *P_P* the execution window for the task *rg1_P* is determined by the offset of the sensor *sen_h1* and the offset of actuator *act_pumps*.

TSL imposes constraints on program expressiveness; they are necessary to execute programs in an unambiguous way and to perform meaningful analysis. The four constraints are

described as follows.

Matching. If a floating task depends on a task invocation, and/or sensor update and/or actuator update then the frequency of invocations or updates should be identical. This constraint forces the i-th instance of the invocations to communicate with each other. In the mode *P_P* floating task *rg1_P* depends upon sensor update *sen_h1*, which has same frequency. The anchored tasks are crucial in this context as they can be used to communicate between tasks with different frequencies.

Acyclicity. Dependencies between floating tasks should be acyclic. In other words, a graph denoting the precedence relation between tasks should be a directed acyclic graph. This is essential for performing schedulability analysis.

Causality. The time constraints on a floating task should be such that the time at which an output is required should not precede that of reading an input. Consider two anchored tasks *t1* and *t2* and a floating task *t* such that *t* reads from *t1* and *t2* reads from *t*. The logical termination of *t1* should be strictly before the logical release of *t2*; this means that *t* gets non-zero logical time to execute. In general, if a floating task *t* (or a set of floating tasks) depends on an anchored task *ti* (or a sensor s update) and the output ports of *t* are being read by an anchored task *to* (or an actuator a update) then the time of termination of *ti* (or the time of updating s) should be strictly less than the time of invocation of *to* (or the time of updating a).

Switchability. The switchability criterion specifies whether a switch is possible between two modes m and m'.

Before presenting switchability criterion I have to define what a *Giotto* mode and a non-*Giotto* mode are:

Giotto mode – is a mode where all offsets are zero, durations for all tasks invoked in that mode are equal to period, and there are no floating tasks.

Non-Giotto mode – is a mode where there is at least one offset greater then zero, or at least one task duration less then period, or at least one floating task.

The switchability criterion is satisfied if one of the following holds true:

1. if m and m' are *Giotto* modes then all task invocations preempted by a mode switch should be present in the target mode with identical frequencies as in the source mode.

By preemption, I mean logical preemption, i.e., preemption anytime during period of invocation of the task.

2. if m and/or m' is a non-*Giotto* mode then either m' includes all task invocations (with identical offsets, durations, period of invocations and precedences) of m OR m includes all task invocations (with identical offsets, durations, period of invocations and precedences) of m'. This implies that there cannot exist two tasks t and t' such that t has been invoked in m and not in m', and t' has been invoked in m' and not in m. Any task invocation preempted by mode switch cannot be removed.

3. if the mode switch dose not preempt any task then the switch is always valid no matter what are the task in source mode and what are the tasks in destination mode.

In the example, from any mode can switch to other mode as no mode switch preempts a task.

1.2.2 TSL Compiler

In this section I will present the compiler for *TSL* programs. For a given input program, the compiler checks well-formedness and schedulability and generates so-called *E code* for the *(E)mbedded Machine* [3]. E code is virtual machine code that specifies the exact times when drivers are called and when tasks are released and terminated. E code does not specify when released tasks actually execute. This is done by an EDF scheduler. E code consists of the following instructions: a $call(d)$ instruction executes the driver d, a $release(t)$ instruction releases the task t for execution by the EDF scheduler, a $future(g, a)$ instruction marks the E code at the address a for future execution when the predicate g evaluates to true, i.e., when g is *enabled*. g is called a *trigger*, which observes events such as time ticks and the completion of tasks. The E machine maintains a FIFO queue of trigger-address pairs. If multiple triggers in the queue are enabled at the same instant, the corresponding E code is executed in FIFO order, i.e., in the order in which the $future$ instructions were executed. An $if(c, a)$ instruction branches to the E code at the address a if the predicate c evaluates to true. We call c a *condition*, which observes port states such as sensor readings and task outputs. A $jump(a)$ instruction is an absolute jump to the address a and a $return$ instruction completes the execution of E code. The existing E ma-

chine implementation, which is written in C and uses POSIX threads, was extended to handle completion events in addition to time ticks.

The compiler divides each mode into uniform temporal segments called unit. An *unit* is defined as the smallest time interval at which any of the following happens: releasing an anchored task, terminating an anchored task, updating a sensor, updating an actuator or switching modes. For a mode m the span of a unit is denoted by $\gamma[m]$ and the total number of units by $\eta[m]$ the relation being $\eta[m] = \pi[m]/\gamma[m]$ where $\pi[m]$ is the period of the mode. For the *TSL* program discussed earlier the unit size is 100 ms and there are 5 units (the mode period being 500 ms). The E-code generated by *TSL* compiler is shown in **Appendix A**. The program compiler starts by emitting calls to initialize all output and private ports. For an output port p there are two drivers: *init*(p) being the driver to initialize the port and *copy*(p) being the driver to copy local output ports to global output ports. For an anchored task there are two sets of output ports, local and global. At termination local ports are copied to the global ports. For floating tasks there is only one set of output ports. Once the ports are initialized a *jump* instruction is emitted to transfer control to the beginning of the E code block corresponding to the start of the program which is the unit 0 of the *P_P* mode. In the example, *P_P* being the starting mode, a jump to $mode\_address[P\_P, 0]$ is emitted; $mode\_address[.,.]$ is a symbolic address which is linked up in the actual E code.

E code for an unit u of a mode m is generated in four stages. In the first stage an E code block is generated (starting with the address $mode\_address[m, u]$) to update task output ports (of anchored tasks that do not precede a floating task), to update actuator ports, to update sensor ports (that are used by mode switches) and to check mode switches that are possible at the corresponding unit. The E code block at $mode\_address[P\_P, 0]$ calls the drivers for the actuators (the function *driver(.)* implements the driver functionalities in the implementation language), updates the actuator ports (the ports are accesses by calling a function *dev(.)*), updates sensor port *sen_e1* for mode switch, and checks for mode switch. If mode switch condition is evaluated to *true* then the control jumps to take necessary action for mode switch (jump to $switch\_address[P\_P, 0, P\_PI, 0, P\_P\_to\_P\_PI]$); otherwise it jumps to the block which releases tasks for the unit (jump to *task_address*[P_P, 0] where *task_address* is a symbolic address).

In the second stage a block of code is emitted to take necessary action if a mode switch is enabled (the block starting with the address $switch\_address[P\_P, 0, P\_PI, P\_P\_to\_P\_PI]$).

The compiler performs a computation of the destination unit u' for the target mode m' and the time to wait δ' before jumping to the target unit. The target unit is computed as close as possible to the end of the destination mode period. If no tasks are preempted the control can jump to the starting unit of target mode without waiting ($u' = 0, \delta' = 0$). If the duration of either u or u' is a multiple of the other then there is no wait time ($\delta' = 0$) else time to wait is computed. The computation is similar to the one presented for the *Giotto* compiler in [4]. If $\delta' = 0$ then a ***jump*** instruction transfers control to the task invocation block of u' starting at the address $task\_address[m', u']$. Note this bypasses the mode switch check at target unit and thus removes the possibility of multiple mode switches at the same logical instant. If $\delta' > 0$, a trigger is emitted to wait for the time instant and then to jump to the required task invocation address. In my example, at mode switch no task is preempted and hence the control jumps to $task\_address[P\_PI, 0]$.

In the third stage code is emitted for releasing tasks invoked at the unit (e.g. code at $task\_address[P\_P, 0]$). First, the sensors updated at u but not read by floating tasks are updated. As one can easily notice the update driver for a sensor can called twice for the same unit, but this is no problem because the drivers are state-full and if a driver is called twice in the same unit it will be actually executed only the first time. Second, the anchored tasks logically starting at the unit are released. The floating tasks with period of invocation coinciding with the unit and with no precedences are also released. Third, the floating tasks with period of invocation coinciding with the unit and with precedence constraints are released. This is done by defining three triggers: 1, call to block of codes that update sensor ports read by the floating tasks, 2, call to block of codes that update output ports (of anchored tasks) that are read by the floating tasks and 3, call to block of codes that release the floating tasks. Note this is required to have ports being read by input drivers of the floating tasks be updated before the invocation of the task. The complexity arises from the fact that the exact instance of floating task release cannot be determined apriori an WCET analysis and the compiler generates code independent of the WCETs of the tasks. An example of the first scenario is updating the sensor *sen_h1* (in mode *P_P*) which precedes a floating task invocation; the update procedure is called by a trigger from code block at $task\_address[P\_P, 0]$ and is updated at the address $sensor\_update\_address[P\_P, 0, \textit{sen\_h1}]$. Note the event for the trigger in the above case are known apriori as time instances for sensor updates and anchored task terminations can be computed without WCET information. Note the

event in the trigger may not be determined here as it consists of completion events of floating tasks which are not known apriori and the trigger event is represented as a set of time triggers and completion events. The last stage of the compiler generates code for the blocks referred to by the above three trigger instructions; the codes are straightforward and updates sensor ports, call copy functions for output ports and release floating tasks.

1.3 SableCC

SableCC[1] is an object-oriented framework that generates compilers (and interpreters) in the Java programming language. This framework is based on two fundamental design decisions. Firstly, the framework uses object oriented techniques to automatically build a strictly-typed abstract syntax tree (AST) that matches the grammar of the compiled language and simplifies debugging. Secondly, the framework generates tree-walker classes using an extended version of the visitor design pattern which enables the implementation of actions on the nodes of the abstract syntax tree using inheritance.
The steps to build a compiler using *SableCC* are:

1. creating a *SableCC* specification file containing the lexical definitions and the grammar of the language to be compiled;
2. launching *SableCC* on the specification file to generate a framework;
3. creating one or more working classes, possibly inheriting from classes generated by *SableCC*, working classes are classes that contain the core compiler functionality;
4. creating a main compiler class that activates lexer, parser, and woking classes;
5. compiling the compiler with a Java compiler.

After launching *SableCC* on the specification file, there will be generated four packages:

- the **lexer** package contains the Lexer and LaxerException classes. These classes are the generated lexer and the exception thrown in case of a lexing error, respectively;
- the **parser** package contains the Parser and ParserException classes. These classes are the generated parser and the exception thrown in case of a parsing error, respectively;

- the **node** package contains all the classes defining the typed AST;
- the **analysis** package contains one interface and three classes. These classes are used mainly to define AST walkers.

The packages that are important for understanding *TSL Compiler* implementaion are **node** and **analysis**. In package **node** for each production rule defined in the specifications file there will be a class named after the production, prefixed with 'P', replacing the first latter with an uppercase, replacing each latter preceded by an underscore with an uppercase, and removing the underscores. If the production has a single unnamed alternative, the alternative class is named like its production class, but the uppercase 'P' prefix is replaced by an uppercase 'A'. When there are more then one alternative, *SableCC* requires a name for each alternative. A name is given to an alternative in the grammar by prefixing the alternative with an identifier between curly brackets. The class name is created as described before the only difference is that the identifier will be used instead of production name. For elements there will be a variable that will be access through getxxx and setxxx methods.

The most important class in **analysis** is the abstract class DepthFirstAdaptor which should be implemented by any depth first walker of the AST. For each alternative there are two methods in the class one (*inAxxx(Axxx node)*) that is called when a node of corresponding type (Axxx) in the AST is reached, and the second (*outAxxx(Axxx node)*) when the node is left. In both cases the parameter represents the node and can be used to get information about the node.

In order to make things clear I will present in the end of this section an example inspired from the *SableCC* specification file for *TSL Compiler*.
I will consider the following production rule:

```
driver_declaration = driver [driver_name]:ident
                     [source_ports]:actual_ports output
                     [destination_ports]:formal_ports
                     l_brace call_driver r_brace ;
```

This is what *SableCC* will generate:

- a class called *PDriverDeclaration* and a class called *ADriverDeclaration*;

- there will be a member of type *TIdent* (this calss was generated for *ident* token) in the above classes, called driverName, and that could be access via method *getDriverName*, such a member will be generated for each element;

- in the class *DepthFirstAdaptor* there will be defined to methods corresponding to this production rule: *public void inADriverDeclaration(ADriverDeclaration node)*, and *public void outADriverDeclaration(ADriverDeclaration node)*.

Chapter 2

Control Engineering Theoretical Support

In this chapter I will present the control engineering theoretical background for this thesis.

2.1 Quality Indicators

In this subsection I will refer only to quality indicators defined using the controlled system response to step signal input and only for dynamic regime. In **Fig. 2.1** are represented the dynamic regime quality indicators. The indicators are:

t_c – control time; represents the time interval form the beginning of the transition regime (t_0), until the answer enters the so called *silent region*, which is defined as $\pm 0.2 \cdot \Delta z_\infty$;

t_1 – first control time; the time interval form the beginning of the transition regime t_0, until the desired value (z_∞) if being reached for the first time;

t_m – time when the maximum value (z_{max}) is reached;

t_r – raising time, defined by the relation: e

$$t_r = t_{0.95} - t_{0.05} \tag{2.1}$$

where $t_{0.05}$ and $t_{0.95}$ represent time moments for which $\Delta z(t)$ reaches 0.05 respectively 0.95 from Δz_∞;

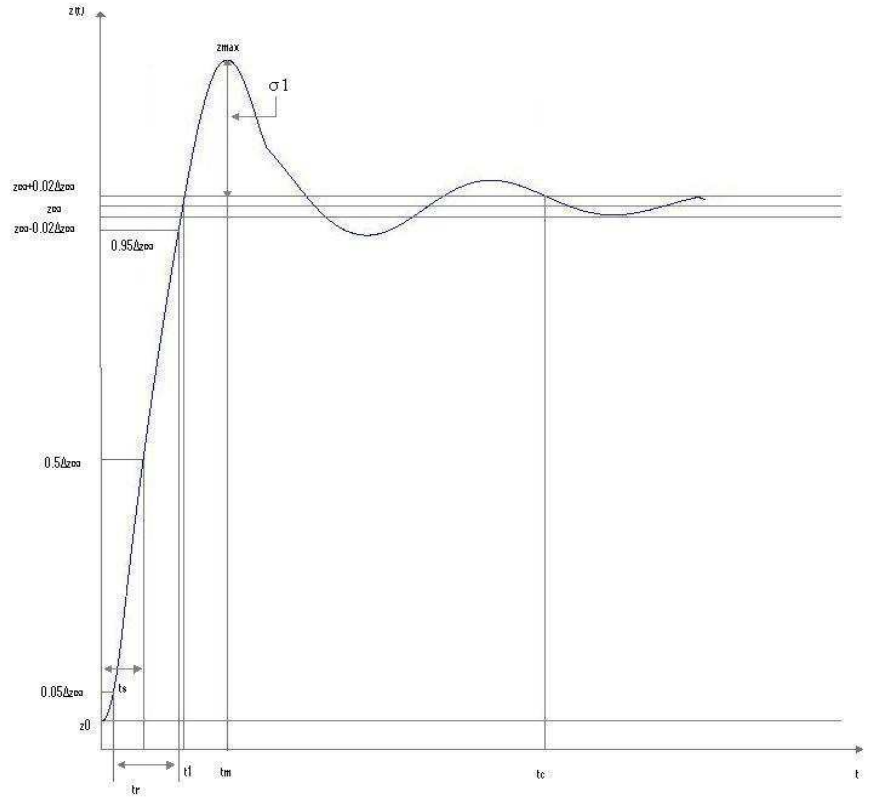


Figure 2.1: Quality Indicators

σ_1 – over-control is defined by the relation:

$$\sigma_1 = \frac{z_{max} - z_\infty}{\Delta z_\infty}, with \Delta z_\infty = z_\infty - z_0, \ or \ in \ percents \ \sigma_1^{\%} = \sigma_1 \cdot 100 \tag{2.2}$$

For more information about quality indicators you should look at [7].

2.2 P Controller

In this section I will present how to design a P Controller for an I plant. In **Fig. 2.2** it is presented the block schema for an I plant controlled with a P Controller.

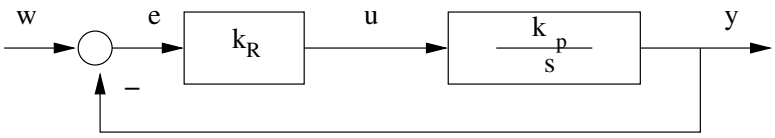


Figure 2.2: P Controller

P Controller transfer function is:

$$H_R(s) = k_R \tag{2.3}$$

I plant transfer function is:

$$H_P(s) = k_P \frac{1}{s} \tag{2.4}$$

Transfer function for controled system is:

$$H_c(s) = \frac{1}{\frac{1}{k_P k_R} s + 1} \tag{2.5}$$

From last equation results that controled system time constant is:

$$T = \frac{1}{k_P k_R} \tag{2.6}$$

Since control time $t_c = 3T$ results that choosing t_c, the controller parameter k_R will be:

$$k_R = \frac{3}{k_P t_c} \tag{2.7}$$

2.3 Frequency Domain Controller Design

Frequency domain controller design in the simplest variant it is based on the frequency logarithmic Bode characteristics, amplitude-frequency and faze-frequency.

The big time constants of the plant are compensated by the controller (for instance if the controller is PI):

$$H_{RG}(s) = \frac{k_r}{s}(1 + T_{dom} \cdot s) \tag{2.8a}$$

$$H_p(s) = \frac{k_p(.....)}{(1 + T_{dom \cdot s})(.....)} \tag{2.8b}$$

Then you have to choose the faze reserve (φ_R) in $[45^o, 60^o]$ where the system is for sure stable and the transition regimes will be acceptable.

Knowing that by definition the faze reserve is:

$$\varphi_R = 180^o + \angle H_0(j\omega_t) \tag{2.9}$$

and using the faze-frequency characteristic (the same for any value of k_r), the cut frequency ω_t will be got.

Then the amplitude-frequency characteristic will be plotted for an initial value of k_r, noted k_r^0. Then for ω_t, from amplitude-frequency characteristic the signed value $AB = |H_0(j\omega_t)|$ will be read and will be used to compute the transfer coefficient (k_r) for the controller:

$$k_r = k_r^0 \cdot 10^{-\frac{AB}{20}} \tag{2.10}$$

Chapter 3

TSL Compiler Implementation

TSL Compiler is made up of four parts:

- the parser which is automatically generated by *SableCC*;
- the symbol table;
- the checker;
- the code generator;

In the next sections I will present only the last three parts, I will not describe the parser because this was already covered in a previous chapter when I had descried *SableCC*.

3.1 Symbol Table

Symbol Table is implemented in the *SymbolTable* class whose declaration is presented in **Code 3.1.1**. For each type of *TSL* declaration there is defined a *public Map* member that will keep

Code 3.1.1 SymbolTable declaration

```
public class SymbolTable extends DepthFirstAdapter
```

track of all defined elements (e.g., *public final Map tasks*, which is being used to keep track of all defined tasks). The key in each such a *Map* member is represented by each defined element name and the object stored is actually the node that represents the declaration. When an instance of this class is fed to parse tree, which was obtained as the result of parsing the program, then

all declarations are being processed one by one, and added to the corresponding *Map*, if two elements that have the same name were declared, an error is reported and the compiler stops. In **Code 3.1.2** it is presented the method that is called before leaving *ATaskDeclaration* node, for other elements it is the same. Everything I have presented so far was already implemented

Code 3.1.2 outATaskDeclaration method

```
public void outATaskDeclaration(ATaskDeclaration node) {
   final String name = node.getTaskName().getText();
   if (declarations.put(name, node) != null) {
      errorRedefined(node.getTaskName(), name);
   }
   if (tasks.put(name, node) != null) {
      errorRedefined(node.getTaskName(), name);
   }
}
```

in classic *Giotto Compiler* what I had to add to the *SymbolTable* class, were two new *public Map* members that are used to keep track of all anchored and float task invocations. The two members are:

- *modesLetTasks*, for each mode there is an element in this Map with the mode name as key and the stored object is another Map that contains all the anchored task invocations in that mode;
- *modesFloatTasks*, the same as *modesLetTasks*, the only difference is that it refers to float task invocations.

DependencyTable (**Code 3.1.3**) it is new introduced in *TSL Compiler* and it is very much alike with *SymbolTable*, the only reason why I haven't combine this two in one single table is that before I build the *DependencyTable* I have to do type checking, which means that I am not able to combine the two tables in a single one. As the name suggests, this table contains

Code 3.1.3 DependencyTable declaration

```
public class DependencyTable extends DepthFirstAdapter{
   /*...*/
   public DependencyTable(SymbolTable pSymbolTable) {
      symbolTable = pSymbolTable;
   }
   /*...*/
}
```

information about the dependency between tasks invocations in a mode. There are two *public Map* members:

- *modesInputDependencies* – for each mode there is an element, which has the mode name as key and the object stored is a *Map* that for each element in the mode keeps an *ArrayList* with all elements on which it depends;

- *modesOutputDependencies* – for each mode there is an element which has the mode name as key and the object stored is a *Map* that for each element in the mode keeps an *ArrayList* with all elements that depends on it.

In order to create the two dependency tables, first for each mode there are created others two dependency maps, that later will be used to find out for what elements a port is an input and for what element a port is an output. Each map maintains an *ArrayList* for each port . In order to compute this two tables, an internal class was defined that will be applied on *AModeDeclaration* instance at the beginning of mode declaration and for each element in the mode declaration the input and the output ports lists are iterated, if they exists, and the node is being put in one of the two maps, depending on what is the relation between the node and the port. In **Code 3.1.4** are presented two methods, one that processes *ASensorUpdate* node and the other one *AActuatorUpdate* node. Then after this tables were computed, for each anchored or float task invocation in the mode, based on its input and output ports and using the two auxiliary maps that were presented before, a list of input and output dependencies is created.

For each mode there is computed a list of task output ports that are outputs for tasks that are being invoked as float tasks. This list is needed because a task can be invoked as an anchored or as a float task in different modes and the output port is doubled buffered if the task is invoked as an anchored task, and it is directly accessed if it is invoked as a float task.

3.2 Checker

Actually this section should be cold “Checkers”, because there are more then one checker. The checkers are:

- *TypeChecker* – performs type checking;

- *DependencyChecker* – checks the program against closed loops;

- *FrequencyChecker* – checks the frequencies, offsets and everything related to time;

Code 3.1.4 Ports dependency computation.

```
public void outASensorUpdate(ASensorUpdate n){
   putPortOutputFor(n.getSensorPortName().getText(), node);
}
public void outAActuatorUpdate(AActuatorUpdate n){
   ADriverDeclaration driver = (ADriverDeclaration) symbolTable.
         drivers.get(n.getDriverName().getText());
   AActualPortList sourcePorts = (AActualPortList) ((AActualPorts)
         driver.getSourcePorts()).getActualPortList();
   LinkedList actualPorts = sourcePorts.getActualPort();
   if (actualPorts != null) {
      ListIterator sourceIterator = actualPorts.listIterator();
      while (sourceIterator.hasNext()) {
         AActualPort port = (AActualPort) sourceIterator.next();
         putPortInputFor(port.getPortName().getText(), n);
      }
   }
}
```

- *ModeSwitchChecker* – checks mode switch condition;
- *TimeSafetyChecker* – checks time safety for each mode;

Next I will present the implementation for each of the checkers.

3.2.1 Type Checker

This checker performs type safety checks on all the elements of a mode. In **Code 3.2.1** is presented the *TypeChecker* class declaration. What this checker dose is:

Code 3.2.1 TypeChecker declaration

```
public class TypeChecker extends DepthFirstAdapter{
   /*...*/
   public TypeChecker(SymbolTable symbolTable,boolean dynamicGiotto){
      this.symbolTable = symbolTable;
      this.dynamicGiotto = dynamicGiotto;
   }
   /*...*/
}
```

- for each sensor update in a mode it checks to see if the sensor port that is updated exists and that there are no to sensor updates in the same mode that refer to the same sensor;
- for each actuator update in a mode it checks to see if the actuator port and the driver being used exist, that there is only one actuator update referring to the same actuator port in a mode, and that the type of the actuator port and the type of the formal port of the driver being used to update the actuator port are the same, and that the driver has only one formal port;

- for each mode switch, first checks for existence of destination mode, then checks that only one mode switch with this destination mode is present in the mode, then checks that the driver being used by the mode switch exists, and finally checks that the number and type of formal ports of mode switch driver are the same with the number and type of destination mode ports;

- for each anchored and float task invocation checks that the task exists, that there are no two task invocations that refers to the same task in the same mode, then checks for the existence of driver used to update the task input ports and finally checks that the number and type of task input port are the same with the number and type of driver formal ports;

- also it checks that there are no two tasks invocations in the same mode that update the same output port

3.2.2 Dependency Checker

This checker, checks the program against closed loops between float task invocations. The algorithm that performs the check is implemented in a recursive method (**Code 3.2.2**) that has two parameters, the first one is the current task name that was reached following the chain of dependency lists and the second is the name of the task on which the function was applied first. Using the *DependencyTable* for the current node the list of node that depends on the current node is obtained, and each node in this list is tested against the start node, and if they match then an error is reported and the compiler stops, else if the node is an anchored or float task invocation, then the method is called again with the start node set to this one.

3.2.3 Frequency Checker

The checks performed by this checker are:

- for each element in a mode, the frequency is checked, there are two types of checks that are performed on the frequency: (1) the frequency can not be zero and (2) it must by among the dividers of the mode period; in **Code 3.2.3** is presented the method that performs this checks

Code 3.2.2 dependencyCheck method

```
private void dependencyCheck(TIdent crrNode,TIdent stNode){
   ArrayList taskDependencies=(ArrayList)modeDependencies.get(
                           crrNode.getText());
   Iterator depIt=taskDependencies.iterator();
   while(depIt.hasNext()){
      Node n=(Node)depIt.next();
      if(n instanceof AFloatTaskInvocation){
         AFloatTaskInvocation flNode=(AFloatTaskInvocation)n;
         if(flNode.getTaskName().getText().compareTo(stNode.getText())==0){
            dependencyError(crrNode,stNode);
         }
         final String flNameTmp=flNode.getTaskName().getText();
         if(floatDep.contains(flNameTmp)){
            dependencyError(flNode.getTaskName(), stNode);
         }
         floatDep.add(floatNameTmp);
         dependencyCheck(flNode.getTaskName(),stNode);
      }
      if(n instanceof ALetTaskInvocation){
         ALetTaskInvocation letNode=(ALetTaskInvocation)n;
         if(letNode.getTaskName().getText().compareTo(stNode.getText())==0){
            dependencyError(crrNode,stNode);
         }
         final String letNameTmp=letNode.getTaskName().getText();
         if(letDep.contains(letNameTmp)){
            dependencyError(letNode.getTaskName(), stNode);
         }
         letDep.add(letNameTmp);
         dependencyCheck(letNode.getTaskName(),stNode);
      }
   }
}
```

Code 3.2.3 checkFrequency method

```
private void checkFrequency(Token frequencyToken) {
   final int modePeriod = Integer.parseInt(modePeriodToken.getText());
   final int frequency = Integer.parseInt(frequencyToken.getText());
   if (frequency == 0)
      errorZero(frequencyToken);
   if ((modePeriod % frequency) != 0)
      errorFrequency(frequencyToken, modePeriodToken.getText());
 }
```

- for each sensor update and actuator update the offset is checked so that it is not greater then period (**Code 3.2.4**), a similar check is performed for anchored tasks also, but in this case you have to consider the duration also;

Code 3.2.4 checkOffset method

```
private void checkOffset(TIdent node,TNumber pOffset,
              TNumber pFrequency){
   boolean checkOk = true;
   if (pOffset != null) {
      int period = modePeriod / Integer.parseInt(pFrequency.getText());
      int offset = Integer.parseInt(pOffset.getText());
      checkOk = offset <= period;
      if (!checkOk) {
         erroOffset(node);
      }
   }
}
```

- for each mode switch there is performed a check so that if the mode switch depends on a sensor update then the sensor update is not allowed to have an offset
- for each float task invocation the list of dependencies is checked so that all the node the task depends on or which depends on the task have the same frequency (**Code 3.2.5**);

Code 3.2.5 checkDependencyFrequency method

```
private void checkDependencyFrequency(TIdent taskName,Token frequency){
        ...
        //check for input dependency
        //take each node that the task represented by taskName depends
        //on and test that the frequencies are the same
        //if they are not the same throw an error and stop
        ...
        //check for output dependency
        //take each node that depends on task represented by taskName
        //and test that the frequencies are the same
        //if they are not the same throw an error and stop
        ...
}
```

- for each anchored task invocation sensor update and actuator update a causality check is performed, in order to make sure that I don’t get zero time to execute a float task; the causality check works as follows: for each anchored task invocation and sensor update I take the dependency list and if I can reach an anchored task, or an actuator update just following the dependency list, and if in the list there is at least one float task invocation, then the causality condition as described in previous chapter are checked; in **Code 3.2.6** I present the methods that perform this checks.

Code 3.2.6 Causality check methods

```
//Search the dependencies until reach to a let task invocation or an
//actuator update then check causality conditiona
private void checkCausality(AFloatTaskInvocation floatTask,Node stNode){
   ...
   if (dependencies != null) {
      Iterator it = dependencies.iterator();
      while (it.hasNext()) {
         Node depNode = (Node) it.next();
         if (depNode instanceof AFloatTaskInvocation)
            checkCausality((AFloatTaskInvocation) depNode, stNode);
         else
            if (depNode instanceof ALetTaskInvocation)
               checkCausality((ALetTaskInvocation) depNode, stNode);
            else
                  if (depNode instanceof AActuatorUpdate)
                     checkCausality((AActuatorUpdate) depNode, stNode);
       }//end while
    }//end if
}
private void checkCausality(ALetTaskInvocation letTask,Node stNode){
    ...
    if (startNode instanceof ALetTaskInvocation) {
           // Let vs Let
           ...
           if (stLetDuration + stLetOffset >= letOffset) //error
    }
    if (startNode instanceof ASensorUpdate) {
           // sensor update vs Let
           ...
           if (stOffset >= letOffset) //error
    }
}
```

3.2.4 Mode Switch Checker

This checker, checks the condition that must hold in order to be able to switch from one mode to another without overloading. There are three possible cases:

- the mode switch does not preempt any task, then there is no condition to be checked;
- the two modes are classic *Giotto* modes, in this case the old *Giotto* mode switch conditions must be checked;
- the two modes are *TSL* modes, in this case the *TSL* mode switch conditions must be checked.

In **Code 3.2.7** is presented the method that is called when leaving the mode switch node. As it can be seen first the preemption test is performed by applying on the *crrMode* an instance of class *CheckPreemption*, which looks at each task invocation frequency in the mode, and if it does not divide by mode switch frequency then *preempts* will be set to *true*, otherwise it will remain *false*. After the preemption test was performed and if this test returned true, then

Code 3.2.7 outAModeSwitch method

```
public void outAModeSwitch(AModeSwitch node){
   ...
   crrMode.apply(checkPreemption);
   if(checkPreemption.preempts){
      // both modes are classic giotto modes
      checkClassicGiotto(destMode, modeSwitchFreq);
      if(!(destModeType && crrModeType)){
      // at least one mode is a giotto+ mode
         checkGiottoPlus(destMode, modeSwitchFreq);
      }
  }
}
```

the classic *Giotto* mode switch condition will be checked. The classic *Giotto* conditions test is implemented in the method **Code 3.2.8**. What this method does is: for each task invocation

Code 3.2.8 checkClassicGiotto method

```
private void checkClassicGiotto(AModeDeclaration destMode,
                                int modeSwitchFreq)
}
```

in the source mode tests to see if the task invoke frequency does divide by the mode switch frequency, if it does not divide tests to see if the task is invoked in the destination mode with the same period, if not an error is thrown an the compiler stops. If at least one of the two modes is a *TSL* mode, then the *TSL* mode switch conditions are tested. In order to find out if a mode is a classic *Giotto* mode or a *TSL* one, an instance of the class *ClassicGiottoMode* is apply on the mode, this class is very similar with *CheckPreemption* class, it has a public memeber that will be true if the mode is a classic *Giotto* mode and false otherwise. The *TSL* mode switch conditions are implemented by **Code 3.2.9**. First the list of anchored and float tasks of the destination mode are got from *SymbolTable*, then the inclusion test is performed (this would be the first *TSL* condition), if this test returns true, then the second test is performed by the method *checkSecondCondition*, the second condition ensure that the dependencies are preserved.

3.2.5 Time Safety Checker

This checker, checks the mode utilization based on mode period and the worse case execution time of each task that is invoked in the mode, if mode utilization is greater then 1 then an error is reported and the compiler stops. The mode utilization is computed as showed in **Code 3.2.10**. Where *wcet* is task worse case execution time and frequency is the task invocation frequency.

Code 3.2.9 checkGiottoPlus method

```
private void checkGiottoPlus(AModeDeclaration destMode,
          int modeSwitchFreq){
   final String destModeName = ...
   final int destPeriod = ...
   final int crrPeriod = ...
   final Map destLetTasks=...
   final Map destFloatTasks=...
   if(Utils.include(crrFloatTasks, destFloatTasks) &&
            Utils.include(crrLetTasks, destLetTasks)){
      checkSecondConditions(crrMode, destMode,
             crrMode.getModeName(), destMode.getModeName());
   }
   else if(Utils.include(destFloatTasks, crrFloatTasks) &&
            Utils.include(destLetTasks, crrLetTasks)){
       checkSecondConditions(destMode, crrMode,
            crrMode.getModeName(), destMode.getModeName());
   }
   else{
       // error
       inclusionError(...);
    }
}
```

Code 3.2.10 Mode utilization computation

```
modeUtilization = modeUtilization
      + ((double) wcet / (modePeriod / frequency));
```

3.3 Code Generator

The code generation is made up of two parts:

- *FTable* – this class is used to generate a C file “f_table.c”, and its corresponding header file “f_table.h”, this files will be compiled into the *E Machine*, the file will contain:
  - for each sensor declaration will be added a variable declaration, that will be associated with the sensor, and a function, that will be called when the sensor is updated;
  - for each actuator will be added again both a variable, and the function that will be called when actuator is updated;
  - for each output port there will be defined two variables, because a port is double buffered, the initialization function, and a copy function that will copy the information from the local variable into the global variable, if the task that writes to an output port is a float task then only the local variable is used;
  - for each task will be added, for each formal port a variable, and also the function that will be executed when the task is released;

  - for each driver there will be added two function one that will be the condition that guards the driver, and the second the function that will be called when the driver is to be called, for each mode in which the driver is used there will be added two new such functions, the reason for this is that a task can be invoked in a mode as a float task, and then in another as an anchored task;
  - a trigger table, usually this table will contain only one trigger, which is the timer;
  - task table, this will contain the list of all the tasks, and what is the function associated with each task;
  - driver table, this will contain the list of drivers, the function associated with each diver;
  - condition table, this will contain the list of condition and the function associated with each condition;
  - port table, this will contain the list of sensor, actuator and output ports and the address of the variable associated with each port;

- *ECode* – this will be used to generate the *E code* for the *Giotto* program.

Further I will not detailed any more the *FTable* class, because I made only a few modifications to it, instead I will detailed the *ECode* class, because I consider it to be more interesting.

Before I present how the *E code* is generated a few observations must be made regarding what I mean when I say that an element is enabled:

- sensor update – a sensor update is enabled if: (1) there is no float task that depends on it and $(|unit - (offset/unitPeriod)| * frequency)\ mod\ nUnit = 0$, or (2) there is a float task that depends on the sensor update then $(unit * frequency)\ mod\ nUnit = 0$
- actuator update – an actuator update is enabled if: $(|unit - (offset/unitPeriod)| * frequency)\ mod\ nUnit = 0$
- mode switch – a mode switch is enabled if $(unit * frequency)\ mod\ nUnit = 0$
- anchored tasks – an anchored task is enabled if $(|unit-(offset/unitPeriod)|*frequency)\ mod\ nUnit$ 0

- float task – a float task is enabled if $(unit * frequency)\ mod\ nUnit = 0$

The *E code* generation works as follows:

- first for each mode the small unit period is computed using an isntance of *ModeUnit* class (the unit period is defined in the previous chapter);
- the period of each mode is divided in a number of unit periods and for each such a unit period the following set of operations is performed:
  - for all anchored tasks on which depends no float task and that are finished at the current unit the copy driver is called;
  - for each actuator update, that is enabled at this unit, first the driver is called to update the port then the corresponding device function is called;
  - for each sensor update, which is enabled and on which there is a mode switch that depends on, the corresponding device function is called;
  - for each enabled mode switch, the mode switch condition check is generated, then on the true branch will be generated the the jump to the address where the mode switch is implemented, and on the false branch the jump to task address is generated;
  - mode switch address, first the mode switch driver is called, then the jump to the task address for desired unit from the destination mode, or a future instruction referring to the same address as the jump, depending on the preempted tasks;
  - task address, first for all enabled sensor updates on which there is an anchored task that depends on, but no dependent float task the device function is called, next for all anchored tasks that are enabled, the update driver is called, then the task is released, next for each enabled sensor update on which there is at least a float task that depends on, a future instruction is generated with the time set to the offset and the address where the sensor device function is called, for each anchored task on which there is a float task that depends on a future instruction is generated with the time set to offset+duration and the address where the copy function for the task output ports is called, for each enabled float task a future instruction is generated with the time set to the biggest offset on which the task depends on, the list of float tasks that this

task depends on (this list is encoded in a 32 bit integer, based on the fact that I can not have more then 32 tasks) and the address where the task is released;

– at the end a last future instruction is generated with the period set to the unit period and the address set to the next unit.

Chapter 4

Case Study

In order to test *TSL* in a control application I have implemented a simulator for the *Three Tanks System (3TS)*, the *3TS Simulator* is written in Java, I have also implemented a controller (3TS Controller), also written in Java, that can be used to control the *3TS* plant. The *3TS Controller*, could also be used as a viewer, for the evolution of the *3TS Simulator* signals. Also I have implemented a reduced version of *3TS Controller* in *TSL*. The communication between the controller (Java or *TSL*) and the simulator is done via sockets, I choose this type of communication because is totally platform independent, in order to do this a protocol (*Control Protocol*) was implemented. Next I will present the *3TS Plant*, then I will present *Control Protocol* and each application in different section.

4.1 Three Tanks System (3TS) Plant

In this section I will present the 3TS plant, I will build a mathematical model for the plant, and I will design the control structures and algorithms.

3TS system is made up of three identical cylindrical tanks (T_1, T_2, T_3), having the same transversal section A. The three tanks are interconnected through pipes having 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 $i \in \{S_1, S_2, g_2, e_1, e_2, e_3\}$.

4.1.1 Mathematical Modeling

In **Fig. 4.1** it is presented the block schema for 3TS plant. Further in this subsection I will present the mathematical model for 3TS system.

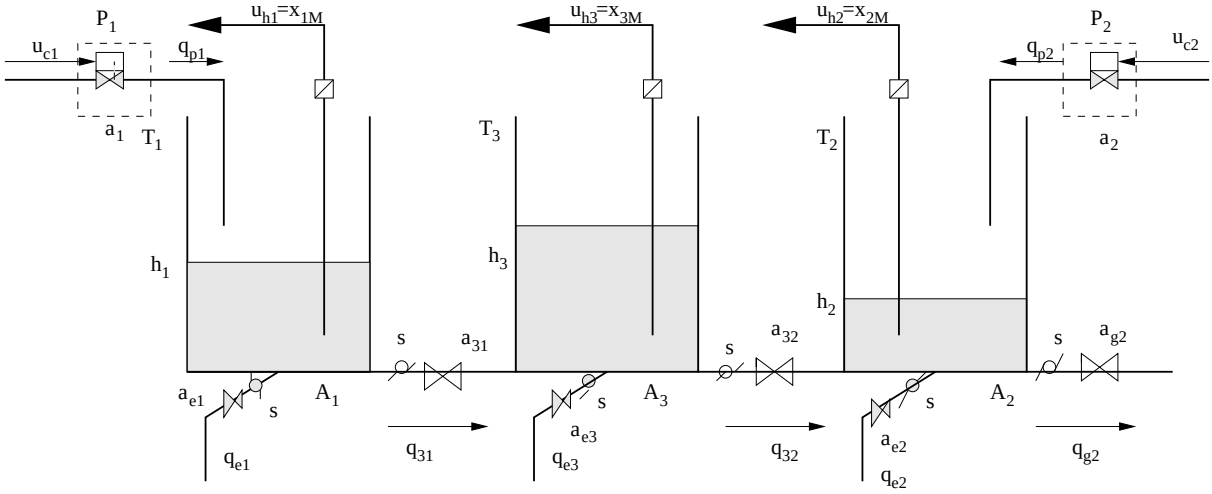


Figure 4.1: 3 Tanks System

The level of the fluid in the the tanks depends on:

- the filling flow capacities of T_1 and T_2 (q_{p1}, q_{p2});
- the draining flow capacities of the six taps:
  - $q_{e1}, q_{e2}, q_{e3}, q_{g2}$ - emptying flow capacities (these represent the perturbations);
  - q_{13}, q_{32} - the interchange 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 process 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.

$$p + \frac{\delta v^2}{2} + \delta g h = const. \tag{4.1}$$

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

$$v \approx \sqrt{2g\Delta h} \tag{4.2}$$

where Δh represents the fluid level deference between interconnected tanks.

First I will model the case when there is only one tank (1TS) then I will model the case when there are two tanks interconnected (2TS) and in the end I will present the mathematical model for 3TS plant.

4.1.1.1 1TS Plant

The plant is presented in **Fig. 4.2**

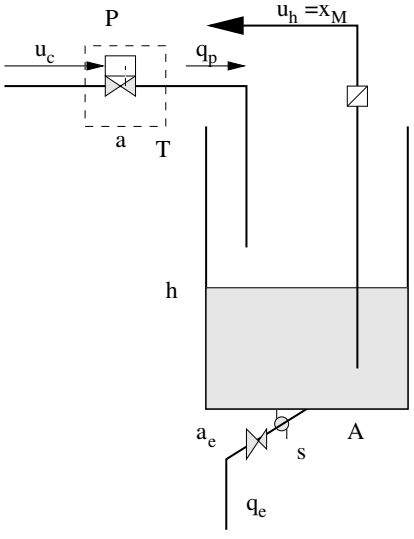


Figure 4.2: One Tank System

The evolution of the volume of the fluid in the tank is characterized by the following equation:

$$\dot{h}A = \sum [q_{in} - q_{out}] = q_{\sum} \tag{4.3}$$

where

- q_{in} represents the in flow capacity, and for for 1TS plant it is represented by pump capacity (q_P);

$$q_P = cu_c \tag{4.4}$$

- q_{out} represents the out flow capacity, and it is represented by the flow capacity of the fluid that drains through μ_e (q_e).

$$q_e = \mu_e S\sqrt{2gh} \tag{4.5}$$

Combining equations (**4.3**), (**4.4**), and (**4.5**) results:

$$\dot{h} = \frac{1}{A}(cu_c - \mu_e S\sqrt{2gh}) \tag{4.6}$$

Linearizing the equation (**4.6**) in the neighborhood of the fixed point (h_0, u_{c0}, and μ_{e0}) results:

$$\dot{\Delta h} = \frac{1}{A}(c\Delta u_c - \mu_{e0} S\sqrt{2g}\frac{1}{2\sqrt{h0}}\Delta h - S\sqrt{2gh_0}\Delta\mu_e) \tag{4.7}$$

In **Fig. 4.3** I present the results of simulating the found mathematical model for the 1TS plant in Matlab,the Simulink schema can be found in **Appendix E**.

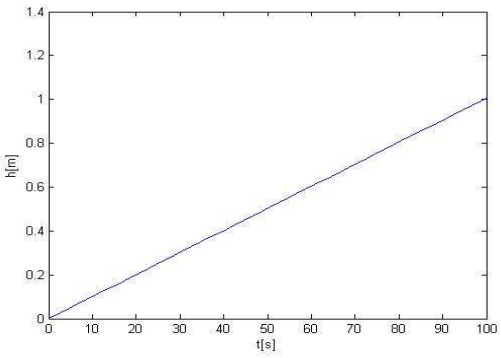
4.1.1.2 2TS Plant

The plant is presented in **Fig. 4.4**.

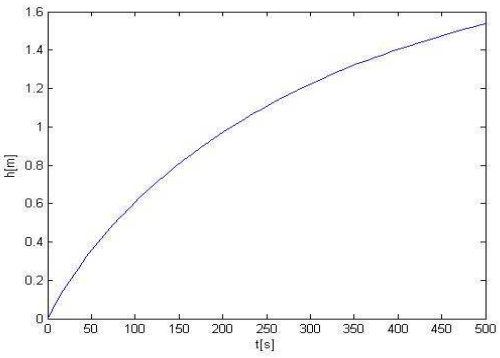
Writing the equation (**4.3**) for each of the two tanks I get:

$$\dot{h1} = \frac{1}{A}(q_{p1} - q_{13} - q_{e1}) \tag{4.8}$$

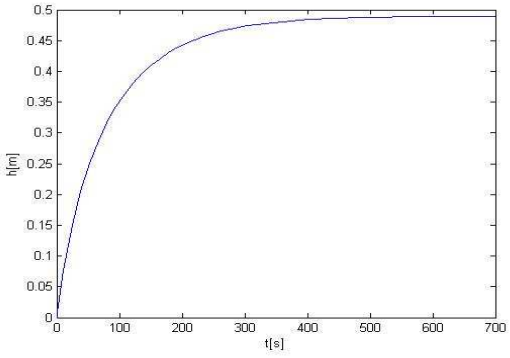
$$\dot{h3} = \frac{1}{A}(q_{13} - q_{e3}) \tag{4.9}$$



u_c=10 u_e=0



u_c=10 u_e=0.5



u_c=10 u_e=1

Figure 4.3: 1TS Matlab simulations

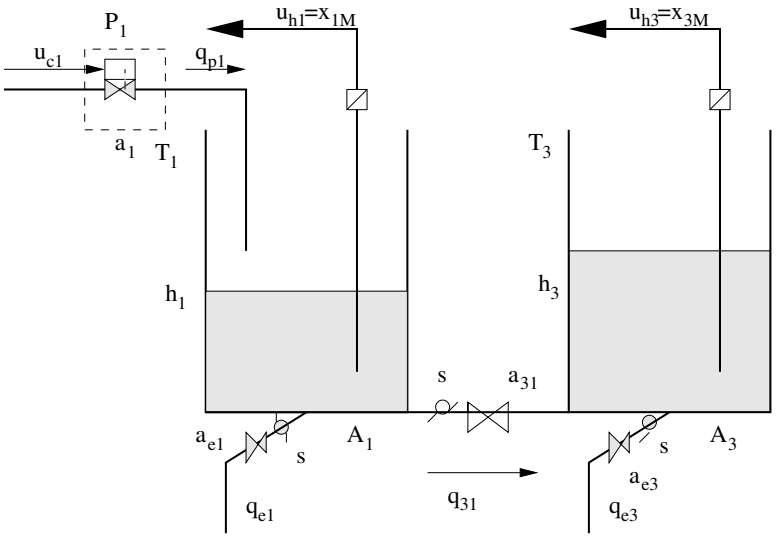


Figure 4.4: 2 Tanks System

where

$$q_{13} = \mu_{S1} \cdot S \cdot sgn(h_1 - h_3)\sqrt{2g|h_1 - h_3|} \tag{4.10a}$$

$$q_{e1} = \mu_{e1} \cdot S \cdot \sqrt{2gh_1} \tag{4.10b}$$

$$q_{e3} = \mu_{e3} \cdot S \cdot \sqrt{2gh_3} \tag{4.10c}$$

$$q_{p1} = c_1 u_{c1} \tag{4.10d}$$

Combining equations (**4.8**), (**4.9**), and (**4.10**) I get:

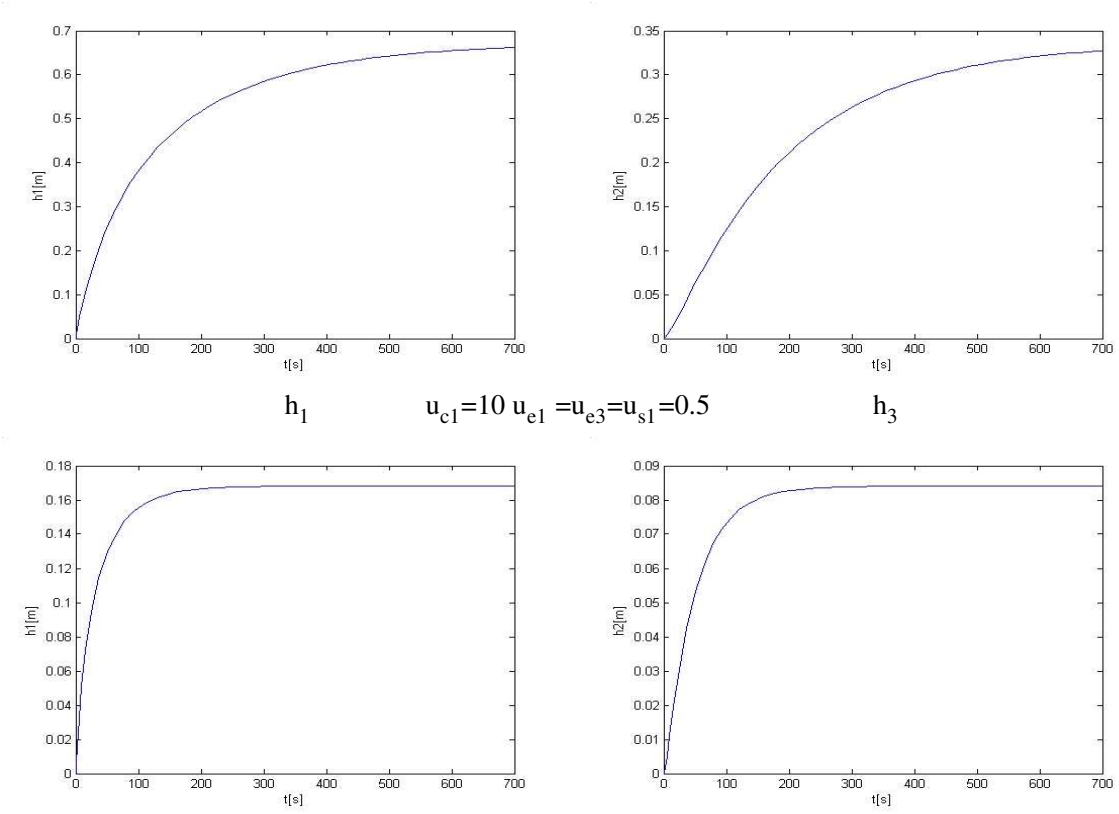
$$\dot{h_1} = \frac{1}{A}(c_1 u_{c1} - \mu_{S1} \cdot S \cdot sgn(h_1 - h_3)\sqrt{2g|h_1 - h_3|} - \mu_{e1} \cdot S \cdot \sqrt{2gh_1}) \tag{4.11a}$$

$$\dot{h_3} = \frac{1}{A}(\mu_{S1} \cdot S \cdot sgn(h_1 - h_3)\sqrt{2g|h_1 - h_3|} - \mu_{e3} \cdot S \cdot \sqrt{2gh_3} \tag{4.11b}$$

Linearizing the mathematical model (**4.11**) in the neighborhood of (h_{10}, h_{30}) I get:

$$\begin{aligned}
\Delta\dot{h}_1 =& -\frac{S\sqrt{2g}}{A} sgn(h_{10} - h_{30}) \cdot \sqrt{|h_{10} - h_{30}|} \cdot \Delta\mu_{s1} \\
&- \frac{S\sqrt{2g}}{A} \mu_{s10} sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10} - h_{30}|}} \cdot \Delta h_1 \\
&+ \frac{S\sqrt{2g}}{A} \mu_{s10} sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10} - h_{30}|}} \cdot \Delta h_3 \\
&- \frac{S\sqrt{2g}}{A} \cdot \sqrt{h_{10}} \cdot \Delta\mu_{e1} - \frac{S\sqrt{2g}}{A} \mu_{e10} \cdot \frac{1}{2\sqrt{h_{10}}} \Delta h_1 + \frac{c_1}{A} \Delta u_{c1} \\
\Delta\dot{h}_3 =& \frac{S\sqrt{2g}}{A} sgn(h_{10} - h_{30}) \cdot \sqrt{|h_{10} - h_{30}|} \cdot \Delta\mu_{s1} \\
&+ \frac{S\sqrt{2g}}{A} \mu_{s10} \cdot sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10} - h_{30}|}} \cdot \Delta h_1 \\
&- \frac{S\sqrt{2g}}{A} \mu_{s10} \cdot sgn(h_{10} - h_{30}) \cdot \frac{1}{2\sqrt{|h_{10} - h_{30}|}} \cdot \Delta h_3 \\
&- \frac{S\sqrt{2g}}{A} \cdot \sqrt{h_{30}} \cdot \Delta\mu_{e3} - \frac{S\sqrt{2g}}{A} \mu_{e30} \cdot \frac{1}{2\sqrt{h_{30}}} \Delta h_3
\end{aligned}$$

In **Fig. 4.5** I present the results of simulating the found mathematical model for the 2TS plant in Matlab,the Simulink schema can be found in **Appendix E**.



h_1 $u_{c1}=10$ $u_{e1}=u_{e3}=u_{s1}=0.5$ h_3

h_1 $u_{c1}=10$ $u_{e1}=u_{e3}=u_{s1}=1$ h_3

Figure 4.5: 2TS Matlab simulations

4.1.1.3 3TS Plant

The plant is presented in **Fig. 4.1**. The mathematical model for 3TS plant is equivalent to the mathematical model tor two 2TS plants and one tank is common to both plants.

2TS$_1$

$$\dot{h_1} = \frac{1}{A}(q_{p1} - q_{13} - q_{e1})$$
$$\dot{h_3} = \frac{1}{A}q_{13} - q_{e3}$$

2TS$_2$

$$\dot{h_2} = \frac{1}{A}(q_{p2} + q_{32} - q_{e2} - q_{g2})$$
$$\dot{h_3} = \frac{1}{A} - q_{32} - q_{e3}$$

Combining the equations for $2TS_1$ and $2TS_2$ I get the following mathematical model for 3TS plant:

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

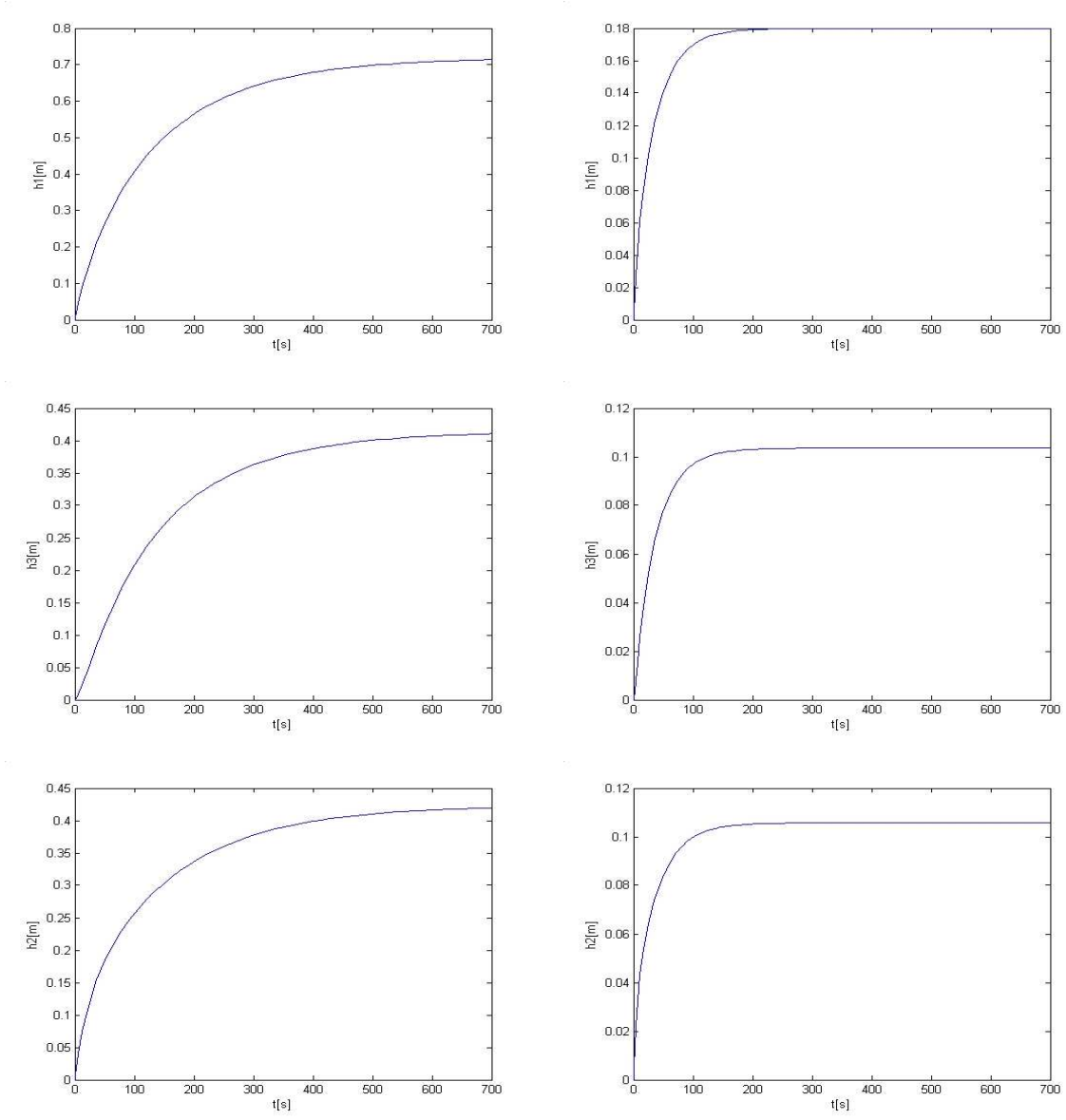
where

$$q_{13} = \mu_{S1} \cdot S \cdot sgn(h_1 - h_3)\sqrt{2g|h_1 - h_3|} \tag{4.16a}$$
$$q_{32} = \mu_{S2} \cdot S \cdot sgn(h_3 - h_2)\sqrt{2g|h_3 - h_2|} \tag{4.16b}$$
$$q_{20} = \mu_{g2} \cdot S \cdot \sqrt{2gh_2} \tag{4.16c}$$
$$q_{ei} = \mu_{ei} \cdot S \cdot \sqrt{2gh_i} \tag{4.16d}$$
$$q_{pi} = c_i u_{ci} \tag{4.16e}$$

Linearizing the model in the neighborhood of (h_{10}, h_{20}, h_{30}) I get:

$$
\begin{aligned}
\Delta\dot{h}_1 =& -\frac{\sqrt{2g}}{A} sgn(h_{10}-h_{30}) \cdot \sqrt{|h_{10}-h_{30}|} \cdot \Delta u_{s1} \\
& -\frac{\sqrt{2g}}{A} u_{s10} sgn(h_{10}-h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_1 \\
& +\frac{\sqrt{2g}}{A} u_{s10} sgn(h_{10}-h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_3 \\
& -\frac{\sqrt{2g}}{A} \cdot \sqrt{h_{10}} \cdot \Delta u_{e1} - \frac{\sqrt{2g}}{A} u_{e10} \cdot \frac{1}{2\sqrt{h_{10}}} \Delta h_1 + \frac{c_1}{A} \Delta u_{c1} \\
\Delta\dot{h}_2 =& \frac{\sqrt{2g}}{A} sgn(h_{30}-h_{20}) \cdot \sqrt{|h_{30}-h_{20}|} \cdot \Delta u_{s2} \\
& +\frac{\sqrt{2g}}{A} u_{s20} sgn(h_{30}-h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_3 \\
& -\frac{\sqrt{2g}}{A} u_{s20} sgn(h_{30}-h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_2 - \frac{\sqrt{2g}}{A} \cdot \sqrt{h_{20}} \cdot \Delta u_{e2} \\
& -\frac{\sqrt{2g}}{A} u_{e20} \cdot \frac{1}{2\sqrt{h_{20}}} \cdot \Delta h_2 - \frac{\sqrt{2g}}{A} \sqrt{h20} \Delta u_{g2} \\
& -\frac{\sqrt{2g}}{A} u_{g20} \frac{1}{2\sqrt{h_{20}}} \Delta h_2 + \frac{c_2}{A} \Delta u_{c2} \\
\Delta\dot{h}_3 =& \frac{\sqrt{2g}}{A} sgn(h_{10}-h_{30}) \cdot \sqrt{|h_{10}-h_{30}|} \cdot \Delta u_{s1} \\
& +\frac{\sqrt{2g}}{A} u_{s10} \cdot sgn(h_{10}-h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_1 \\
& -\frac{\sqrt{2g}}{A} u_{s10} \cdot sgn(h_{10}-h_{30}) \cdot \frac{1}{2\sqrt{|h_{10}-h_{30}|}} \cdot \Delta h_3 \\
& -\frac{\sqrt{2g}}{A} sgn(h_{30}-h_{20}) \cdot \sqrt{|h_{30}-h_{20}|} \cdot \Delta u_{s2} \\
& -\frac{\sqrt{2g}}{A} u_{s20} \cdot sgn(h_{30}-h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_3 \\
& +\frac{\sqrt{2g}}{A} u_{s20} \cdot sgn(h_{30}-h_{20}) \cdot \frac{1}{2\sqrt{|h_{30}-h_{20}|}} \cdot \Delta h_2 \\
& -\frac{\sqrt{2g}}{A} \cdot \sqrt{h_{30}} \cdot \Delta u_{e3} - \frac{\sqrt{2g}}{A} u_{e30} \cdot \frac{1}{2\sqrt{h_{30}}} \Delta h_3
\end{aligned}
$$

In **Fig. 4.6** I present the results of simulating the found mathematical model for the 3TS plant in Matlab,the Simulink schema can be found in **Appendix E**.



$u_{c1}=u_{c2}=10$ $u_{e1}=u_{e2}=u_{e3}=u_{g2}=u_{s1}=u_{s2}=0.5$ $u_{c1}=u_{c2}=10$ $u_{e1}=u_{e2}=u_{e3}=u_{g2}=u_{s1}=u_{s2}=1$

Figure 4.6: 3TS Matlab simulations

4.1.2 Controllers Design

The aim of the controller is to maintain constant the level of the fluid in tanks T_1 and T_2 (h_1 and h_2, respectively), using the fill flow capacities q_{p1} and q_{p2} and the tank T_3. The perturbations are considered to be:

- the level of the fluid in tank T_3;
- drain of the fluid throughout emptying taps, marked by flow capacities: q_{e1}, q_{e2}, q_{e3}, q_{20}.

There are two possible situations that should be consider: (1)if there is no perturbation, then as showed in **Section 4.1.1.1** the mathematical model has an integrator behavior, and a good controller for such a mathematical model would be a P Controller, and (2) if there are some perturbations in the system then will be used a PI Controller.

4.1.2.1 P Controller

This controller will be used only if there is no perturbation acting up on the tank for which the controller is working. Since there is no perturbation, the tank can be consider to be independent, resulting that can be used the 1TS mathematical model (**4.6**) for the controller design.
Transfer function for 1TS mathematical model without perturbations is:

$$H_P(s) = \frac{c}{A}\frac{1}{s} \tag{4.18}$$

Controller transfer function is:

$$H_R(s) = k_R \tag{4.19}$$

Using equation (**2.7**) and choosing $t_c = 60s$ I get:

$$k_R = \frac{A}{20c} \tag{4.20}$$

In **Fig. 4.7** I present the results of simulating the control of 1TS plant using the P Controller I design for a reference value of 0.6m,the Simulink schema can be found in **Appendix E**.

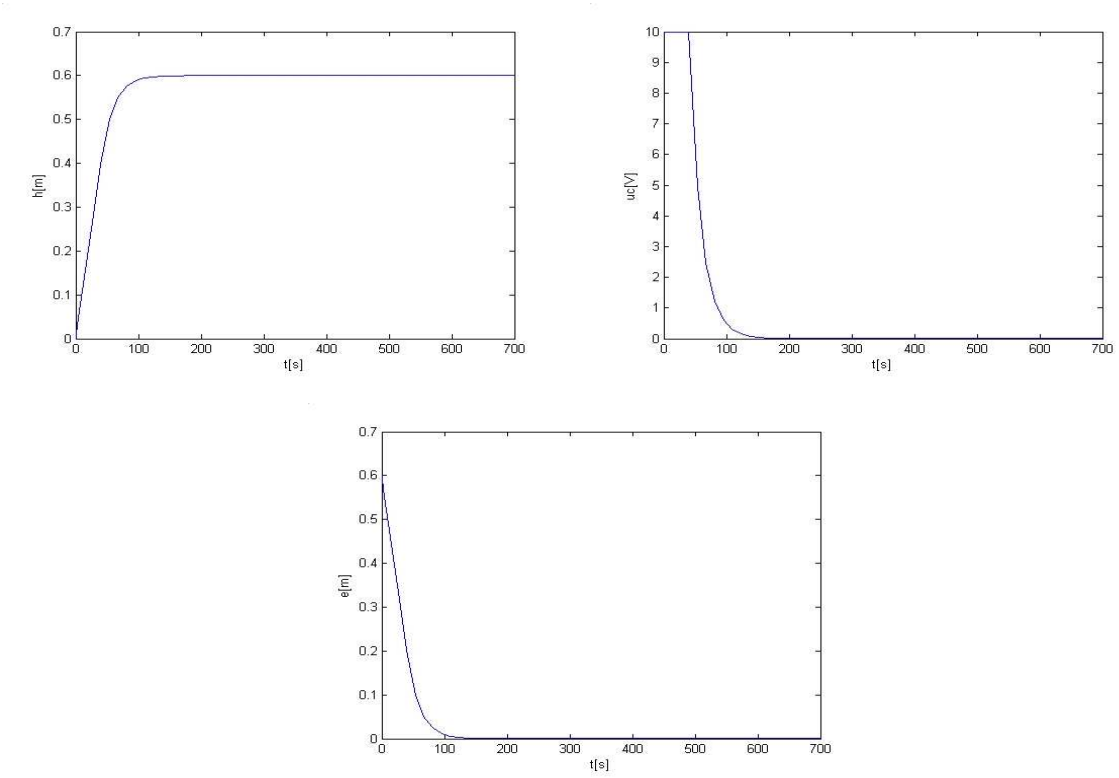


Figure 4.7: 1TS with P Controller simulation results

4.1.2.2 PI Controller

The PI controller will be used if there is any perturbation acting up on the tank for which the controller is working.Since as I showed in **Section 4.1.1.3** the 3TS plant is equivalent to two 2TS plants, having a common tank, I will use the 2TS plant mathematical model for controller design. The controller for the first tank (T1) will be designed using the mathematical model of the 2TS plant made up of T1 and T3, while the controller for the second tank(T2) will be designed using the mathematical model of the 2TS plant made up of T2 and T3. The method used to design the two controllers is *The Frequency Domain Controller Design* method.

PI Controller transfer function is:

$$H_R(s) = \frac{k_R}{sT_i}(1 + sT_i) \tag{4.21}$$

PI Controller for T1 - Bode diagrams for system transfer function and closed system transfer function are presented in **Fig. 4.8**.

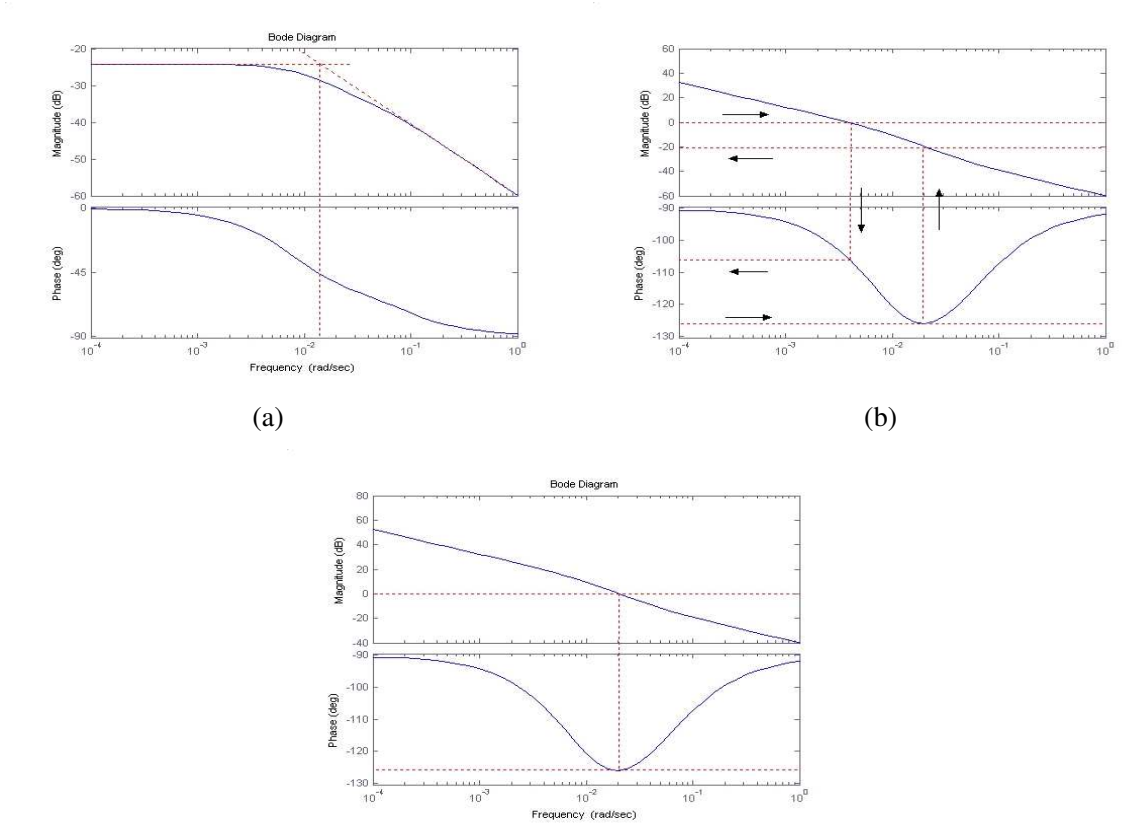


Figure 4.8: Bode diagrams:(a)system; (b)closed system, $k_R = 1$ (c)closed system, $k_R = 10$

From **Fig. 4.8a** can be read the dominant time constant (T_{dom}) for the plant: $T_{dom} = 70s$ I choose $T_i = 15s$ and I plot the Bode diagrams for closed system for $k_R = 1$ (**Fig. 4.8b**). If I choose $\varphi_{res} = 45^o$ then from figure (b) I can see that I have to raise the diagram with 20db, which is equivalent to multiply by 10, thus results $k_R = 10$. Hence for tank T_1 I get the following PI Controller:

$$H_R(s) = \frac{10}{15s}(1 + 15s) \tag{4.22}$$

In **Fig. 4.9** I present the results of simulating the control of 2TS(T_1-T_3) plant using the PI Controller I design for a reference value of 0.4m and all perturbations set to 0.5, in Matlab,the Simulink schema can be found in **Appendix E**.

PI Controller for T2- Bode diagrams for system transfer function and closed system transfer function are presented in **Fig. 4.10**.

From **Fig. 4.10a** can be read the dominant time constant (T_{dom}) for the plant: $T_{dom} = 65s$ I choose $T_i = 15s$ and I plot the Bode diagrams for closed system for $k_R = 1$ (**Fig. 4.10b**). If I

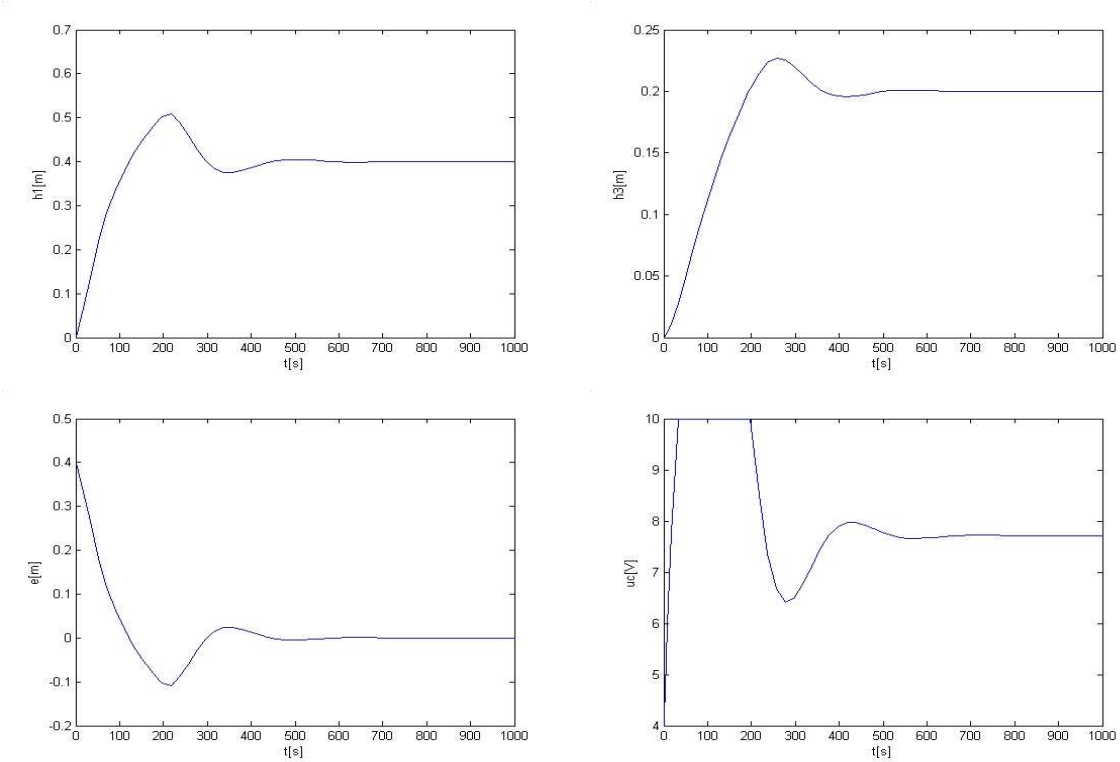


Figure 4.9: 2TS(T_1-T_3) with PI Controller simulation results

choose $\varphi_{res} = 45^o$ then from figure (b) I can see that I have to raise the diagram with 17.5db, which is equivalent to multiply by 5.3, thus results $k_R = 5.3$. Hence for tank T_1 I get the following PI Controller:

$$H_R(s) = \frac{5.3}{15s}(1 + 15s) \tag{4.23}$$

In **Fig. 4.11** I present the results of simulating the control of 2TS(T_2-T_3) plant using the PI Controller I design for a reference value of 0.4m and all perturbations set to 0.5,the Simulink schema can be found in **Appendix E**.

The PI controllers will be implemented so that the P and I components are distinct as shown in **Fig. 4.12**.

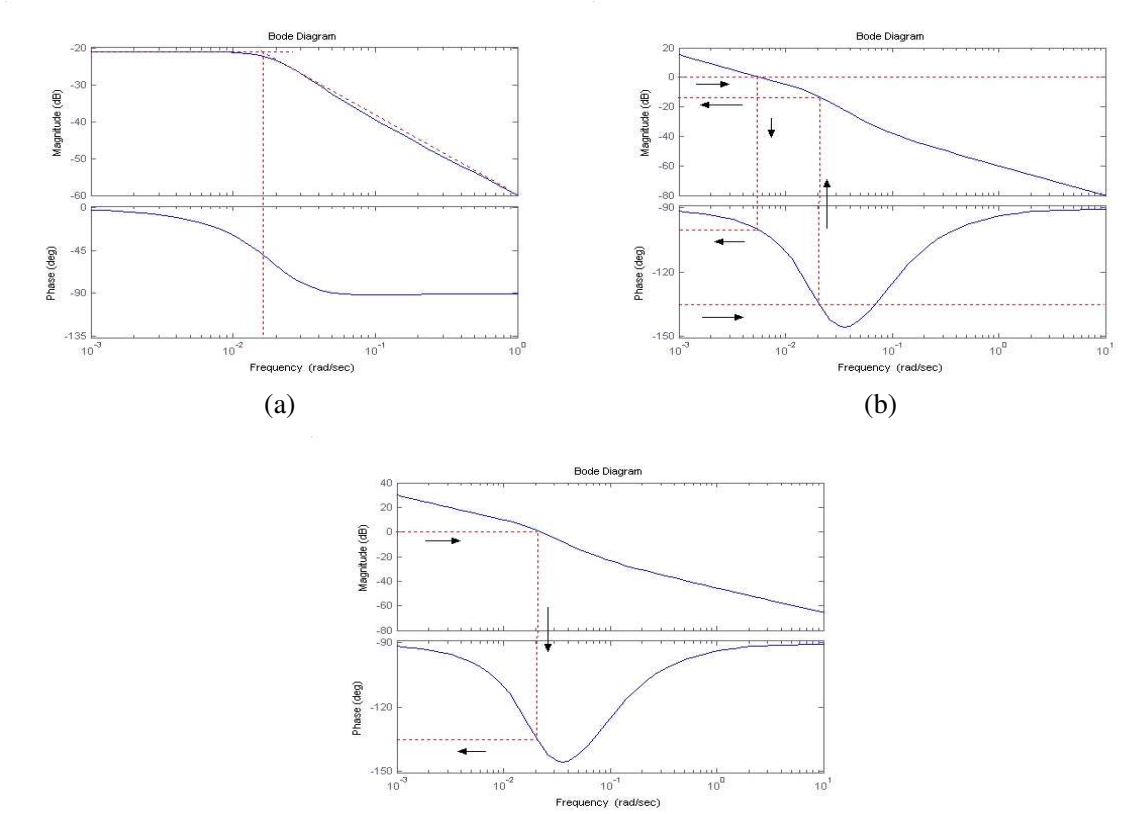


Figure 4.10: Bode diagrams:(a)system; (b)closed system, $k_R = 1$ (c)closed system, $k_R = 10$

In **Fig. 4.13** I present the results of simulating the control of 3TS plant using the PI Controllers I design before, for a reference value of 0.5m for h_1 and 0.4 for h_2, and all perturbations set to 0.5,the Simulink schema can be found in **Appendix E**.

As it can be seen from the simulation results $\sigma_1 > 20\%$, which is too much so I have introduce Anti Windup Reset (AWR) in order to reduce the σ_1. The block schema for the controller with AWR is presented in **Fig. 4.14**.

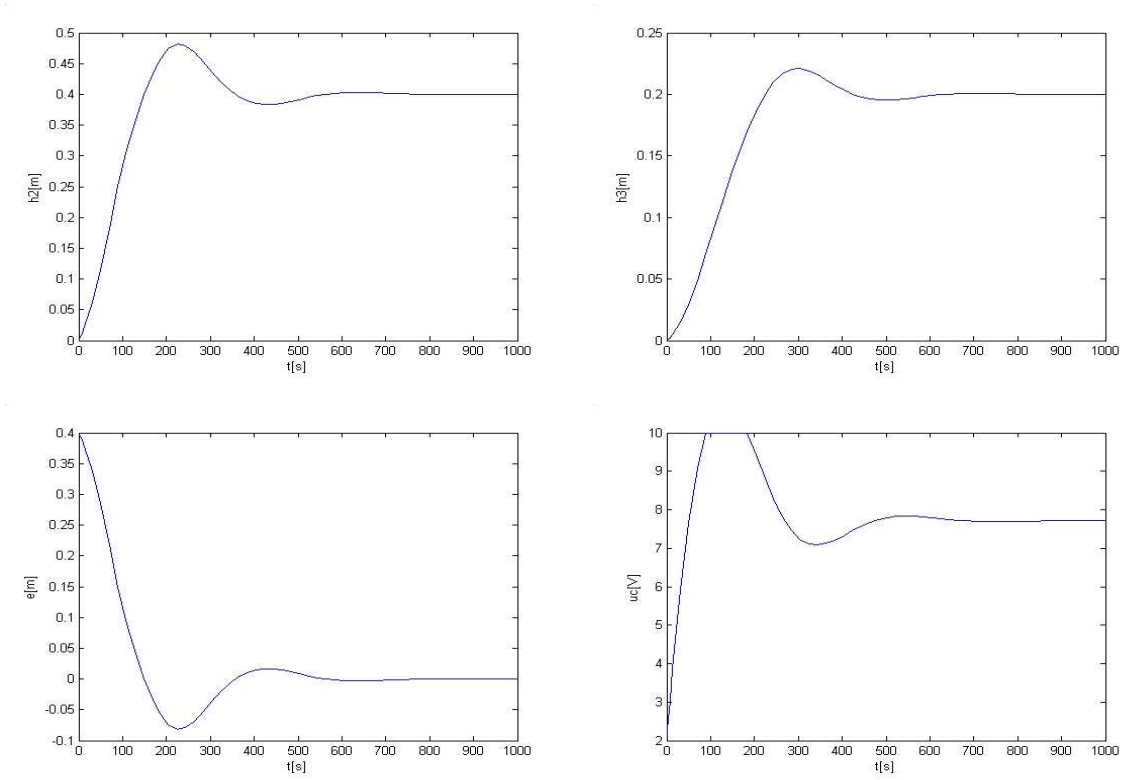


Figure 4.11: 2TS(T_2-T_3) with PI Controller simulation results

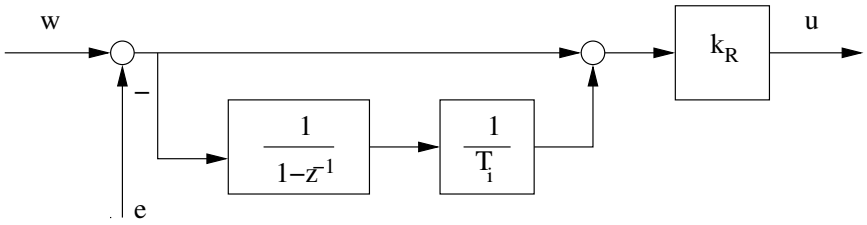


Figure 4.12: PI Controllers implementation

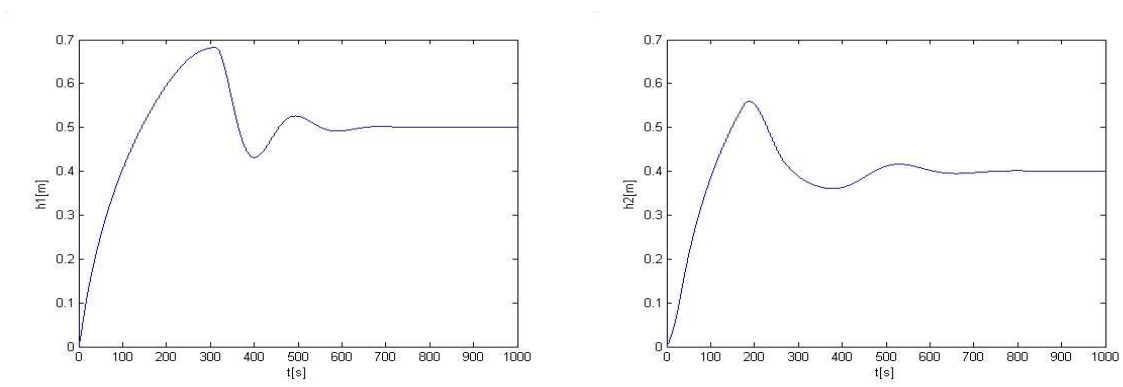


Figure 4.13: 3TS with PI Controllers simulation results

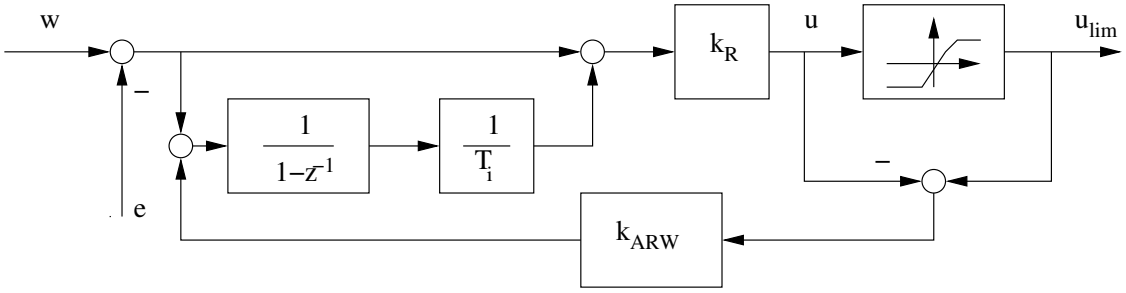


Figure 4.14: PI Controller with ARW

In **Fig. 4.15** I present the results of simulating the control of 3TS plant using the PI Controllers I design before with ARW, for a reference value of 0.5m for h_1 and 0.4 for h_2, and all perturbations set to 0.5,the Simulink schema can be found in **Appendix E**.

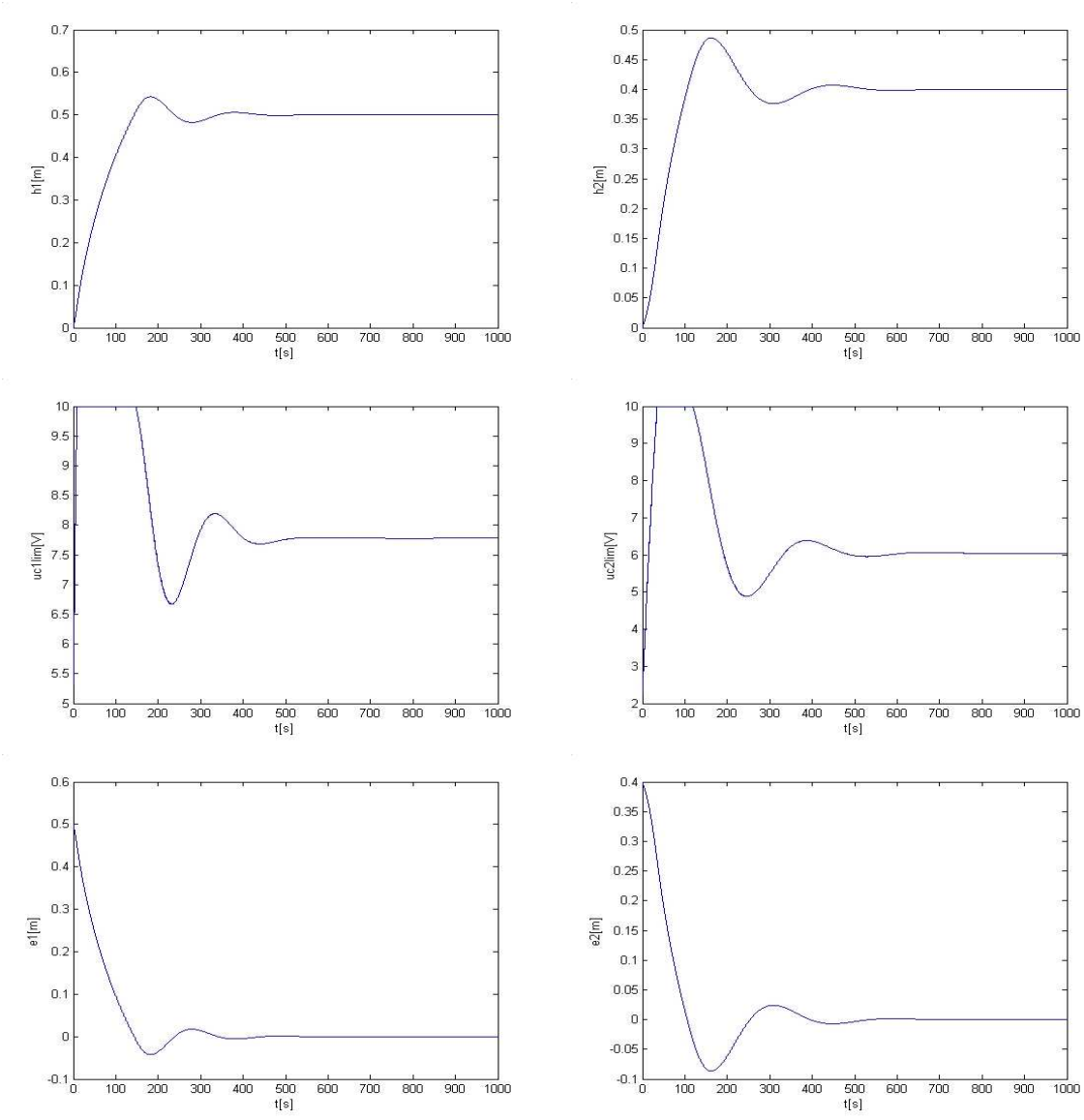


Figure 4.15: 3TS with PI Controllers with ARW simulation results

4.2 Control Protocol

I have implemented this protocol in order to have a simple way to communicate between the regulator client and the process server. The protocol is build over TCP/IP. I have implemented a full version of the protocol in Java and in C I have implemented only the regulator client.
In **Fig. 4.16** I present the structure of a packet from *Control Protocol*. As it can be seen from

Figure 4.16: Control Protocol packet structure

the figure the first byte in the packet will represent the packet type, after this byte there can be a fix or a variable number of bytes representing packet data. If packet data has a fix number of bytes then data length is known from the type (a type can not be used both with fix and variable data length). If the packet data has a variable length then the first byte after the type byte will represent data length. For a full description of the protocol you should look at **Appendix B**. Further I will present the Java and C implementation of the protocol.

4.2.1 Java implementation

Control Protocol was implemented in the package *control.net*, the package contains only four classes:

- *ControlProtocol* – in this class are defined as *public static final byte* members all the possible packet types and all the error codes, there are also defined a few utility *static* methods;
- *ControlProtocolException* – this represents an exception, the class extends *java.lang.Exception*;
- *ProcessServer* – this class represents the server (the process is considered to be the server); the constructor for this class takes a single parameter that represents the port on which the server will listen, still in the constructor a *ServerSocket* is created for the specified port, but the server will not start listening for clients until the *start()* method is called; what *start()* method does, is to create a new thread in which the server will start

listening for clients, and if a client is accepted then an instance of *ServerWorker*, which is an internal class, will be created and put in a list of clients, in order to take care of the client in a separate thread, while the server will continue to wait for others clients; what worker dose is: wait for packets from the client and for each packet if it is necessarily it will raise an event to notify the presence of the packet; in the server class there are also defined method that allows to the upper layer to send a packet to all client (broadcast); there is also a stop method that will send a packet to all clients telling them that the server is going down and then it will close all the workers and the server main thread.

- *RegulatorClient* – this class represents the client (the regulator is considered to be the client); the constructor will take two parameters: (1) a *String* that will represent the server address, and (2) an *int* that will represent the port on which the server is listening, in the constructor there will be created a *Socket*, that will connect to server, after the connection was establish a synchronization packet is send to the server that will have to reply with the same packet, otherwise the client will shutdown; there is also a *start()* and a *stop()* method, the *start()* method will create a new thread, where the client will wait for packets send by the server and it will raise an event if necessarily to notify the upper layer, the *stop()* method will send a packet to the server telling it that the client is going down and then stops the thread; there are also others methods defined, that can be used by the upper layer to send a packet to the server.

4.2.2 C implementation

It is just a partial implementation of the regulator client, but still enough to be able to control the process. The implementation is made in one file C file *rg_cleint.c*, and the corresponding header file *rg_client.h*. The implementation consists of the following functions:

- *init_rg_client()* – initialized the connection and sends the synchronization packet;
- a couple of functions that are used to send a particular type of packet;
- *read_sensors()* – used to read a packet send by server and that represents the new fluid levels in the three tanks.

As it can be seen the error cases are not treated and also the only valid packet that should be send by the server is the one that represents the new signals values, if any other packet is sent, then the regulator will report a broken protocol and will stop. This condition is satisfied because the server can accept to types of clients (*giotto client* and *java client*), the only difference between them is that to *giotto clients* the server will send only packets that represents new signals value and will ignore all other packets.

4.3 3TS Simulator

3TS Simulator it is implemented in Java. The main class is *simulator.Simulator3TS*. The user guide for the application is presented in **Appendix C**. When implementing the simulator I tried to decouple *User Interface* as much as possible from the logic. Having this idea in mind, I have split the application in three packets:

- simulator.model – contains all classes used to simulate the process;
- control.net – this was presented before and what is used from this is the server.
- simulator.ui – contains all classes used to create user interface;

4.3.1 Package simulator.model

This package contains all the class used to simulate 3TS process. In **Fig. 4.17** I present the UML diagram for the classes in this package.

As it can be seen from the UML diagram all the models are driven from the same class (*Model*), which extends another class *MyPropertyChangeSupport*. Next I will describe each class:

MyPropertyChangeSupport – this class is from the packet *control.util*, it is an abstract class, its main purpose is to add support for *PropertyChange* event in the subclasses;

Model – this is also an abstract class and it is extended by all models;

TapModel – this class implements the logic for a tap, it has a *double* member that represents the open coefficient, it also keeps two references to a *TankModel*, if both are not *null* then

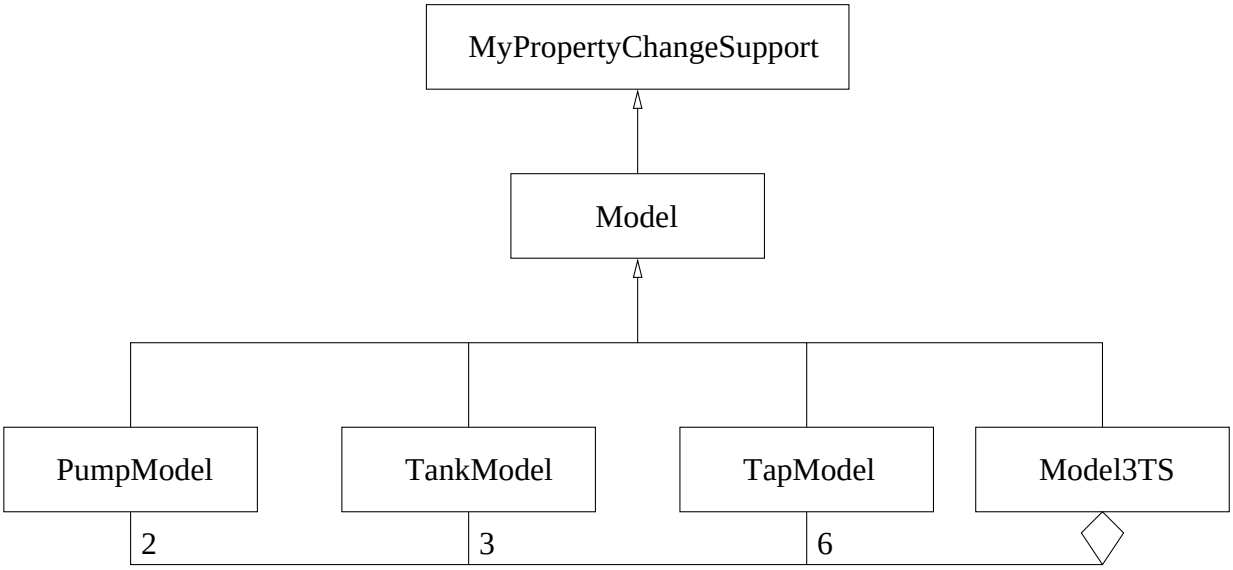


Figure 4.17: UML diagram for simulator.model package

the tap is between two tanks else it is an emptying tap, this class has also a method that based on the tanks references and the open coefficient computes the flow capacity through the tap, the class can also trigger an event every time the open coefficient changes ;

PumpModel – this class implements the logic for a pump, this class has three *members*: pump capacity, the command given to the pump (in [V], and the pump debit computed on the first two), the class can trigger an event when command has changed or debit has changed;

TankModel – this class implements the logic for a tank, this class has a *Set* of taps that are connected to the tank, a *Set* of pumps that are connected to the tank, a *double* member representing the hight of the tank, and another *double* member representing the fluid level, there is also a method that takes as a parameter the sample time, and based on pumps and taps debit it will compute the speed of variation of the fluid, that will be integrated to obtain the actual fluid level, the class can trigger an event every time the fluid level changes;

Model3TS – this class implements the logic for the 3TS process simulation and it uses the previous three models in order to do this, the class has an *int* member representing the sample time, in order to simulate 3TS process a thread is created that after sample time milliseconds will compute the new fluid level in each tank;

4.3.2 Package simulator.ui

For each mode there is a graphical component used to control that model parameters. In **Fig. 4.18** it is presented the UML diagram of the class in this package.

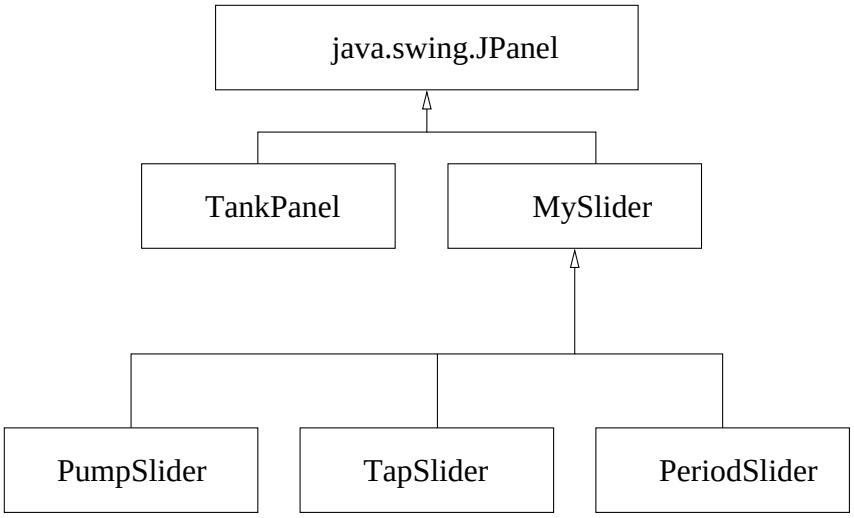


Figure 4.18: UML diagram for simulator.ui package

Next I will present each class in the packet:

MySlider – this class will be used only as a base class, it represents a slider;

PumpSlider – extends *MySlider* and it is used to update the command given to the pump, it has a *PumpModel* member for which the command is given;

TapSlider – extends *MySlider* and it is used to update the open coefficient of a tap, there is a *TapModel* member for which the coefficient is controlled;

PeriodSlider – extends *MySlider* and it is used to update the simulation sample time, there is a *Model3TS* member for which the sample time is updated;

TankPanel – this is used to graphically represent the the evolution of the fluid level in a tank, there is a *TankModel* member which is represented by the component.

4.4 3TS Controller

4.4.1 3TS Controller

3TS Controller is implemented in Java. The main class is *controller.Controller3TS*. The user guide for the application is presented in **Appendix D**. As *3TS Simulator* this too is implemented with the idea of separating the user interface from the application logic. It is made up of three packets:

- controller.model – contains all classes used to implement the regulator logic;
- control.net – this was presented before and what is used from this is the client;
- controller.ui – contains all classes used to create user interface.

Next I will present each packet except *control.net* in a separate subsection.

4.4.2 Package controller.model

The classes in this package implements the logic of the application. In **Fig. 4.19** can be seen the UML diagram for this package. Further I will detail each of this classes:

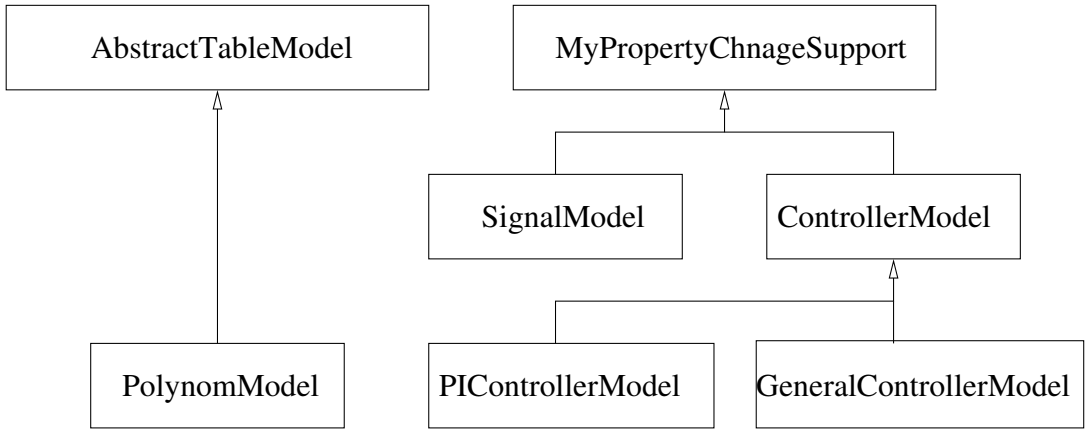


Figure 4.19: UML diagram for controller.model package

SignalModel – this class implements a model for a signal, it has an *ArrayList* member which is used to store data, what is interesting about this class is that when accessing an element whose index is not in the rage then zero will be return and no Exception is thrown, also it can trigger an event when a new value is added;

ControllerModel – this is the base class for all controllers, it has an *int* member that represents regulator sample time, it also has an *ArrayList* of SignalModel for commands, another one for feedback signals, and another one for error signals and an array of doubles for references and new computed commands, it can trigger an event when the sample time has changed;

PIControllerModel – this class implements the PI Controller with ARW, it has as members k_R and T_i based on this it will compute the numerical algorithm parameters, it also implements the abstract method *computeCommands* that is inherit from *RegulatorModel*, in this method, based on the feedback signals, error signals, and commands signal the PI numerical algorithm is implemented;

PolynomModel – this class extends *AbstractTableModel* because it is also used as a model for a *JTable* component, it has an *ArrayList* member that will be used to store polynom coefficients;

GeneralControllerModel – this class implements a general numerical control algorithm, it has two members of type *PolynomModel*, one for *p* coefficients and another one for *q* coefficients, the class implements *computeCommands* from the superclass, in this method actually is implemented the general numerical algorithm based on *p* coefficients, *q* coefficients, feedback signals, error signals, and commands signals, in **Code 4.4.1** I present the *computeCommands* method for the general algorithm (*computeCommands* method for PI controller is just a particular case of this).

4.4.3 Package controller.ui

The classes in this package are used to create user interface for the *3TS Controller*. In **Fig. 4.20** can be seen the UML diagram for this package. Further I will detail each of this classes:

DiagramPanel – this is used for plotting a signal, the signal is represented by a *SignalModel* member, every time a value is added to the signal the component is repainted;

DiagramDialog – this class is used to display a dialog, that will draw the diagram for a signal and it will compute the quality indicators if it is the case;

Code 4.4.1 computeCommands method for general numerical control algorithm

```
public void computeCommands() {
   //initializations
   ...
   double yk=ySignale.getScaledValue(ySignale.getSize()-1);
   double ek=referances[0]*ySignale.getScaleFactor()-yk;
   // add p coefficients
   final Iterator p=pCoefficient.iterator();
   int index=pCoefficient.getRowCount();
   while(p.hasNext()){
       inal double crrP=((Double)p.next()).doubleValue();
       newCommands[0]+=-crrP*uSignale.get(uSignale.getSize()-index);
        index--;
    }
    //add p coefficients
    final Iterator q=qCoefficient.iterator();
    index=qCoefficient.getRowCount();
    while(q.hasNext()){
       final double crrQ=((Double)q.next()).doubleValue();
       if(index==1)
          newCommands[0]+=crrQ*ek;
       else
           newCommands[0]+=crrQ*eSignale.get(eSignale.getSize()
                          -index+1);
       index--;
    }
    ....
    // limit the command [0,10]
    ....
    // store command and error
}
```

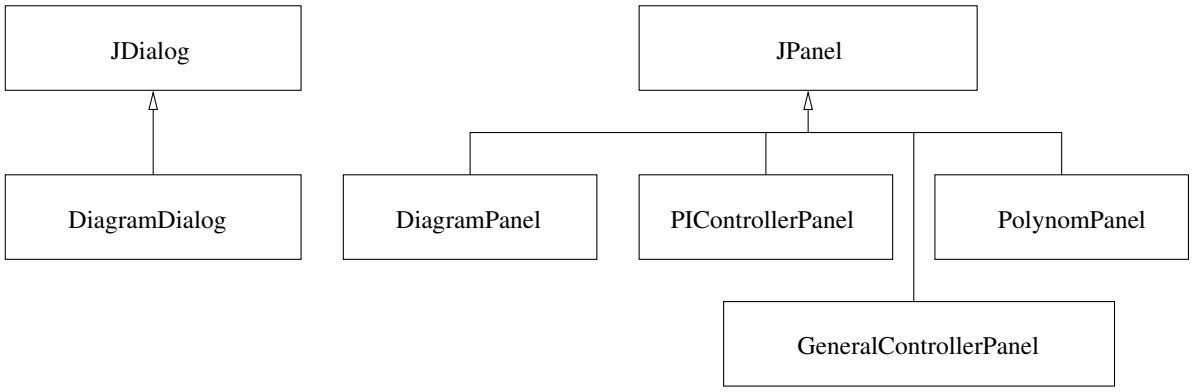


Figure 4.20: UML diagram for controller.ui package

PIControllerPanel – this class is used to create a graphical component that can be used to set the parameters for a *PIControllerModel* instance;

GeneralControllerPanel – this class is used to create a graphical component that can be used to set the parameters for a *GeneralControllerModel* instance;

PolynomPanel – this class is used to create a graphical component that can be used to add, remove, or change a coefficient from a *PolynomModel* instance.

4.4.4 TSL 3TS Controller

I have implemented only the PI Controller with ARW algorithm. The implementation consists of two pars:

- one that is written in C, and implements the control algorithm as well as the communication with the plant, this represents the functionality;
- and the part that is implemented in *TSL*, representing the timing.

4.4.5 C implementation

C implementation consist of: (1)*Control Protocol* implementation, this was presented in **Subsection 4.2.2** and (2) the regulator implementation and the implementation of the driver used to read sensors and write actuators.

In **Code 4.4.2** I present the C implementation for the PI with ARW control algorithm. One can clearly see the separation between P and I components. All the parameters of the algorithm (k_R, T_i, and sample time) are read from a file in the initialization function (discussed in **Subsection 4.2.2**).

4.4.6 TSL implementation

In **Fig. 4.21** I present the structure of the program. As it can be seen from the figure the program has four modes: *P_P*, *P_PI*, *PI_P*, and *PI_PI*. All four modes have:

- the same period;

Code 4.4.2 P_Rg function

```
double P_Rg(double w,double y, double kR,double kRi,
   double Tri,type_circular_array *u,type_circular_array *ulim,
   type_circular_array *ui,type_circular_array *e){
  double uk;
  double ek;
  double uik;
  ek=w-y;
  uk=kR*ek;
  uik=(uk-kRi*ek)*Tri/kRi;
  circular_array_add(u,uk);
  if(uk<0.0){
    uk=0.0;
  }
  if(uk>10.0){
    uk=10.0;
  }
  circular_array_add(ulim,uk);
  circular_array_add(ui,uik);
  circular_array_add(e,w-y);
  return uk;
}
```

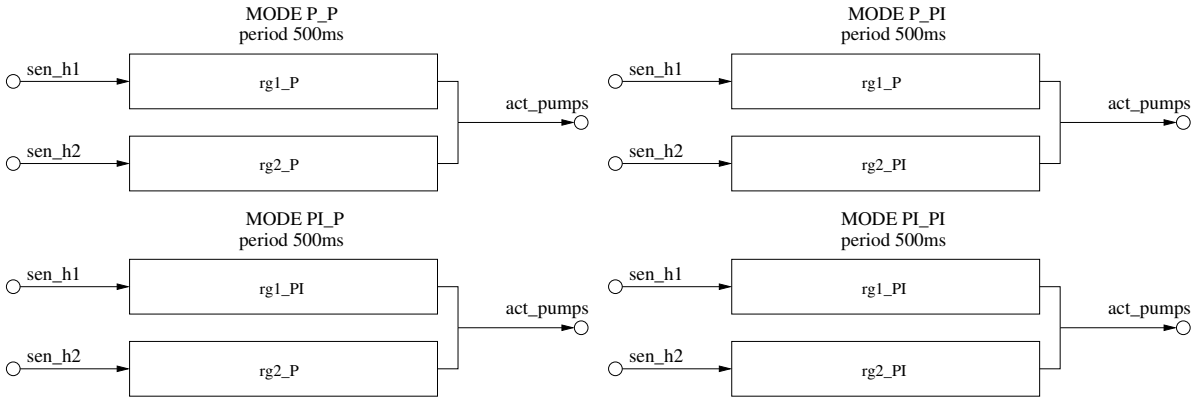


Figure 4.21: TSL program structure

- two tasks;
- two sensors update with an offset of 100ms (sen_h1 and sen_h2);
- six sensors update with zero offset;
- one actuator update (act_pumps);
- three mode switches.

The execution will always start with the *P_P* mode. The program can switch from any mode to any other mode.

P_P mode - this mode is used when there is no perturbation acting up on T_1 or T_2, the control lows for the two tanks are of P type;

P_PI mode - this mode is used when there is no perturbation acting up on T_1, but there is perturbation acting up on T_2, for T_1 there will be used a P control low, while for T_2 there will be used a PI control low

PI_P mode - this mode is used when there is no perturbation acting up on T_2, but there is perturbation acting up on T_1, for T_2 there will be used a P control low, while for T_1 there will be used a PI control low

PI_PI mode - this mode is used when there is perturbation acting up on T_1 or T_2, the control lows for the two tanks are of PI type;

Full *TSL* program and the *E code* generated for it can be seen in **Appendix A**.

4.4.7 Simulation results

In this section I will present the results of using *TSL* Controller.

In **Fig. 4.22** I present the results of using the PI Controller for tank T_1 and P Controller for tank T_2. The simulation conditions are:$h_{10} = 0.5m$, $h_{20} = 0.4m$, $\mu_{e1} = 0.5$, and $\mu_{e2} = \mu_{e3} = \mu_{s1} = \mu_{s2} = \mu_{g2} = 0$.

In **Table 4.1** and **Table 4.2** can be found the quality indicators for T_1 and T_2.

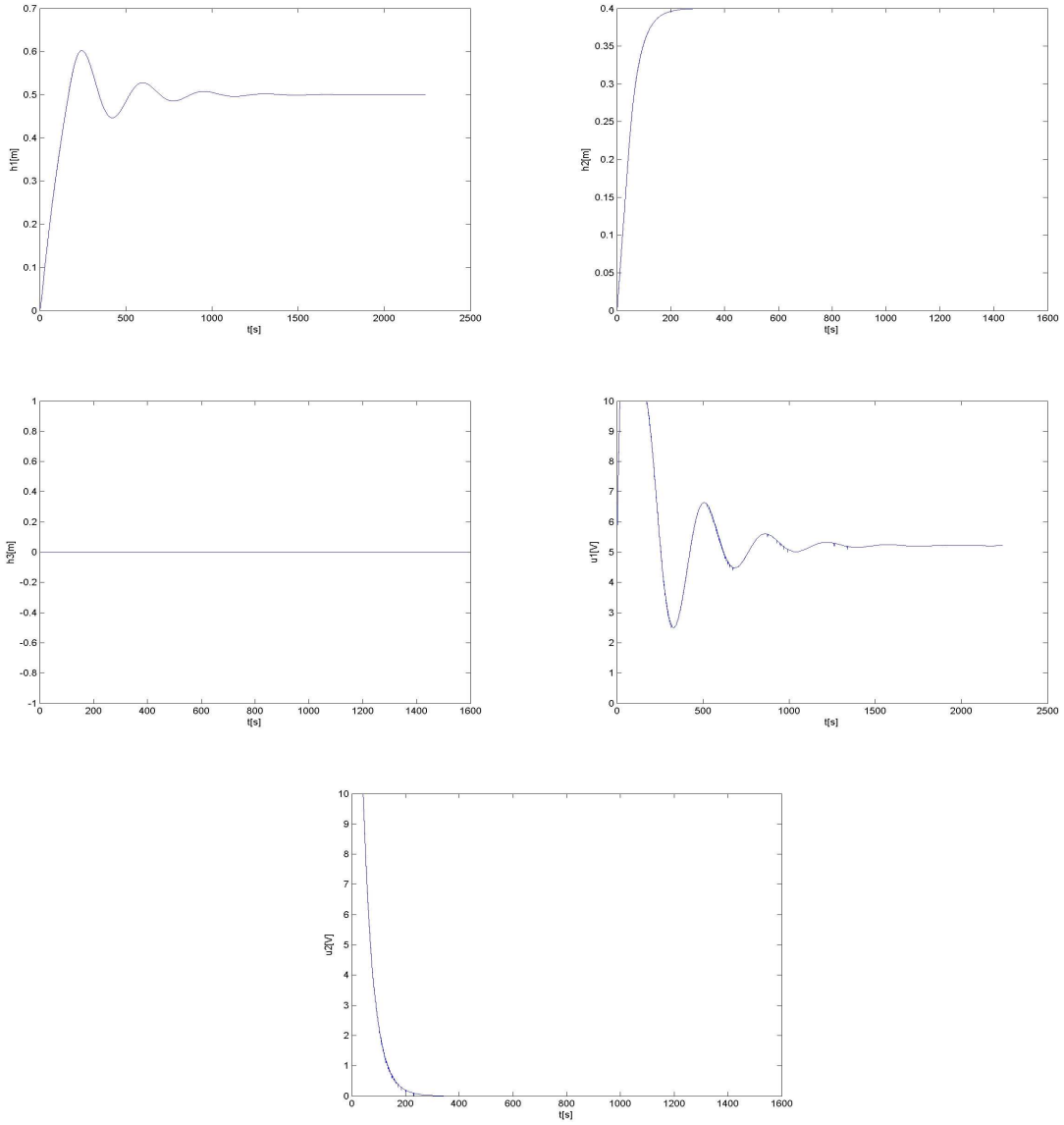


Figure 4.22: PI_P mode simulation results

σ_1	20%
t_s	72[s]
t_r	147[s]
t_1	167[s]
t_m	244[s]
t_c	818[s]

Table 4.1: PI_P T_1 quality indicators

σ_1	0%
t_s	42[s]
t_r	129[s]
t_1	342[s]
t_m	342[s]
t_c	120[s]

Table 4.2: PI_P T_2 quality indicators

In **Fig. 4.23** I present the results of using the PI Controller for tank T_1 and PI Controller for tank T_2. The simulation conditions are:$h_{10} = 0.5m$, $h_{20} = 0.4m$, $\mu_{e1} = \mu_{e2} = \mu_{e3} = 0.5$,$\mu_{s1} = 0.3$,$\mu_{s2} = 0.4$, and $\mu_{g2} = 0$.

In **Table 4.3** and **Table 4.4** can be found the quality indicators for T_1 and T_2.

σ_1	8%
t_s	114[s]
t_r	208[s]
t_1	302[s]
t_m	379[s]
t_c	639[s]

Table 4.3: PI_PI T_1 quality indicators

σ_1	17%
t_s	119[s]
t_r	217[s]
t_1	266[s]
t_m	381[s]
t_c	835[s]

Table 4.4: PI_PI T_2 quality indicators

In **Fig. 4.24** I present the results for the following simulation conditions:$h_{10} = 0.5m$, $h_{20} = 0.4m$, I start with all perturbation set to zero, then I set $\mu_{e1} = 0.4$, $\mu_{s1} = 0.5$, and $\mu_{e3} = 0.6$, and in the end I set $\mu_{ue2} = 0.5$, and $\mu_{us2} = 0.6$.

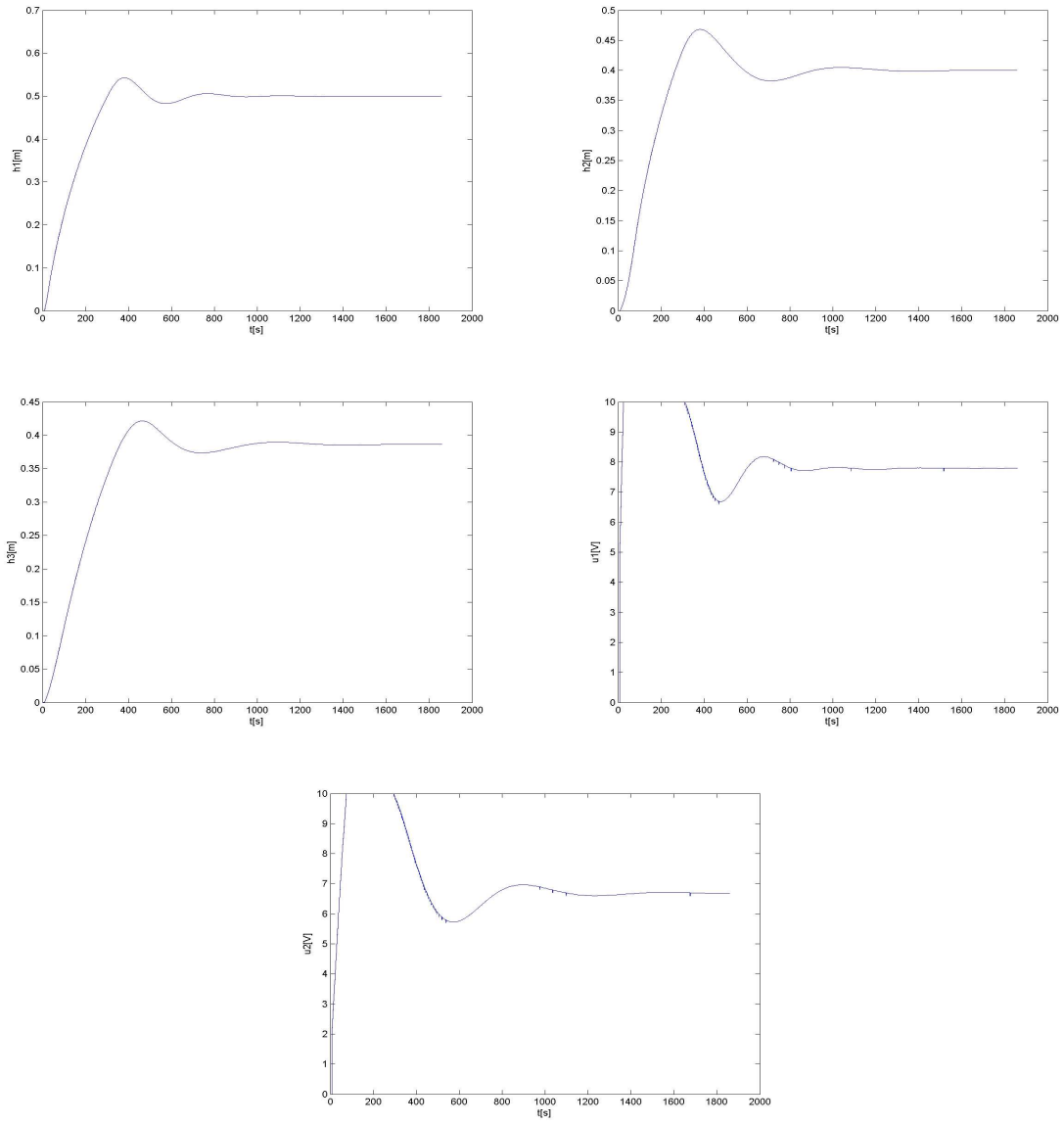


Figure 4.23: PI_PI mode simulation results

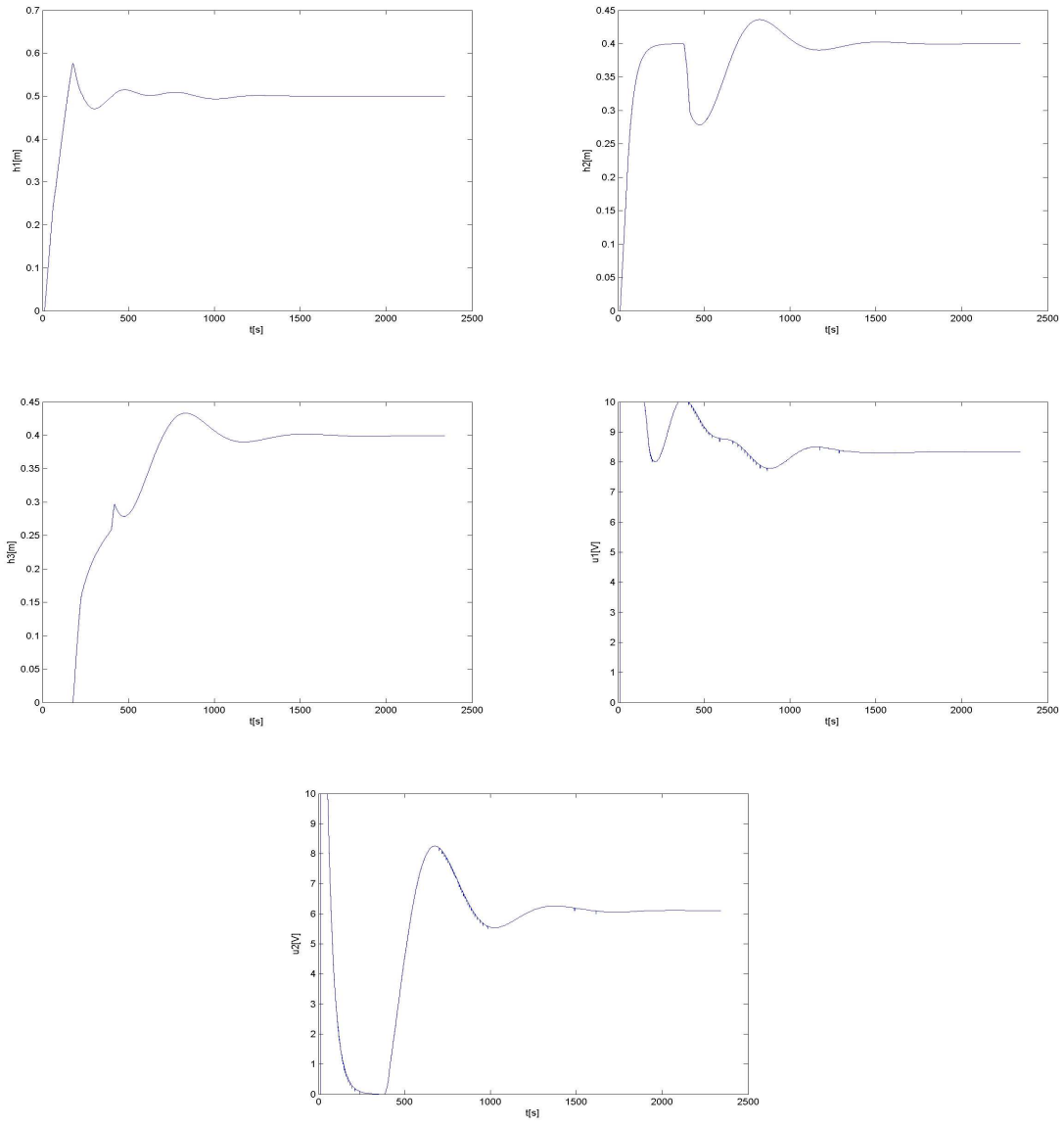


Figure 4.24: Multi- mode simulation results

Chapter 5

Conclusions

In this thesis I have presented how I had implemented the compiler for the *TSL*, starting from a previous implementation of a *Giotto* compiler. Then I had presented the solution to a control problem using *TSL* for the implementation of the control low. From the case study one can easily observe that there are three main steps that must be followed when solving a control problem using *TSL*:

- plant modeling and controller design;
- implementation of the control low in C, this represents the functionality;
- writing the *TSL* program, which represents the timing.

5.1 Outlook and Future Work

The anchored tasks were introduced in *TSL* because of the need to communicate between groups of tasks that have different elements, otherwise the float tasks would have been enough. In a future version we plan to remove anchored tasks and to have only float tasks, but in order to do this a mechanism to be able to communicate between float tasks groups with different frequencies is needed. A solution for this could be to use generalized ports (communicators) that can be both read and write by tasks, an actuator is a special case of port that can only be written, while a sensor is a special port that can only be read. Then in a *TSL* program there will be a *communicator update*, similar to a sensor or actuator update, that will have a frequency and will use a driver to update the port (the driver will read from some tasks).

In the current *TSL* implementation, there can be executed only one mode, a good idea would be able to run a number n of modes in parallel, but a mode will contain only tasks that have the same frequency.

An important limitation of *TSL* is that when switching from one mode to another then either the tasks are removed or added, but not both, in other words you are not able to replace tasks when you switch from one mode to another. In order to overcome this limitation for each task there will be specified a nominal execution time (NET), this of course is platform independent and it will allow to compare relatively tasks. The idea for this NET comes from the fact that if two tasks t_1 and t_2 are given, and if on a platform t_1 is faster then t_2, then this will be the case on any other platform.

In the current implementation of the *TSL Compiler* and *E Machine* the compiler generates C code for each *TSL* program, that has to be compiled and static linked in the *E Machine*. This implies that the *E Machine* must be compiled after each compilation of *TSL* program. A solution to this would be to use dynamic linking.

Appendix A

TSL 3TS Regulator Program

The *TSL* program that implements the regulator is:

```
sensor
c_double sen_h1 uses getH1; //reads h1
c_double sen_h2 uses getH2; //reads h2
c_int sen_e1 uses getE1; //read T1 e1 evacuation valve state
c_int sen_e2 uses getE2; //read T2 e2 evacuation valve state
c_int sen_e3 uses getE3; //read T2 e3 evacuation valve state
c_int sen_g2 uses getG2; //read T2 g2 evacuation valve state
c_int sen_s1 uses getS1; //read T1-T3 s1 inverconnection valve state
c_int sen_s2 uses getS2; //read T2-T3 s2 inverconnection valve state

actuator
c_double_arr act_pumps uses updateCommand; // update pumps command

output
c_double out_u1 := c_zero_double;  //command for first pump
c_double out_u2 := c_zero_double;  //command for second pump

//tasks

//General regulator for h1
task rg1_PI(c_double h1) output(out_u1) state(){
release PI_Rg1(h1, out_u1)
}

//General regulator for h2
task rg2_PI(c_double h2) output(out_u2) state(){
release PI_Rg2(h2, out_u2)
}

//P regulator for h1
task rg1_P(c_double h1) output(out_u1) state(){
release P_Rg1(h1, out_u1)
}

//P regulator for h2
task rg2_P(c_double h2) output(out_u2) state(){
release P_Rg2(h2, out_u2)
}

//drivers

//update rg1 input
driver update_rg1(sen_h1) output(c_double h1){
if c_true() then double_to_double(sen_h1,h1)
}
```

```
//update rg2 input
driver update_rg2(sen_h2) output(c_double h2){
if c_true() then double_to_double(sen_h2,h2)
}

//update command limit task input
driver update_comm_limit(out_u1,out_u2) output(c_double u1, c_double u2){
if c_true() then double2_to_double2(out_u1,out_u2,u1,u2)
}

//actuator update
driver update_act_pumps(out_u1,out_u2) output(c_double_arr commands){
if c_true() then double2_to_double_arr(out_u1,out_u2,commands)
}

//mode switch drivers
driver P_P_to_P_PI(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if P_P_to_P_PI_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver P_P_to_PI_P(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if P_P_to_PI_P_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver P_P_to_PI_PI(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if P_P_to_PI_PI_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver P_PI_to_P_P(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if P_PI_to_P_P_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver P_PI_to_PI_P(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if P_P_to_PI_P_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver P_PI_to_PI_PI(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if P_P_to_PI_PI_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver PI_P_to_P_P(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if PI_P_to_P_P_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver PI_P_to_P_PI(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if PI_P_to_P_PI_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver PI_P_to_PI_PI(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if PI_P_to_PI_PI_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver PI_PI_to_P_P(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if PI_PI_to_P_P_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}

driver PI_PI_to_P_PI(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if PI_PI_to_P_PI_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
```

```
}

driver PI_PI_to_PI_P(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2) output(){
if PI_PI_to_PI_P_cond(sen_e1,sen_e2,sen_e3,sen_g2,sen_s1,sen_s2)
then empty_switch_driver()
}
start P_P{

mode P_P() period 500{
//sensor update
senfreq 1 do sen_h1 offset 100;
senfreq 1 do sen_h2 offset 100;
senfreq 1 do sen_e1 offset 0;
senfreq 1 do sen_e2 offset 0;
senfreq 1 do sen_e3 offset 0;
senfreq 1 do sen_g2 offset 0;
senfreq 1 do sen_s1 offset 0;
senfreq 1 do sen_s2 offset 0;

//actuator update
actfreq 1 do act_pumps(update_act_pumps) offset 300;

//mode switch
exitfreq 1 do P_PI(P_P_to_P_PI);
exitfreq 1 do PI_P(P_P_to_PI_P);
exitfreq 1 do PI_PI(P_P_to_PI_PI);

floatfreq 1 do rg1_P(update_rg1);
floatfreq 1 do rg2_P(update_rg2);
}

mode P_PI() period 500{
//sensor update
senfreq 1 do sen_h1 offset 100;
senfreq 1 do sen_h2 offset 100;
senfreq 1 do sen_e1 offset 0;
senfreq 1 do sen_e2 offset 0;
senfreq 1 do sen_e3 offset 0;
senfreq 1 do sen_g2 offset 0;
senfreq 1 do sen_s1 offset 0;
senfreq 1 do sen_s2 offset 0;

//actuator update
actfreq 1 do act_pumps(update_act_pumps) offset 300;

//mode switch
exitfreq 1 do P_P(P_PI_to_P_P);
exitfreq 1 do PI_P(P_PI_to_PI_P);
exitfreq 1 do PI_PI(P_PI_to_PI_PI);

floatfreq 1 do rg1_P(update_rg1);
floatfreq 1 do rg2_PI(update_rg2);
}

mode PI_P() period 500{
//sensor update
senfreq 1 do sen_h1 offset 100;
senfreq 1 do sen_h2 offset 100;
senfreq 1 do sen_e1 offset 0;
senfreq 1 do sen_e2 offset 0;
senfreq 1 do sen_e3 offset 0;
senfreq 1 do sen_g2 offset 0;
senfreq 1 do sen_s1 offset 0;
senfreq 1 do sen_s2 offset 0;

//actuator update
actfreq 1 do act_pumps(update_act_pumps) offset 300;

//mode switch
exitfreq 1 do P_P(PI_P_to_P_P);
```

```
exitfreq 1 do P_PI(PI_P_to_P_PI);
exitfreq 1 do PI_PI(PI_P_to_PI_PI);

floatfreq 1 do rg1_PI(update_rg1);
floatfreq 1 do rg2_P(update_rg2);
}

mode PI_PI() period 500{
//sensor update
senfreq 1 do sen_h1 offset 100;
senfreq 1 do sen_h2 offset 100;
senfreq 1 do sen_e1 offset 0;
senfreq 1 do sen_e2 offset 0;
senfreq 1 do sen_e3 offset 0;
senfreq 1 do sen_g2 offset 0;
senfreq 1 do sen_s1 offset 0;
senfreq 1 do sen_s2 offset 0;

//actuator update
actfreq 1 do act_pumps(update_act_pumps) offset 300;

//mode switch
exitfreq 1 do P_P(PI_PI_to_P_P);
exitfreq 1 do P_PI(PI_PI_to_P_PI);
exitfreq 1 do PI_P(PI_PI_to_PI_P);

floatfreq 1 do rg1_PI(update_rg1);
floatfreq 1 do rg2_PI(update_rg2);
}
}
```

The *E code* generated by the compiler for the previous program is:

```
//output port initialization
call(init(out_u1))
call(init(out_u2))

//jump to start mode
jump(mode_address[P_P,0])

//mode P_P
//mode_address[P_P,0]
call(dev(sen_s2))
call(dev(sen_g2))
call(dev(sen_e3))
call(dev(sen_e2))
call(dev(sen_e1))
call(dev(sen_s1))
if(condition_P_P_P_PI,switch_address[P_P,0,P_PI,0,P_P_to_P_PI])
if(condition_P_P_PI_P,switch_address[P_P,0,PI_P,0,P_P_to_PI_P])
if(condition_P_P_PI_PI,switch_address[P_P,0,PI_PI,0,P_P_to_PI_PI])
jump(task_address[P_P,0])

//switch_address[P_P,0,P_PI,0,P_P_to_P_PI]
call(driver(P_P_to_P_PI))
jump(task_address[P_PI,0])

//switch_address[P_P,0,PI_P,0,P_P_to_PI_P]
call(driver(P_P_to_PI_P))
jump(task_address[PI_P,0])

//switch_address[P_P,0,PI_PI,0,P_P_to_PI_PI]
call(driver(P_P_to_PI_PI))
jump(task_address[PI_PI,0])

//task_address[P_P,0]
future(100,sensor_update_address[sen_h1,P_P,0])
future(100,sensor_update_address[sen_h2,P_P,0])
future(100,task_release_address[rg1_P,P_P,0])
```

```
future(100,task_release_address[rg2_P,P_P,0])
future(100,mode_address[P_P,1])
return

//sensor_update_address[sen_h1,P_P,0]
call(dev(sen_h1))
return

//sensor_update_address[sen_h2,P_P,0]
call(dev(sen_h2))
return

//task_release_address[rg1_P,P_P,0]
call(driver(update_rg1))
release(task(rg1_P))
return

//task_release_address[rg2_P,P_P,0]
call(driver(update_rg2))
release(task(rg2_P))
return

//mode_address[P_P,1]
future(100,mode_address[P_P,2])
return

//mode_address[P_P,2]
future(100,mode_address[P_P,3])
return

//mode_address[P_P,3]
call(driver(update_act_pumps))
call(dev(act_pumps))
future(100,mode_address[P_P,4])
return

//mode_address[P_P,4]
future(100,mode_address[P_P,0])
return

//P_PI mode
//mode_address[P_PI,0]
call(dev(sen_s2))
call(dev(sen_g2))
call(dev(sen_e3))
call(dev(sen_e2))
call(dev(sen_e1))
call(dev(sen_s1))
if(condition_P_PI_P_P,switch_address[P_PI,0,P_P,0,P_PI_to_P_P])
if(condition_P_PI_PI_P,switch_address[P_PI,0,PI_P,0,P_PI_to_PI_P])
if(condition_P_PI_PI_PI,switch_address[P_PI,0,PI_PI,0,P_PI_to_PI_PI])
jump(task_address[P_PI,0])

//switch_address[P_PI,0,P_P,0,P_PI_to_P_P]
call(driver(P_PI_to_P_P))
jump(task_address[P_P,0])

//switch_address[P_PI,0,PI_P,0,P_PI_to_PI_P]
call(driver(P_PI_to_PI_P))
jump(task_address[PI_P,0])

//switch_address[P_PI,0,PI_PI,0,P_PI_to_PI_PI]
call(driver(P_PI_to_PI_PI))
jump(task_address[PI_PI,0])

//task_address[P_PI,0]
future(100,sensor_update_address[sen_h1,P_PI,0])
future(100,sensor_update_address[sen_h2,P_PI,0])
future(100,task_release_address[rg1_P,P_PI,0])
future(100,task_release_address[rg2_PI,P_PI,0])
```

```
future(100,mode_address[P_PI,1])
return

//sensor_update_address[sen_h1,P_PI,0]
call(dev(sen_h1))
return

//sensor_update_address[sen_h2,P_PI,0]
call(dev(sen_h2))
return

//task_release_address[rg1_P,P_PI,0]
call(driver(update_rg1))
release(task(rg1_P))
return

//task_release_address[rg2_PI,P_PI,0]
call(driver(update_rg2))
release(task(rg2_PI))
return

//mode_address[P_PI,1]
future(100,mode_address[P_PI,2])
return

//mode_address[P_PI,2]
future(100,mode_address[P_PI,3])
return

//mode_address[P_PI,3]
call(driver(update_act_pumps))
call(dev(act_pumps))
future(100,mode_address[P_PI,4])
return

//mode_address[P_PI,4]
future(100,mode_address[P_PI,0])
return

//mode PI_P
//mode_address[PI_P,0]
call(dev(sen_s2))
call(dev(sen_g2))
call(dev(sen_e3))
call(dev(sen_e2))
call(dev(sen_e1))
call(dev(sen_s1))
if(condition_PI_P_P_PI,switch_address[PI_P,0,P_PI,0,PI_P_to_P_PI])
if(condition_PI_P_P_P,switch_address[PI_P,0,P_P,0,PI_P_to_P_P])
if(condition_PI_P_PI_PI,switch_address[PI_P,0,PI_PI,0,PI_P_to_PI_PI])
jump(task_address[PI_P,0])

//switch_address[PI_P,0,P_PI,0,PI_P_to_P_PI]
call(driver(PI_P_to_P_PI))
jump(task_address[P_PI,0])

//switch_address[PI_P,0,P_P,0,PI_P_to_P_P]
call(driver(PI_P_to_P_P))
jump(task_address[P_P,0])

//switch_address[PI_P,0,PI_PI,0,PI_P_to_PI_PI]
call(driver(PI_P_to_PI_PI))
jump(task_address[PI_PI,0])

//task_address[PI_P,0]
future(100,sensor_update_address[sen_h1,PI_P,0])
future(100,sensor_update_address[sen_h2,PI_P,0])
future(100,task_release_address[rg1_PI,PI_P,0])
future(100,task_release_address[rg2_P,PI_P,0])
future(100,mode_address[PI_P,1])
```

```
return

//sensor_update_address[sen_h1,PI_P,0]
call(dev(sen_h1))
return

//sensor_update_address[sen_h2,PI_P,0]
call(dev(sen_h2))
return

//task_release_address[rg1_P,PI_P,0]
call(driver(update_rg1))
release(task(rg1_PI))
return

//task_release_address[rg2_P,PI_P,0]
call(driver(update_rg2))
release(task(rg2_P))
return

//mode_address[PI_P,1]
future(100,mode_address[PI_P,2])
return

//mode_address[PI_P,2]
future(100,mode_address[PI_P,3])
return

//mode_address[PI_P,3]
call(driver(update_act_pumps))
call(dev(act_pumps))
future(100,mode_address[PI_P,4])
return

//mode_address[PI_P,4]
future(100,mode_address[PI_P,0])
return

//mode PI_PI
//mode_address[PI_PI,0]
call(dev(sen_s2))
call(dev(sen_g2))
call(dev(sen_e3))
call(dev(sen_e2))
call(dev(sen_e1))
call(dev(sen_s1))
if(condition_PI_PI_P_PI,switch_address[PI_PI,0,P_PI,0,PI_PI_to_P_PI])
if(condition_PI_PI_PI_P,switch_address[PI_PI,0,PI_P,0,PI_PI_to_PI_P])
if(condition_PI_PI_P_P,switch_address[PI_PI,0,P_P,0,PI_PI_to_P_P])
jump(task_address[PI_PI,0])

//switch_address[PI_PI,0,P_PI,0,PI_PI_to_P_PI]
call(driver(PI_PI_to_P_PI))
jump(task_address[P_PI,0])

//switch_address[PI_PI,0,PI_P,0,PI_PI_to_PI_P]
call(driver(PI_PI_to_PI_P))
jump(task_address[PI_P,0])

//switch_address[PI_PI,0,P_P,0,PI_PI_to_P_P]
call(driver(PI_PI_to_P_P))
jump(task_address[P_P,0])

//task_address[PI_PI,0]
future(100,sensor_update_address[sen_h1,PI_PI,0])
future(100,sensor_update_address[sen_h2,PI_PI,0])
future(100,task_release_address[rg1_PI,PI_PI,0])
future(100,task_release_address[rg2_PI,PI_PI,0])
future(100,mode_address[PI_PI,1])
return
```

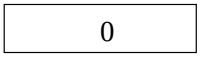
```
//sensor_update_address[sen_h1,PI_PI,0]
call(dev(sen_h1))
return

//sensor_update_address[sen_h2,PI_PI,0]
call(dev(sen_h2))
return

//task_release_address[rg1_PI,PI_PI,0]
call(driver(update_rg1))
release(task(rg1_PI))
return

//task_release_address[rg2_PI,PI_PI,0]
call(driver(update_rg2))
release(task(rg2_PI))
return

//mode_address[PI_PI,1]
future(100,mode_address[PI_PI,2])
return

//mode_address[PI_PI,2]
future(100,mode_address[PI_PI,3])
return

//mode_address[PI_PI,3]
call(driver(update_act_pumps))
call(dev(act_pumps))
future(100,mode_address[PI_PI,4])
return

//mode_address[PI_PI,4]
future(100,mode_address[PI_PI,0])
return
```

Appendix B

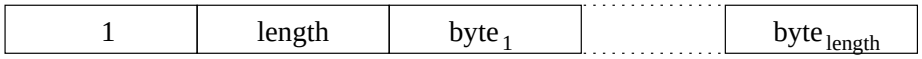
Control Protocol

Control Protocol valid packets:

- **synchronize request** – it is send by the client and the server will respond with the same packet:

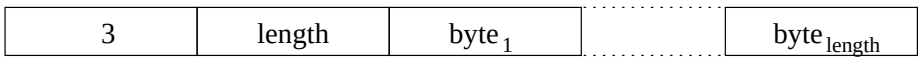


- **command** – this represents a command packet, data has a variable length:



- **set period** – this is sent by the client and tells the server what is the period for sending the new values of the signals, the length of data is constant and is 2:

- **new values** – this is a packet that is send by the server and represents new signals values it has a variable length:



- **net delay** – this is a packet that it is used to determine the delay introduced by the network, the client will send this packet and the server will response with a packet that has the same type but no data:

4	ID

- **error** – this is an error packet, it can be send both by the server and client and it has only one byte of data, that will represent the error code:

5	error code

- **stop** – this can be send both by the server or client and tells the partner that it is going down, it has no data:

6

- **sensor request** – this is send by the client, and tells the server to send the new signals values, it has no data:

7

- **regulator request** – this is send by the client and tells the server that the client wants to be a regulator, initially the client is considered to be in the view mode, if there is already a regulator then the server sends an error packet, else responses with the same packet, this packet has a single byte of data representing the regulator type:

8	RG type

- **disconnect regulator** – this is send by the client and tells the server that if it is a regulator then it wants to be disconnected, this packet has no data:

9

Appendix C

3TS Simulator User Guide

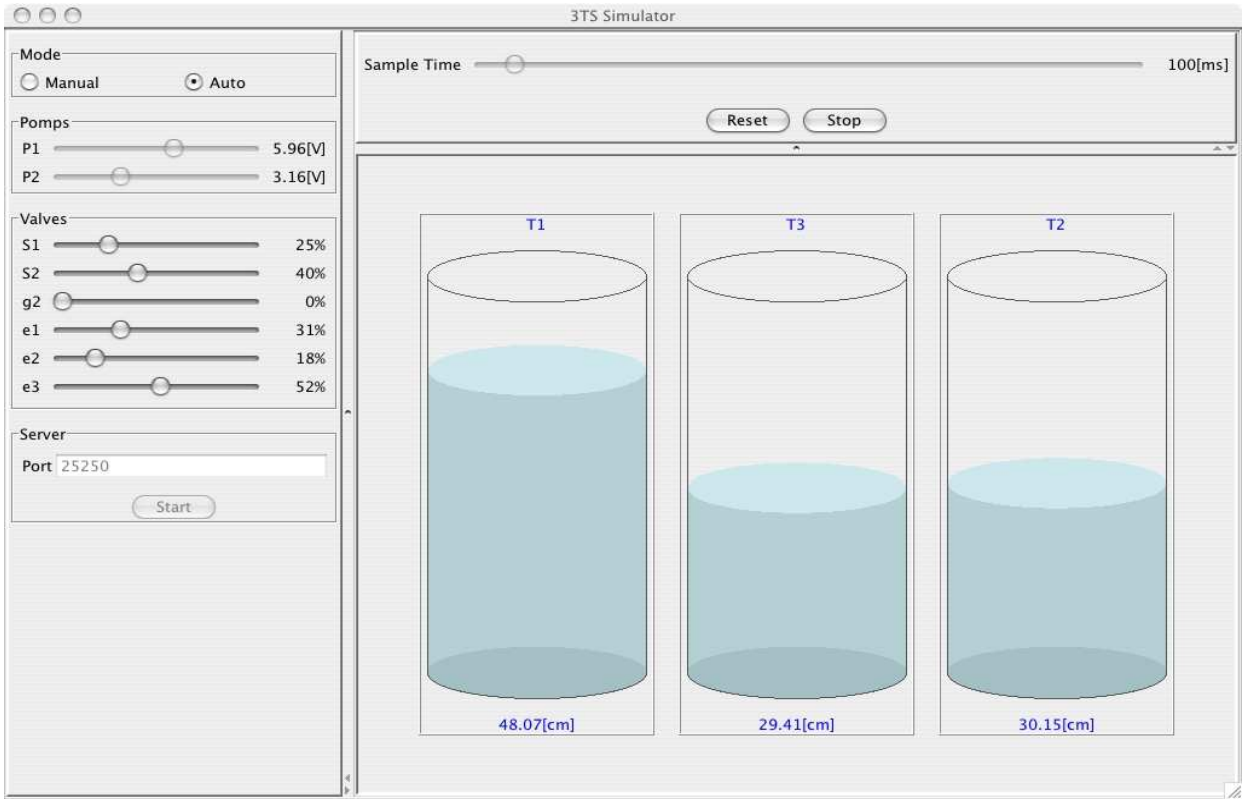


Figure C.1: 3TS Simulator screenshot

As it can be seen from the screenshot **Fig. C.1** the application main window is split in 3 panels:

control panel – this panel allows the user to:

- set the functioning mode, the mode can be *manual* or *auto*, if the simulator will be use without a regulator then it should be used in manual mode, else it should be in

auto mode; in auto mode the user will not be able to control the command given to the pumps;

- change the command given to pumps, this is possible only in manual mode;
- change the opening coefficient of the valves;
- set the port on which the server will listen and start the server;

simulation control panel – this panel allows the user to:

- set the simulation sample time;
- start, stop, or restart the simulation

simulation panel – this panel takes no input from the user it is just a viewer for the simulation.

Appendix D

3TS Controller User Guide

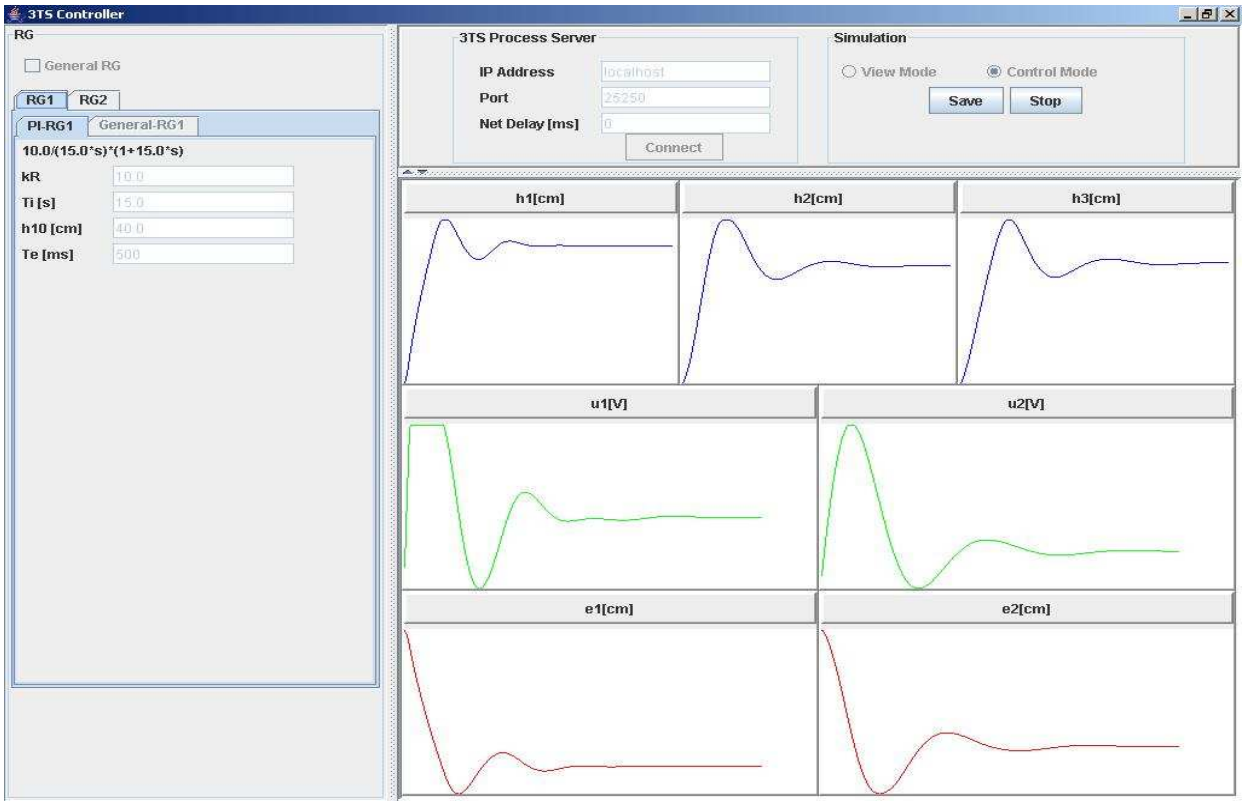


Figure D.1: 3TS Controller screenshot

As it can be seen from the screenshot **Fig. D.1** the main application window is split in 3 panels:

control panel – allows the user to:

- set the address and port on which the server is listening and connect to the server, if the connection to the server was not establish then the user can't do any thing since

all others controls are disabled until the connection is established;

- switch between the view and control mode, in view mode the program receives data from the server and plots it, while in the control mode the program actually controls the 3TS simulated process;
- start or stop the controller and save simulation data; simulation data will be saved in two formats:(1) each signal will be saved in a file located in the application directory, named after the signal, and having the extension *dat*, the file will contain all the values for the signal separated by semicolon, and (2) each signal will be saved in a *jpg* image file, this file will be also locate din the application directory and will be named after the signal it will represent ;

controller panel – this panel will be activated only if the program is in control mode and the controller is stop, the user can choose between two types of controllerss: (1)a PI controller and (2) a general controller, in both cases the user will be able to set the reference, controller parameters,and sample time ;

view panel – in this panel are plotted the main signals, if one double clicks on one of the diagrams then a dialog will be showed for that diagram (**Fig. D.2**).

In **Fig. D.2** can be seen the *Diagram Dialog* for h_2. The dialog is split in two parts, the first one is represented by a table that will contain the quality indicators for the signal, and the second part where the signal diagram is drawn. If the user selects an indicator from the table, then a line will be drawn on the diagram to represent that indicator, there can be more the one indicators selected at the same time. For instance in the figure there are selected h_{inf} and t_r, this will result in drawing to lines on the diagram, a horizontal one for h_{inf} and a vertical one for t_r. There is also drawn a supplementary line representing the reference for the computed time indicators. There is also a save button that will save the diagram in a *jpg* image file, the file will be located in the application directory and will be named after the signal it is representing plus the suffix *fancy*.

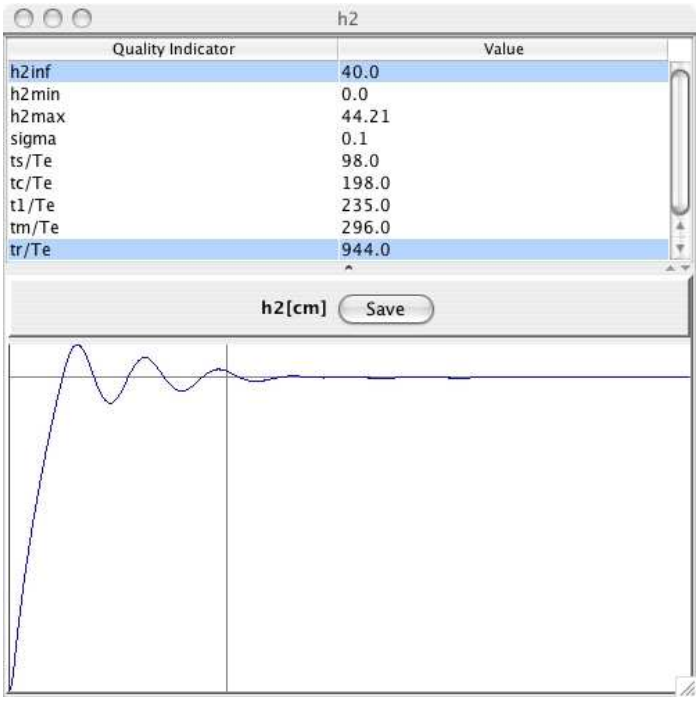


Figure D.2: Diagram Dialog screenshot

Appendix E

Matlab programs and Simulink schemes

1TS-MM - Simulink schema for 1TS plant and the Matlab program associated with it:

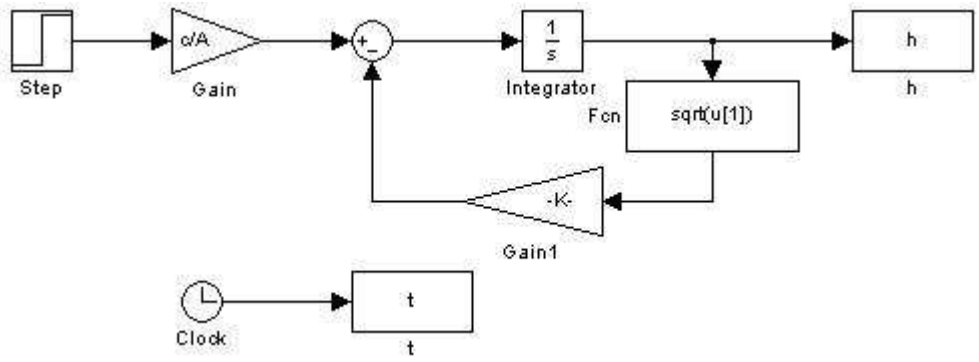


Figure E.1: 1TS Simulink schema

Matlab const.m program:

```
%File with constants for mathematical modeling of the 1TS plant

%Inputs
c=0.0000155;            %m3/(V*sec)
uc=10; %V
u=0.5;

%Constants
A=0.0154; %m2;
S=0.00005; %m2;
g=9.81; %m/sec2
```

2TS-MM - Simulink schema for 2TS plant and the Matlab program associated with it:

Matlab const.m program:

```
c=0.0000155;            %m3/(V*sec)
uc=10; %V
```

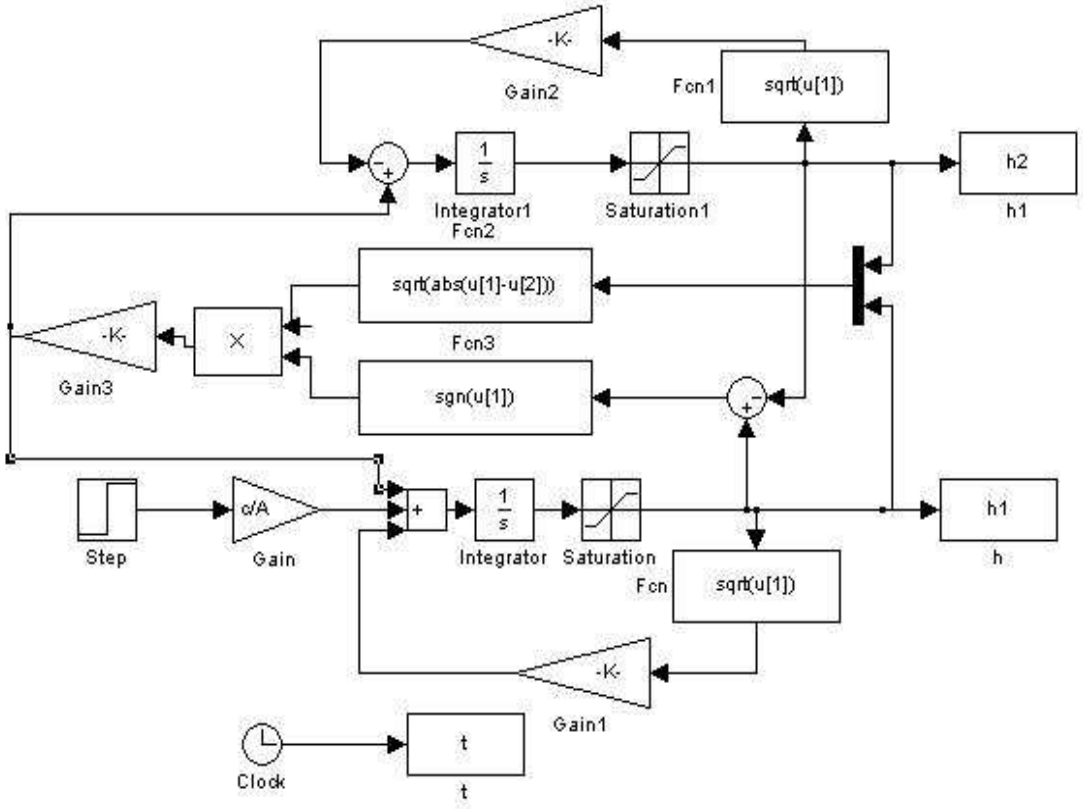


Figure E.2: 2TS Simulink schema

```
u1=0;
u2=0;
us=0;

A=0.0154; %m2;
S=0.00005; %m2;
g=9.81; %m/sec2
```

3TS-MM - Simulink schema for 3TS plant and the Matlab program associated with it:
Matlab const.m program:

```
c=0.0000155;            %m3/(V*sec)
A=0.0154; %m2;
S=0.00005; %m2;
g=9.81; %m/sec2
Km=27;

ue1=1;
ue3=1;
us1=1;
ue2=1;
ug2=1;
us2=1;

uc1=10;
uc2=10;
```

1TS-Controller - Simulink schema for 1TS plant with P Controller and the Matlab program associated with it:
Matlab const.m program:

```
c=0.0000155;            %m3/(V*sec)

u=0;

A=0.0154; %m2;
S=0.00005; %m2;
g=9.81; %m/sec2
Km=27;

h0=0.3; %m
u0=0

%regulator P
tr=60;
kR=3*A/(tr*c
```

2TS-Controller - Simulink schema for 2TS plant with PI Controller and the Matlab program associated with it:
Matlab const.m program:

```
c=0.0000155;            %m3/(V*sec)
A=0.0154; %m2;
S=0.00005; %m2;
g=9.81; %m/sec2
Km=27;
```

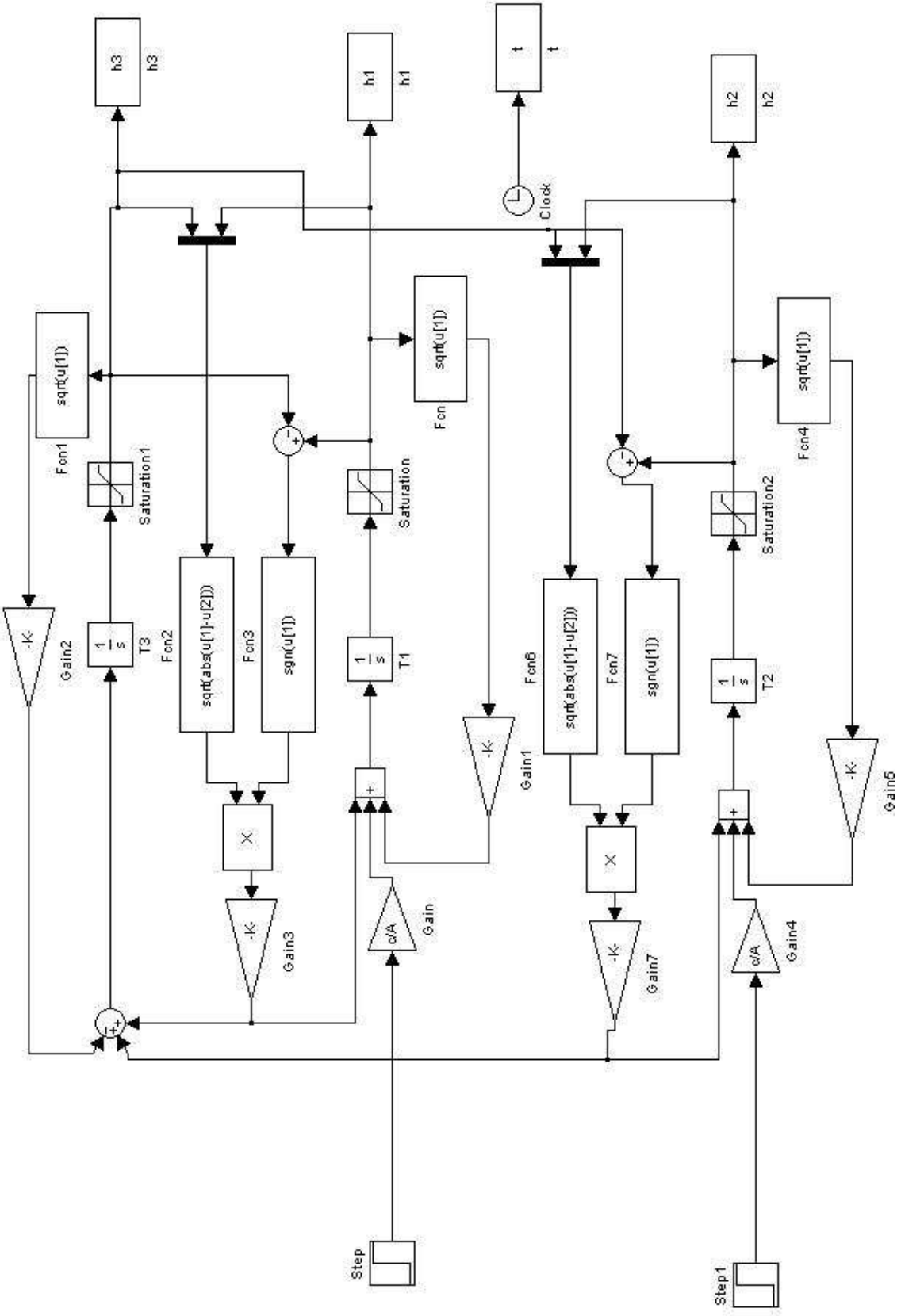


Figure E.3: 3TS Simulink schema

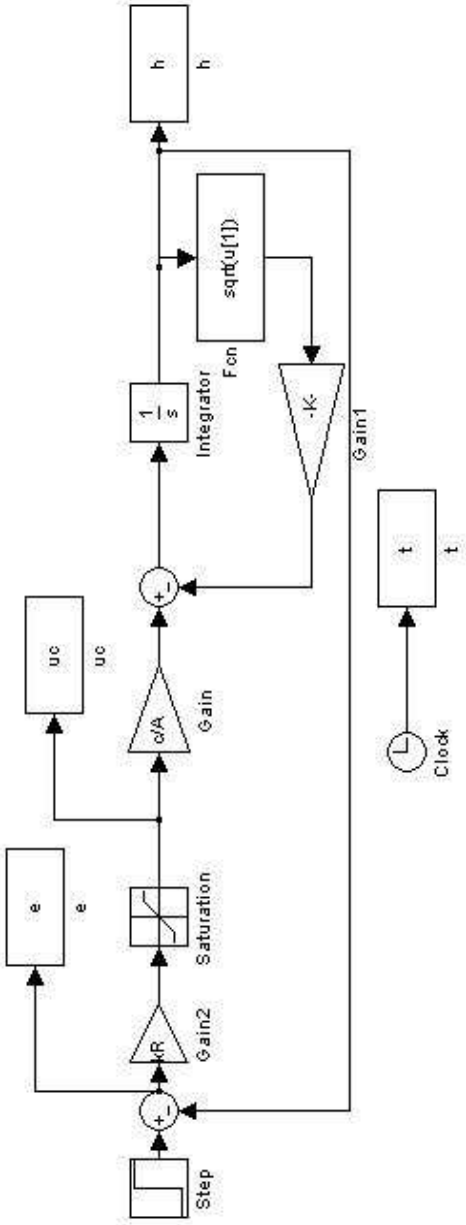


Figure E.4: 1TS with P Controller Simulink schema

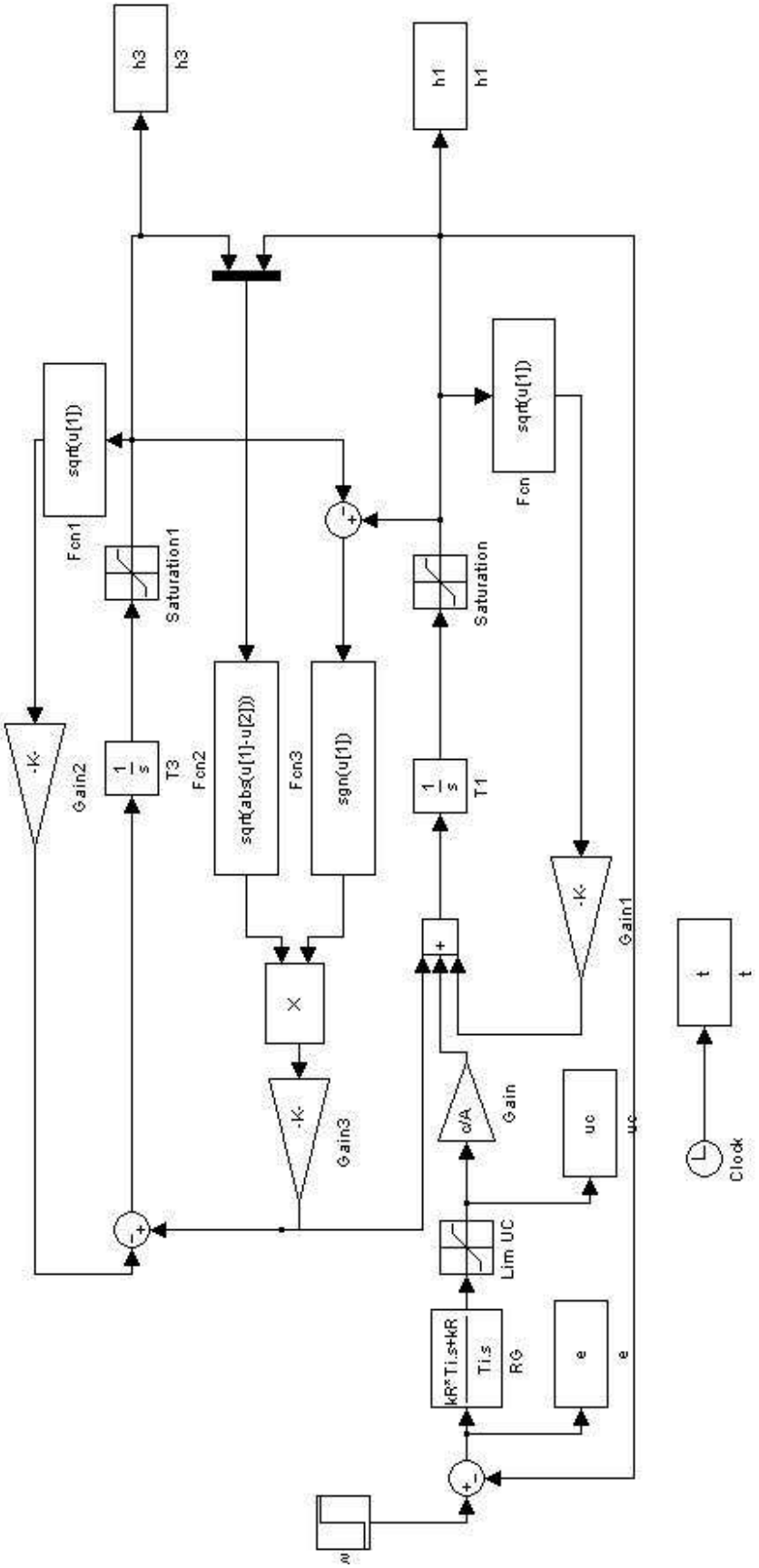


Figure E.5: 2TS with PI Controller Simulink schema

```
h10=0.4;
h30=0.3;
ue10=0.5;
ue30=0.5;
us10=0.5;

%A matrix
a11=-S/A*sqrt(2*g)/2*(us10*sign(h10-h30)/sqrt(abs(h10-h30)) + ue10/sqrt(h10));
a12=us10*S/A*sign(h10-h30)*sqrt(2*g)/2/sqrt(abs(h10-h30));
a21=us10*S/A*sign(h10-h30)*sqrt(2*g)/2/sqrt(abs(h10-h30));
a22=-S/A*sqrt(2*g)/2*(us10*sign(h10-h30)/sqrt(abs(h10-h30))+ue30/sqrt(h30));

%b matrix
b1=c/A;
b2=0;

%c matrix
c1=1;
c2=0;

A=[a11 a12;a21 a22];
b=[b1;b2];
c=[c1 c2];
d=[0];

%compute TF
[bb,aa]=ss2tf(A,b,c,0,1);
H11=tf(bb,aa);

bb1=[bb11 bb10];
aa1=[aa12 aa11 aa10];

H111=tf(bb1,aa1)

%controller parameters
Ti=15;
kR=10;
br0=kR;
br1=kR*Ti;
ar0=0;
ar1=Ti;
ar=[ar1 ar0];
br=[br1 br0];
Hr=tf(br,ar)
H0=Hr*H11

%SRA simulation parameter
ue1=0.5;
ue3=0.5;
us1=0.5;
%overridde values do not remove
c=0.0000155;             %m3/(V*sec)
A=0.0154; %m2;
S=0.00005; %m2;
g=9.81; %m/sec2
Km=27;
```

3TS with controllers - Simulink schema for 3TS plant with PI Controllers and the Matlab program associated with it:

Matlab const.m program:

```
c=0.0000155;             %m3/(V*sec)
A=0.0154; %m2;
S=0.00005; %m2;
```

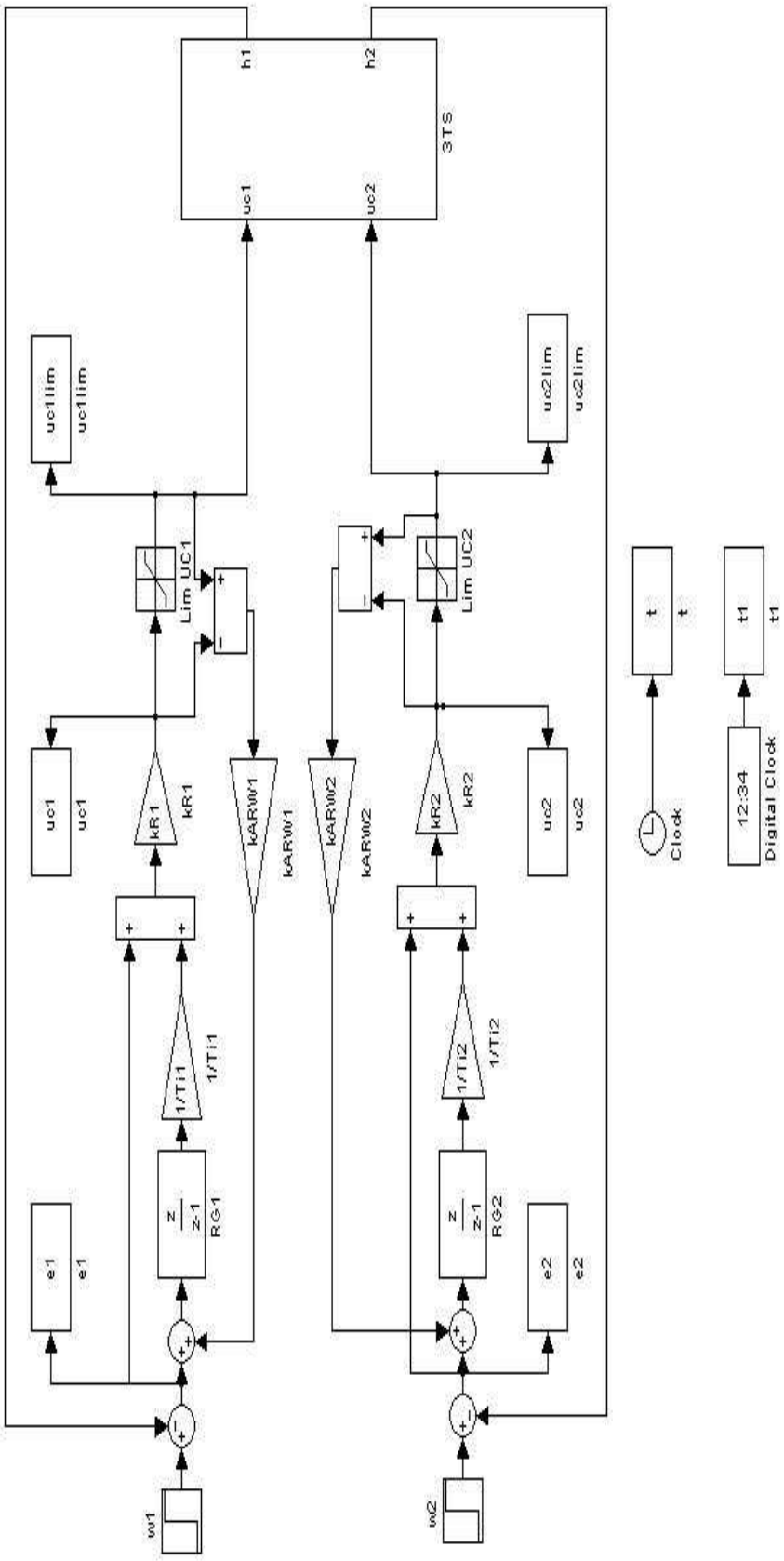


Figure E.6: 3TS with PI Controller Simulink schema

```
g=9.81; %m/sec2
Km=27;

ue1=0.5;
ue3=0.5;
us1=0.5
ue2=0.5;
ug2=0.0;
us2=0.5;

Ti1=15;
kR1=10;

Ti2=15;
kR2=5.3;

h10=0.5;
h20=0.4;

Te=0.5;

kARW1=1/Ti1
kARW2=1/Ti2
```

List of Figures

List of Tables

List of code sequences

Bibliography

[1] Etienne Gagnon. Sablecc, an object-oriented compiler framework, 1998.

[2] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. GIOTTO: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[3] T. A. Henzinger and C. M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proceedings of Programming Language Design and Implementation*, pages 315–326. ACM Press, 2002.

[4] T. A. Henzinger, C. M. Kirsch, R. Majumdar, and S. Matic. Time-safety checking for embedded programs. In *EMSOFT 02: Embedded Software*, Lecture Notes in Computer Science 2491, pages 76–92. Springer-Verlag, 2002.

[5] j2se. http://java.sun.com/j2se/.

[6] Stefan Preitl. Control engineering course notes.

[7] Stefan Preitle and Radu-Emil Precup. *Introduction in Control Engineering*. Editura Politehnica, 2001.