Design and Implementation of Analysis and Optimization Tool for Embedded Systems aiming at Increased Interoperability

Adam Kordianowski

Supervised by
Paul Pop
Andrzej Napieralski

Kongens Lyngby 2010

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Embedded systems are nowadays present in every area of people lives. Their complexity is constantly increasing and they have tight requirements in terms of performance, energy consumption, reliability, size and cost. Hence, the task of designing embedded systems is getting more difficult.

Designers today use complex tools that help them in synthesizing an implementation that fulfills all the requirements. Several tools have to work in conjunction to produce the desired result. In this thesis we evaluate three interoperability approaches: (i) full integration at the source-code level using shared data structures, (ii) tool integration using XML files and (iii) tool integration using object-oriented databases for data exchange.

To evaluate the interoperability approaches, we have focused on three design tasks: (i) mapping, (ii) voltage scaling and (iii) schedulability analysis. The first design task is to find the best mapping of functions to the processing elements in the given architecture. Dynamic voltage scaling, which is the second design task, allows to choose appropriately voltage levels for all functions so that the power consumption is minimized without performance or reliability degradation. The third design task, schedulability analysis, is used to determine whether all the functions in the system will meet their desired deadlines.

We have designed and implemented an analysis and optimization tool that performs all three design tasks. The optimization is performed using a Simulated Annealing meta-heuristic. The integration of the tool modules implementing the

analysis, design transformation and optimization functions has been performed using all three mentioned interoperability approaches. We have discussed both advantages and disadvantages of each considered interoperability approach and have evaluated their impact on the performance of the implementation in terms of runtime and the quality of the produced solution.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Embedded system design

Embedded systems have become much more complex over last years. Following the growth of their complexity, models used in design and methodologies have to change as well.
As described in [18], first (in 1970s) embedded systems designers used transistor models. Such way of presenting the systems was suitable, however, only for simple systems, consisting limited number of transistors. As the systems grew, the more abstract models were used. And so throughout the years the design models changed through gate-level models (1980s), register transfer level (RTL in 1990s), to designing at the system level nowadays. The difference between those models lays in the level of abstraction that hides lower level details from the designer, therefore allowing him not to be concerned about them.
When it comes to design methodology used in the embedded system, it can be divided into three main levels:

- functional level
- mapping level
- implementation level

At functional level the behaviour and main requirements of the system are described. Here also the verification of the models is performed making sure that all constraints will be satisfied (e.g. system will not enter the deadlock). At the same time multiple hardware architectures, that will enable the system to fulfill its task, are composed.
Further the mapping of the functional model onto the hardware architecture is done. In this step designer performs the design space exploration attempting to choose the best of composed architectures that will fulfill performance (speed, power consumption), and other requirements.
Last level in the design methodology is implementation of the embedded system. Here the lower-level specification of the system is created. This level may not necessarily end by having ready product. It rather generates the specification and requirements for next phase in development.
Note that all those levels in the methodology may give the input to the higher level, taking design back to that level. Such feedback from lower levels, where more details are available and some of the problems become visible, aims at creating better final design, and so the product.
As it is crucial to have very low time-to-market of the embedded system, mistakes and possible problems should be found and fixed as early as possible in the design process. Fault found at the implementation level, means in many cases going back to the higher design levels and fixing it there. This results in necessity to redo the earlier stages, which in turn results in the longer design process and higher time-to-market.
There are plenty of tools helping the designers in their task. Lavagno and Passerone in [18] mention tools for functional design like Simulink ([19]) or IBM Rational Rose RealTime ([14]). They also give examples of function-architecture codesign tools like Artisan Software Real Time Studio ([22]).

1.2 Motivation

As described in previous section, the design of the embedded system is the complex and challenging process. Designer's decisions during first stages highly influence the overall design, and, if done wrongly, have a significant impact on the product success.
One of the most important and difficult decisions is the mapping of functional tasks to the processing elements (NP-hard problem, as discussed in [25]). This decision impacts not only the schedulability of the system, but also the power consumption and reliability, as different processors may have different energy characteristics and mean time between faults.
As embedded systems become more portable with continuous grow of complexity (e.g. mobile phones), the battery life becomes the issue. Therefore, together

with mapping of tasks, designers may want to use dynamic voltage and frequency scaling (DVFS), and make some tasks being executed at lower voltage making the energy consumption smaller as well. Such decision, on the other hand, makes the tasks being executed for longer time (voltage is proportional to the frequency), and be more vulnerable to faults (less reliable). As discussed in [30] the static *reliability-aware power management (RA-PM)* is again NP-hard.
Note that both mapping and deciding on voltage levels for tasks have to be done together, as one influences another, and both impact the schedulability, as well as reliability and power consumption.
This thesis aims to create the design tool that will cope with above two problems. Furthermore, an attempt is made to create the framework that will allow joining multiple tools, and make them cooperate with each other. Such framework would allow using simple tools that work together to achieve complex task, instead of creating complex systems that are capable of solving limited number of problems.

1.3 Overview

Following chapters will describe work done throughout this project. Following this chapter, in chapter 2 we will cover preliminary knowledge that will be used in this thesis, namely used theoretical model of embedded system (section 2.1) and meta-heuristics used for in optimization process (section 2.2).
In chapter 3 we will design data structures that should represent the embedded system model. We will discuss how tasks (section 3.2), processing elements (section 3.3) and finally mapping of software to architecture is represented (section 3.4).
Further in chapter 4 we will take a closer look at possibilities that we have when creating interoperability framework. We will first look at technologies allowing data sharing among the tools (section 4.1), and communication between them (section 4.2).
Having discussed all aspects needed for design of both integrated tool and interoperable tool set, let us in chapter 5 make the actual design. First in section 5.1, let us distinguish independent modules, that could run as separate applications in case of tool set. This section also already includes design aspects of integrated tool. All additional information for designing it are provided in further section 5.2. That chapter concludes with discussion of different aspects of designed tool set (section 5.3).
Next, in chapter 6 we move our attention to actual implementation. We, however, limit our attention here to only some major aspects and decisions that were done during this phase of the project. Full understanding of all implementation details is, therefore, left for looking at actual source code. In this chapter we will

first justify choice of programming language used (section 6.1). Further we will look first at implementation of embedded system representation (section 6.2), which was designed in chapter 3. Chapter will conclude with implementation issues regarding interoperability framework (section 6.3).
In last but one, chapter 7, we will compare all three approaches that were taken to achieve the same task, namely integrated tool, and two tool sets, one using files and second using database. We will also discuss reasons for obtaining such results.
This thesis will conclude with providing in chapter 8 general ideas for future development and improvement for tools that were made.

Chapter 2

Preliminaries

In this chapter the preliminaries for the thesis will be presented. This includes defining of system models (in 2.1), which will cover both hardware and software architecture. Afterwards, the theory regarding the optimization method shall be presented, together with the heuristic algorithm actually used (2.2).

2.1 System model

2.1.1 Hardware architecture model

Let us consider the hardware platform first. It consists of multiple processing elements (PE_i), connected with each other by communication channels e.g. buses. Each processing element is capable of dynamic voltage scaling and has well-defined set of operating modes, $M = \{\Lambda_0, \Lambda_1, ..., \Lambda_n\}$. Each operating mode is defined by: $\Lambda_k = \{s_k, f_k, p_k\}$, where:

- s_k is the speed of the processing element when running at operating mode k measured in Hz
- f_k is relative voltage level of the operating mode

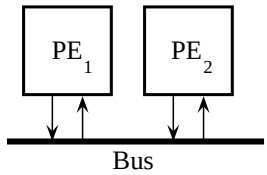

Figure 2.1: Architecture

- p_k being the power consumption when processing element runs at given operating mode.

Example architecture is shown in the figure 2.1. Each processing element runs the real-time operating system with the static priority scheduler (e.g. rate monotonic). Any faults within the execution of the task can be detected after it is finished.

2.1.2 Fault model

During their operations, embedded systems may be affected by various faults. These faults may happen due to hardware errors (usually permanent faults), electromagnetic fields affecting the processing elements and/or memory, or others. Faults can be divided into two categories: permanent and transient. When permanent fault occurs, it can be removed only by changing the affected hardware. Until that the fault will be present with every subsequent execution. Transient faults, on the other hand, usually disappear before the next execution of the task. As the transient faults happen more often ([30]), focus of this thesis will be on them.
Transient faults follow the Poisson distribution with the rate λ. This will be referred as the fault rate. Fault rate as described in [31] is affected by the operating mode (voltage level) and given by formula:

$$\lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} \tag{2.1}$$

Response time analysis (2.1.4) also require minimum time between faults (also referred to as the fault period). Fault period is given by the designer and corresponds to the application reliability goal (see section 2.1.5). If time between two faults happening is not less than the fault period, then the reliability goal for the application shall be met. This follows the approach presented in [8].

2.1.3 Application model

Having architecture and fault model, let us move to the application model. The application consists of set of tasks. For simplicity of the response time analysis let those tasks be independent and not share any resources. These two assumptions, however, can be loosen, if more advanced analysis techniques shall be applied.
Each task of the application, $P_i = \{c_i, T_i, D_i\}$, is defined by three parameters:

- c_i being the number of cycles required to execute the task in the worst case (for simplicity we assume that this value is independent of processing element on which the task is running)
- task period, T_i, measured in seconds
- deadline, D_i, for the task to be executed.

Having worst-case execution cycles, we can calculate rather simply worst-case execution time, C_i^k, using equation:

$$C_i^k = \frac{c_i}{s_k} \tag{2.2}$$

In case the system detects an error during the task execution, the task is re-executed at the maximum speed. Priority statically assigned to this task is not changed.

2.1.4 Response time analysis

One of most important requirement for the embedded systems is that all tasks have to meet their deadlines, i.e. in the worst-case scenario their execution must terminate within the deadline. Analysis that verifies fulfilling of this requirement is response time analysis.
As described in [5] we can find the worst-case response time of the task using the following formula:

$$r_i^k = C_i^k + B_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i^k}{T_j} \right\rceil C_j^{k_j}$$

As already assumed, tasks do not share any resources. Therefore, we can take $B_i = 0$[1].
Following the approach presented in [8] we assume that an error can happen in any of the tasks, however, not more often than a fault period T_F (see 2.1.2). Response time calculation taking this into account is as follows:

$$r_i^k = C_i^k + \sum_{j \in hp(i)} \left\lceil \frac{r_i^k}{T_j} \right\rceil C_j^{k_j} + \left\lceil \frac{r_i^k}{T_F} \right\rceil \mathbf{max}_{j \in hp(i) \cup i} \left(C_j^k \right) \tag{2.3}$$

Let us also define the degree of schedulability, as presented in [20]. This will help us in driving the heuristics towards better solutions in case the current one is not schedulable (see 2.2). Degree would be then defined as:

$$degree = \begin{cases} c_1 = \sum_{i=0}^{n} \mathbf{max}(0, r_i - D_i) & \text{if } c_1 > 0 \\ c_2 = \sum_{i=0}^{n} r_i - D_i & \text{if } c_1 = 0 \end{cases} \tag{2.4}$$

Note that in case of the unschedulable system, value of degree of schedulability is always positive (condition for first equation is satisfied, and there exists at least one task for which $r_i - D_i$ is greater than 0; schedulable tasks in this sum do not count, as $r_i - D_i$ for them is negative).
For schedulable systems, value of degree of schedulability is always non-positive (negative or zero).

2.1.5 Reliability of application

As already described in section 2.1.2, the designer of the system specifies the reliability goal: value representing the minimum reliability that must be met by the embedded system.
To calculate the reliability of the application let us define first the reliability of the task as in [21]. Reliability of the task is the probability that it executes successfully.

$$R_i = e^{-\lambda(f_k) C_i^k} \tag{2.5}$$

In case of faults the tasks can be re-executed. The reliability of the task with the possibility of re-execution is the probability of any execution of the task to be successful. In general the task can be re-executed multiple (k_f) times. As the re-executed task is run at the maximum speed, the $\lambda(f)$ is in fact λ_0.

[1] In case of tasks sharing resources the appropriate resource sharing protocol would have to be set (like Priority Inheritance Protocol or Priority Ceiling Protocol); then B_i is calculated accordingly. See [9] for details.

The reliability of the re-execution can be therefore calculated using following formula:

$$R_i = 1 - \left(1 - e^{-\lambda_0 C_i}\right)^{k_f} \left(1 - e^{-\lambda(f_k) C_i^k}\right) \tag{2.6}$$

In case of important tasks, the system designer may want some them be replicated to another processing element or elements. In such a case both replicas may be voltage scaled differently. The general formula for the reliability of replicated tasks is as follows:

$$R_i = 1 - \prod \left(1 - e^{-\lambda_{f_{k_i}} C_i^{k_i}}\right) \tag{2.7}$$

Reliability of application is the probability of all tasks executed successfully. It is therefore product of reliabilities of all tasks:

$$R = \prod R_i \tag{2.8}$$

Normal convention for providing the value of reliability is to specify number of nines that must follow the decimal point. And therefore the reliability: 0.9999994 is referred to as *six nines (and four).*

2.1.6 Energy consumption

When it comes to the energy consumption, let us first remind well-known formula from physics that energy is product of power and time.

$$E = Pt$$

This formula can be directly used to calculate the energy consumed by the task, as we have the power consumption for each operating mode that task could run at, as well as its worst-case execution time, when run at this operating mode. Note, however, that in such a case the tasks that run rarely but for long time would dominate the energy consumption over the tasks that run for shorter time but frequently. As this is not the case in real-life systems, we shall make some adjustments to the formula. It should take into consideration also how often the tasks run comparing to others.
To do this, let us first define the application period, (T_{APP}), as the smallest common multiple of all the tasks periods. Then let us define for each task the coefficient that will catch how many times the task is executed within the application period:

$$\varphi_i = \frac{T_i}{T_{APP}} \tag{2.9}$$

We shall, therefore, calculate the energy consumption for the application period. Energy consumption for the task is given by the formula:

$$E_i = p_i^k C_i^k \varphi_i \tag{2.10}$$

Total energy consumption is the sum of energies consumed by all the tasks within the application period:

$$E = \sum E_i \tag{2.11}$$

Note that when calculating the energy consumed by the system, we assume that no faults occur. We shall, therefore, try to minimize the best-case energy consumption.

2.2 Optimization and meta-heuristics

As described in section 1.2, the tool designed and implemented as part of this thesis attempts to solve NP-hard problems, namely mapping application to the hardware architecture, and setting statically the voltage levels for all tasks within the application.
To make it possible to solve problems from the NP-hard set, tool must perform the design space exploration attempting to find the optimal solution for given problem. To achieve this use of meta-heuristic algorithms is necessary. This will not guarantee finding the optimal solution, but such guarantee cannot be given in reasonable amount of time. In this thesis the simulated annealing (first presented in [17]) is used for this purpose.
In general meta-heuristic algorithms aim at finding the minimum or maximum value of the function of multiple independent variables. This function is usually called the *cost function*, and this name will also be used in this thesis. Cost function should capture how good the current solution is i.e. how far is it from the optimal one. This makes defining the right cost function crucial for finding solution close to optimal.
Very important thing for cost function is that it must catch whether the current solution satisfy the constraints or not. If any of the constraint in is not met (solution is not feasible), the cost function should give appropriate penalty, therefore making it far from the optimal one. This will prevent such solutions to be considered, even if all other parameters make the value of the functions close to the optimal value. Furthermore, as already discussed in section 2.1.4, cost function should drive the algorithm towards better solutions even if the current solution is not feasible. This means that the value of cost function in case any constraint not being met should not be the same in all cases.
In our case, two constraints are present:

- system must be schedulable
- reliability goal must be satisfied

In case of system being schedulable, as already mentioned, degree of schedulability captures the penalty size. In case of reliability goal, the size of penalty may be dependent on the difference from the reliability goal (similarly to degree of schedulability).

2.2.1 Simulated annealing

Simulated annealing takes its name from the metallurgical process: annealing, in which the metal is repeatably heat and cooled, aiming at creating near-perfect crystal structure (see [28]).
The algorithm uses the neighbourhood search technique. In this technique the move is done always to the neighbour of the current solution. Algorithm terminates when it reaches the maximum number of failed attempts to find better solution then the current.
The algorithm is given in Algorithm 1.

As one may see, simulated annealing uses the randomize method to look for the better solution in the neighbourhood. Moreover, when looking for better solutions in the neighbourhood of current one, it may accept, under certain conditions, move to worse solution (of course without updating the best solution found). Possibility of such moves is reduced with time, as the temperature is lowered (this is the similarity to the annealing process in metallurgy, in which with lower temperature the possibility for an particle to move in crystal structure gets smaller).

Algorithm 1 Simulated annealing algorithm

{ $maxNA$ - maximum number of attempts to find the better solution among the neighbours
x - current best solution
f - cost function
T - current temperature
ϵ - cooling rate }

input
$maxNA, x, T, \epsilon$

while $NA \neq maxNA$ **do**
  for $i = 0$ to TL **do**
    $x_{new} \leftarrow random\_neighbour(X)$
    $\Delta = f(x) - f(x_{new})$
    if $\Delta > 0$ **then**
      $NA \leftarrow 0$
      $x \leftarrow x_{new}$
      if $f(x) > f(x_{best})$ **then**
        $x_{best} \leftarrow X$
      end if
    else if $e^{\Delta/T} > random\_real(0, 1)$ **then**
      $NA \leftarrow 0$
      $x \leftarrow x_{new}$
    else
      $NA \leftarrow NA + 1$
    end if
  end for
  $T \leftarrow \epsilon * T$
end while

return x_{best}

Chapter 3

System model representation

Having described models of the embedded systems, let us now consider their representation in the tool that is created in as a result of this thesis. Therefore, we shall start with general overview of approach taken to achieve this (3.1). Next the representation of application model (3.2) followed by representation of the hardware architecture model (3.3) is described. The chapter concludes with the description of mapping of application model onto the architecture (3.4).

3.1 Overview

One of the main requirements when doing design of the system model representation is the separation of the input models from the processed ones. The aim in designing the data structures was to clearly distinguish data provided by the designer, and do not change them as the tools run.
Data provided by the designer that shall not change consists mostly of:

- task characteristics including task deadline and its period
- processing elements characteristics with operating modes that it consists available

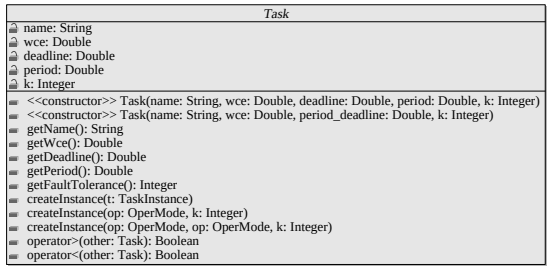

Figure 3.1: Class diagram of `Task`

On the other hand, data that changes throughout the tool run mostly consists of:

- mapping of tasks to the processing elements
- operating mode at which the task is going to be run, and what is related to this, its worst case execution time

Another important aspect is the robustness of data structures that should allow easy further development on the top of those structures. Therefore, data structures created should be prepared for being the base classes for more specialized classes.
Note that presented design of data structures is not final, and may be a little changed during further design. This chapter only attempts to link the system model from section 2.1. Any further changes that will be necessary will be described in following chapters. Following sections will go more in depth with all data structures.

3.2 Task representation

3.2.1 Task

As already mentioned in previous section, data structures representing the tasks should separate the designer input data, that does not change, from data changing while the program runs. Let us, therefore, define abstract class `Task` that will be container for all constant data related to tasks. This class is presented in figure 3.1.
As this shall be the base class (it is abstract), notice that it contains protected

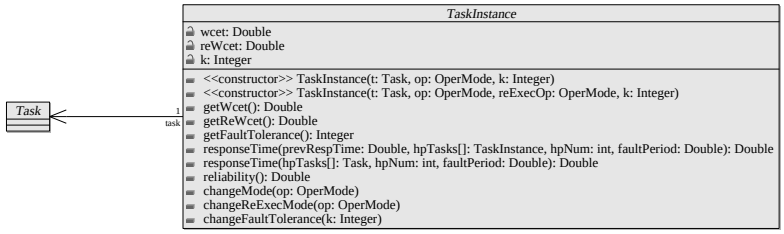

Figure 3.2: Class diagram of `TaskInstance`

data present in most models of tasks encountered in embedded systems: worst-case execution[1], deadline and period. Additionally, to allow fault tolerance task designer may set for each task number of faults that it should tolerate (k)[2] to achieve the reliability goal. For identification purposes there is name of the task as well.
Class have two constructors setting all its attributes. One of them is used to simplify creating task in case the deadline and period are the same. Also note that there are no methods for setting protected attributes, but only for obtaining them (`get*`). This guarantees that tasks are read-only for application.
As it will be necessary to tell the task with the higher priority (e.g. when calculating response time analysis), methods overloading $<$ and $>$ operators (or others performing the same task) seems reasonable for this purpose.

3.2.2 Task instantiation

Having defined basic class for the task representation, let us move on to representing instance of this task. It will contain additional information about the task that is already scheduled, relate it to the operating mode at which it shall be executed, and contain information about necessary number of re-executions of the task. For this purpose let us define class `TaskInstance`, diagram of which is presented in figure 3.2.
`TaskInstance` can be created having object of `Task` class (in fact one needs the object of non-abstract class derived from `Task`) using one of `createInstance()` methods. You then need to provide operating modes (for normal execution and for re-execution) at which the task should be run, as well as number of possible re-executions.
It is again an abstract class, and requires implementation of its methods. It con-

[1] As described in section 2.1.3 we consider in this thesis worst-case execution time. However, this parameter can also be treated as worst-case execution time, if operating modes in processing element have relative speeds, and not absolute ones (see section 3.3 for details)
[2] This fault tolerance can be achieved by both re-execution and replication.

tains two worst-case execution times, one when the task is executed normally, second in case it needs to be re-executed (as stated in section 2.1.3 it should be re-executed at maximum speed)[3]. Both times are calculated using equation 2.2 (inputing speed from related operating mode) when the instance is created, or the operating mode is changed. There shall be no possibility to change it directly.
Class also give the methods to calculate worst-case response time of the task. Method requires set of higher priority tasks, and fault period (if applicable). Recalling equation 2.3, one may notice that to obtain response time multiple iterations have to be made. Normally one shall start setting initial response time to worst case execution time of the task (worst-case response time cannot be smaller then the worst-case execution time), and iterating continues until the response time converges. Notice, however, that response time is not smaller then the response time of higher priority tasks. Therefore, one may as well start iterating from the response time of higher priority task.
As the reliability of the task requires only information about worst-case execution time and value of $\lambda(f)$ for the operating mode (see formulas 2.5 and 2.6), it is possible to calculate it here as well. To achieve this, let us define method `reliability()` that shall calculate the reliability depending on the value of `k`, using either only equation 2.5 (for `k=0`), or equation 2.6 (for `k>0`). It is, however, not possible to calculate at this level reliability of replicated tasks, unless including in class `Task` notion of all objects of class `TaskInstance` that are based on it. This is not meant, as would require changing the `Task` object while the tool is running.

3.3 Processing element representation

Having represented the application model let us now move to the hardware architecture representation. As described in section 2.1.1 hardware model consists of multiple processing elements connected by some communication channel. As application contains set of independent tasks that do not share resources, no communication channel between the processing elements is necessary to represent here.

[3]As `TaskInstance` does not connect task to the processing element, but only to operating modes, at this level of abstraction re-execution at maximum speed cannot be enforced.

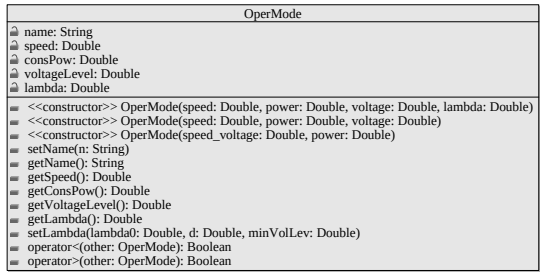

Figure 3.3: Class diagram of `OperMode`

3.3.1 Operating mode

Let us start by defining class `OperMode` representing operating mode of processing element. It is presented in figure 3.3.
Class attributes represent all parameters of the operating mode: speed, power consumption per time unit, and relative voltage level. Note, however, that speed may be both absolute and relative, therefore making it possible to use in the task definition worst-case execution time and worst-case execution cycles. If the speed of operating mode is relative ($0 \leq s \leq 1$), the worst-case execution in task representation stands for worst-case execution time. Otherwise, worst-case execution in task representation stands for worst-case execution cycles[4]. In both cases formula 2.2 holds.
Taking advantage of $\lambda(f)$ depending only on the relative frequency (or voltage) that is constant in each operating mode, we shall give the opportunity to calculate it once only (using formula 2.1), and store it as an attribute `lambda` in this class. It shall decrement the number of necessary calculations during heuristic search.
To make it possible to compare operating modes with respect to their speeds in simple way, as it was with comparing priorities of tasks, the `<` and `>` operators shall be defined (or dedicated methods providing the same functionality).
Note the existence of method allowing change of name of operating mode. This is due to fact that the name of operating mode shall depend also on the processor to which operating mode belongs. However, at the time object of this class is created, no processing element is defined yet (see following subsection for constructing processing element object), therefore making it necessary to provide set method.

[4]Correctness of input data is responsibility of person using the tool. Tool does not check if the corresponding attributes match

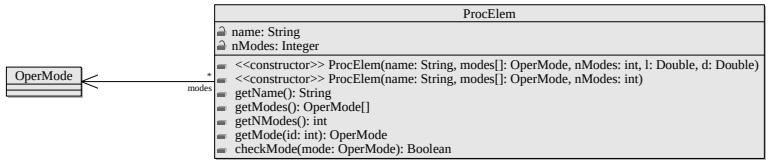

Figure 3.4: Class diagram of `ProcElem`

3.3.2 Processing element

Next let us move to definition of `ProcElem` class, presented in figure 3.4.
Class `ProcElem` may be considered mostly as the container of operating modes (objects of `OperMode` class) that it can run. Therefore, most of logic connected to processing element is not here, but in class already discussed in previous section. This class mostly provides accessing methods for this collection. Note that again putting operating modes is not allowed, and may be done only in the constructor.
Object of this class can be constructed either only with name and set of operating modes. In this case, set of operating modes need to be complete (all data, including value of $\lambda(f)$ function, in operating modes is present, and operating modes are already sorted). Otherwise, one needs to provide data necessary for calculating value of $\lambda(f)$ specific for this all operating modes. In this case the collection of operating modes provided as argument is first sorted from the slowest to the fastest, and then each operating mode has the `lambda` variable calculated and set by means of `setLambda()` method. This is recommended way of creating the processing element.

3.4 Mapping of tasks onto hardware architecture

Having defined basic data structures to represent the task (together with its instance that is scheduled for execution) and processing element (with it's operating modes), let us now move to definition of how the application model is mapped to particular hardware architecture. Such mapping is done at two levels of abstraction. Firstly, instances of tasks, already being related to the operating mode with which they are going to be run, are related to particular instance of processing element. Secondly, all instances of processing elements are related together into one mapped architecture. Both levels are described in following subsections.

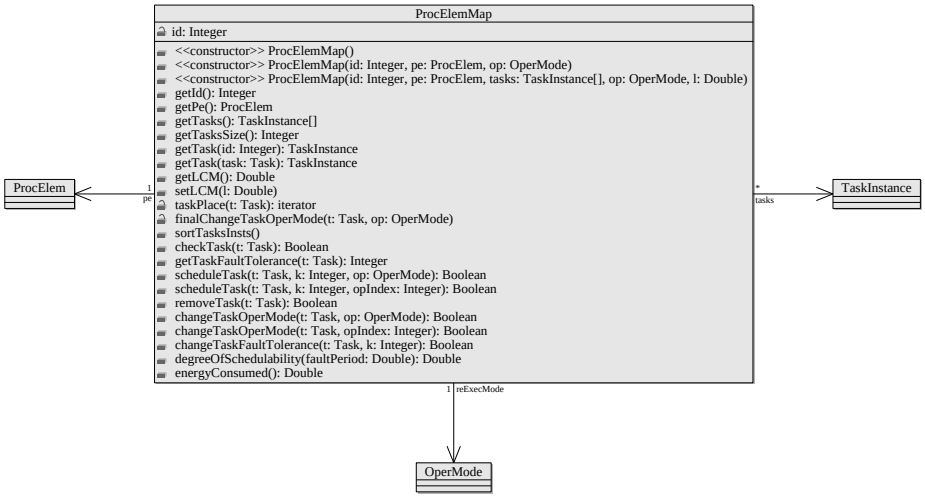

Figure 3.5: Class diagram of `ProcElemMap`

3.4.1 Mapping onto processing element

First abstraction level groups all tasks that shall be executed at one physical processing element (instance of processing element) from the architecture. Let us therefore define class `ProcElemMap` presented in figure 3.5
This class shall contain the reference to object of `ProcElem` class, which stores all the information about the type of processing element being used, including collection of its operating modes. `ProcElemMap` also contains the collection of `TaskInstance` objects, representing all the tasks that are scheduled on this unit. This class should provide methods that will allow to schedule and remove the task, as well as change the operating mode or number of tolerated faults for each of them. Both operations should be safe to use. It should not be possible to schedule task that is already on given processing element, or attempt to remove the task that is not there. Moreover, changing the operating mode of task, should be possible only to operating mode that is present on the processing element to which tasks are mapped (this cannot be validated on the task instance level, as there is no notion of processing element).

At this level of abstraction one may perform already some analysis, therefore moving most of calculation effort to lower levels of abstractions (if possible). Let us point out that for set of independent tasks response time of one task depends on tasks having higher priority and executed on the same processing

element. As all necessary data in our case is already available, we may at this level find response time for all tasks, using methods `responseTime()` from class `TaskInstance`.
Taking the closer look at the definition of degree of schedulability (formula 2.4), one may see that the value of degree is calculated using c_1 formula if system is not schedulable, and using c_2 otherwise. Also note that c_1 takes into consideration only tasks that missed their deadlines, and ignores rest of them. It is also true that system is not schedulable if task set on any of processing elements is not schedulable (i.e. if task sets on all processing elements are schedulable, then the system is also schedulable). Therefore, it is possible to redefine the degree of schedulability making it being sum of partial degrees, if all of them are smaller or equal 0, or sum of positive partial degrees otherwise. Such definition, which is equivalent to original formula, allows us to calculate partial degree of schedulability already at this level of abstraction. Partial degree may be calculated using the original formula (2.4). Calculation of both, response times for all tasks and partial degree of schedulability, is done in `degreeOfSchedulability()` method.
Next possibile analysis at this level is calculating energy consumed. Recalling equation 2.11 total energy is sum of energies of individual tasks. As energy consumed on one of the processing elements does not influence energy consumption of another, we may define partial energy consumption at the processing element level. Then the total energy consumed for the system is the sum of those partial consumptions. This is done using `energyConsumed()` method. Note that for finding the energy consumption one needs the application period (in fact number of times task occurs within the application period, see formulas 2.9 and 2.10). Therefore, when scheduling tasks least common multiple of task periods needs to be calculated. Such calculation is done first at the processing element mapping level, and then on system level (and updated downwards if necessary).
There is no possibility of calculating further reliability at this level of abstraction. One could be tempted to do so, recalling formula 2.8, which makes the reliability the product of individual reliability of tasks. However, reliability of task may depend on its instances on multiple processing elements (in case of replication), information about which is available only on higher level of abstraction described next.

3.4.2 Processing elements collection

Highest abstraction level is the collection of all processing elements (architecture) with tasks mapped onto them. Class representing this level is `Mapping`, and is presented in figure 3.6.
This class contains collection of objects of lower level of abstraction, namely of `ProcElemMap` class. It gives methods for adding and removing new processing elements. In this way it should be possible to look at different architectures

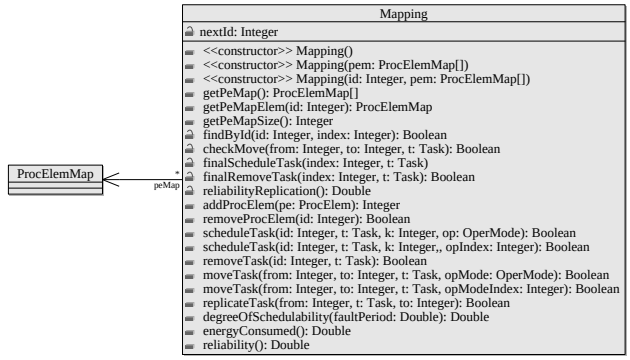

Figure 3.6: Class diagram of `Mapping`

while looking for optimal solution for the embedded system design. Moreover, it provides set of wrapper methods that allow scheduling and removing of tasks (at this point it is necessary to specify the processing element on which operation should be performed). Note that, unlike tasks being uniquely scheduled on processing elements, architecture may contain of multiple processing elements of the same type. Therefore, when adding new processing element unique identifier should be assigned to it. Also for tasks once scheduled on processing element methods to move it to another are available. It is also possible to replicate the task by removing one fault tolerated by re-execution and making a replica of it on another processing element. The same method can be used for moving number of tolerable faults between two processing elements (in case the task is already scheduled on the target processing element).

As this is the highest level of abstraction, all analysis have to be finalized (final results should be available). As already mentioned in previous section, final energy consumption calculations are made by consolidating (summing) the partial results of them (in method `energyConsumed()`). When it comes to degree of schedulability (method `degreeOfSchedulability()`), the summation is performed as long as all processing elements have the partial degree smaller or equal to zero. In case of any processing element having value above zero, system is known to be non-schedulable. In such case previous sum of partial degrees is discarded, and from this point only partial degrees that are greater then 0 are taken into consideration (including first encountered).
Here we may also finally calculate the total reliability of the system. To achieve this, mapping object must be able to tell number of replicas of all tasks in the system together with processing elements to which they are scheduled. Having this information, the product of all tasks' reliabilities is being made as stated in formula 2.8 (in method `reliability()`). If a task has more then one instance

(is replicated), the total reliability of all replicas is calculated using formula 2.7 in method `reliabilityReplication()`. Otherwise, method from task instance is used (as described in section 3.2).

Chapter 4

Interoperability

Let us now move to the different area of investigation in this thesis: attempting to make the framework for interoperability of multiple tools. Following section (4.1) will describe major issues and ways of sharing data among different tools. In particular two possible approaches are brought to reader's attention. First method is using files to exchange information (4.1.1), and the second using external database system (4.1.2). Finally, as the tools must inform each other about the operation status, a look at possible communication methods (4.2) is made.

4.1 Sharing data between tools

Working environment of embedded system designer usually consists of multiple tools, each performing either single very specialized task (e.g. calculate the worst-case execution of given task), or complex ones that help during whole design process. Those tools are usually written using different programming languages and technologies. Even in case of complex and almost complete design tools, it may be still necessary to use the result produced by one tool in another, or even to make them work together to achieve goal set by the designer. To make it possible for multiple tools to cooperate with each other, they must

share information, and in many cases communicate with each other (e.g. send the information regarding successfully performed task). In case of tightly coupled tools both things can be achieved using the same technique, i.e. tool could send the result of its work together with information about task being successfully performed (see 4.2 for more details on such approach). This, however, requires all tools to be aware of each other, and produces problem when new needs to be incorporated or replace existing one. In such situation, not only new tool would need to be prepared for such cooperation, but most likely the older ones would also require some modifications. Moreover, in some cases output of one tool is required by multiple tools, of which producing application may be not aware, and not prepared for.
Examples of such cases may be easily found when looking at the embedded systems design methodology, as described in 1.1. Let us consider mapping level of the design as an example[1]. To perform the mapping, tool requires input from functional design level, which is both functional model of application and possible hardware architectures. Result of this stage must be verified and can be considered as an input for the next design step. However, as already mentioned, further phases of the process will not only use the output of the mapping, but may also impact the mapping. In such situation making all separate tools able to communicate effectively possible, would be tremendous task. Much smarter idea would be to provide the data store into which the tools may put produced data, and where they may find the required ones. For tools tightly interconnected this will not make communication unnecessary, but will make the whole set of tools more interoperable with others by making results available in common datastore.

When it comes to possible storage of data one may consider mostly two options: files, that would be readable for other tools, or putting all necessary data into some database. They are described in more details in following subsections (files in 4.1.1, and databases in 4.1.2). One may also consider memory sharing between tools. This, however, is very dependent on both platform and programming language that one uses. As usage of one platform and one programming language only cannot be assumed in our case, this approach is not developed further in this thesis.

[1]Designed tool will work exactly at this level in design methodology.

4.1.1 Files

One of easiest way to create datastore is to put all information into file or files. Such files may be either binary or in ASCII format. Great advantage of binary files comparing to those in text format is their size: data stored in binary format is usually smaller then text representation of the same structure (unless structures themselves consist only information in text form). Binary files require good description about the structure of the file that may depend on size of basic data types used to create such file. As simple example let us look at sizes of integer variables in C programming language. C standard (see [16] for latest committee draft) defines all data types, however, about their maximum sizes it states only that *their implementation defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign*[16, p. 21]. This makes the size of `long` different when using 32 and 64-bit GCC compiler. In first case size of `long` would be 4 bytes (32-bits), whereas in second 8 bytes (64-bits). This difference could make the binary data file created on 32-bit processors will be unreadable on 64-bit machines, unless special measures will be applied in the file format that will define strictly its structure, therefore making file format definitions much more complex.
Unlike binary format, text file format provides human readability, and when size is not an issue, seems to be more and more preferred option. Text files are more platform independent, as text representation will be read by all the systems in the same way[2]. One of the most popular textual formats nowadays is XML (Extensible Markup Language), created by W3C in 1996 (current 5th edition of the 1.0 recommendation can be found in [27]). It provides the possibility to represent arbitrary data structures using tree structured elements. Each element falls between opening and closing tags. All child elements are placed within parent element. Example XML file representing books by author is presented in listing 4.1. As one may see, the root object in example is `books` and contains two child elements `author`. Each of those child contains also attribute `name`. Further, each `author` element contains three child elements `book` that have in turn child elements `title` and `year`.
Strong advantage of XML format is large number of parsers giving opportunity to be used in most of programming languages without necessity of creating your own.
XML format is base for other more specialized formats that shall represent and store particular structures or information like GraphML or SVG.
When talking about design of embedded systems one should also consider TGFF (Task Graphs For Free) (first defined in [10]) format. TGFF allows to create

[2]This is not true when file format is dependent on line endings, as three most popular operating systems use different characters to represent this: Linux and most of UNIX systems use linefeed (0x0A), MacOS up to version 9 carriage return (0x0D), and Windows both in mentioned order. Note that current versions of MacOS use the same representation as Linux.

Listing 4.1: XML example - books collection

```
<?xml version="1.0" encoding="UTF-8" ?>
<books>
        <author name="Tom Clancy">
                <book>
                        <title>The Hunt for Red October</title>
                        <year>1984</year>
                </book>
                <book>
                        <title>Red Storm Rising</title>
                        <year>1986</year>
                </book>
                <book>
                        <title>Patriot Games</title>
                        <year>1987</year>
                </book>
        </author>
        <author name="Robert Ludlum">
                <book>
                        <title>The Bourne Identity</title>
                        <year>1980</year>
                </book>
                <book>
                        <title>The Bourne Supremacy</title>
                        <year>1986</year>
                </book>
                <book>
                        <title>The Bourne Ultimatum</title>
                        <year>1990</year>
                </book>
        </author>
</books>
```

pseudo random task graphs that are useful as benchmarks for the research purposes, in particular hardware-software co-design. In TGFF files multiple processing elements are defined, together with specification for execution of specific simple tasks (like basic calculations). From those tasks tool may generate the task graphs that could be inputed into analysis tools. As all graphs are composed of tasks that worst case executions (and possibly other parameters) are given for each architecture, the analysis and finding e.g. optimal mapping can be performed. Example TGFF file, taken from the Embedded Systems Synthesis Benchmarks Suite (E3S) version 0.9[3], is shown in listing 4.2.
As one May see, there is first definition of task graph consisting of 6 connected tasks. Task graph has defined period and hard deadline. Further, there are definitions of processors with overall characteristics (like price or power) followed by specification of different tasks that can be executed. In this example there are two AMD processors. TGFF also defines possible communication links that can be used. In this example it is VME and USB 2.0.

Although file storage seem very reasonable and is permanent, there is quite big drawback of using them in the tools, namely time. Files are constantly written to and read from hard drive which significantly increases time required to access data. If files shall be used for exchange information in the design space

[3]File was cut to reasonable size. Original file is much larger, and would not fit as an example.

Listing 4.2: TGFF example - auto-indust-cords from E3S

```
@HYPERPERIOD 0.0009

@COMMUN_QUANT 0 {
0 4E3
1 8E3
2 15E3
3 1E3
}

@TASK_GRAPH 0 {
PERIOD 0.0009

TASK src TYPE 45
TASK can1 TYPE 0
TASK fp TYPE 1
TASK can2 TYPE 0
TASK pulse TYPE 12
TASK sink TYPE 45

ARC a0_0 FROM src TO can1 TYPE 0
ARC a0_1 FROM can1 to fp TYPE 0
ARC a0_1 FROM fp TO can2 TYPE 0
ARC a0_2 FROM can2 TO pulse TYPE 0
ARC a0_3 FROM pulse TO sink TYPE 1

HARD_DEADLINE d0_0 ON sink AT 0.0003
}

# AMD ElanSC520-133 MHz
@PROC 0 {
# price buffered preempt_power commun_energy_bit io_energy_bit idle_power
  33    1        1.6          0                 0              0.16
#-------------------------------------------------------------------------
# type version valid task_time preempt_time code_bits task_power
# Angle to Time Conversion
0       0       1       9e-06       150E-6        6.9e+04    1.6

# Basic floating point
1       0       1       2.3e-05     150E-6        5.8e+04    1.6

# Pulse Width Modulation
12      0       1       4.5e-06     150E-6        1.4e+04    1.6

# src-sink
45      0       1       1e-05       150E-6        80         1.6

}

# AMD K6-2 450
@PROC 1 {
# price buffered preempt_power commun_energy_bit io_energy_bit idle_power
  88    1        11.3         0                 0              1.1
#-------------------------------------------------------------------------
# type version valid task_time preempt_time code_bits task_power
# Angle to Time Conversion
0       0       0       0           150E-6        0          11

# Basic floating point
1       0       0       0           150E-6        0          11

# Pulse Width Modulation
12      0       0       0           150E-6        0          11

# src-sink
45      0       1       1e-05       150E-6        80         11

}

@LINK 0 {
# use_price contact_price  packet_size  bit_time    power    contacts
# VME (Tundra SCV64 + transceiver)
  0         180            1            2.27E-9     10.35    4
}

@LINK 1 {
# use_price contact_price  packet_size  bit_time    power    contacts
# USB 2.0 (Cypress CY7C68013-100AC)
  0         14.19          1            2.08E-3     0.66     4
}

@MEMORY 8388608 1
```

exploration, those time penalty will significantly decrease the performance of the search. One should therefore look for the alternative way of storing data to be shared.

4.1.2 Databases

When storage and sharing of data is necessary one should definitely consider usage of database systems. There are multiple types of databases. Normally the one that are thought of are relational databases, that store information in well defined tables, and allow relations between them. Those databases use SQL for defining queries. Most popular database systems of this type are MySQL (see [1]), PostgreSQL (see [3]).
There are, however, more databases types to consider. Some of them are promoted by NoSQL movement (see [29]), like key/value or object oriented databases. All those database types try to break the popularity of relational databases, offering different models for storing data that in many cases is more convenient to use, at the same time fulfilling database requirements, like ACID[4] for database transactions.
In key/value, each information is associated with particular key. Their advantage is quickness of finding required information, as search is done only through keys, that are usually sorted. One should not, however, think of them as having very limited functionality. Most of those databases allow storing typical data structures (like lists) as value of the key. Problem with those databases is difficulty of storing more complex data structures, for which creating key is not obvious. Known examples are: Cassandra from Apache Foundation[5] (see [11]), or Redis (see [4]).
With grown popularity of object-oriented programming languages, the necessity for them to use databases appeared. Despite language built-in support to store objects in relational databases (like it is in Ruby) most languages that support object-orientation put the necessity of transformations between table records and objects on the developer. Although such transformations are possible, they require additional work to be done, and in case of companies additional costs of developing the application. Therefore, the need for databases allowing storage and retrieval of objects appeared. Such databases shall provide most of the features of data structures used in object-oriented programming: classes, instances and inheritance.
According to [2] object database standards were defined by Object Data Management Group (ODMG), that completed it's work by publishing ODMG 3.0 standard [6] in 2001. It consists of following major components:

[4] ACID - atomicity, consistency, isolation, durability
[5] Used by e.g.: Facebook or Twitter

- Object Model defining common data model, which should be supported by ODMG 3.0 applications
- Object Specification Languages (namely Object Definition Language, ODL, and Object Interchange Format, OIF); first one is used for definitions of data structures in the database, whereas second one defines the file format in which those data may be stored
- Object Query Language (OQL) for searching and updating the database; similar to SQL
- C++, Smalltalk and Java bindings of ODMG implementations to specific object-oriented programming language

Further work on this standard is currently done by Object Management Group (OMG) that according to [2] plans to release 4th version of the standard that would follow up with most recent changes in database systems.
As the place where many resources related to object database management systems, ODBMS.ORG provides the list of available applications, both commercial (like Objectivity/DB [15] or Caché [24]) and free (like EyeDB [23]).

4.2 Communication between tools

All tasks performed together by multiple agents require communication between them. It includes asking for performing specific operation (together with input data for it), information about it's status and signaling that operation was completed (either successfully or with an error). As sharing data was already discussed, in this section we will concentrate on ways to ask different task to perform particular operation (i.e. Remote Procedure Calls). In cases when tools are tightly interconnected, methods described here can replace already defined in 4.1 for data exchanging purposes, as calling different methods require argument passing. However, as already discussed sharing data could be seen in more general way.
When it comes to calling another process, one may consider various APIs[6] that will help in achieving this. One may consider in this case Java RMI, or .NET Remoting. Although it would be tempting to use one of them, due to their easiness in implementation phase, one should remember that both of them are language specific APIs, and using them enforces using specific programming language in the interoperability framework. As this is against our main requirement, neither of those technologies may be used in this case.

[6] Application Programming Interface

More general API with support for multiple programming languages and operating systems is CORBA[7]. Defined by Object Management Group (OMG) in middle 90s standard, allows applications running not necessarily on the same machine, to interact with each other[8]. Available to variety of programming languages (including C/C++, Java, Smalltalk, Ruby and Python), and not dependent on the system running, CORBA would seem great solution for the purpose of interoperability framework.
However, multiple technical issues with CORBA made it not being used widely in more recent years, leaving place for Web technologies like SOAP. Those issues and problems are described in [13]. Firstly, CORBA's API is very complex, making it hard to use by the developers, which in turn increases both development time and number of defects. Another important aspects of CORBA pointed out by Henning are security, and more importantly lack of versioning, that allowed only software updates that were backward compatible.
In case when in procedure calls whole objects do not need to be send to another application, but only their unique identifiers (like in case when the database system is used), one may consider usage of sockets to transfer messages. Those messages could follow both well-known protocol (like XML-RPC or SOAP) or simple protocol defined for the purpose of the application. Socket usage does not depend on the operating system that is running underneath, nor the programming language used for creating of the application. Moreover, various libraries provide methods that may simplify usage of rather low level sockets. Moreover, if well-known protocol is to be used, one may also find the library that supports it.

[7] Common Object Request Broker Architecture

[8] In fact most technologies used for remote procedure calls treat communication with process running on the same host is special case of communication with process running on different host on the network.

Chapter 5

Tool design

Having described how the system model is going to be represented and methods for making multiple tools interoperable, let us now move to the design of the tool itself. Firstly, let us divide functionality of the tool, and therefore itself into separate modules (5.1). Then let us take a brief look at the design of the integrated tool (5.2), followed by design of set of tools that should work together (5.3). The interoperability design is further divided into communication between those tools (5.3.1), and use of files (5.3.2) and database (5.3.3) for data sharing.

5.1 Separation into modules

When dividing the functionality into parts, let us first recall what the tool actually should do. As described in 1.2 the tool should perform the design space exploration and try to find out the optimal solution for mapping of tasks onto processing elements together with deciding voltage levels for each of them[1].
Design space exploration can be done using heuristic algorithm that takes the initial solution, and outputs the best solution that was found during the search. As described in 2.2 the goodness of given solution is captured by the cost function, therefore allowing the algorithm to look for extreme value of it.

[1] As problems are NP-hard, no guarantee of finding the optimal solution could be given.

During the search the solutions needs to be modified to find more and more optimal ones. Those transformations should produce one random solution from the neighbourhood of current, as it is required in case of simulated annealing. The design however should not limit the possibility to only random transformation, but shall allow the generating also the whole neighbourhood, if required. Distinction shall be, however, made between random solution and set of solutions.
Note here that algorithm does not require the knowledge of structures used, and may leave it to methods calculating the cost function or performing solution transformation.
Taking all above into account, let us distinguish here four modules into which the functionality of the tool can be divided:

- Requesting module
  This module is responsible mostly for interactions with the user. In cases when user does not specify himself the initial mapping, it shall also generate the initial solution that will be an input to the heuristic algorithm.
- Heuristic algorithm module
  This module should implement heuristic search algorithm (in our case it would be Algorithm 1). It does not require any informations about representation of system model.
- Cost function module
  This module calculates the value of the cost function for given solution.
- Transformation module
  This module modifies given solution by changing one of the parameters, therefore creating the new solution being in the neighbourhood. It should also be able to create the set of solutions (whole neighbourhood), in case the heuristic algorithm requires that.

How the modules are interconnected precisely depends whether all of them are integrated into one tool, or they are standalone tools that cooperate with each other. Both situations are described in following sections (5.2 and 5.3).
Design of each of those modules is quite straightforward. Let us therefore look only at two of them, namely cost function and heuristics.

5.1.1 Cost function module

At first glance quite simple module seems to be the one that calculates value of cost function. Most importantly it consists of one method that is responsible

for doing the calculations of the cost function, based on energy consumed by the questioned solution, its schedulability and reliability. In the simple case we are interested only that designed system fulfills its requirements and consumes minimal energy. Let us therefore define the cost function as follows:

$$c(d, r, E) = W_d d + W_r r + W_e E \tag{5.1}$$

As one see in equation 5.1 cost function depends on three parameters. One of them is (E) is the energy consumed by the system (with its weight W_e). As we are in this case interested only that degree of schedulability is smaller or equal 0, let us define parameter d as follows:

$$d = \begin{cases} degree & \text{if } degree > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

Such defined parameter will make the cost function ignore the degree of schedulability for systems that are schedulable and make the penalty on the systems that are not. This penalty is controlled bye W_d in equation 5.1.
Similar approach to one from equation 5.2 is taken when defining the parameter representing the reliability of the system (equation 5.3). Here we are, however, interested in knowing how far our reliability is from the reliability goal. If it does not fulfill this requirement, appropriate penalty is given.

$$r = \begin{cases} R_g - R & \text{if } R_g - R > 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

By controlling the weights of parameters in cost function we can drive the heuristic search towards the results that we want to obtain.
However, when designing such module one should make it more configurable, therefore, allowing user to find the best solution for him. Consider that e.g. in case of incremental design, we are also interested in leaving time slots for possible future tasks to be scheduled on the same architecture. Therefore, cost function module should allow changing not only weights, but also way that parameters are calculated. To provide such functionality let us redefine formulas for calculating parameters (5.2 and 5.3) and slightly cost function (5.1) to following.

$$d = \begin{cases} degree - d_g & \text{if } degree - d_g > 0 \text{ or } C_d = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

$$r = \begin{cases} R_g - R & \text{if } R_g - R > 0 \text{ or } C_r = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

$$e = \begin{cases} E - E_g & \text{if } E - E_g > 0 \text{ or } C_e = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

$$c(d, r, e) = W_d d + W_r r + W_e e \tag{5.7}$$

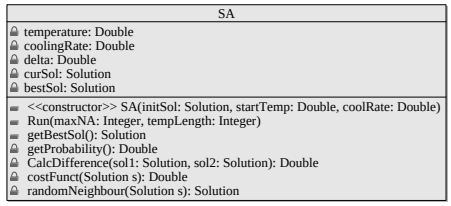

Figure 5.1: Class diagram of `SA`

One may notice that we can distinguish two types of parameters: those that we want to minimize (degree and energy), and one that we want to maximize (reliability). Due to this we have different order of subtraction: in case of maximizing value we subtract actual value from goal, and in case of minimizing it is another way round.
Note that for $C_d = 0$ and $d_g = 0$ formula 5.4 is the same as 5.2. Also for $C_r = 0$, formula 5.5 is the same as 5.3. Also in case when $E_g = 0$ cost function presented in 5.7 is the same as one shown in 5.1. Therefore, by controlling variables C_d, C_e and C_r, as well as weights W_d, W_e and W_r we may ignore or take into consideration each of the parameters, and so customize cost function to our needs. All goals shall be included in the representation of highest level of system model, i.e. in the `Mapping` class.

5.1.2 Heuristic algorithm module

As described in section 2.2 heuristic algorithm used here is simulated annealing. Module that implements this is made of class `SA` presented in diagram 5.1.
On the class diagram one may notice presence of private methods for getting random neighbouring solution and cost function. Those methods are responsible for interfacing with other modules only, and do not perform those tasks themselves. From the class design one may notice that obtaining the result requires three steps to be taken. Firstly the object of the class can be constructed defining initial solution, starting temperature and cooling rate. Then one should run the algorithm, defining additionally two parameters that are dependent on each run of the algorithm, namely temperature length and maximum number of not accepted moves. Finally, after the algorithm terminates one may obtain the best solution.
Data type `Solution` is type that depends on type of tool (whether it is one integrated tool, or many tools operate together). The same type is used in transformation module.

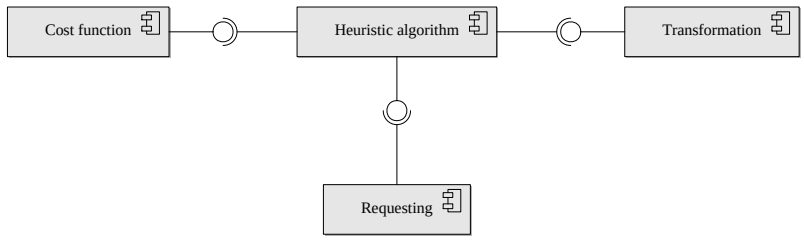

Figure 5.2: Integrated tool design

5.2 Integrated tool

As one may see from the overall design of integrated tool, shown in figure 5.2, the central module is the one responsible for heuristic algorithm. It provides an interface to the requesting module, allowing it to start design space exploration. Furthermore, the heuristic module requires two modules that help them achieving its task. Therefore, it is a client to both transformation module and cost function module.
In case of integrated tool all communication is done by calling methods that do exist in the same address space, and therefore no special communication method nor protocol is required. Therefore, modules here do not require any additions comparing to those presented before, and can be seen in the same way as were presented in section 5.1.
In case of integrated tool, type `Solution` refers to class `Mapping`, defined in section 3.4.

5.3 Inter-operation of tools

Let us now take a closer look at the design of interoperation of multiple tools that will perform the same task. The overall design of the tool set is presented in figure 5.3.
First thing to notice comparing to figure 5.2 is encapsulation of each of modules presented in section 5.1 into server components (and applications). Additionally one may see presence of main server application, to which client connects and sends requests. Main server is then responsible for routing of messages between server applications actually performing their specific tasks.
It would be tempting to think that this approach adds unnecessarily complexity by placing functionality that could be in heuristic algorithm server into separate application. Note, however, that this makes each application providing only

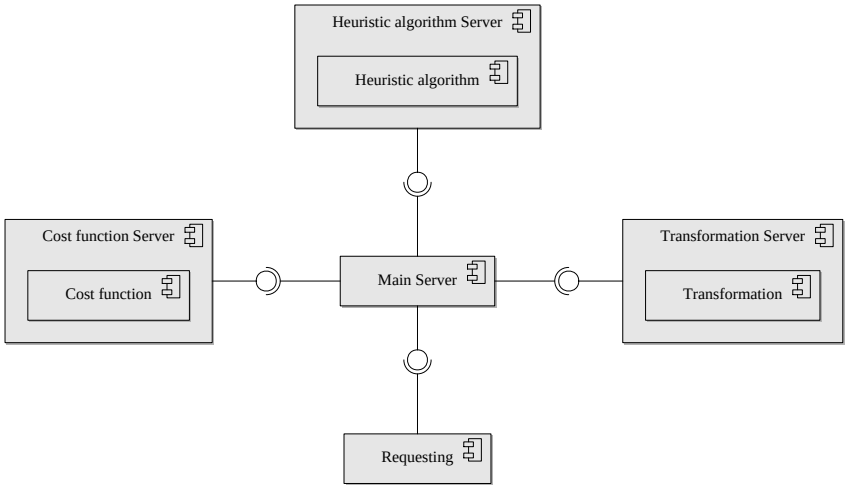

Figure 5.3: Inter-operation of tools design

necessary service. It also allowed to create each of the servers in simple way. As presented in figure 5.4 each of server classes inherit from common class `Server` that consists all necessary logic of the server, like sending and receiving messages, and common handler for incoming messages.
Design of the modules themselves should not be much affected by the fact that they are separate applications now. However, in some cases additional elements were necessary to be added. Such case is e.g. information about the socket in the heuristic module to which all requests should be sent (this is socket connection with main server application). Additionally, in case of tool set, type `Solution` cannot be `Mapping` anymore, as tools do not share address space. It is string containing unique identifier instead.
Let us now move into more details regarding more specific issues with set of tools, namely communication between them (section 5.3.1) and then handling of data sharing, first by means of files (5.3.2) and then using database management system (5.3.3).

5.3.1 Communication

From considerations in section 4.2 let us choose sockets with own protocol as the way of communication between tools. This solution is not hard to implement and does not require much more work comparing to other possibilities. Moreover, as already mentioned, it does not depend on the platform, as notion of sockets is independent of the platform.

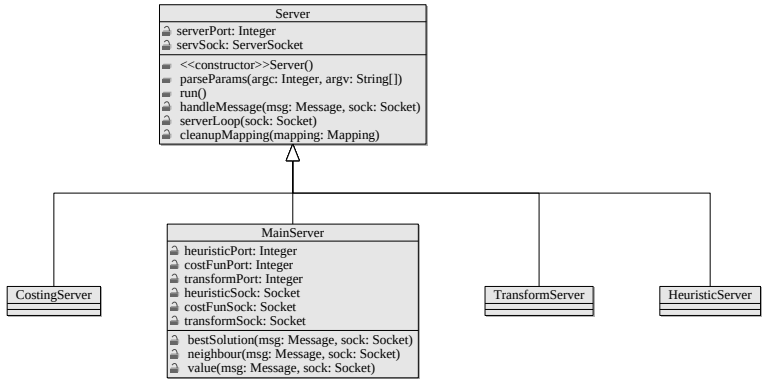

Figure 5.4: Class diagram of server classes

Let us therefore define the following basic protocol rules for communication:

1. All messages are in plain ASCII format. No termination symbol is used[2].
2. Each message consists of three fields:
   - type of message
   - command being sent
   - payload
3. Payload is subdivided into two fields:
   - number of items in the payload
   - items themselves
4. Fields of the message, as well as the items in payload are separated with space character (0x20 in ASCII code). Therefore, no space character is allowed in any of the fields.
5. Not recognized messages are dropped.
6. Messages are not acknowledged.

Note that protocol does not contain any error handling built-in, as well as no guarantees about message transmission. This makes the protocol simple both to design and implement, and is enough for purpose of this thesis.
For full definition of all possible messages (types and commands), please see the appendix A.

[2]This guarantees the same behaviour on different operating systems. Message completeness, however, is not taken care of in this case.

5.3.2 Using files for objects exchange

As discussed in section 4.1.1 there are quite few possibilities when it comes to storing data in files, including binary and text format. Solution that seems to be the most reasonable for purpose of this thesis, is the XML file format. XML makes it possible to define arbitrary structures and variety of parsers for most programing languages.
To make conversion straightforward, defined XML schema should be as close to those representing system model as possible. Therefore, let the top element be `tool` (XML allows only one top element in file and we want to have only one file containing all necessary information). Its child elements shall represent tasks and processing elements, as well as the whole mapping. As operating modes are strictly connected to processing element, they should be defined as child elements of them. Whole mapping should be also included in one element and its children.
Whole structure of the XML file is defined using XML schema that can be found in appendix B.1.

5.3.3 Using database for objects exchange

As discussed in section 4.1.2 despite well know and commonly used relational databases, there are plenty of other database types that could be used for data exchange. As the design of data structures follow object-oriented approach, using object-oriented database seems reasonable choice for this thesis. As there exists standard for such database management systems, let us take advantage of it, and create design of used data structures in ODL. Class specifications are presented in appendix B.2. This representation is direct translation of classes described in chapter 3 of this thesis to Object Definition Language.
Worth mentioning here is that although database system software may allow to generate code in desired programming language, such code may strongly be dependent on the database used. As we aim to have the same implementation of the data structures for all possible cases, there will be still translation from generated classes to those defined and used in other variants of the tool. Still having such transformations might be seen as losing an advantage of object databases, as in case of more popular relational databases such transformation would be needed anyway. Note, however, that transformations in case of relational databases seem more complicated, and possibly would require more work during implementation.

Chapter 6

Implementation

Having discussed design of the tool, let us now move on to description of the implementation. It is, however, not an intention of this chapter to describe everything in detail. Instead an overview and most important decisions and non-trivial issues are described.
Following sections will discuss firstly the choice of implementation language. Further they will describe implementation issues with representation of system models (section 6.2), that were designed in chapter 3. After this we shall move to interoperability framework (6.3), first describing sharing of data (6.3.1), and then communication between processes (6.3.2).

6.1 Choice of language

Let us start this discussion with the choice of programing language. This decision affects implementation process itself as well as the final tool. On the other hand, it is also affected by the design methodology chosen at the beginning. In case of this thesis, object-oriented design was chosen, leaving still choice between multiple languages, as many that are used today offer object-oriented techniques. Most popular, and worth mentioning here are C++, Java, C#, and Python. Although those languages offer possibility to make object-oriented

code, they differ in the level of abstraction, and e.g. usage of resources like memory. Good example here is C++ that unlike others does not offer garbage collection, leaving memory management to the developer. On the other hand, it does not require any runtime environment or interpreter (despite dynamically linked libraries[1]) that run underneath. Being compiled directly to binary representation has also a drawback: it is dependent on the architecture and operating system on which it shall be run. Therefore, it is necessary to build such code from the sources, before it can actually be used. As this is an academic project it is not an issue, as sources are freely available.

Due to different levels of abstraction, the complexity and size of the code is different for each of those languages. The same functionality implemented in Python will for sure take less lines of code than when written in Java, which in place will take less code than C++. Although measurement of lines of code is issue of many discussions, it is well-known fact, that maintenance cost increases with amount of code that is written. Furthermore, with complexity increment possibility of making errors occurring also increases.

Choosing programing language also affects the performance of created application. It is a fact that with the increased level of abstraction at which language operates, the performance is lower. In case of mentioned languages that require runtime environments, worse performance seems straightforward: additional layer is present that needs to be translated first to processor operations. In case of Python it is directly the code written by developer (as it is interpreted at the runtime), whereas Java and C# compiles original source to byte-code, that is then interpreted by the JRE[2]. As the tool that is going to be implemented performs design space exploration, which takes significant amount of iterations in heuristic algorithm before terminating, performance is major concern. Therefore to provide reasonable performance, despite using optimized algorithms, one should consider using language offering low level of abstraction[3].

Taking all above into consideration, the decision was made that programing language that should be used is C++. It offers object-oriented programming, does not require any additional runtime environment, and provide probably the lowest level of abstraction from all languages offering object-orientation.

Before moving to actual description of the implementation, let us also point out, that all tools that are going to be developed (integrated tool, and two sets of interoperable tools) in all cases when it is possible, should have the same code base, therefore, limiting number of repetition in the sources. In cases of small differences between, we will use pre-processor instructions and different compilation flags to execute appropriate code for the tool.

[1] It should be, however, noted here that unless using very specialized libraries, standard ones are present in all operating systems nowadays.

[2] Java Runtime Environment

[3] It should be stated, however, that creating such tool in assembly language is not reasonable choice, as code size would be enormous and difficult to maintain, and todays compilers are in many cases much better in optimizing higher level languages (like C), then years ago.

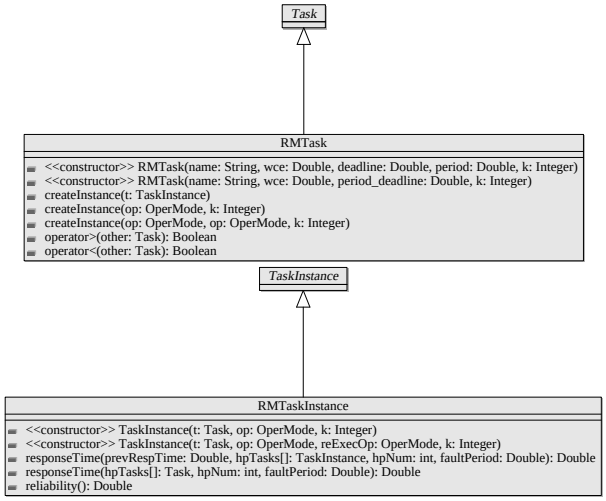

Figure 6.1: Class diagrams of `RMTask` and `RMTaskInstance`

6.2 System model representation

In chapter 3, we divided there constant data (like information about tasks or processing elements) from the one that changes during design space exploration (mapping). Let us here follow the structure of chapter 3, showing the most interesting parts of the implementation.

6.2.1 Task representation

There are two classes representing tasks of the system model. First one, `Task` represents constant data, and in general is read-only for the environment. Second class is `TaskInstance` that represents scheduled instance of task and relates to the operating mode at which it should be executed. As shown in figures 3.1 and 3.2 both classes should be abstract, leaving methods specific to scheduling (like calculating of response time analysis and deciding which task has higher priority) to inheriting classes. Therefore, let us define classes that inherit from `Task` and `TaskInstance` and implement rate-monotonic scheduling. Those classes, with methods that they implement, are shown in figure 6.1.

Implementation of `Task` and `RMTask` classes does not contain any issues that need a discussion. Classes are simple, and their implementations straightforward. More interesting part here is implementation of task instantiations methods for calculating worst-case response time. As described in 3.2.2 there are two methods that allow such calculation, as we noticed there that worst-case response time of task cannot be smaller then the one of higher priority task. Therefore, one of methods start with argument representing response time of one priority higher task. Second method (one without this argument) is only wrapper for the first one. It would be tempting here to use default arguments values allowed by C++ language. Note, however, that default arguments must be present as last ones in argument lists, and if there are more then one, they need to have different types. In `responseTime` methods there is already default argument for fault period (defaulting to 0), which is of type `double`. As response times are of the same type, using this technique would result in ambiguity and compilation error. If `faultPeriod` argument is greater than 0 (if provided value is smaller than 0, it is set to 0 beforehand), `responseTime()` method first finds greatest value of worst-case execution time of re-execution of higher priority tasks, that can contribute to re-execution necessity. Then it loops using equation 2.3. This equation enforces convergence as stop condition (two subsequent iteration produce the same result). However, to avoid infinite loops, the counter is introduced that will terminate the loop if it exceeds certain maximum number of iterations. Both methods are presented in listing 6.1.

Listing 6.1: `responseTime` method from class `RMTaskInstance`

```
double RMTaskInstance::responseTime(double prevRespTime,
                const TaskInstance **hpTasks, int hpNum,
                double faultPeriod) const
{
        /* Maximum worst-case execution of the higher priority tasks and current
         * task */
        double maxWcet = reWcet;
        if (faultPeriod < 0.0) faultPeriod = 0.0;
        if (faultPeriod > 0.0) {
                for (int i=0; i<hpNum; i++) {
                        if (hpTasks[i]) {
                                // Get methods have to be used here, as we use
                                // pointer to TaskInstance class (error that
                                // reWcet is protected)
                                // Alternatively casting could be used
                                maxWcet = (hpTasks[i]->getReWcet() > maxWcet &&
                                                hpTasks[i]->getFaultTolerance()
                                                > 0) ?
                                        hpTasks[i]->getReWcet() : maxWcet;
                        }
                }
        }

        /* Response time is initially set to worst case execution time */
        double responseTime = prevRespTime;
        double old;
        unsigned counter = 0;

        /* Iterate as long as the response time is different from previously
         * found or the maximum number of iterations is exceeded */
        do {
                old = responseTime;
                responseTime = wcet;

                /* Interference from higher priority tasks due to preemption */
                for (int i = 0; i < hpNum; ++i) {
                        if (hpTasks[i]) {
                                // Get methods have to be used here, as we use
```

```
                                // pointer to TaskInstance class (error that
                                // wcet is protected)
                                // Alternatively casting could be used
                                responseTime +=
                                        ceil(old/hpTasks[i]->task->getPeriod())
                                        * hpTasks[i]->getWcet();
                        }
                }

                /* Possible re-execution due to fault */
                if (faultPeriod > 0.0) {
                        responseTime += ceil(old/faultPeriod) * maxWcet;
                }

                // Increment counter
                counter++;

        } while (responseTime != old && counter < MAX_ITERATIONS);

        return responseTime;
}

double RMTaskInstance::responseTime(const TaskInstance **hpTasks,
                int hpNum, double faultPeriod) const
{
        return responseTime(wcet, hpTasks, hpNum, faultPeriod);
}
```

Also worth looking at is method for calculating local (meaning only taking into account instance of this task on one processing element) reliability of task. It calculates first the normal reliability of task, using equation 2.5, and then if necessary uses it to calculate reliability of task being re-executed.
In case of reliability we are using numbers of type `long double` that are very close to 0 or 1. To avoid treating such numbers as 0 or 1, we need to create protection mechanism. In C++ `cfloat` header provides constants characterizing given system about the precision and accuracy regarding floating point values for each type. Useful here is constant `LDBL_EPSILON` that defines the smallest possible value that if added to 0 will create result recognized as different from 0. Same value if decremented from 1 will make the result different from 1. Therefore, to avoid reliability being 0 or 1, we should make sure that results we are getting are within range `0+LDBL_EPSILON` and `1-LDBL_EPSILON`. This method is presented in listing 6.2.

Listing 6.2: `reliability` method from class `RMTaskInstance`

```
long double RMTaskInstance::reliability() const
{
        long double reliability = 0.0;

        long double normalReliability =
                exp(-opMode->getLambda() * wcet);

        // It can be so close to 1, that will be treated as it. Make it then
        // distinguishable from 1.
        if (normalReliability > 1 - LDBL_EPSILON) {
                normalReliability = 1 - LDBL_EPSILON;
        }

        if (k == 0) {
                reliability = normalReliability;
        } else {
                long double reExecReliability =
                        exp(-reOpMode->getLambda() * reWcet);

                // It can be so close to 1 that subtracted from 1 will produce 0
                // result. Make the difference then minimum distinguishable from
                // 0.
```

```
                if (reExecReliability > 1 - LDBL_EPSILON) {
                        reExecReliability = 1 - LDBL_EPSILON;
                }

                long double reExecPower = pow(1 - reExecReliability, k);

                // It can be so close to 0 that multiplied will produce 0, and
                // so reliability of 1. Make the difference then minimum
                // distinguishable from 0.
                if (reExecPower < LDBL_EPSILON) {
                        reExecPower = LDBL_EPSILON;
                }

                long double probAllFail = reExecPower * (1 - normalReliability);
                if (probAllFail < LDBL_EPSILON) {
                        probAllFail = LDBL_EPSILON;
                }

                reliability = 1 - probAllFail;
        }

        return reliability;
}
```

Note that this approach is followed in all methods that calculate reliability. One may argue that it does not provide fully accurate results. However, it should be mentioned that results in this case are lowered, leaving designer on the safe side.

6.2.2 Processing element representation

Unlike representing of tasks, classes designed in section 3.3 are not abstract, and therefore we may implement their functionality directly without need of class inheritance.
Implementation of first class described in the design (`OperMode`) was quite straightforward and did not require any specific work. Note only that overloaded operators < and > differentiate modes on three levels: first speed is taken into account then, voltage level, and if both mentioned are equal, consumed power. In the second class (`ProcElem`) worth mentioning seems to be the constructor. It needs first to sort operating modes increasingly, leaving the slowest with lowest voltage level first in the array. Sorting was done using bubble sort algorithm. After that failure rate (λ) together with name (needed when put into XML format) is set for each operating mode. Constructor is presented in listing 6.3.

Listing 6.3: Constructor of `ProcElem` class

```
ProcElem::ProcElem(std::string n, OperMode **om, unsigned nm, double l,
                double d)
        : name(n), modes(om), nModes(nm)
{
        /* These must be non-zero to continue */
        if (!modes || !nModes) {
                return;
        }

        /* Sort operating modes (bubble sort) */
```

```
        bool changed = false;
        do {
                changed = false;
                for (unsigned i = 0; i < nModes - 1; ++i) {
                        if (*(modes[i]) > *(modes[i+1])) {
                                OperMode *tmp = modes[i];
                                modes[i] = modes[i+1];
                                modes[i+1] = tmp;
                                changed = true;
                        }
                }
        } while (changed);

        /* Minimum voltage level */
        double minVolLev = modes[0]->getVoltageLevel();

        /* Set failure rate for each operating mode */
        for (unsigned i = 0; i < nm; ++i) {
                stringstream ss;
                ss << name << "OpMode" << i;
                modes[i]->setName(ss.str());
                modes[i]->setLambda(l, d, minVolLev);
        }
}
```

6.2.3 Mapping of tasks onto hardware architecture

As described in section 3.4 mapping is done at two levels. First instances of tasks are mapped to processing element in `ProcElemMap` class, and then all processing elements with tasks mapped are collected in `Mapping` class.
Let us first take a look at some parts of `ProcElemMap` implementation. As one may see from figure 3.5, all methods available for handling scheduled tasks use class `Task` for referencing what task should be affected. In C++ such referencing is done using `const *Task`, which allows to pass only pointer (which is faster then copying object) and guarantees that this pointer will not be used to change the object itself.
To allow quick finding of higher priority tasks that are scheduled on the same processing element (this is necessary when calling already described `responseTime()` method), all task instances are sorted by the priority. To prevent repeatable sorting of vector storing tasks, `scheduleTask` methods use `taskPlace` method that finds where in the vector new task should be inserted. This method uses binary search, and is presented in listing 6.4. Same method is used in order to check if task is already scheduled on this processing element, when second argument, defaulted to `false`, is set to `true`.

Listing 6.4: `taskPlace()` method in `ProcElemMap` class

```
std::vector<TaskInstance*>::iterator ProcElemMap::taskPlace(const Task *task,
                bool onlyFound)
{
        // Place when the task should be put in the vector of tasks
        std::vector<TaskInstance*>::iterator place = tasks.begin();

        // Left and right limits
        unsigned l = 0;
        unsigned r = tasks.size();

        // Find the place where the task should be placed in vector
```

```
        while (l <= r && r != 0) {
                unsigned mid = (l + r) / 2;
                std::vector<TaskInstance*>::iterator tmp = place + mid;
                const Task *t = (*tmp)->task;
                const Task *nt = NULL;
                if (mid + 1 < tasks.size()) {
                        nt = (*(tmp+1))->task;
                }

                // Cover special case when task with same priority is present.
                // This is also the case when we want to find this particular
                // task.
                if (!(*t < *task) && !(*t > *task)) {
                        place = tmp;
                        break;

                // Check if the place found is exactly between lower and higher
                // priority task
                } else if (nt) {
                        if (*t > *task && *nt < *task) {
                                if (onlyFound) {
                                        place = tasks.end();
                                } else {
                                        place = ++tmp;
                                }
                                break;
                        }
                }

                // Prepare for next iteration
                if (*task > *t) {
                        r = mid;
                        if (r > 0)
                                r--;
                } else {
                        l = mid + 1;
                }

                // Cover case when task should be placed at the end
                if (l == tasks.size()) {
                        place = tasks.end();
                        break;
                }
        }

        return place;
}
```

`taskPlace` needs to be defined twice, as in different situations we require usage of `iterator` or `const_iterator`. Difference between both of them is only type of iterator used.

Let us also take a look at method calculating degree of schedulability for the processing element using equation 2.4. As described before, it takes advantage of fact that worst-case response time is greater then worst-case response time for higher priority task. Therefore it first calculate response time for highest priority task, not specifying starting value (in this case method will take the worst-case execution time of the task as starting point), and providing empty array of higher priority tasks. Afterwards it prepares array for higher priority tasks that will be filled in before calculating response time for each task. In this loop we will make use of second `responseTime()`, as we keep previously calculated response time. We should also remember whether system is so far schedulable or not. In case it becomes unschedulable degree of schedulability calculated so far should be discarded, and from this moment, we add only differences between response times and deadlines for unschedulable tasks. Code for this method is presented in listing 6.5.

Listing 6.5: `degreeOfSchedulability()` method in `ProcElemMap` class

```
double ProcElemMap::degreeOfSchedulability(double faultPeriod) const
{
        double degree = 0.0;
        double prev;
        bool schedulable = true;

        // If there are no tasks scheduled, then for sure it is 0
        if (tasks.size() != 0) {
                // Calculate response time for task with highest priority
                degree = tasks[0]->responseTime(NULL, 0, faultPeriod);
                prev = degree;
                degree -= tasks[0]->task->getDeadline();

                // Check if schedulable so far
                if (degree > 0) {
                        schedulable = false;
                }

                // There can be at most number of tasks - 1 higher priority
                // tasks
                const TaskInstance **hpTasks = (const TaskInstance**)
                        new TaskInstance*[tasks.size()-1];

                // Calculate response time for all lower priority tasks
                for (unsigned i = 1; i < tasks.size(); ++i) {
                        double deadline = tasks[i]->task->getDeadline();

                        // Add the higher priority task to the array
                        hpTasks[i-1] = tasks[i-1];

                        // Calculate response time for the task
                        prev = tasks[i]->responseTime(prev, hpTasks, i,
                                        faultPeriod);

                        // Check if schedulable
                        if (prev > deadline && schedulable) {
                                schedulable = false;
                                degree = 0.0;
                        }

                        // Add this task to degree of schedulability unless this
                        // task is schedulable and the system is not
                        // if (! (prev <= deadline && !schedulable) ) {
                        if (prev > deadline || schedulable) {
                                degree += prev - deadline;
                        }
                }

                // Free memory reserved for the higher priority tasks
                delete [] hpTasks;
        }

        // Return found degree of schedulability
        return degree;
}
```

Calculation of total degree of schedulability is done in `Mapping` class, discussed below, and only sums up results of this method for each of processing elements. It of course also keeps track whether system is schedulable or not, similarly to methods presented here.

As other methods in this class are straightforward, let us move to highest abstraction level in the system model, namely `Mapping` class. As described in 3.4.2 each processing element is assigned unique id, using which it will be referenced. This produces similar problem to finding tasks scheduled on processing element, so let us look at method responsible for it: `findById()`. Similarly to `taskPlace()` method, it uses search algorithm from divide and conquer group, that is enhanced version of binary search adopted to our identifier policy. The

only difference lays in the way we divide the set of processing elements. In case of binary search we always divided it into half. However, let us observe that mapping assigns new identifier to processing element by incrementing previous one starting from 0, making them equal to indexes in the vector of `ProcElemMap` objects. Therefore, by checking first index that is equal to id, algorithm will terminate successfully immediately, unless some processing elements were removed from vector. If on the other hand there were removed objects, iteration will find the object with higher identifier than the one that we looked for. It is then possible to find out number of objects that were removed, and change the lower limit accordingly by this number. This will make the algorithm converged faster to the solution that we looked for (if it exists). Implementation of this algorithm is presented in listing 6.6.

Listing 6.6: `findById()` method in `Mapping` class

```
bool Mapping::findById(unsigned id, unsigned &index) const
{
        bool status = false;
        unsigned left = 0;
        unsigned right = peMap.size() - 1;

        // Our first guess would be the case when no removals from array did
        // happen, i.e. index == id
        unsigned pointer = id;
        if (pointer > right) {
                pointer = right;
        }

        /*
         * We moved the binary search condition to the end of the loop. As the
         * first guess is the actual id we are looking for. In case of no
         * removals, it will break immediately.
         */
        while (true) {
                // Store the id in temporary variable not to call function many
                // times
                unsigned tmp = peMap[pointer].getId();

                if (tmp == id) {
                        index = pointer;
                        status = true;
                        break;
                } else if (tmp > id) {
                        /*
                         * This will increase also the lower bound of the search
                         * and therefore further reduce number of iterations
                         * necessary.
                         */
                        // This value will always be at least 0
                        left = pointer - (tmp - id);
                        right = --pointer;
                } else {
                        left = ++pointer;
                }

                // Binary search condition
                if (right < left) {
                        break;
                }
                pointer = (right - left) / 2;
                pointer += left;
        }

        // Return the status; index is returned through arguments
        return status;
}
```

Another methods worth looking at are those calculating reliability. We already saw methods that calculate local reliability of the task (listing 6.2), and here

let us see how the reliability is calculated when replication occurs. As shown in equation 2.8 final reliability is product of reliabilities of all tasks in the system. However, as seen from 2.7, we should not consider replicas of tasks as independent ones. We should at this level keep information about all replicas of tasks. Let us, therefore, create in the `Mapping` class `multimap` called `replicas` object that will have `Task` as its key, and identifier of processing element as value. In C++ STL[4] `multimap`, unlike `map`, allows multiple values for one key, and provides methods to count and access them. It is therefore ideal container for storing information about task replicas. To make the comparison simple, without necessity to define equality operator for class `Task`, let us use pointer to this structure as a key. Then when calculating total reliability we should first verify, whether task exists more then once in `replicas`, and then use method `reliabilityReplication()` that finds reliability of single task being replicated. As we stored identifier of processing element to which it is mapped, it is easy to obtain the task instance itself. Those methods also guarantee that we will not consider any task more then once (`multimap` is sorted by its key). We also used here the same methods to overcome limited accuracy of floating point values. Implementation of both methods are presented in listing 6.7.

Listing 6.7: `reliability()` and `reliabilityReplication()` methods in `Mapping` class

```
 */
long double Mapping::reliability() const
{
        long double reliability = 1.0;

        const Task *prev = NULL;
        std::multimap<const Task*, unsigned>::const_iterator it;
        for (it=replicas.begin(); it!=replicas.end(); ++it) {
                if (it->first == prev) {
                        continue;
                }
                if (replicas.count(it->first) > 1) {
                        reliability *= reliabilityReplication(it);
                } else {
                        const TaskInstance *t = peMap[it->second].getTask(it->first);
                        reliability *= t->reliability();
                }
                prev = it->first;
        }

        return reliability;
}

long double Mapping::reliabilityReplication(std::multimap<const Task*,
                unsigned>::const_iterator ptr) const
{
        long double reliabilityProduct = 1.0;
        std::multimap<const Task*, unsigned>::const_iterator it;
        for (it=replicas.equal_range(ptr->first).first;
                        it!=replicas.equal_range(ptr->first).second;
                        ++it) {
                const TaskInstance *t = peMap[it->second].getTask(it->first);

                long double taskReliability = t->reliability();

                // It can be so close ot 1 that subtracted from 1 will produce 0
                // result, and so reliability product of 0. Make the difference
                // then minimum distinguishable from 0.
                if (taskReliability > 1 - LDBL_EPSILON) {
                        taskReliability = 1 - LDBL_EPSILON;
```

[4]Standard Template Library

```
            }

            reliabilityProduct *= (1 - taskReliability);
        }

        // It can be so close to 0 that subtracted from 1 will produce 1 result,
        // and so reliability of 1. Make the difference then minimum
        // distinguishable from 0.
        if (reliabilityProduct < LDBL_EPSILON) {
            reliabilityProduct = LDBL_EPSILON;
        }

        return 1.0 - reliabilityProduct;
```

Please also note here, that maintaining `replicas` object is done inside methods responsible for scheduling, moving and replicating tasks. Therefore, it is always up-to-date, and it is not necessary to recreate it every time we calculate the reliability.

6.3 Interoperability framework

Let us now move on to implementation of interoperability framework. We will concentrate here on the ways in which data is shared between tools (section 6.3.1), namely, how it is converted between different representations, and then on the communication methods (section 6.3.2).

6.3.1 Sharing data framework

Unless tools are using shared memory, they need to convert data to the form that will be readable for other tools. As decided in section 5.3 tool will use two types of shared medias for data exchange: XML files (design of this was presented in section 5.3.2), and object-oriented database (design of this was presented in section 5.3.3). Both methods are done in similar way, and therefore will be described together. Note, however, that storing and retrieving information from database is only possible in case of tool using database system[5]. Therefore, our discussions should concentrate on this case, as it contain implementation of both database and XML conversions.
Before describing how transformations were done, let us first mention tools that are used with this transformation. In case of database, EyeDB database management system was used [23]. It was already described in section 4.1.2. For parsing of XML files Xerces-C++ XML Parser [12], an open source parser supporting both DOM and SAX parsing methods.

[5]This can be, however, easily changed if necessary.

Every class representing system model (and described in 6.2) have two methods that are responsible for storing objects into requested format: `toXML()` and `toDB()` (further they both of them will be referenced as `to()` methods for simplicity). Each of those methods start with check if it is within the transaction (which basically defines whether method was called from outside, or from another `to()` method) and if necessary starts the transaction. Note that in case of XML format, `toXML()` should not be called for classes `TaskInstance`, `ProcElemMap` and `OperMode`, as XML schema defines them as childs of other objects. In case of storing data in XML format, methods create string representation of data structures and then output it to file. In case of EyeDB output conversion, methods create objects of classes that were created by EyeDB tools from ODL description, and then input them to database. If particular `to()` method begun the transaction, after successful execution it also ends it. Note that in case of EyeDB, objects are available in database only after the transaction is finished.
Note that none of `to()` methods define the target to which it should be stored. For databases it is understandable, as database creates the identifier itself, but in case of files, one should be able to define output file to which it should be stored. This is done using global identifier parameter that could be set or retrieved. It also keeps the identifier of just stored object.
Example of `to()` methods for `ProcElem` class are shown in listings 6.8 and 6.9.

Listing 6.8: `toXML()` method of `ProcElem` class

```
void ProcElem::toXML() const
{
        bool started = false;
        if (!in_transaction) {
                beginOutputTransaction();
                started = true;
        }

        file << "<" << procElemName << "␣";

        file << "name=\"" << name << "\"␣";

        file << ">" << endl;

        for (unsigned i=0; i<nModes; ++i) {
                modes[i]->toXML();
        }

        file << "</" << procElemName << ">" << endl;

        pes_oids[name] = const_cast<ProcElem *>(this);

        if (started) {
                endOutputTransaction();
        }
}
```

Listing 6.9: `toDB()` method of `ProcElem` class

```
void ProcElem::toDB() const
{
        bool started = false;
        if (!in_transaction) {
                db->transactionBegin();
                started = true;
                in_transaction = true;
        }
```

```
        eyedb_tool::ProcElem *pe = new eyedb_tool::ProcElem(db);
        pe->setName(name);
        pe->setModesCount(nModes);
        for (unsigned i=0; i < nModes; ++i) {
                const eyedb::Oid *x = opMode_oid(modes[i]);
                if (!x) {
                        modes[i]->toDB();
                        x = oid;
                }
                pe->setModesOid(i, *x);
        }

        pe->store();
        clearOid();
        setOid(pe->getOid());
        pes_oids[*oid] = const_cast<ProcElem*>(this);

        if (started) {
                db->transactionCommit();
                in_transaction = false;
        }

        pe->release();
}
```

Storing data using `to()` methods is simple. More complicate seems to be retrieving object from the data store, as we do not have object on which such method could be performed, we define `static` methods `fromDB()` and `fromXML()` (further mentioned as `from()` methods). Additionally, for each defined class we define function pointer, that will be used to run method from appropriate class. Therefore, it will be possible to create `RMTask` and `RMTaskInstance` objects, even though other classes may not know them. Implementation of `from()` methods is then similar to `to()` methods: setting up transaction if necessary, obtaining the object either from database or from parser, creating new object and ending transaction, if we begun it. Also similar approach, by using global variable, is used to maintain identifier of object that should be retrieved.
Example of `from()` methods for `ProcElem` class are shown in listings 6.10 and 6.11.

Listing 6.10: `fromXML()` method of `ProcElem` class

```
ProcElem *ProcElem::fromXML()
{
        ProcElem *result = NULL;

        bool started = false;
        DOMElement *procElem = NULL;
        if (!in_transaction) {
                beginInputTransaction();
                started = true;
                std::vector<DOMElement *> childs = parser->getChilds(elem, procElemName);
                if (childs.size() <= curProcElemId) {
                        endInputTransaction();
                        throw (unsigned)childs.size();
                }
                procElem = childs[curProcElemId++];
        } else {
                procElem = elem;
        }

        std::vector<DOMElement *> childs = parser->getChilds(procElem,
                        opModeElemName);
        unsigned numberModes = childs.size();
        OperMode **modes = new OperMode*[numberModes];

        DOMElement *temp = elem;
        for (unsigned i=0; i<numberModes; ++i) {
```

```
                elem = childs[i];
                modes[i] = operModeFromXML();
        }
        elem = temp;

        const XMLCh *name = xmlch("name");
        result = new ProcElem(
                        xmlch(procElem->getAttribute(name)),
                        modes,
                        numberModes
                        );

        pes_oids[xmlch(procElem->getAttribute(name))] = result;
        xmlrel(name);

        if (started) {
                endInputTransaction();
        }

        return result;
}

OperMode *OperMode::fromXML()
```

Listing 6.11: `fromDB()` method of `ProcElem` class

```
ProcElem *ProcElem::fromDB()
{
        ProcElem *result = NULL;
        bool started = false;
        if (!in_transaction) {
                db->transactionBegin();
                started = true;
                in_transaction = true;
        }

        eyedb::Object *o;
        db->loadObject(*oid, o);

        eyedb_tool::ProcElem *pe = eyedb_tool::ProcElem_c(o);

        OperMode **modes = new OperMode*[pe->getModesCount()];
        for (unsigned i=0; i<pe->getModesCount(); ++i) {
                eyedb::Oid *old = oid;
                eyedb::Oid x = pe->getModesOid(i);
                oid = &x;
                modes[i] = (*operModeFromDB)();
                oid = old;
        }

        result = new ProcElem(
                        pe->getName(),
                        modes,
                        pe->getModesCount()
                        );

        if (started) {
                db->transactionCommit();
                in_transaction = false;
        }

        pes_oids[*oid] = result;

        pe->release();

        return result;
}
```

Quite big advantage of creating such general methods, is the possibility of exchanging them easily. If it would be required to use different type of database, or change the format of XML file, the only `to()` and `from()` together with other interfacing methods (for handling transactions) need to be reimplemented accordingly.

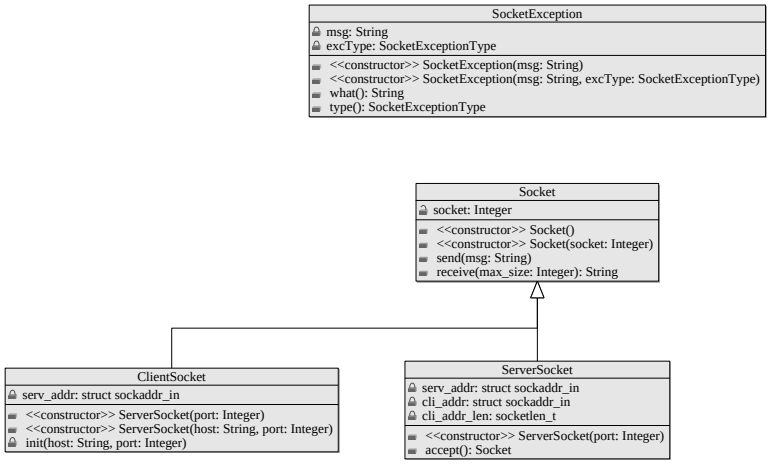

Figure 6.2: Class diagram of socket library

6.3.2 Communication

Unlike Java, C++ standard libraries do not offer any higher abstraction for socket communication then C. Therefore, one have to use the same primitives as in C. As in the tool we are creating multiple applications that should communicate using sockets, let us define library with classes that would simplify basic support for socket communication, like setting up connection, and sending or receiving messages. Class diagram of such library is presented in figure 6.2.
As one sees there is general class socket, that implements sending and receiving messages, and two specific classes handling server and client side of the connection. Implementation follows well-defined usage of methods and structures defined in `sys/socket.h`, `arpa/inet.h` and `sys/types.h` header files, which is not issue of this thesis. This gives us easy to use abstraction to implement servers.
Another important issue with communication is implementation of the protocol itself. For this we should define structure `Message` that will be responsible for message exchange. Definition of this class can be seen in listing 6.12. There are also two `enum` types defined that represent message type and command that is going to be send. Value of enumeration defines then index in array of strings, from which messages are created.

Listing 6.12: `Message` structure

```
enum Type { NULL_TYPE, ASK, RETURN, ERROR, MGMT, H_CONFIG, C_CONFIG, CONF_ACCEPT };

enum Command { NULL_COMMAND, BEST_SOLUTION, NEIGHBOUR, ALL_NEIGHBOURS, VALUE,
        ALL_VALUES, CLOSE_CONNECTION, START_TEMP, COOL_RATE, TEMP_LNGTH,
```

```
        NA_MOVES, W_DEGREE, C_DEGREE, W_RELIABILITY, C_RELIABILITY, W_ENERGY,
        C_ENERGY };

struct Message
{
        enum Type type;
        enum Command command;
        std::vector<std::string> payload;

        std::string toString();
        void fromString(std::string str);

        void send(Socket *sock);
        void receive(Socket *sock);

        Message()
                : type(NULL_TYPE), command(NULL_COMMAND) {}
};
```

As one may see `Message` structure keeps the value for message type and command that is send in it, and provides methods for converting message between text format and structure object. Additionally, it uses already discussed sockets to transmit and receive messages.
Protocol definition also provides general messages for constructing and retrieving most used valid messages. Therefore, all logic responsible for correct handling of protocol is kept in one place.

Chapter 7

Comparison of approaches

Having created three tools (or set of tools) doing exact the same thing, let us compare them now. In following sections we will first describe overall results of comparison from testing them (7.1), and then move specifically to comparison of both created tool sets (7.2).

7.1 Overall comparison

When doing comparison here we will not look at results obtained when tool was run, as code base during implementation for all tools was kept as close as it was possible, and in case of algorithms performing search it was exactly the same. Moreover, randomization in the heuristic algorithm, makes results obtained from two subsequent runs different. Instead our focus here will be placed on performance of each tool (or tool set). We also discuss here what had greatest impact on such results.

Let us here look at how long it took the application to actually perform the search. As heuristic search does perform different number of iterations, that in case of simulated annealing can be control by number of non-acceptable moves, temperature length and cooling ratio, we will need to consider the time needed for execution per iteration. Assuming high number of iterations of heuristic

Table 7.1: Timing results for tools

Tool	Average total time	Average number of iterations	Average time per iteration
Integrated tool	1.75 seconds	55975	3.13E-5 seconds
Database tool	575 seconds	56250	0.010 seconds
XML tool	847 seconds	55850	0.015 seconds

algorithm, it could be shown that its execution will dominate the time required for processing input data and providing output to the user. Averaged results of performed tests are shown in table 7.1.
All above tests were performed on Linux host with Intel® i7 Core^{TM} 2 x 2.6 GHz with HT=2[1], 4GB of RAM and SATA II 7200rpm disc.

As one might have expected, the integrated tool was a lot faster in performing the task then other to solutions and provided results almost immediately. This is due to fact that it does not needed any conversion between different formats, nor the multiprocess communication overhead. All calls for obtaining transformed model or result of cost function are simple method calls, and models are stored within the same address space, and accessible within whole applications by pointers or references. Taking this into account, it is quite impossible to compare it separately with other two. It is clear that in situations when time needed to obtain solution is crucial, integrated tool is always the best choice.
Also let us point here out that this is minimal time needed by the given system to perform such task. Time difference between integrated tool and set of tools, may be therefore treated as a penalty of using distribution.

7.2 Tool sets comparison

Surprising could be the difference between tool set using database and one using files for data exchange. When using EyeDB database management system, there was only small difference comparing to XML files being exchanged. To explain this let us look here into nature of both approaches.
Let us begin here with discussing how files are handled. As it is well-known

[1]HT - abbreviation for Hyper Threading; Intel technology allowing almost simultaneous execution of multiple threads on one processor core.

Table 7.2: Timing results depending of CPU power management policies

Policy	Total time for files	Time per iteration for files	Total time for database	Time per iteration for database
Power-save	1015 second	0.018 seconds	939 seconds	0.016 seconds
Performance	847 seconds	0.015 seconds	575 seconds	0.010 seconds

from computer architecture, all operations performed by processor access only its registers, or computer memory (possibly with using caching techniques), to which processor has access. If disk access is required, CPU orders bring data from disk to memory. When such data is changed, depending on the writing policy, it can be either written back to disk immediately or such writing could be delayed. Moreover, todays operating system when having unused memory, perform disk caching, that will allow better performance. As in case of doing heuristic search, files are written, and read by another process in short period of time, it is possible that this file is still kept in memory, and do not need to be brought from hard drive again. In such case, normal penalty that is considered when using files is minimized.
Let us now move to the database usage for sharing data. As described in [26] EyeDB database management system uses CORBA as a technology for sharing data with its client applications[2]. This in turn means that data needs to be send from server to client, and client stores it's own object in its private address space in memory, which gives transmission overhead, related to most of database systems. What is more important here is that whenever database is asked for object that is stored there, it needs to perform search through it's records, that takes CPU time. In case of files this time could be used by the tool itself. Therefore, unlike files that are disk-bound, usage of database is CPU-bound.

Very good example that shows how time needed for completing task is impacted in case of both CPU-bound and disk-bound applications, could be comparison of results of two runs that were done with two different power management policies for CPU. Results of them are shown in table 7.2.
On test machine difference between those two power management policies, was that in case of power-saving policy, CPU frequency was kept as low as possible throughout run of the program. Basically, frequency governor was mostly keeping it at lowest possible level, being 1.2GHz for each processor. On the other hand, in case of performance CPU was kept high, whenever there was a process using it. In general it was mostly over 2GHz for each processor.

[2]CORBA was already discussed in 4.2.

As one sees, time difference between both tool sets is much smaller in case of power-saving policies. However, still we see that solution that uses database management system is faster than the one using files.

Chapter 8

Conclusions

In this thesis we created tool allowing design space exploration and finding optimal solution for embedded system design. This task was achieved in three ways, allowing comparison of different techniques for sharing data between multiple tools.
Presented results show clearly that none of distributed tools can be compared with the integrated one with respect to performance. Integrated tools It also showed that choice technique in which data is being shared between tools between multiple tools is not always clear, as it may depend on the system on which the tool is going to be run. In general, however, we consider using database for data exchange as better solution, due to both its performance and transaction security mechanisms.

8.1 Future work

Work done during this thesis open many possibilities for future work, and much space for further improvements. First of all, heuristic algorithm that is used, do return only one solution, which it found the most optimal (basing on the provided cost function). However, this may make the cost function very complex if it was to catch all aspects of design. In most cases there are many possible solutions considered as optimal, and differing in aspects that might not be visible

only by looking at final value of cost function, As an example, let us consider two cases. In first one the energy consumption is the smallest, however, tasks do not leave much time space for possible further additions. Second case is opposite of the first one: energy is higher, however, tasks scheduled leave time space for additions that may come with time. Let us in this example assume, that differences between energy and degree of schedulability in those two cases, are such that cost function returns very close (or even equal) result. For the heuristic algorithm there will be no difference between those two solutions, and they will be equally optimal. However, those differences matter for the designer, and it would be much better if both such solutions would be presented allowing designer to choose among them. Such set of optimal solutions from which designer should be able to choose, is called Paretto optimal set of solutions. Although in current implementation it is possible to obtain the Paretto optimal solutions from the tool by running it multiple times with different settings of cost function, possible improvement would be to obtain them in one go.
Furthermore, system model used throughout this thesis and in created tool is quite simple, and therefore not very realistic. We considered here only set of independent tasks, that do not share resources, whereas in embedded systems known from everyday life tasks are interconnected and dependent on each other. They also share resources, for example communication bus or memory. Moreover, model of architecture considered here is relatively simple. Architecture consists only of multiple processing elements, and does not include buses that allow tasks them to communicate. We assumed in addition that tasks can be run at only discrete speeds that are available in processing elements. [7] describes, however, possibility of running tasks on processing elements, with discrete voltage levels, at virtually any speed by creating so called PWM-modes. Such PWM-mode can be created by running task at two different voltage levels, therefore creating illusion of continuous speed possibilities. Such considerations require also more realistic models of the processors that allow dynamic voltage scaling. Major things that our simple model lacks are the penalties that are present during operating mode changes: time required by the processor to switch to another operating, as well as power consumed by the processor during switching.
All above make the system model closer to reality. In turn it would allow to analyse complex embedded systems with more accuracy, and should be considered to be done in further tool development.

Another area of improvement is the way that tools in the set are connected together. This covers both communication between them, as well as sharing data. First thing to consider here is the use of more advanced protocol or communication method. As pointed in section 5.3.1, current protocol does not provide any error handling, and if used in distributed manner in the network environment (and not only on localhost) may possibly fail. Also the construction of server

application enforces that in case of unpredicted message received, connection is terminated, after sending the error message to second party. This effectively does not allow any error recovery.
Also note lack of parallelism in server applications. Each server is capable of handling only one connection at the time. If one chooses to use such tools in distributed manner, he will like to make them available to multiple parties, therefore, making the parallelism in the applications crucial for operation.

Appendix A

Protocol definition

Table A.1: Definition of valid messages used in the protocol.

Type	Command	Description
NONE	NONE	This is empty message. It does not have any specific type nor payload.
ASK	BEST_SOLUTION	Message that is sent towards heuristic search module asking for the optimal solution. Payload should consist one identifier of initial solution.
	NEIGHBOUR	Message that is sent towards transformation module asking for random neighbour to the given solution. Payload should consist one identifier of the solution for which neighbour should be found.
	ALL_NEIGHBOURS	Although not used in simulated annealing, this message requests from transformation module all neighbouring solutions to the current one. Payload represents identifier of current solution.

	VALUE	Message that is sent towards cost function module asking for value of cost function for the current solution. Payload should consists of one identifier of the solution in question.
	ALL_VALUES	Similarly to ALL_NEIGHBOURS command, it is not used in simulated annealing. Like previous command it is sent to cost function module asking for values of cost function for all solutions that are send. Payload in this case may consist of multiple identifiers of solutions.
RETURN	BEST_SOLUTION	Reply to ASK BEST_SOLUTION message. Its payload is identifier of the optimal solution that was found.
	NEIGHBOUR	Reply to ASK NEIGHBOUR message. Its payload is identifier of random neighbour that was found by transformation module.
	ALL_NEIGHBOURS	Reply to ASK ALL_NEIGHBOURS message. Its payload consists of set of identifiers that represent all neighbouring solutions that were found by transformation module.
	VALUE	Reply to ASK VALUE message. Its payload represents the value of the cost function calculated by costing module.
	ALL_VALUES	Reply to ASK ALL_VALUES message. Its payload consists of set of values of cost function for the solutions that were in question. Note that order of values of cost function must be the same as the order of solutions from the asking message.
ERROR	–	Message is reply to not understand message, or indicates any other error. After this message server closes connection.
MGMT	CLOSE_CONNECTION	Message requesting closure of connection. After this message is send, the link is removed.

HEUR_CONFIG	START_TEMP	Configures start temperature in heuristic module with simulated annealing. Payload contains new value of start temperature.
	COOL_RATE	Configures cooling rate in heuristic module with simulated annealing. Payload contains new value of cooling rate.
	TEMP_LENGTH	Configures temperature length in heuristic module with simulated annealing. Payload contains new value of temperature length.
	NA_MOVES	Configures number of not accepted moves during heuristic run with simulated annealing. Payload contains new value of non-accepted moves.
COST_CONFIG	WEIGHT_DEGREE	Configures weight of degree of schedulability in cost function module. Payload contains new value of this weight.
	CONFIG_DEGREE	Configures whether degree of schedulability should be taken into account when satisfying its goal in cost function module. Payload is either TRUE or FALSE.
	WEIGHT_RELIABILITY	Configures weight of reliability in cost function module. Payload contains new value of this weight.
	CONFIG_RELIABILITY	Configures whether reliability should be taken into account when satisfying its goal in cost function module. Payload is either TRUE or FALSE.
	WEIGHT_ENERGY	Configures weight of energy in cost function module. Payload contains new value of this weight.
	CONFIG_ENERGY	Configures whether energy should be taken into account when satisfying its goal in cost function module. Payload is either TRUE or FALSE.
CONF_ACCEPT	–	Message indicating that the module accepted the configuration change.

Appendix B

Data sharing

B.1 XML schema

Listing B.1: XML schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <xs:element name="tool" type="toolInfo"/>

        <xs:complexType name="toolInfo">
                <xs:sequence>
                        <xs:element name="task" type="taskInfo"
                                minOccurs="1"/>
                        <xs:element name="procElem" type="procElemInfo"
                                minOccurs="1"/>
                        <xs:element name="mapping" type="mappingInfo"
                                maxOccurs="1"/>
                </xs:sequence>
        </xs:complexType>

        <xs:complexType name="taskInfo">
                <xs:attribute name="name" type="xs:string"/>
                <xs:attribute name="wce" type="xs:decimal"/>
                <xs:attribute name="deadline" type="xs:decimal"/>
                <xs:attribute name="period" type="xs:decimal"/>
                <xs:attribute name="k" type="xs:integer"/>
        </xs:complexType>

        <xs:complexType name="taskInstanceInfo">
                <xs:attribute name="task" type="xs:string"/>
                <xs:attribute name="wcet" type="xs:decimal"/>
                <xs:attribute name="reWcet" type="xs:decimal"/>
                <xs:attribute name="opMode" type="xs:string"/>
                <xs:attribute name="reOpMode" type="xs:string"/>
                <xs:attribute name="k" type="xs:integer"/>
        </xs:complexType>

        <xs:complexType name="procElemInfo">
                <xs:attribute name="name" type="xs:string"/>
                <xs:sequence>
```

```
                        <xs:element name="opMode" type="opModeInfo"
                                minOccurs="1" />
                </xs:sequence>
        </xs:complexType>

        <xs:complexType name="opModeInfo">
                <xs:attribute name="name" type="xs:string" />
                <xs:attribute name="speed" type="xs:decimal" />
                <xs:attribute name="consPow" type="xs:decimal" />
                <xs:attribute name="voltageLevel" type="xs:decimal" />
                <xs:attribute name="lambda" type="xs:decimal" />
        </xs:complexType>

        <xs:complexType name="procElemMapInfo">
                <xs:attribute name="id" type="xs:integer" />
                <xs:attribute name="procElem" type="xs:string" />
                <xs:attribute name="lcm" type="xs:decimal" />
                <xs:attribute name="reExecMode" type="xs:string" />
                <xs:sequence>
                        <xs:element name="taskInstance" type="taskInstanceInfo" />
                </xs:sequence>
        </xs:complexType>

        <xs:complexType name="mappingInfo">
                <xs:attribute name="nextId" type="xs:integer" />
                <xs:attribute name="faultPeriod" type="xs:decimal" />
                <xs:attribute name="degreeGoal" type="xs:decimal" />
                <xs:attribute name="reliabilityGoal" type="xs:decimal" />
                <xs:attribute name="energyGoal" type="xs:decimal" />
                <xs:sequence>
                        <xs:element name="procElemMap" type="procElemMapInfo"
                                minOccurs="1" />
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

B.2 ODL description

Listing B.2: ODL description

```
class Task
{
        string name;
        double wce;
        double deadline;
        double period;
        int k;
};

class TaskInstance
{
        Task *task;
        double wcet;
        double reWcet;
        OperMode *opMode;
        OperMode *reOpMode;
        int k;
};

class ProcElem
{
        string name;
        OperMode *modes[];
};

class OperMode
{
        string name;
        double speed;
        double consPow;
        double voltageLevel;
        double lambda;
};

class ProcElemMap
{
        int id;
```

```
        ProcElem *pe;
        double lcm;
        OperMode *reExecMode;
        TaskInstance *tasks[];
};

class Mapping
{
        int nextId;
        double faultPeriod;
        double degreeGoal;
        double reliabilityGoal;
        double energyGoal;
        ProcElemMap *peMap[];
};
```

Appendix C

Required applications

To be able to build or run created tool, two dependencies need to be installed in the system first: Xerces-C++ XML parser (version 3.1.0 was used in the project) and EyeDB object-oriented database management system (version 2.8.8 was used).
For installation instructions of Xerces-C++ XML parser, please consult Xerces project webpage (http://xerces.apache.org/xerces-c/) for instructions regarding installation on your operating system. For users of Linux systems it is advised to check your system repository first.
For installation instructions of EyeDB, please consult EyeDB documentation package[1] and most importantly, EyeDB Installation.

Note that in this project we assumed the installation of both database management system and parser system wide, making all libraries and header files accessible.

[1] http://downloads.sourceforge.net/project/eyedb/EyeDB%20documentation/2.8.8/eyedb-doc-2.8.8.tar.gz

Bibliography

[1] Mysql. `http://www.mysql.com/`.

[2] Object database management systems, ODBMS.ORG. `http://www.odbms.org/`.

[3] Postgresql. `http://www.postgresql.org/`.

[4] Redis. `http://code.google.com/p/redis/`.

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[6] M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The object data standard: ODMG 3.0.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[7] E. Bini, G. Buttazzo, and G. Lipari. Minimizing cpu energy in real-time systems with discrete speed management. *ACM Trans. Embed. Comput. Syst.*, 8(4):1–23, 2009.

[8] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. *Real-Time Systems, Euromicro Conference on*, 0:0029, 1996.

[9] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[10] R. P. Dick, D. L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101, Washington, DC, USA, 1998. IEEE Computer Society.

[11] Apache Software Foundation. Cassandra. `http://cassandra.apache.org/`.

[12] Apache Software Foundation. Xerces-C++ XML Parser. `http://xerces.apache.org/xerces-c/`.

[13] M. Henning. The rise and fall of corba. *Queue*, 4(5):28–34, 2006.

[14] IBM. Rational rose realtime. `http://www.rational.com/products/rosert`.

[15] Objectivity Inc. Objectivity/DB. `http://www.objectivity.com/pages/objectivity`.

[16] ISO/IEC 9899:TC3. C99 standard with Technical Corrigendum 1, 2, 3 – Committee draft. Technical report, International Organization for Standardization, Geneva, Switzerland., 7. September 2007.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[18] L. Lavagno and C. Passerone. Design of embedded systems. In R. Zurawski, editor, *Embedded Systems Handbook*. CRC Press, Inc., 2004.

[19] Mathworks. Simulink and stateflow. `http://www.mathworks.com`.

[20] P. Pop. *Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems*. PhD thesis, Department of Computer and Information Science Linköpings universitet, 2003.

[21] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 233–238, New York, NY, USA, 2007. ACM.

[22] Artisan Software. Real time studio. `http://www.artisansw.com`.

[23] SYSRA. Eyedb. `http://www.eyedb.org/`.

[24] Inter Systems. InterSystems caché. `http://www.intersystems.com/cache`.

[25] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: an np-hard problem made easy. *Real-Time Syst.*, 4(2):145–165, 1992.

[26] E. Viara, E. Barillot, and G. Vaysseix. The eyedb oodbms. *International Database Engineering and Applications Symposium 1999*, pages 390–402, 1999.

[27] WC3. Extensible Markup Language (XML) 1.0 (Fifth edition). Technical report, The World Wide Web Consortium, 26. November 2008.

[28] The Free Encyclopedia Wikipedia. Annealing. `http://en.wikipedia.org/wiki/Annealing_(metallurgy)`.

[29] The Free Encyclopedia Wikipedia. NoSQL. `http://en.wikipedia.org/wiki/NoSQL`.

[30] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Transactions on Computers*, 99(PrePrints):1382–1397, 2009.

[31] D. Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 35–40, Washington, DC, USA, 2004. IEEE Computer Society.