Arts'Codes:
A New Methodology for the Development
of Real-Time Embedded Applications for
Control Systems

by

Aryeh Teitelbaum

B. Tech. & Appl. Sc. in computer Science

Submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

School of Aerospace, Mechanical and Manufacturing Engineering
RMIT University
Victoria Australia

March, 2006
DECLARATION

The author declares that:

• Except where due acknowledgement is made, the work is that of the author alone.

• The work has not been submitted previously, in whole or in part, to qualify for any other academic award.

• The content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

• Any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Aryeh Teitelbaum

Date …..26-February-2007..................
Acknowledgments

Thanks are due to G-od who sustains me, gave me the opportunity to live in His magnificent world, and supported me to accomplish this thesis. Thanks are due to all His emissaries:

To Prof. G. Mendelbaum who convinced me to enter in this research, guide me, and solved encountered problems.
To Prof. Herszberg for his loyal, patient guidance and help.
To Dr. R. Gallant and Dr. Y. Badihi for their advices.

Thanks to all engineering students who took part in this wide research, helping me on feasibilities of several approaches:
Thanks to Y. Blushtein and A. Toronovski on their help in the design evaluation.
To M. Goldsman, M. Shechter and S. Z. Zingerevitz on their attempt to apply rules of compilation.
To B. Avramovsky and M. Milshtein from the electronic-department, on their attempt to implement the tri-processor platform.

Thanks to the engineers who programmed with me the graphical editor:
To Y. Gordin, A. Serov and L. Muzychenka.

Thanks to the Jerusalem College of Technology for their help in my studies, and the grants they gave in order to achieve the method:
To Prof. J. Bodenheimer (president), Prof. Y. Friedman and Prof. M. Steiner (rectors).

Thanks to the RMIT University for the opportunity to accomplish this PhD thesis.

Thanks to my mother for her encouragement.
Thanks to my wife Tzippy for her assistance, and to my children: Moishi, Menahem, Yonni, Yoeli, Yossi, Libby, Eliezer, Hananel, Hilel and Taly, for their understanding on my long nights of work.
Thanks again to G(od) for all Thou gave me, thanks a lot; and I ask from Thou to continue giving.

תמה ושלום שבמה לבורה עולם
Contents

Summary xv
Preface 1
Part I: Introduction 2
1 Real Time Systems 3
   1.1 Profile 3
   1.2 Real-time problems 4
   1.3 Features of an ideal Real-time system 5
2 Diagrammatic Reasoning 6
   2.1 Evidences 6
   2.2 Diagram Role 7
   2.3 Ideal Diagram 7
   2.4 Criterion for Evaluation 8
   2.5 Cognitive Dimensions 8
   2.6 Cognitive Dimensions Questionnaire 8
3 Research Goals 9

PART II: Real-Time Design 10
4 Literature review on Design 11
   4.1 Finite State Machine 11
      4.1.1 Definition 11
      4.1.2 Mealy machine 11
      4.1.3 Moore machine 12
      4.1.4 Modern FSM 13
      4.1.5 Deterministic and non-deterministic FSM 14
      4.1.6 Advantages 14
      4.1.7 Disadvantages 14
      4.1.8 FSM evaluation for programming issues 15
      4.1.9 FSM example 15
   4.2 Petri’s Net 18
4.2.1 Petri's basics 18
4.2.2 Formal definition 18
4.2.3 Dynamic behavior 19
4.2.4 Example 20
4.3 Harel’s Statechart 22
4.3.1 Hierarchy 22
4.3.2 Concurrency 23
4.3.3 History pseudo-state 23
4.3.4 Communication 24
4.3.5 Formal representation 24
4.3.6 Example 25
4.4 Selic’s ROOM 27
4.4.1 Structure 27
4.4.2 Behavior 28
4.4.3 Example 30
4.5 UML 32
4.5.1 Purpose 32
4.5.2 Diagrams 32
4.6 Components 36
4.6.1 The UML definition 36
4.6.2 Other component definitions 37
4.6.3 Component relationships 37
4.6.4 Arts’Codes Definitions 40
4.7 Lavi’s Context Diagram 41
4.8 Analysis and discussion 42
4.8.1 Advantages and disadvantages of the various approaches 42
4.8.2 Adopted features 43

5 Proposal for the Design 44
5.1 Arts’Codes Visual Programming Language (VPL) 44
5.1.1 General presentation 44
5.1.2 Principles 45
5.1.3 Discussion on Arts’codes dynamic model 51
5.1.4 Graphical representation 53
5.1.5 Process of development 55
5.1.6 Example 59
5.2 Arts’Codes VPL formal representation 65
5.2.1 Formal representation 65
5.2.2 Dynamic model for execution 67
8.4 Analysis and discussion
  8.4.1 Simple Approach
  8.4.2 Multi-threading
  8.4.3 Pre-run scheduling
  8.4.4 Synchronous
  8.4.5 Adopted features

9 Proposal for the Execution
  9.1 Abstract
  9.2 Time domain
    9.2.1 Complexity
    9.2.2 The time model
    9.2.3 Rationality
    9.2.4 Restrictions
    9.2.5 Heavy Reactions
  9.3 Arts execution platform
    9.3.1 Reactive Kernel (RK)
      9.3.1.1 Concurrency robustness and determinism
      9.3.1.2 Execution by priority
    9.3.2 Environment Kernel (EK)
    9.3.3 The Synchrony Co-Processor (SC)
      9.3.3.1 Step pace making
      9.3.3.2 Heavy-Reactions execution
      9.3.3.3 Host communication
  9.4 Arts Principles
    9.4.1 Abstract
    9.4.2 The Arts'Codes Synchrony Hypothesis version
      9.4.2.1 Reminding the original Synchrony Hypothesis
      9.4.2.2 Complexity
      9.4.2.3 The Arts version
      9.4.2.4 Synchronous and Multi-threading platforms interleaving
    9.4.3 The Arts reactive-cell version
      9.4.3.1 The Andre's original reactive-cell
      9.4.3.2 Realization of the original approach
      9.4.3.3 Arts'Codes Inter-level transitions complexity
      9.4.3.4 The Arts'Codes improved reactive-cell
    9.4.4 The Arts "Java's Thread::Run" version
    9.4.5 The Arts improved "Parnas's pre-run-time scheduling"
      9.4.5.1 On-line versus off-line scheduling
11.2.5 Heavy Transitions 157
11.2.6 Hierarchic Component Activation 158
11.2.7 Exception Handling 158
11.2.8 The between steps actions 160
11.2.9 Virtual and Physical devices (VD & PD) 160
11.2.10 Events 161
11.3 Software Execution Model 162
11.3.1 Abstract 162
11.3.2 Reactive Kernel (RK) 162
11.3.3 Environment Kernel (EK) 167
11.3.4 Synchrony Co-Processor (SC) 169
11.4 Rules of Compilation 170
11.4.1 Derivation of classes 171
11.4.2 Addition of members 171
11.4.3 Abstract method implementation 173
11.4.4 States 174

12  CHILD Language 177
12.1 Abstract 177
12.2 CHILD Language Syntax 177
12.2.1 Component declaration 177
12.2.2 Inputs and outputs declaration 179
12.2.3 Sub-components declaration 179
12.2.4 Manager and Guard 179
12.3 Examples 180

13  Compiler Rules to provable Temporal Logic 182
13.1 Abstract 182
13.1.1 Principle of the proving 182
13.2 Compiling rules 183
13.2.1 Concordance Rules between CHILD and TCOM/PTL 183
13.3 Compiler architecture 187
13.3.1 CHILD to TCOM compilation 187
13.3.2 TCOM to PTL compilation 189
13.3.3 PTL specifications validation 192
PART V: Case Study

14 Arts'Codes Components Editor (ACE)

15 The washing machine case study

15.1 Statement of needs (SON)
   15.1.1 The needs
   15.1.2 The Washing Programs
   15.1.3 Control Panel and display
   15.1.4 Physical definitions

15.2 Design
   15.2.1 1st phase: Environment wrapping
   15.2.2 2nd phase: Hierarchic building of the component structure
   15.2.3 3rd phase: Behavior Molding
   15.2.4 4th phase: Exception Handling
   15.2.5 Virtual and Physical devices

15.3 Code generation
   15.3.1 The ReactiveKernelRoot structural translation
   15.3.2 The ReactiveKernelRoot behavior translation

15.4 Execution

Part VI: Summary

16 Conclusions
   16.1 Natural and formal design
   16.2 Visual Programming Language abilities
   16.3 Full and robust executable code
   16.4 Full and robust execution platform
   16.5 Fast and known response time
   16.6 Evaluation

17 Main Contributions
   17.1 Design Contribution: Codes
   17.2 Execution model Contribution: Arts
   17.3 Arts'Codes Component editor
18 Further Researches  
18.1 Components polymorphism  
18.2 Arts'Codes pre-defined design patterns  
18.3 Tri-processor platform  
18.4 Specific C++ features translation to ANSI C  

Bibliography  

Appendices  

Appendix A  Execution Platform Classes  
Appendix B  ACE screen views  
Appendix C  CHILD syntax rules
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Finite State Machine</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Mealy machine</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>The Conveyor belt FSM</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>Petri's transition (firing) rule: (a) before, (b) after</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>Petri's non-determinism</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>Petri's concurrency pattern</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>Petri's loop representation</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>A Petri net state-machine representation</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Statechart's hierarchy</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>Statechart's concurrency</td>
<td>23</td>
</tr>
<tr>
<td>11</td>
<td>Statechart's history pseudo-state</td>
<td>24</td>
</tr>
<tr>
<td>12</td>
<td>Multi-alarm watch example</td>
<td>26</td>
</tr>
<tr>
<td>13</td>
<td>Actors binding and ports</td>
<td>27</td>
</tr>
<tr>
<td>14</td>
<td>ROOM's Finite State Machine</td>
<td>29</td>
</tr>
<tr>
<td>15</td>
<td>ROOM's concurrency notation (comparing to Statechart)</td>
<td>30</td>
</tr>
<tr>
<td>16</td>
<td>Rudimentary computer structure</td>
<td>31</td>
</tr>
<tr>
<td>17</td>
<td>Class Diagram contents</td>
<td>33</td>
</tr>
<tr>
<td>18</td>
<td>Classes interconnections</td>
<td>34</td>
</tr>
<tr>
<td>19</td>
<td>Sequence Diagram</td>
<td>34</td>
</tr>
<tr>
<td>20</td>
<td>ECSAM's Context Diagram</td>
<td>41</td>
</tr>
<tr>
<td>21</td>
<td>Sockets, Holes and interconnections</td>
<td>53</td>
</tr>
<tr>
<td>22</td>
<td>Component's structure</td>
<td>54</td>
</tr>
<tr>
<td>23</td>
<td>Manager graphical representation</td>
<td>55</td>
</tr>
<tr>
<td>24</td>
<td>The Guard graphical representation</td>
<td>55</td>
</tr>
<tr>
<td>25</td>
<td>Assertion's dialog boxes</td>
<td>56</td>
</tr>
<tr>
<td>26</td>
<td>Home Heater System Environment</td>
<td>60</td>
</tr>
<tr>
<td>27</td>
<td>Reactive Kernel Root static-view</td>
<td>61</td>
</tr>
<tr>
<td>28(a)</td>
<td>Reactive Kernel Root dynamic-view</td>
<td>62</td>
</tr>
<tr>
<td>28(b)</td>
<td>Motor dynamic-view</td>
<td>63</td>
</tr>
<tr>
<td>28(c)</td>
<td>Circulator dynamic-view</td>
<td>64</td>
</tr>
<tr>
<td>28(d)</td>
<td>Furnace dynamic-view</td>
<td>64</td>
</tr>
<tr>
<td>29(a)</td>
<td>Furnace Exception handling</td>
<td>65</td>
</tr>
<tr>
<td>29(b)</td>
<td>Motor Exception handling</td>
<td>65</td>
</tr>
</tbody>
</table>
Figure 30: SyncChart Notations 101
Figure 31: SyncChart example 101
Figure 32: SSM Recursive/Parallel algorithm 105
Figure 33: Misleading example with state-transitions 107
Figure 34: Misleading example with history connector 108
Figure 35: Tri-processor Arts prototype block-diagram 138
Figure 36: The ComponentT class 147
Figure 37: The VirtualDevice class 148
Figure 38: The ClockT class 149
Figure 39: The AssertionT class 150
Figure 40: The HeavyReactions class 151
Figure 41: The Arts tri-processor execution platform 165
Figure 42: The Pump component 181
Figure 43: CHILD to TCOM Compilation Part Structure 188
Figure 44: Step-by-step menu opening 188
Figure 45: CHILD 2 TCOM Compiler Error Notification 189
Figure 46: TCOM to PTL Compilation Part Structure 190
Figure 47: Example of TCOM Program’s Hierarchical Structure 191
Figure 48(a): Result Of the Hierarchy splitting: “a.tcom” 191
Figure 48(b): Result Of the Hierarchy splitting: “b.tcom” 191
Figure 48(c): Result Of the Hierarchy splitting: “main.tcom” 191
Figure 49: The washing-machine physical devices 205
Figure 50: Washing-machine environment definition 206
Figure 51: The reactive-kernel-root component 207
Figure 52: The ProgCtrl component 208
Figure 53: The PanelCtrl component 209
Figure 54: Third level components 210
Figure 55: The Regulatr Manager 211
Figure 56: The Motor Manager 211
Figure 57: The ProgCtrl Manager 212
Figure 58: The reactive-kernel-root Manager 213
Figure 59: The ProgCtrl Guard 214
Figure 60: Button Virtual Device (static and dynamic views) 215
List of Tables

Table 1: Manager – Guard different models comparison 52
Table 2: Questionnaire results 80
Table 3: Questionnaire results grouped by CDs 81
Table 4: Questionnaire results 82
Table 5: Questionnaire No. 2 results grouped by CDs 82
Table 6: Questionnaires comparison 83
Embedded real-time applications have to allow interaction between the control computer and the controlled environment. Controlling the environment requires in particular to take into account its time constraints and critical logical conditions. One of the main programmer efforts in real-time application's development is to trace the incoming events, and to perform reactions based on the current system status, according to the application requirements. All this have to be handled, although external events may come in the middle of a critical reaction, which may disturb it.

This problem involves two difficulties:
- The cognitive efforts to percept the problem, and consequently to express the solution.
- The correct translation of this solution to code.

Two requirements were defined in this research in order to achieve high-quality performance: clearness and robustness, clearness in the design, and robustness in the execution.

In this work the author proposes a methodology and a tool for real-time application's development that uses or implies an innovated form of design based on natural-cognitive researches. This design method has clear compilation's rules to produce an Object-Oriented light-code, suitable for embedded platforms. These compilation's rules introduce to the code implicit security and synchronization's elements, to support robust execution.

In this methodology, clear development phases were defined, using a high-degree of reuse and even polymorphism, which were emphasized in the research. Several existing ideas were improved/adapted and synthesized together with the author's innovation, creating the Arts'Codes method for real-time application development.

The work includes cognitive evaluations, assuring the natural skills of the design.

Arts'Codes method proposes a natural VPL (Visual Programming Language) for real-time applications, based on hierarchic components. This VPL is built on a
minimum of diagrams: one for the static architecture and one for the dynamic behavior, with a similar restricted notation at all levels. These two diagrams (static architecture and dynamic behavior) are interleaved in a unified view. This method was implemented by building a suitable graphic editor, which automatically compiles the applications diagrams in a light and robust Object-Oriented code (based on Parallel Automata FSM), and by building an execution compact software platform.

Furthermore, the parallel automata FSM are translated automatically in PTL temporal formula defining the goals and the behaviors of the components, permitting to prove a-priory that the components behaviors are consistent to their goals.

The execution platform is based on a restricted implementation of the synchrony hypothesis and on a powerful model of execution: the parallel automata FSM. These Parallel Automata describe the dynamic behaviors of the components and allows to implement run-time exceptions handling too.
In addition, the research proposes a tri-processor execution hardware platform, which supports a hybrid synchronous/multi-threading execution.
This method will contribute to versatile, clear and robust real-time application’s development.
Preface

The Arts'Codes method, developed in this thesis, is an executable Visual Programming Language for real-time application's development. It emerged from cognitive skills and real-time system researches. On one hand, it molds the software design according the natural cognitive skills, without deteriorating execution performance. On the other hand, it proposed also a robust execution platform which takes into account only dry technical restrictions which does not depend on the design cognitive problems.

A bridge between these two contradictory dimensions (cognition vs. execution) was built; by defining well-defined rules, in order to conserve the advantages of each dimension.

This thesis is split into six parts:
Part I, "Introduction", introduces the reader into this research field, which in fact joins two opposite parts: Diagrammatic Reasoning and Real-Time Systems; finishing with the statement list of the research goals.

Part II, "Real-Time Design", presents the design aspect of the methodology, named Codes, prefaced by the suitable literature review, and finishing with the method evaluation.

Part III, "Real-Time Execution Platform", presents the execution principles of the proposed platform named Arts. These execution principles are well-based on nowadays research, with improvements and innovations.

Part IV, "From Design to Execution", describes the methodology of building an automatically bridge for the creation of suitable light executable code, molded by the Codes design, to be run in the Arts platform.

Part V, "Examples", demonstrate the effectiveness of the method by applying a wide case-study.

Part VI, "Summary", discusses the research contribution, and powerful features for further researches.
Part I

Introduction

This part introduces the reader into the two main fields which escort this research, being: Real-Time Systems and Diagrammatic Reasoning. After a short tour on these fields, clear research goals are listed.

Chapters for this part:
1  Real Time Systems
2  Diagrammatic Reasoning
3  Research Goals
1

Real Time Systems

1.1 Profile

"Real time is a level of computer responsiveness that a user senses as sufficiently immediate or that enables the computer to keep up with some external process (for example, to present visualizations of the weather as it constantly changes). Real-time is an adjective pertaining to computers or processes that operate in real time. Real time describes a human rather than a machine sense of time." [1]

What exactly is meant by real-time? It is really a subjective adjective?

Let’s compare the previous definition with a more objective one:

"A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred." (Donald Gillies) [2]

A good example is a soda factory where the bottles are located on a conveyor belt. The conveyor belt role is to bring the bottle through two stations: filling and sealing. When the bottle arrives under the filling station the conveyor belt stops, the tap is opened and the bottle is filled. Then the conveyor belt moves again to bring the bottle to the sealing station.

For this simple example let us assume that there is only one bottle on the conveyor belt at a time, and there is a robot that puts an empty bottle on the empty belt from one side, and a second robot that receives the filled bottle on the other size.

Let us define a time restrictions for the conveyor belt as follows:

The conveyor belt must stop immediately when the bottle is under the filling/sealing station.

If this time restriction is not fulfilled, e.g. the conveyor belt stops a little bit later, then the bottle will not be filled, and the soda will be spilled.
Note that in this case all the reactions were performed in a correct way, but not at the correct time. In a real-time system the correctness depends also upon the time that is performed.

Real-time is obviously an objective adjective.

For Operating Systems, Real-time is defined as:
"The ability of the operating system to provide a required level of service in a bounded response time" [3]

Then we can emphasize this definition:
"One will also see references to real-time systems when what is meant is just fast. It might be worth pointing out that real-time is not necessarily synonymous with fast; that is, it is not the latency of the response per se that is at issue (it could be of the order of seconds), but the fact that a bounded latency sufficient to solve the problem at hand is guaranteed by the system." [2]

Real-time systems are in fact reactive systems with timeliness restrictions. A Reactive system is one that is in continuous interaction with its environment. A Reactive program is a program that maintains a permanent interaction with their environment, reacting to inputs coming from this environment by sending outputs to it. A Reactive Kernel is the part of the reactive program that contains the logic of the system. It handles logical inputs and outputs. It decides what computations and what outputs must be generated in reaction to the inputs.

### 1.2 Real-time problems

Real time systems have to allow interaction between the control computer and the controlled environment. Controlling the environment requires in particular to take into account its time constraints and critical logical conditions.

One of the main programmer efforts in real time applications development is to trace the incoming events, and to make reactions based on the current system status, according to the application requirements.

The "art" of defining and developing real-Time applications, may be a hard task. The events may come in parallel or sequentially, they may be implied in logical conditions or not. The real-time application may be made of many small sub-components, each one having its own behavior; they can be triggered by shared events. The sub-components behavior may be also connected through logical
conditions. The various parts of the application can run as parallel tasks and necessitate synchronizations between them. Some tasks may have critical sections.

External events may come in the middle of a critical reaction, and disturb it.

In summary, the design method should be sufficiently clear, to define the application in the most simple manner, and be sufficiently robust, to execute the application accurately according to its requirements.

The system must be sufficiently fast, to take into account the timing requirements and the synchronization definitions.

The proof of timing requirements, may be also a hard task, and almost impossible in a multitasking system.

The reaction times must be defined and measured, including the possible disturbance by other tasks running in the same system.

1.3 Features of an ideal Real-time system

§ **Clearness:** the behavior is defined in a well readable way.

§ **Reactiveness:** it has to provide a well defined output for each input

§ **Determinism:** it has to produce only one output for each input.

§ **Correctness:** it does the right thing all the time.

§ **Robustness:** it does a right thing under unplanned circumstances.

§ **Timeliness:** it has to produce the reactions meeting with the time constraints.

§ **Concurrency:** it enables to model the parallel nature of the application.

There are also other requirements that are not directly deduced from the pure real-time needs, but in fact they exist.

For instance, embedded systems are real-time systems which must be adapted to small systems poor in memory and processor capabilities [4].

These restrictions dictates the following requirements:

§ **Compact Code (compactness):** the code and variables must be shortest and efficient as possible

§ The Real Time Operating Systems (RTOS) must be **scalable**, i.e. the RTOS is structured so that only the needed components are included.
In this section a literature survey concerning to human cognition, specially concentrated on the aspect of diagrams as an aid for programming.
This field supplies tools for the definition and evaluation of diagram’s features in the design stage of the new proposed methodology.

2.1 Evidences
As an introduction to diagrammatic reasoning a set of evidences is presented. These evidences were well analyzed by Petre and Blackwell [15].
"There is widespread anecdotal evidence that expert programmers make use of visual mental images when they are designing programs. This evidence is used to justify the use of diagrams and visual programming languages during software design."

Many anecdotes were presented by Lammers [16] who interviewed well-known programmers, the following interview is from Charles Simony:
"The first step in programming is imagining ... I like to imagine the structures that are being maintained, the structures that represent the reality I want to code ... The code for the most part writes itself, but it’s the data structures I maintain that are the key. They come first and I keep them in my mind throughout the entire process."

The next one is from the famous Bill Gates:
"You have to simulate in your mind how the program’s going to work, and you have to have a complete grasp of how the various pieces of the program work together."

The following anecdotes are brought by Petre’s [17] who also studies expert programmers (this paper was not published but was quoted by Petre and Blackwell [15]):
"One of the earliest things is to visualize this structure in my head, a dynamic structure, so I can think about how things fit together and how they work ... and once I have the structure fairly strong and clear in my mind, I move it around and move around inside it, examining it and tweaking it ... "

2
Diagrammatic Reasoning

"
"I think of these systems of relationships as alive, interacting beings ... I make this dirty dynamic mental representation, a sort of organic thing ... "

Petre and Blackwell [15] summarize these anecdotes emphasizing what it is called in software design: the static and dynamic view. "Each of these anecdotes includes two key elements: a structure of information, and how it works. These mental structures, part of initial solution planning, apparently embody information inter-relationships and take into account manipulations and functions to be performed in processing information."

From these anecdotes the author adopted three principles in real-time applications design [18]:

§ The first guiding principle is the Component-Oriented approach.
§ The second principle is the Static and Dynamic views of the design "Each of these anecdotes includes two key elements: a structure of information, and how it works"[15].
§ The third principle is the diagrammatic interleaving of these Static and Dynamic views, "These mental structures, part of initial solution planning, apparently embody information interrelationships and take into account manipulations and functions to be performed in processing information" [15].

2.2 Diagram Role

Diagrams in association to computers have two mainly roles [19]:

§ "as an external representation employed as a private aid to thought"
§ "a communication medium between members of a software project team".

These roles are able to be realized by diagrams because diagrams "provide non-visual information in a visual form". Furthermore "the processes involved in the visual perception of the real world and the processes involved in the visual perception of pictures are identical" [20][21].

Diagrams have another advantage, following Stenning & Oberlander (1995) because "diagrams aid cognitive processing because of their specificity - the way in which they limit abstraction. Diagrams have fewer interpretations, so are more tractable than unconstrained textual notations" [21].

2.3 Ideal Diagram

A definition of the ideal diagram:

"Solving a problem simply means representing it so as to make the solution transparent" (Simon).
2.4 Criterion for Evaluation

There may be several approaches to evaluate cognitive aspects of a diagram, one of them is by defining a set of parameters giving a score for each of them. This set of parameters is called by Blackwell et al. [23] Cognitive Dimensions. "Cognitive Dimensions of Notations (CDs) is a framework for describing the usability of notational systems (e.g. word processors, computer-aided design tools, or music notation) and information artifacts (e.g. watches, radios or central heating controllers)".

These Cognitive Dimensions includes "definitions of notations and notational systems, characterization of the human activities involving notational systems, a description of the ways that multiple notations can interact within a single system, and a minimal process for applying the resulting insights in a design context for use in evaluating and improving a design." [23]

2.5 Cognitive Dimensions

The following list of Cognitive Dimensions is defined by Blackwell et al. [23]:

- **Viscosity**: resistance to change.
- **Premature commitment**: constraints on the order of doing things.
- **Hidden dependencies**: important links between entities are not visible.
- **Role expressiveness**: the purpose of an entity is readily inferred.
- **Error-proneness**: the notation invites mistakes and the system gives little protection.
- **Abstraction**: types and availability of abstraction mechanisms.
- **Secondary notation**: extra information in means other than formal syntax.
- **Closeness of mapping**: closeness of representation to domain.
- **Consistency**: similar semantics are expressed in similar syntactic forms.
- **Diffuseness**: verbosity of language.
- **Hard mental operations**: high demand on cognitive resources.
- ** Provisionality**: degree of commitment to actions or marks.
- **Progressive evaluation**: work-to-date can be checked at any time.

2.6 Cognitive Dimensions Questionnaire

A questionnaire model for cognitive evaluation is proposed by Blackwel and Green [22], based on the Cognitive dimensions "showing that a generalised CDs questionnaire indeed a suitable tool for user evaluation. Not surprisingly, some problems emerged as well".
Research Goals

The design method and implementation of Real-Time (RT) applications must be sufficiently clear to define the application in its simplest form, and be sufficiently robust to execute reliably according to the system requirements.

In this research the main goal is, to develop a clear and robust embedded real-time application methodology for the design and the code generation, and to propose a light-code execution platform.

This main goal was split in the following sub-goals:

1. Finding a formal method to design real time applications in a clear and simple way, based on a natural cognitive human’s approach, including modularity and encapsulation.
2. Defining translation rules, to automatically compile the design diagrams into a validation formal language and into a light but robust code.
3. Finding a type of code enabling expression of: parallelism, synchronization, events, conditions and clocks, with emphasis on robustness.
4. Building a new type of execution platform, which allows executing the mentioned code. This platform must support implicit synchronization and robust mechanism (transparent to the programmer), in order to enhance the robustness with a minimum programmer effort.
5. Optimizing the execution platforms to assure a known and reliable response time.
6. Presenting clear metrics to evaluate the proposed methodology.
PART II

Real-Time Design

Arts'Codes method proposes a Visual Programming Language (VPL) for real-time applications. As implied, the programming style must be graphical.

At the beginning of the research many existing methods for application's design were study, and many attempts of synthesis and innovations were considered.

In this part the Author presents a literature review of the relevant existing methods, the proposed method and its evaluation.

Chapters for this part:

4 Literature review on Design
5 Proposal for the Design
6 Evaluation of the Design by users
7 Summary of Design contribution
In the following sections a relevant reactive applications' methodologies review is presented, in order to discuss later, their properties. These methodologies were the contributors of the Arts'Codes method, which emerged on their merits, synthesizing existing features together with author's innovations.

4.1 Finite State Machine

4.1.1 Definition

"A model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated with transitions (Mealy machine) or states (Moore machine), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (nondeterministic finite state machine), one or more states designated as accepting states (recognizer), etc." [6]

![Figure 1: Finite State Machine](image)

4.1.2 Mealy machine

The Mealy machine [9] is named on its promoter G. H. Mealy. He define a FSM by 6 components as follows:

1. a finite set of states \( S \)
2. a finite set of inputs \( \Sigma \)
3. a finite set of outputs \( \Lambda \)
4. a transition function \( T : S \times \Sigma \rightarrow S \).
5. an output function \( G : S \times \Sigma \rightarrow \Lambda \).
6. an initial state \( s \)
The reactions of a Mealy machine are defined for each couple of source state and input, having each couple an output and target state. This definition is the most popular and we will discuss it later [11].

![Figure 2: Mealy machine][1]

In Fig. 2 we can see a typical Mealy machine. It has three states (s0, s1 and s2), it receives two inputs (0 and 1) and produces two outputs (0 and 1). The arrows show the transitions by connecting two states in a specific direction, specifying in this way the source and target states. On each arrow there are two numbers separate by a slash, these are the input and the correspondent output for the couple source-state/input.

For example referring to Fig. 2, if the input ‘0’ arrives when the Mealy machine is in state s1, then it produces the output ‘1’ and it changes to state s2. But if the same input arrives when the Mealy machine is in state s0, then it produces the output ‘0’ and changes to state s2. The state don’t must be changed, for example if the input ‘0’ arrives when the Mealy machine is in state s2, it produces the output ‘0’ and stays in the same state.

### 4.1.3 Moore machine

In contrast to the Mealy machine, the output of a Moore finite state machine depends only on the current state and does not depend on the current input [11]. The Moore machine was proposed by E. F. Moore [10], and is defined by (also) 6 components:

1. a finite set of states (Q)
2. a finite set of inputs (Σ)
3. a finite set of outputs (Δ)
4. a transition function (δ : Q × Σ → Q)
5. an output function (λ : Q → Δ)
6. a initial state (q0)
As we can see a transition of a Moore machine (see component 4) does not defines an output. The output (see component 5) is the same for all transitions and depends only by the target state (each state has a constant output).

4.1.4 Modern FSM

Selic & Garth Gullekson [8] defines the FSM as a composition of:

1. A finite set of input events
2. A finite set of output events
3. A finite set of states
4. A function that maps states and input to output
5. A function that maps states and inputs to states
6. the initial state

As we can see they adopt the Mealy machine definition, because the output depends on the transition.

For a simplification and in order to translate the FSM elements to a more similar software design, let us define the following general elements:

1. A set of events/Conditions
2. A set of actions
3. A set of states
4. A set of Transitions compound of:
   - source state
   - trigger (condition/event)
   - action
   - destination state.
5. The initial state.

Mendelbaum, Teitelbaum et al. [50] defined a parallel automaton on a hierarchic basis as a set of transitions where parallel conditions induce parallel reactions and sub-automata activations.

Each transition of the parallel automaton table is written in a product $\pi$ form:

/event-flags, conditions/ /state/
/reaction, output/ /new subAutom/ /new state/
/$\pi(ef_i)^{C_j \wedge \nu_k}/S_0/$ /$r_n \wedge \pi(e_q)/-A_y+A_z/ /S_d/$

And it is read as following:

When all the expected events $\pi(f_i)$ arrived and the conditions $c_j \wedge \nu_k$ is true, then the automaton reacts according to its current state $S_0$ by executing a reaction
function $r_{ir}$ emitting output-events $\pi(e_{ir})$, updating its state $S_d$ and activating and deactivating automata $-A_r+A_r$.

### 4.1.5 Deterministic and non-deterministic FSM

A deterministic FSM has only one transition for each state/input couple. In a non-deterministic FSM, there can be more than one transition for each state/input couple.

The non-deterministic FSM has to be converted in a deterministic one, in order to be implemented, certainly by adding more states [13].

#### 4.1.6 Advantages [14]

**Simplicity**

The design by a FSM is simple. Due to their simplicity, the design of a small application by FSM is quick and can be implemented in a short time. Novices are able to design their applications based on a light tutorial. Furthermore due to the FSM simplicity also their execution on a microprocessor is efficient (low CPU consumption and small storage) and simple to debug.

The FSM translation to code is very simple, e.g. one of the FSM implementation methods is by a two dimensional matrix. The events are represented in the rows and the states by the columns. Each row/column intersection holds the action to be executed and the number of the target state.

**Predictability**

In a deterministic FSM given a series of inputs and the current state, the outputs series can be predicted including the target state. These advantages are important for critical systems where non-predicted events may cause damage.

**Long History**

FSM was use for a long time, and was researched by many people. Many analyzers and verifications tools were built, and therefore it is assumed to be a very stable tool for real-time systems.

#### 4.1.7 Disadvantages [14]

**Complex systems**

For large and complex systems FSM implementation may be a hard task, and certainly difficult to manage. State explosion (a very large number of states) will increase the cognitive effort for understanding. The FSM simplicity will fall by
state additions. Also adding more transitions will cause a fair degree of "spaghetti-factor" increasing the cognitive efforts of transitions tracking.

**Sequentiability of execution**
Real-time applications have some times parallel programming nature, this nature of programming simplifies the design and maintenance.
FSM don’t involves parallel design it has purely sequential robotic properties. The concurrent behavior of a system it is out of the FSM scope.

**Event driven**
FSM is built for event-driven systems. Not all problems have an event nature, e.g. not all systems are suitable to be defined in terms of events, states and outputs.

**Predictability**
Prediction is a positive property probably for most systems but, there are systems that precisely must be unpredicted, e.g. computer games.

### 4.1.8 FSM evaluation for programming issues
Benveniste and Berry [12] evaluates the FSM for programming goals as follows: "the automata (FSM) have numerous advantages: they are deterministic, efficient, they can be automatically analyzed by numerous available verification systems. However, they have a severe drawback: they do not directly support hierarchical design and concurrency. A small change to a specification can provoke a complete transformation of an automaton. When they are put in cooperation, separately small and pretty automata can yield a big ugly one. As soon as they are large, automata become impossible to understand for human beings".

### 4.1.9 FSM example
Returning to the soda factory example, let us design an FSM for the conveyor belt controller.
Let us define the FSM by the 5 elements as proposed in section 4.1.4:

**Events**
- $e\_\text{BottleEnter}$: occurs when the robot puts an empty bottle on the empty belt.
- $e\_\text{UnderFilling}$: occurs when the empty bottle is under the filling station.
§ e_Full: occurs when the bottle was filled.
§ e_UnderClosing: occurs when the full bottle is under the closing station.
§ e_Closed: occurs when the bottle was closed.
§ e_EmptyBelt: occurs when the filled and closed bottle was taken from the belt, and now the belt is empty.

Actions
§ MoveBelt: turn on the conveyor.
§ StopBelt: turn off the conveyor.
§ OpenFilling: open the tap
§ CloseFilling: close the tap
§ DoClose: Put a seal

States
§ isEmptyBelt: there is no bottle on the belt (the conveyor is off).
§ isMovingToFilling: waiting that the bottle arrives to the station filling (the conveyor is on).
§ isFilling: the bottle is in a filling process (the conveyor is off).
§ isMovingToClosing: waiting that the bottle arrives to the station closing (the conveyor is on).
§ isClosing: the bottle is in a closing process (the conveyor is off).
§ isMovingOut: the bottle is ready and it is moving out (the conveyor is on).

Transitions
1. isEmptyBelt (state), e_BottleEnter -> isMovingToFilling, MoveBelt: If the conveyor belt is in the isEmptyBelt state, and a bottle enters (e_BottleEnter), then the conveyor belt transits or passes to the isMovingToFilling state, and the MoveBelt action is performed.
2. isMovingToFilling, e_UnderFilling -> isFilling, StopBelt, OpenFilling: If the conveyor belt is in the isMovingToFilling state, and the bottle pass under the filling station (e_UnderFilling), then the conveyor belt transits to the isFilling state, and the StopBelt followed by the OpenFilling actions are performed.
3. isFilling, e_Full -> isMovingToClosing, MoveBelt: If the conveyor belt is in the isFilling state, and the bottle is full (e_Full), then the conveyor belt transits to the isMovingToClosing state, and the CloseFilling followed by the MoveBelt actions are performed.
4. isMovingToClosing, e_UnderClosing -> isClosing, StopBelt, DoClose: If the conveyor belt is in the isMovingToClosing state, and the bottle pass under the closing station (e_UnderClosing), then the conveyor belt transits to the isClosing state, and the StopBelt followed by the DoClose actions are performed.

5. isClosing, e_Closed -> isMovingOut, MoveBelt: If the conveyor belt is in the isClosing state, and the bottle is closed (e_Closed), then the conveyor belt transits to the isMovingOut state, and the MoveBelt action is performed.

6. isMovingOut, e_EmptyBelt -> isEmptyBelt, StopBelt: If the conveyor belt is in the isMovingOut state, and the bottle was taken out (e_EmptyBelt), then the conveyor belt transits to the isEmptyBelt state, and the StopBelt action is performed.

**Initial State**

isEmptyBelt

We can see at Fig. 3 the conveyor belt FSM.

The rounded rectangles are the states. The isEmptyBelt is the initial state.

The label on a transition has two parts separated by a slash. The first part is the name of the event that triggers the transition. The second part is the name of the action to be performed once the transition has been triggered.
4.2 Petri’s Net

"Petri nets are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic" [66].

Carl Adam Petri working as a scientist at the University of Bonn introduced Petri nets principles in 1962. Later in the 1970's Petri nets were introduced in the United States and other European countries.

4.2.1 Petri’s basics

Petri introduced just one rule, which seems to be very simple but its implication is deep and complex: the rule for transition enabling and firing.

A Petri net is a kind of directed graph with an initial marking (state) named $M_0$. The graph is compound of three main elements:

$s$ Places: holds the transition's input and output.
$s$ Transitions: holds the condition to enable passing Place's inputs to the respective Places (outputs).
$s$ Arcs: connects Places to Transitions and vice versa.

In summary we can say that the directed graph is composed of two kinds of nodes, places and transitions, connected by arcs.

Places are marked to hold a nonnegative number, defining the system state as a set of places marked with different values. Formally we say that the place $p$ is marked with $k$ tokens when a marking assigns to it the nonnegative number $k$.

Arcs have weights (which are positive numbers) that defines the condition to enable the passing/flow; i.e. how much input tokens are requested to enable the passing through this arc.

Graphically places are drawn as circles while transitions as rectangles. Arcs connect places to transitions (and transitions to places) by directed arrows, and are labeled with their weight.

Place's marking is graphically represented by placing black dots in the place.

Formally let us say that it exist a vector $M$ with $m$ entries, each one representing a specific place, where $M(p)$ is the numbers of tokens in place $p$. The vector $M$ represents the actual system state.

4.2.2 Formal definition

A Petri net is formally defined as $PN = (P, F, T, W, M_0)$, where:

$s$ $P$ is a finite set of places
$s$ $T$ is a finite set of transitions
$s$ $F$ is a set of arcs
§ W is the arc's weight function
§ M₀ is the initial marking

A Petri structure \( N = (P, T, F, W) \) represents a net without an initial marking, while \( (N, M₀) \) represent a net with it.

### 4.2.3 Dynamic behavior

#### Firing rules

The rules of firing (transitions) were defined by Murata [66] as follows:

1. "A transition \( t \) is said to be enabled if each input place \( p \) of \( t \) is marked with at least \( w(p, t) \) tokens, where \( w(p, t) \) is the weight of the arc from \( p \) to \( t \)."
2. An enabled transition may or may not fire (depending on whether or not the event actually takes place).
3. A firing of an enabled transition \( t \) removes \( w(p, t) \) tokens from each input place \( p \) of \( t \), and adds \( w(t, p) \) tokens to each output place \( p \) of \( t \), where \( w(t, p) \) is the weight of the arc from \( t \) to \( p \)."

In Fig. 4 there is an illustration of a transition firing rule: (a) illustrates the marking before firing and (b) after the transition rule was applied.

![Figure 4: Petri's transition (firing) rule: (a) before, (b) after [66]](image)

#### Non-determinism

A place may have more than one outgoing arcs. This pattern is referred to as a conflict, decision or choice. This structure exhibits non-determinism. Fig.5 shows such a pattern.
**Concurrency**

Concurrency is represented by two or more arcs out coming from a common transition, creating two parallel branches. The concurrency is cancelled when the arcs of each branch arrives the same transition (see Fig. 6).

**Loops**

Loops are represented in Petri nets by connecting arcs circularly between a set of two places and two arcs. At one of the transition must appear an exit condition via an additional arc, otherwise it becomes an infinite loop (see Fig. 7).

### 4.2.4 Example

*Finite State-Machine (FSM)*
Petri nets can represent equivalent state machine-diagrams.

The methodology is simple:

§ States are replaced by places.

§ Transition's triggers are replaced by labeled transitions with input conditions.

§ The number of incoming and outgoing arcs from a transition is limited to one, which represents the state's transition.

In Fig. 8 we see a Petri net representation of a vending machine FSM. This vendor machine accepts two kinds of coins (5 and 10 cents), and sells two kinds of products with prices of 15 and 20 cents respectively.

![Figure 8: A Petri net state-machine representation](#)
4.3 Harel’s Statechart

Statechart is a visual formalism for complex systems. It is an extension of the conventional formalism of FSM, suitable for the design of complex reactive systems such as a real-time application.

Three extensions were introduced:

- Hierarchy
- Concurrency
- Communication

This extension adds to the conventional FSM highly structural features, it reduces the design's volume and improves the expressiveness.

Transitions are defined by an event, a guard condition and the action to be performed.

This methodology is more behavioral than structural, and it doesn't make any distinction between static and dynamic views.

4.3.1 Hierarchy

"To be useful, a state/event approach must be modular, hierarchical and well-structured" [65]. When implementing hierarchy the number of transitions can be reduced by grouping states with common reactions in a super-state, in place of defining the same reaction for each state of this group.

Hierarchy enables also decomposition of a complex behavior in a well-defined tree structured model. Transitions are separated for each sub-FSM node/leaf in this tree model, but inter-level transitions are also permitted. Inter-level transition is a powerful tool but very hard to implement in a formal and systematic form, as it will be discussed later (see section 9.4.3.3).

High-level state's transitions influence low-level states, because their activeness is totally dependent on their super-state's activation, and therefore any high-level state's left deactivates inner levels states.

By using hierarchy, states are decomposed only in a XOR form, e.g. the system will be only in an unique state.

Graphical hierarchy representation can be seen in Fig. 9.
4.3.2 Concurrency

The modular approach permits separation of modules which will concurrent react; or in Harel's terms: enables "orthogonally" execution.

Statechart introduces discernment in state decomposition: AND and OR states. OR or XOR form of decomposition was explained in the hierarchy section; concurrency adds the AND form of decomposition.

AND decomposition means that "being in a state the system must be all of its AND components" [65].

The super-state who holds the concurrent FSMs is split into different sections by dashed lines. Each section allocates a different FSM which reacts in parallel. Concurrency avoids the possible explosion of number of states when combining different states, by enabling two or more current states, creating dynamic current states combinations. The super-state is then defined as the "orthogonal product" of its internal subsections (A and D in Fig. 10)

4.3.3 History pseudo-state
An interesting innovation is the history pseudo-state which remembers the last current state of the reactivated FSM.

When the super-state is left then its sub-FSMs are left also, when returning the programmer can choose to reach the default (initial) state or the previous current-state.

In Fig. 11(a) when arriving the super-state the default state is reached unless the transition labeled $a$ is activated, then the history pseudo-state helps to return to the previous left state. In Fig. 11(b) any transition to the super-state will return to the history pseudo-state.

![Figure 11: Statechart’s history pseudo-state](image)

Statechart has two different interpretations of the history pseudo state:

- Deep history
- Shallow history

Deep history cascades the history returning to all FSM’s descendant. In the shallow version the history return is only implemented at the specified level, and not more.

### 4.3.4 Communication

The communication between concurrent FSM’s is implemented by broadcasting messages to all parallel modules, no direct communication is implemented.

### 4.3.5 Formal representation

Formally the graphical Statechart representation (without the history pseudo-state) can be described as [69]:

\[ M_{StateChart} = \{ E, S, A, L, T, V, C \} \]

where:
- $E$ is a set of events
- $S$ is a set of states
- $A$ is a set of actions
- $V$ is a set of variables
- $C$ is a set of conditions
- $L$ is a set of labels; $L = E \times A$
- $T$ is a set of transitions:
  - $T = \{ S_{\text{Source}}, I, S_{\text{Target}} \}$, $I \in L$,
  - $T \subseteq 2^S \times L \times 2^S$, $S_{\text{Source}} \subseteq S$, and $S_{\text{Target}} \subseteq S$

At a specific time $t$, the Statechart current configuration is described as:

$$SC_t = \{X_{\text{StateChart}}, \Pi_{\text{StateChart}}, \Theta_{\text{StateChart}}, \xi_{\text{StateChart}}\}$$

where:

$X_{\text{StateChart}}$ is the maximal state configuration at the last transition at step $t-1$.

$\Pi_{\text{StateChart}}$ is a set of external events that arrived during the time step $t-1$.

$\Theta_{\text{StateChart}}$ is a set of conditions true at step $t$.

$\xi_{\text{StateChart}}$ is a function returning the value of a variable at step $t$; $\xi(\text{variable}) = \text{value}$.

4.3.6 Example

In Fig. 12 we can see a design of a multi-alarm watch.

Main and dead states represent on/off modes, where the alarms services appears in the main state.

Each section in main represent a concurrent task, handling with two alarms, chime announcing the new incoming hour, light, power support, and the display handling.

Battery's insertion/removing, turns on/of the system.
Figure 12: Multi-alarm watch example [65]
4.4 Selic’s ROOM

The Real Time Object-Oriented Modeling (ROOM) methodology was developed for distributed systems. Its main aim is to describe high-level parallel behavior creating a variation of Statechart visual formalism, in an Object-Oriented domain. This methodology proposed also a form of implementation. The model includes structural and dynamic (behavior) design.

4.4.1 Structure

The Actor

The principal concept in the structural design is the actor. "An actor represents an active concurrent entity with a specific responsibility" [28].

The concept of concurrency at ROOM "means that an actor can exist and operate in parallel with the actors in the same environment" [28].

The Actor encapsulates its implementation from other outer elements, such as other actors or environment.

The structure is hierarchical thus complex actors "may be broken down further into component actors, each responsible for a subset of the overall functionality" [28].

Ports

Actors communicate with their environment through ports. Ports are means for in/out information flow. This information is packaged into messages.

In order to hold a communication, a protocol has to be associated to the port; which consist of a set of valid message types, and a set of valid message exchange sequence on that port.

Binding (see Fig. 13) "is an abstraction of an underlying communication channel which carries messages from one actor to another" [28]. Bindings may link ports which hold a common protocol.

This communication model dictates a pure message-based communication model, no other possibilities of data interchange, like shared variables, exist.

![Figure 13: Actors binding and ports](28)
Implementation

Actors are translated to classes that hold other sub-actors (also traduced to classes). Ports are also implemented by classes adopting in such way the OO paradigm.

4.4.2 Behavior

"Actors are structural entities which provides the logical containers for behavior" [28].

The behavior is an actor attribute likewise a port is.

"The linkage between behavior and structure is achieved through ports" [28], which means that ports are accessible by the Finite State Machine, which describes the behavior. Note that only end-ports (with no internal connection) are accessed by the behavior. If a port is connected to a sub-actor, then the handling of this port is propagated to this specific sub-actor.

Communication

Events are defined as incoming messages at some port that are compound of four fields:

1. Port: the port from through the message entered.
3. Priority: the dispatched priority.
4. Data: any optional information attached to this event.

When the behavior of an actor sends a message it can block until the receiver replies, applying in this way a synchronous communication; otherwise the communication is defined as asynchronous. Replied messages have high priority and are immediately queued to the sender-actor message queue, skipping the current messages in the queue.

Execution Model

ROOM adopts "the run-to-completion programming model for the behavior of actors" [28].

An actor is normally continuously in a receiving mode, waiting for incoming events.

When an event arrives it is processed. After the processing the actor returns to its receiving waiting mode.

Any incoming event which arrived during the processing is queued, and processed immediately after the previous event processing.
Priorities are attached to events and not to actors, breaking the FIFO rudimentary order; enabling a dynamic real-time event-driven processing.

**Design Implementation**

The *run-to-completion* execution model is implemented by a Finite State Machine (FSM) (see Fig. 14).

The receiving mode of an event is mapped into states, while the trigger is an event that is processed by an action.

More detailed, the FSM consist in the following elements:

- A set of end ports
- Service Access Points (same as ports, but differentiated in their structural issue).
- Internal functions
- A set of states
- Local state's variables
- A set of transitions compound of:
  - Trigger in a format of <port, signal, guard>
  - Code segment for event handling
- An initial point

![Figure 14: ROOM's Finite State Machine](image)

**States**

In ROOM states are also encapsulations in addition to the actor's one. This encapsulation consists of the following elements:

- Local variables
§ An entry action
§ An exit action
§ A set of sub-states (enabling in such way a hierarchical structure of states)
§ A set of transitions

The Statechart's history pseudo-state is also supported only in its deep version. ROOM doesn't supports concurrent states like in Statechart. The concurrency was taken out to be implemented by two different actors "running" their behavior in parallel (see Fig. 15). This approach exhibits the interconnection between concurrent executions.

![Figure 15: ROOM's concurrency notation (comparing to Statechart) [28]](image)

**Inheritance**

Inheritance is applied in ROOM by drawing addition transitions to the parent's actor, or even by adding new "leaf" states to it. This form of inheritance preserves the parent behavior, extending the FSM to handle with specialized cases.

**4.4.3 Example**

In this section an example of the ROOM's structure methodology is brought, modeling a global glance of a rudimentary computer.

The example (see Fig. 16) designs a CPU board connected to an IDE card for ports handling (serial and parallel).

Note that all the ports are handled by the IDE card, while the CPU just connects to the IDE card for already processed data.
Figure 16: Rudimentary computer structure [67]
4.5 UML

UML stands for Unified Modeling Language. UML is an Object-Oriented system of notation. This method of notation evolved from the work of Grady Booch, James Rumbaugh, Ivar Jacobson, and the Rational Software Corporation. These renowned computer scientists integrate their respective technologies into a single, standardized model.

Today, UML is accepted by the Object Management Group as the standard for modeling object oriented programs.

4.5.1 Purpose

The following are the primary goals of UML as brought in the UML tutorial of the Kennesaw State University [63].

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.

4.5.2 Diagrams

UML provides a set of diagrams that shows different system views. The following is a partial list of the UML proposed diagrams.

- Use Case Diagram: models the functionality of the system. Enumerates the services provided by the system.
- Class Diagram: models the program structure. Shows the participant classes, including their attributes and method. Also CD provides means for class interconnections such inheritance, association and aggregation.
- Sequence Diagram: Shows the dynamic objects interconnections. Demonstrates a sequence of method calling and message passing between objects.
- Statechart: Shows the behavior of a class by defining different states on reaction. According to these states the relevant reactions are activated for the suitable incoming event or conditions.
Class diagrams may define the static view of the program, while Statecharts and Sequence diagrams defines the dynamic view.

**Class Diagram**

As mentioned the class diagram includes the attributes and method full description, it may also tell if an attribute is static or not (see Fig. 17). Even can specify if a method if abstract or not. Visibility is also handled in class diagrams using different signs (+, - and #) to define the type of access permission (public, private or protected).

![Class Diagram contents](image)

Interconnections are also handled in CD. The various types of Object-Oriented classes' interconnection are graphically implemented (see Fig. 18). Association, which is a weak link between classes, is represented by a single arrow.

Aggregation, which represents a *part-of* association, is represented by an arrow with a diamond at the end.

Composition, which represents an aggregation with time life dependencies, is represented as an aggregation with a filled diamond.

Generalization, which represents inheritance, is represented by a triangle pointing to the parent class.
Sequence Diagrams

The sequence diagram expresses interaction between objects.

Figure 18: Classes interconnections

Figure 19: Sequence Diagram

The objects are positioned in the top line, enlarging from each one a dashed line called the timeline (see Fig. 19). The timeline describes the interaction as a function of the time advancement.
Asynchronous messages are represented by simple arrows, while synchronous messages by filled ones.

UML provides wide offers of loops and conditions to include in SD.

*Statechart*

See section 4.3.
4.6 Components

Various definitions of the term *component* have been proposed. The degree of rigor in each definition is indicative of the prominence of the concept *component* in the lexicon of the authoring body.

4.6.1 The UML definition

The UML component definition is clearly at the low end of the spectrum. "A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics)" [31: section 4].

UML 2.0 addresses the composition and behavioral aspects of components in very generic terms [31: section 8.1] "The Packaging-Components package focuses on defining a component as a coherent group of elements as part of the development process. It extends the concept of a basic component to formalize the aspects of a component as a ‘building block’ that may own and import a (potentially large) set of model elements". Similarly, UML 2.0 speaks of component behavior in the most general terms. Behaviors may be attached to interfaces, connectors, or the component itself. What is missing in UML 2.0 is an explicit link between architecture and behavior. It is all well and good to view behavior at the various levels as a series of contracts between collaborating elements. But there is no requirement, or guidance in UML 2.0 regarding compliance between architecture and behavior goals, perhaps because UML is a notation standard and not a development methodology.

In keeping with this bare-bones definition, UML provides a notation to show the organizations and dependencies of components (Component Diagram) and for the deployment of components onto a physical architecture (Deployment Diagram). Deployment diagrams relate to components only as black boxes, deployed to physical nodes. Component diagrams are, in effect, class diagrams, representing components as classifier boxes, marked with a textual or iconic "component" stereotype. Since component diagrams use the notation of class diagrams, component structure is indicated by the elements used in class diagrams: association, dependency, port and interface. Internal component structure can be represented in one of two ways [31: section 8.3.1]

1. Realization dependencies between a complex component and the classifiers that realize it [31: figure 85].
2. Nesting of classifiers within the complex component [31: figure 86].

4.6.2 Other component definitions

The Interface "Façade" definition
A definition of a component is to see it as "an object + an interface". Object-Oriented practitioners have used the "façade design pattern" [70], to distinguish between the external interface and the underlying implementation. Component users interact with the public operations of a façade class, which delegates the implementation of these operations to hidden constituent classes of the component. This pattern facilitates implementation of component-oriented applications in classical Object-Oriented languages.

The ITEA's Interface definition
The group "Information Technology for European Advancement" [29] has given a much more elaborate definition of components. In addition to the "syntactic interface level", it defines a "semantic interface level" as well as a "synchronization interface specification level." These latter two characteristics clearly encompass behavioral aspects. However, the ITEA's proposed use of UML to represent components is restricted to structural aspects. Accordingly, when using the UML Class notation to represent component blueprints, ITEA specifies two specific list compartments: provided interfaces and required interfaces. Similarly, for component instances, ITEA uses the UML component notation, in conjunction with the UML interface notation, modified to distinguish between provided and required interfaces.

The Behavioral-contract definition
In keeping with the capabilities and emphasis of Eiffel, Bertrand Meyers [30] stresses the behavioral aspects of component definition as a contract definition. Such concepts as pre and post conditions are directly represented in Eiffel, and Meyers demands no less of a rigorous definition of components behavior.

4.6.3 Component relationships
In a component oriented design, components have to cooperate. The means of cooperation are components relations.

UML Relationships [26]
In the UML notation method many types of relationships were defined between classes.
In the UML Class Diagram, Classes are linked by Associations that are defined as "a relationship that may occur between instances of classifiers".

Aggregation is "a special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part" [26]. There is more fine-tuned Association, the composition. The Composition is defined as "a form of aggregation which requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts. Composition may be recursive" [26].

**Six Different Kinds of Composition** [32]

James Odell defined six different kinds of composition based on a combination of three basic properties. This classification of composition enables him to decide if an operation can be propagated from the component "ancestor" to its "childs". Odell came to a conclusion that if the composition form remains the same, "transivity exist and propagation of operations can be inferred", otherwise is not guaranteed.

In this work we defined the composition term as:

components parts that its time life scheduling is controlled by its container. Furthermore its container is defined as its environment and it may control the subcomponents behavior. Later in the "our proposal" section our composition term is discussed.

Odell's definitions are brought here with little differences in order to translate his ideas in term of components.

The properties are:

1. **Configuration**: the components bear a particular functional or structural relationship.
2. **Homeomerous**: the components are compound by the same type
3. **Invariance**: The relation between the components can be disconnected.

From these properties he derived six different kinds of composition:

1. **Component-integral**: a configuration of parts which bears a particular functional or structural relationship, as well to the component they constitute.
   
   For example wheels are part of a grocery cart.

2. **Material**: an invariant configuration of parts.
   
   For example bread is partly flour.
3. **Portion**: a composition where the components are compound by the same type.
   For example a slice of bread is a portion of a loaf of a bread.

4. **Place-area**: an invariant and homeomeric composition. In addition to the Portion composition the Place-area composition is invariant.
   For example a peak is part of a mountain.

5. **Member-bunch**: a collection of components, with no any of the three properties.
   For example a tree is part of the forest.

6. **Member-partnership**: an invariant a collection of components. It is an invariant form of the Member-brunch composition.
   For example Stan Laurel is a part of Laurel and Hardy.

**Non-compositional Relationships** [32]
Composition is confused with other kinds of relationships. In this section five non-compositional forms are enumerated.

1. **Topological inclusion**: a relationship between a container and the component contained. In this form of relationship the component contained is not a part of the container it is only contained by it.
   For example the customer is in the store.
   Commonly confused with the place-area composition.

2. **Classification inclusion**: a collection of components with common concepts.
   For example *Gone with the Wind* is a part of the set of objects with book concepts, but it is not a part of the book.
   Commonly confused with Member-brunch composition, however the Member-brunch has spatial, temporal or social connection (tree is part of the forest).

3. **Attribution**: Components properties like the height of a Lighthouse are not a part of the Lighthouse, but they are a part of the properties of the Lighthouse.

4. **Attachment**: Component attachment does not guarantee composition.
   For example Earring are attached to the Ear but they are not a part of the ear.

5. **Ownership**: It is often confused with composition. For example a Bicycle has Wheels and they are a part of the Bicycle, but if John has a Bicycle is not say that the Bicycle is a part of John.
4.6.4 Arts’Codes Definitions

**Component**
A component in Arts’Codes is defined as a type which can be instantiated and also inherited.

This type is an abstraction of a responsibility delegated by the programmer, who characterized the component by defining inputs, outputs and gates as the interface, and its behavior.
A component is a concurrent entity.

**Composition**
In Arts’Codes the components are defined hierarchically; while the composition relation is defined between them statically and dynamically.

Statically, the component's environment is defined by its host, having no any connection to the real world. The component is a part of its host like a fetus in his mother.

Dynamically the component behavior is a sub-part of its host's behavior. Moreover the component behavior is scheduled by its host.

In summary we say that composition can be described as a "part-of" association, where one component is defined to be a sub-component of a second one, named the container so that:

- Its time life and scheduling is controlled by its container.
- Its environment is provided by its container.
- Its behavior is linked to (and controlled by) its container's behavior.

This innovated composition definition merges the sub-component into its container, without breaking its abstraction, but the whole sub-component is totally dependent on its container, and is a part of its structure and behavior.
4.7 Lavi's Context Diagram

ECSAM is an analysis method for embedded and computer-based systems, developed by J. Lavi and J. Kudish [27]. It models the system and specifies its requirements.

In this work the author partially based his component approach specification, on the Context Diagram of the ECSAM method.

Context Diagram deals with conceptual architecture. It helps the top-level specification.

Its main objectives are:

§ Identification of the system's scope.
§ Determination of the system boundaries.
§ Attainment of initial common understandings among customers and developers.

Graphically (see Fig. 20) the system is drawn as a solid-line rectangle, and the environmental devices are drawn as dashed-line rectangles. Flows of information are represented by labeled arrows, which connect inputs from external devices on the left side, and outputs to external devices on the right side.

In our design work we transformed the environmental devices to virtual ones, being a part of the code, and not just a specification.

As we will see later, the separation between the application and the environment is one of the 11 principles of Arts'Codes, inspired on the Lavi's Context Diagram.

Context diagrams involve more internal data specification not used in our work.

Figure 20: ECSAM’s Context Diagram
4.8 Analysis and discussion

4.8.1 Advantages and disadvantages of the various approaches

In the previous section some of state-machine based methods were presented, for reactive behavior description; some of them having also structure description support.

Real-time applications are obviously reactive due to their nature; therefore a design methodology built on a state-machine variation seems to be suitable. Of course not the Mealy FSM version! This is not compatible with complex applications, but some variance of parallel automata or Statechart with concurrency and hierarchy.

Researches around the Diagrammatic Reasoning field yield to the fact that the programmer helps himself, in the first stages of design, with a data structure image which appeared in his brain. Based on this imaginary data structure, operations are designed modeling the data structure behavior.

Modern methodologies such as UML comprehend that at least two views must be designed: the static view for structure description, and the dynamic view for the behavior molding. In Statechart's and Petri's diagrams for example, static design is missed.

UML approach as said uses the static and dynamic separation, but this separation was implemented in a high degree of notation splitting the design in multiple different views (at least nine kinds of diagrams), and in consequence much cognitive efforts are used to track and correlate between the static and dynamic diagrams.

Abstraction like ROOM seems to help also for the reuse and concurrency, exhibiting in such a way concurrent entities' interconnections. What lacks in ROOM is its composition definition, i.e. a sub-actor is too much independent. The sub-actor connection to its host is just by receiving the responsibility of a port, it is not a part of the host behavior; it is almost a concurrent and independent entity.

Well-defined definitions and separation between entities is missed almost in all methodologies, e.g. separation between:

§ Application and environment appears only in ECSAM.
§ Different concurrent entities appear only in ROOM and UML.
§ Normal and excepted behaviors is missing in all methods.

Arts'Codes was built as a synthesis of these suitable features collected from the reviewed methodologies, in addition to author's innovation, in order to suit a natural cognitive method. This natural/cognitive approach may alleviate cognitive
efforts, simplify the diagram, and therefore will facilitate the design and readability.

4.8.2 Adopted features

The following existing features were adopted in Arts'Codes approach:

- Support of static and dynamic design.
- Component encapsulation and its hierarchical structure.
- A behavior is attached to each component.
- This approach exhibits the interconnection between concurrent executions of the components. The behavior is described as a hierarchical automaton (eliminating Statechart state's concurrency), but without state encapsulation.
- Supporting of deep history.
- Inputs and outputs responsibilities' propagation to sub-components.
- Inheritance.
- Well-defined separation between environment and application.

Arts'Codes can be seen as an evolution in state-machines, its near ancestors were ROOM and Statechart.
Proposal for the Design

In the following sections the design method is presented, based on the literature brought in the previous chapter. For the reader’s comfort the method description is split and presented both visually and formally.

5.1 Arts'Codes Visual Programming Language (VPL)

In the following sections the innovated Arts'Codes design method is presented. A general presentation is brought, followed by an enumeration of the Arts'Codes principles. In continuation the graphic elements are presented, followed by the development process description, an example and finishing with a short discussion.

5.1.1 General presentation

The preliminary Arts'Codes design method was presented by the author for the first time in an internal seminar at JCT [24] in 2002. Later a full version was presented at the IEEE SwSTE ’03 conference in Herzlia, Israel [18] in 2003.

Arts'Codes is the acronym of:

§ Parallel-Automata Real-Time Systems &
§ Component Oriented Design Methodology.

This method includes the real-time design method – "Codes", and a real-time execution platform – "Arts".

In the following section the "Codes" part is presented, which offers a design method for real-time applications.

As mentioned in the Diagrammatic Reasoning section (see section 2.1), we adopted from the programmer's anecdotes 3 principles in real-time applications design:

1. The component-oriented approach.
2. The static and dynamic views of the design "a structure of information, and how it works".
3. The diagrammatic interleaving of these static and dynamic views.
The design approach was also inspired from printed (digital electronic) circuits. The circuit diagram defines the static view, with its sockets and their interconnections. On each socket, a component-type selected from a component catalog, is inserted, creating in this way an instantiation of the component type. The pins are the components’ input/outputs. Furthermore, each electronic component has its Data Sheet describing its internal dynamic behavior.

This approach enables a component hierarchy structure, defining sub-components in its ancestors’ way (compound by the static & dynamic view). The Arts’Codes method implements the hierarchy in a homothetic way, this means that: at each stage or level, the design is defined in the same homogenous way.

5.1.2 Principles
The Arts' design was built according to the following eleven principles

1. Separation between Application and Environment
2. Component Abstraction
3. Hierarchy and Homothetic Design
4. Interleaving of Structural and Behavioral Models
5. Separation of Normal and Exception behaviors
6. Clear scheduling definition
7. Behaviors links through Gates
8. Components Concurrency
9. Synchronous execution
10. Determinism
11. Reuse

1st principle: Separation between Application and Environment
There is a well-defined separation between the reactive rules (application) and the environment. All the system is encapsulated in the root-component, which is the only one that has connection to the environment through the Virtual Devices (see forward Synchronous execution).
This approach helps the identification of the system's scope, and the determination of its boundaries, as defined in the Lavi's context diagram [27].

2nd principle: Component Abstraction
This model specifies that the design is made through encapsulated "black boxes" called Components. This approach enables module abstraction, which facilitates the design process, and the reusability. When the designer wants to insert a sub-
component in a component, he must first use a socket, which defines the number and types of links with the local/external data (like connection pins). Then, the component-type is inserted graphically onto the socket fitting it to these data links.

3rd principle: Hierarchy and Homothetic Design
This approach enables a component hierarchy structure, defining sub-components in its ancestors' way (compound by the static & dynamic view).
The Arts'Codes method implements an homothetic approach, which means that: at each stage or level, the design is defined in the same way. At each level in the hierarchy component structure the components are define in the same way. The same notations are used to define the structure which includes the I/O arrows, shared variables, sub-components and interconnections arrows. This homothetic approach exists also in the dynamic behavior defined at each level by its Manager and Guard using a hierarchic automaton including exception-Assertions and control linking-Gates.

4th principle: Interleaving of Structural and Behavioral Models
Arts'Codes enables the interleaving of the static architectural structure (component structural model) and the dynamic behavior (behavioral model), and facilitates human cognition by providing clear correspondence between different graphical views.

Structural Model
It describes graphically the static architecture of the components (see later Fig. 22 in section 5.1.4).
It allows drawing the:
§ Inputs/Outputs (to/from component)
§ Sub-Components (components hierarchy)
§ Manager (contains the normal behavior)
§ Guard (exception assertions and security rules)
§ Shared variables
§ Data interconnections (data links between input, outputs, sub-components and manager)
The system’s structure does not change often, and hence, it is relatively stable.

Behavioral Model
It describes graphically (see later Fig. 23) the component's dynamic behavior. It defines the system reactions for internal and external inputs, by transmission of internal and external outputs. It allows drawing inside:

- Hierarchic Automata (designed as a Component Oriented Statechart (CoST))
- Sub-Components Activation in the states (see later)
- Local Variables
- Gates (to allow control-links with other components managers, see later)

The system’s behavior which is described in the Manager, may have to be changed (or adapted) many times.

5th principle: Separation of Normal and Exception behaviors

The behavior design is split into two main parts: Manager and Guard co-manager. The Manager is responsible of the normal component behavior and the Guard tries to assure this normal behavior in case of exception. Assertions are means for behavior verification. They can verify correct behavior or detect critical status. Guards are responsible of the exception handling.

The Manager role
The Manager has three principal tasks:
1. Reacting to inputs
2. Sub-component scheduling
3. Assertions activation

Assertions and the Guard co-manager
The Guard is responsible of the exception handling. Exceptions are detected by the various Assertions which are inserted and tested in the various Manager’s states.

The Guard is designed in the same manner that the Manager using the same notation of CoST (Component Oriented Statechart).

The Assertions are defined by a trigger (when to check) and a condition to be checked.

About the responsibilities distribution between the Manager and The Guard, and consequently their interleaving mode, many patterns of Manager and Guard interleaving were proposed and compared (see section 5.1.3). The "Guard xor Manager" model was selected by the following definition:

When an assertion raises an exception, the Manager exits through the Exception gate and the Guard-co-manager is then entered through the corresponding Assertion gate. The Guard (after tests and repair actions) can come back to a
Manager's entry-gate (continuing the normal behavior), or exit through the Critical gate (for emergency stop).

6th principle: Clear scheduling definition
The sub-components are activated in the states of the hierarchic automaton of the Managers.
The principal innovation is that sub-components activation can be graphically inserted in a state of its host-component, extending in such a way the Statechart's notation (named CoST (Component Oriented Statechart)). The sub-component has a controlled life; it is non-dormant only when it appears in an active state of its host-component.
In this CoST extended Statechart, when a transition occurs, all the sub-components contained by the source father state are paused, and then all the sub-components contained by the destination state are activated.
A sub-component is influenced also by its father's history, i.e. the sub-components returns to its previous state together with the history host-component's state.
According to this approach (and the Behaviors links through Gates principle), all sub-components behaviors are in fact sub-behaviors of the composite father's behavior, encapsulated and organized in a hierarchical tree structure. Their activation is controlled by the father component.

7th principle: Behaviors links through Gates
In this CoST extended Statechart, gates enable the controlled entry & exit links between the manager's behaviors, and its sub-component's behavior. This innovation increases the sub-components' control by the host-component, without breaking the component abstraction principle.
It also enhanced the linking of all the hierarchies' behaviors in order to unify them in a large modularized definition.
The programmer may define two main kinds of gates:
- An entry-gate which is linked to an initializing state of the component’s manager. This kind of gate is used by the host-component in order to demand a specific entry-mode.
- An exit-gate which outcomes from the component's Manager, telling a specific exit-mode. This gate may be is used by the host-component Manager as a trigger of a transition.
Furthermore there are many other predefined gates in order to deal with the activation/deactivation of sub-components when entering/exiting a state:
§ Pause: when the host-component leaves the sub-component’s host-state, the sub-component’s Manager is exited via this exit-gate. Before leaving, the current-state is saved (history pseudo-state implementation).
§ Resume: is the default re-entering entry-gate after leaving via the Pause exit-gate.
§ Continue: a gate that avoids deactivation of subcomponents when passing from one state to another (not implemented yet).

In addition, the designer can also define new specific control gates for the needs of his application.

8th principle: Components Concurrency
The static structure of sub-components implies components concurrency.
Note that the sub-components concurrency is activated by the dynamic behavior, since it is a consequence of the dynamic view, e.g. two different sub-components which appear in a common state are acting in parallel. Concurrency is also implied even when inserting only one sub-component in a state, since it is acting in parallel to its host.

9th principle: Synchronous execution
The Arts’Codes approach implements the Synchrony Hypothesis [12] (see section 8.2 for a detailed presentation).
The strict Synchrony hypothesis demands that reactions are performed in zero time.
This approach is implemented defining a discrete time domain; i.e. the time advance according to steps, and during the steps the time is frozen. Time advancing is performed only between steps.
This time model not only influences the time, but the whole I/O environment. If the time doesn’t advance, the environment stays unchangeable.
By this definition, all reactions executed during the step are reliable, because they react in a stable I/O environment.
The strict Synchrony hypothesis is impossible to implement (because of the zero time reactions). Our approached implementation of this Synchrony hypothesis is as follows:
A controller defines the reactive-kernel-root which is the root component, together with its environment which is compound of a list of input/output virtual-devices; and their mutual interconnections (i.e. how the reactive-kernel-root connects to the virtual-devices).
The reactive-kernel-root (RKR) component is the program core; it receives inputs and sends outputs via the virtual-devices (VD). It is the root of the components hierarchy.

The VDs represent the software interface to the external physical-devices (PD). VD’s supply to the Reactive Kernel means for environment interface, by providing virtual inputs and outputs. These inputs/outputs are in fact an image of the current PDs.

An execution cycle (for each step) is processed as follows:

§ Collect all VD inputs
§ Scan automata structures and execute reactions
§ Emit out all outputs

When the reactions are executed all outputs are saved temporally in the VDs; later, between the steps they are emitted to the suitable PDs.

**10th principle: Determinism**

Arts’Codes execution is deterministic, i.e. it produces only one output for each input. The order, in which the transitions of the hierarchic FSM (Manager's implementation) are tested when an input arrives, may influence the output. In order to provide determinism, the transitions are tested in a fixed order.

Furthermore the order in which the components' behaviors are executed when an input arrives, may also influence the output; therefore, the components are tested in a fixed order.

Clear definition of priorities for components, transitions, assertions and sub-states enables determinism (detailed expiations can be found in section 11.2.4).

**11th principle: Reuse**

Reuse of components can be applied into two forms:

1. By inserting the same component type into two different sockets, then the same behavior will be applied two different sets of input/output.
2. Applying inheritance, e.g. defining a base-component with abstract reactions. Later this base-component can be inherited, implementing the abstract reactions in the derived components, in different ways.
5.1.3 Discussion on Arts' codes dynamic model

This section describes the various options and their properties followed by a discussion about the best model selection, concerning with the Manager and Guard interleaving.

Possible models

1. **Super Manager**: The Guard is encapsulated in a sub-component. The Manager is the super automaton and activates sub-components including the Guard. The Manager handles the exceptions. The Guard task is only to test and alert.

2. **Super Guard**: The Manager is encapsulated in a sub-component. The Guard is the super automaton and activates sub-components including the Manager. The Guard tasks are test, alert and exception handling.

3. **Guard inherits Manager**: The Guard inherits the manager and it adds alerts and exception handling.

4. **Guard parallel to Manager**: Both the Guard and the manager are encapsulated in sub-components separately. The Guard works in parallel to the Manager. The Guard tasks are: test, alert and handle the exception including by sub-component activation.

5. **Watch Dog**: Both the Guard and the manager are encapsulated in sub-components separately. The Guard works in parallel to the Manager. The Guard task is only test & alert. The Manager receives the alerts and handles with the exceptions.

6. **Guard xor Manager**: Both the Guard and the manager are encapsulated in sub-components separately. The Guard inserts assertions at the Manager automaton, which they make exceptions causing Manager's deactivation, and then Guard activation. The Guard handles with the exceptions, returning back to the Manager after completing with the exception handling. Never the Manager and the Guard works at the same time.

Models Properties

A set of properties (criterions) was defined in order to estimate the models and compare between them.

The following properties were defined:

- Split of Normal and Exception behavior.
- The Guard can explicitly test assertions at a specific state.
- The Normal behavior has component properties.
The Normal and the Exception behavior runs in an exclusive-or mode.
Simplicity.

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Split of Normal and Exception behavior</th>
<th>The Guard can explicitly test assertions at a specific state</th>
<th>The Normal behavior has component properties</th>
<th>The Normal and the Exception behavior runs in an exclusive-or mode</th>
<th>Simplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Manager</td>
<td>Bad</td>
<td>Excellent</td>
<td>Bad</td>
<td>Good</td>
<td>Bad</td>
</tr>
<tr>
<td>Super Guard</td>
<td>Very good</td>
<td>Bad</td>
<td>Excellent</td>
<td>Bad</td>
<td>Good</td>
</tr>
<tr>
<td>Guard inherits Manager</td>
<td>Very good</td>
<td>Good</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
</tr>
<tr>
<td>Guard parallel to Manager</td>
<td>Excellent</td>
<td>Bad</td>
<td>Excellent</td>
<td>Very bad</td>
<td>Very good</td>
</tr>
<tr>
<td>Watch Dog</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
<td>Bad</td>
</tr>
<tr>
<td>Guard xor Manager</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Excellent</td>
<td>Excellent</td>
</tr>
</tbody>
</table>

Table 1: Manager – Guard different models comparison

Models Comparison
Table 1 compares the different models by their properties.
The properties receive score at each model in order to obtain a criterion for model selection. The levels of scores are: Excellent, very good, good, bad and very bad.
The "Guard xor Manager" model was selected by the following definition:
When an assertion raises an exception, the Manager exits through the Exception gate and the Guard-co-manager is then entered through the corresponding Assertion gate. The Guard (after the exception handling) comes back to a Manager-entry-gate (continuing the normal behavior), or exit through the Critical gate (for emergency stop).
5.1.4 Graphical representation
This section introduces the graphical representation of the different design's elements.

Sockets
Sockets are means for virtual-devices and components instantiation.
Sockets have holes for pin insertion.
Pins are the component's inputs/outputs.
Interconnections are made between sockets via their holes.
The sockets and interconnections are usually constant, and virtual-devices and components can be changed (plugged/unplugged to/from their sockets), creating in such a way a form of reusing and easy maintenance.
The graphic elements can be seen in Fig. 21.

![Sockets Diagram]

Figure 21: Sockets, Holes and interconnections

Components
Components are composed of input/output pins, sub-components (including the correspondent sub-component's sockets), shared variables and Manager/Guard.
The component's graphic elements are shown in Fig. 22.
**Manager**

The manager includes the gates, local variables, state's hierarchy, transitions, sub-component activation and exceptions-assertions for the Guard.

We can see in Fig. 23 the Manager graphical elements. Note that transitions are continued sometimes by dotted lines, which represents entry-gate selection. Dotted lines coming from a sub-component exit-gate, represents a trigger transition (doesn't appear in the mentioned figure). Reaction can be executed before exiting the Manager via exit-gates, by just labeling the exit-gate arrow with the reaction name.

**Guard**

The Guard has similar properties as the Manager, with the difference that:

- Entry-gates are Assertion's entries.
- Exit-gates are Manager's entry-gates.

Fig. 24 demonstrates the Guard graphic representation. Note that the Guard can activate sub-components.

The Guard and Manager behave in an exclusive-or mode.
Figure 23: Manager graphical representation

Figure 24: The Guard graphical representation

**Assertions**

Assertions use textual and not graphic design. They are edited via dialogs boxes, allowing entering their specifications in several text-boxes (see Fig. 25).
5.1.5 Process of development

**Instantiations**

Before presenting the development process, it has to be known that the Arts'Codes method is supported by the Arts'Codes Component Editor (ACE) which will be explained later (section 14). ACE holds three kinds of catalogs for:

1. Components
2. Virtual devices
3. Physical devices

*Codes* design approach gives emphasis to the reuse principle by these catalogs. In these catalogs, different patterns of virtual-devices and component's types are listed in order to be used in various applications. The instantiation of these catalog's elements is made by dragging a type from the catalog into a specific socket in the application's diagram.

A virtual-device has to be also *inherited* by some *physical-device* in order to be implemented. This *physical-device* binds the *virtual-device* pattern to a specific hardware.

**The four phases**

Researches concerning to the Diagrammatic Reasoning field (see section 2.1) yield that the programmer helps itself, in the first stages of design, on a data structure image appeared in his brain. Based on this imaginary data structure operations are designed, modeling the data structure behavior.
According to this conclusion, a development process with four basic phases was defined, which are:

1. Environment wrapping
2. Hierarchic building of the components’ structure
3. Behavior molding
4. Exception handling

1st phase: Environment wrapping
The first design stage is to define the application’s environment, and how the environment connects to the application.

The environment description was inspired on the Lavi’s ECSAM Context-diagram [27].

In the ECSAM approach the application is defined as a black-box, drawing physical input devices at the left side, and the outputs ones at the right side of the black-box. Physical devices are connected to the black-box through arrows, defining in such a way the inputs and the outputs. This black-box is developed converting it to a white-box, exhibiting internal structures.

In Codes design, physical devices are replaced by virtual-devices (VD), which are interfaces between the physical ones and the application.

The VDs aim is to:

- Hold a constant environment image during the step, and
- Supply processed input/output data according to the application's needs.

Each VD has inputs and outputs which were assigned to a specific variable type, such as an event or an integer.

VD's inputs represent application's outputs connected to the VD, and VD's outputs are outcome data from the VD to the application. When saying application we refer to the root component called `reactive-kernel-root`.

The VDs themselves receives their environment's inputs and outputs through specific actions, such as ports in/outs, that are defined as abstract reactions until the VD instantiation (instantiation is made by inserting the VD in a specific socket).

When the VD is instantiated, the abstract actions must be implemented, by a specific physical device (PD).

PDs are implementations of VDs abstracts action, forming a form of inheritance. In fact PDs are translated (in the Arts execution platform) as subclasses of VDs.

The following checklist is to be performed in order to complete the first phase of the development process:

1. Create a socket for the root-component.
2. Create VDs sockets.
3. Define numbers of input and output holes for each VD socket.
4. Drag suitable (usually existing) VD templates from the catalog, into the VD's sockets.
5. Define numbers of input and output holes for this component socket (later these holes will receive the component's pins).
6. Connect VD's sockets holes to the component's socket ones.

2nd phase: Hierarchic building of the component structure

Now we start building the root-component (in a top-down approach) by:

- Enumerating its inputs and outputs, according to the already defined environment.
- Splitting responsibilities by creating internal sub-component sockets.
- Define numbers of input and output holes for each sub-component socket.
- Add internal shared variables.
- Add connections between the sub-component's socket holes and:
  - Internal shared variables
  - External input/output
- Add external input/output to the Manager (this component responsibility).
- Recursion, i.e. apply the same phases of development for the sub-component's built.

When the recursion process arrives the bottom's leafs, components are dragged to their suitable socket in the back-recursion-track.

Note that some sub-components will appear already in the components catalog, speeding-up the develop process.

Now the application architecture is ready, and we have the natural tools for starting thinking on which operations this structure has to complete (Diagrammatic reasoning evidences, see section 2.1).

3rd phase: Behavior Molding

This phase has to define the Manager, which is the normal behavior of each component. The bottom-up approach may be used with no compromise.

This phase involves:

- Entry and exit-gates definitions
- State's hierarchy definition
- Transitions between states (or gates, or state/gate combinations), each one defining
  - Source gate/state.
4th phase: Exception Handling
The exceptions aims are to assure real outputs, or to alert when it is not achievable.
This phase includes:

$\quad$ Assertions definitions, including:
$\quad$ o Trigger (when to test).
$\quad$ o The condition to be tested.
$\quad$ o The position (at the begin/end of the step).

$\quad$ The Guard, including:
$\quad$ o An entry for each Assertion.
$\quad$ o An automaton behavior connected to each entry.
$\quad$ o An exit-gate, which is in fact a Manager's entry-gate.

5.1.6 Example

**Statement Of Needs (SON)**
In order to complete the Arts'Codes design method presentation, the Home Heating System controller is designed here, according to the specifications brought by Rozenblit et al [69], as follows:

1. If Room Temperature $<$ (Desired Temperature ($H_d$) - 2), then Motor Command = ON after 1 second.
2. If Motor Speed $>$ Predefined Motor Speed ($S_d$), then turn on furnace (Oil Valve = OPEN and Ignite).
3. If Water Temperature $>$ Predefined Water Temperature ($T_w$), then Circulation = ON.
4. If the Fuel or Combustion Sensor detects errors (Combustion_Error, Fuel_Low), then turn off furnace (Oil Valve = CLOSE, wait 5 seconds, Circulation = OFF and Motor Command = OFF).
5. If Room Temperature $>$ ($H_d$ + 2), then turn off furnace (Oil Valve = CLOSE, wait 5 seconds, Circulation = OFF and Motor Command = OFF).

Additional Constraints are:
The minimum time between Motor Command = OFF and Motor Command = On is 300 seconds.

The Furnace cannot be on continuously for more than maxonTime seconds.

Process of development

1st phase: Environment wrapping
At this phase we separate the environment from the application fixing well-defined bounds. Therefore we design the Vds and reactive-kernel-root sockets and their interconnections (see Fig. 26). In this example we skip the trivial VDs design.

2nd phase: Hierarchic building of the component structure
In this phase the full static view has to be designed. For this, the reactive-kernel-root component is defined, including its inputs, outputs, shared variables, sub-components and their interconnections.
In this application only two levels of component's hierarchy are needed. Each sub-component receives a responsibility, and therefore the suitable inputs/outputs (see Fig. 27). The DMS shared variable, is the acronym of Desired Motor Speed, an auxiliary variable for event sending from the Motor sub-component to the *reactive-kernel-root*.

![Diagram of the reactive-kernel-root static-view](image)

**Figure 27: Reactive Kernel Root static-view**

3\(^{rd}\) phase: Behavior Molding

Now we have to mold the components' behavior, the *Manager*. This includes entry/exit gates, state hierarchy, transitions and sub-components' activation.

Fig. 28(a) presents the *reactive-kernel-root* Manager. As it can be seen, a three levels of state hierarchy are designed. The first/top level includes the *ReqTemp* and the *NeedHeating* states, the second level the SlowSpeed and the
DesiredSpeed states and the bottom level is formed by the WaterHeating and the Delay5 states.

![Diagram](image)

**Figure 28(a): Reactive Kernel Root dynamic-view**

When the condition It'sCold (RoomTemp < DesiredTemp-2) value is true the Manager enters the SlowSpeed (and therefore the NeedHeating) state and the Motor sub-component is activated.

Motor sends DMS event when the motor reaches the desired speed, then the Manager enters the WaterHeating (and therefore the DesiredSpeed) state activating the Circulator and Furnace sub-components.

When the room temperature is desirable the Manager leaves the WaterHeating (which will deactivate the Furnace) state and after 5 seconds returns to the initial state ReqTemp (which will deactivate both the Motor and Circulator).

Note that the sub-components have their own behavior which are reacting in parallel to their host when they are active (see Fig. 28(b), 28(c) and 28(d)). All they, turn off their physical devices when they are paused, before leaving through the predefined exit-gate Pause.

4th phase: Exception Handling
The last phase handles with the exceptions. *Assertions* have to be defined, and inserted in the *Manager*, at the suitable states. Then, the *Guard* can be defined by handling properly each *assertion* (see Fig. 29(a) and 29(b)).

![Motor dynamic-view](image)

*Figure 28(b): Motor dynamic-view*

As we can see in Fig. 28(d), two critical *assertions* were inserted: the *Maxtime* and the *Errors*. The *Maxtime* rises when the `maxonTime` times out (see the SON section); and the *Errors* assertion rises when the fuel or combustion sensors report an error.

The handling of these assertions can be seen in Fig. 29(a). When the *assertion* rises the manager is left through the *Exception gate* turning off the OilValve. For a Maxtime assertion rising e.g., the *Guard* is entered through the *Maxtime gate*, it waits in the *WaitSecureInterval* state, and returns to the *Manager* through the *Initial gate*. 
Figure 28(c): Circulator dynamic-view

Figure 28(d): Furnace dynamic-view
5.2 Arts'Codes VPL formal representation

In this section we present a formal definition of the Arts'Codes graphic design. This definition may also help as a basis model, in order to build the suitable model of compilation; and for proving the consistency of the components design.

5.2.1 Formal representation

A formal presentation of the Arts'Codes Visual Programming Language (VPL) is brought here, splitting the definition into: static and dynamic models.
**Static architecture model**

The Arts'Codes static model can be represented as a set of $N$ extended FSMs structured in a tree pattern. These extended FSMs are in the form of parallel automaton [50]. One of these FSMs is the root of the tree named $fsm_{root}$.

Formally we say that:

$$\mathbf{M}_{\text{Static}} = \{FSM_{\text{Arts'Codes}}, \text{SUB}_{\text{Arts'Codes}}, fem, N, I, O\},$$

where $FSM_{\text{Arts'Codes}}$ is a set of fsm of type FSM:

$$FSM_{\text{Arts'Codes}} = \{fsm_0..fsm_N\},$$

$N$ is the number of FSMs,

$I$ is a set of environment inputs,

$O$ is a set of environment outputs,

$\text{SUB} = \{\text{sub}(fsm_0)..<\text{sub}(fsm_N)\}$, such as:

for each $fsm_i$ exist a set $\text{SUB}(fsm_i)$ that includes all immediate FSM sub-level, and for each $\text{SUB}(fsm_i)$ and $\text{SUB}(fsm_j)$: $\text{SUB}(fsm_i) \cap \text{SUB}(fsm_j) = \{\varepsilon\}$ for $i,j:0..N$ and $i \neq j$.

$fsm_{root} \in FSM_{\text{Arts'Codes}}$ where $\text{SUB}(fsm_i) \cap \{fsm_{root}\} = \{\varepsilon\}, i:0..N$; and for each $fsm_i \in FSM_{\text{Arts'Codes}}$ exist only one $fsm_j$ where $fsm_i \in \text{SUB}(fsm_j)$ for $i,j:0..N$, $i \neq root$ and $i \neq j$.

**Dynamic model**

The Arts'Codes dynamic model for the Managers and the Guards is more complicated. It has to define the FSM which is in fact: a hierarchical parallel automaton, with a set of active states, activating/deactivating sub-FSMs in some state, controlling sub-FSM's initial modes via entry-gates, and triggering its transitions also via sub-FSM's exit-gates; all this with the addition of exception handling.

For the simplification we can see the Manager, the Guard and the exception-assertions as a unified (merged) hierarchical parallel automaton, where the assertions are conditions that trigger the Manager to transit to the Guard extension, and coming back after the handling.

The activation of the dynamic model, which models the application behavior, begins with the $fsm_{root}$ behavior, activating all other FSMs, limited to the restrictions supplied by the static architectural model.

Therefore the dynamic model will define the Arts'Codes Managers, described graphically in CoST diagrams.

The behavior execution begins activating the $fsm_{root}$ via its initial gate $G_{root}$.

$$\mathbf{M}_{\text{Dynamic}} = \{fsm_{root}, G_{root}\},$$

Finally we define the visual FSM with the following parameters:
**FSM(I, O, P, L, SUB, G, S, C, A, T),** where:

**I** is the set of input variables,

**O** is the set of output variables,

**P** is the set of shared/protected variables,

**L** is the set of local variables,

**SUB** is the set of sub-FSMs,

**G** is the set of entry/exit gates,

**S** is a set of states structured in a tree, where the FSM itself is the root state,

**C** is the set of conditions based on I, O, S, L and G (self entry/exit or sub-FSM exit);

**A** is a set of reactions based on I, O, S, L, and G (sub-FSM entry);

**T** is a set of transitions \( T = \{t_0..t_m\} \) where for each \( t_i \in T \), when at some current state \( S_C \), some condition \( C_k \) value is true then:

- the reaction \( A_t \) will be performed,
- the set \( SUB_l \) will be deactivated,
- the set \( SUB_n \) will be activated, and finally
- the state \( S_C \) (of an entry-gate) is left reaching the target state \( S_T \) (or an exit-gate).

Formally represented as \( t_i = (C_k, S_C) (A_t, SUB_l, SUB_n, S_T) \).

In fact there may be more than one current-state, one for each state-level, which provokes to define the \( C_k \) and \( S_T \) as a set of states. This topic is fully discussed and solved in our proposal of *improved Andre’s reactive-cell* (see section 9.4.3).

### 5.2.2 Dynamic model for execution

The proposed model for the visual representation of the Managers and Guards (seen in the previous section) must be close to an internal model ready for execution. For this, we proposed [72] the Abstract Parallel Automata (APA) model, which is the most generalized kind of parallel automata.

In the APA abstraction a transition is expressed as:

\[
\text{test}_i(VV) \rightarrow \text{assign}_i(VV),
\]

where:

- states, events, inputs, flags, timers, actions, outputs, etc; are represented as a vector \( VV \) of values to \( \text{test} \) or to \( \text{assign} \).

We can say that parallel reactions may have the product \( n \) form, such:

\[
n_1 / \text{test}_1 / \rightarrow n_k / \text{assign}_k / 
\]

where:

- \( \text{test}_i \) are boolean relation tests composed of:
  - event, input signal or flag arrived.
state or internal variable or time condition. 

Assign are setting of values to variables, such as:

- setting a state, a internal variable or a clock.
- execution of actions or functions.
- output of flags or sending of events.

A more formal definition of abstract parallel automata APA can be:

APA = (I, V, O, K, R), where:

- I = {x₁,...,xᵣ}  a finite set of input variables.
- V = {y₁,...,yₛ}  a finite set of internal variables.
- O = {z₁,...,zₜ}  a finite set of output variables.
- K = the finite range of values of each variable.
- R = a finite number of transition rules such:
  - test the values of variables → assign to variables

The execution of APA rules takes place in a succession of cycles:

- In each cycle, each automata rule is scanned once only.
- During each cycle, variables are tested and assigned.
- The new value of a variable I ∪ V ∪ O → {0,1,2,..., K} becomes available in the following cycle.

This definition of such an executable automata is compatible with the formal representation of the Arts'Codes dynamic: FSM(I, O, P, L, SUB, G, S, C, A, T):

- The variables I, O, P, L, S CoSt diagrams, can all be internally represented by (correspond to) the APA executable variables I, V, O.
- In the same way, the gates G and the activations of SUB FSMS can also be implemented for execution by flag variables in V.
- The CoSt diagrams conditions C and the actions A can be implemented as the functions test and assign correspondently.

The transitions of the Arts'Codes visual dynamic CoSt diagrams

\[ t_i = (C_k, S_C) \quad (A_r, SUB_l, -, SUB_n, +, S_T) \]

are compatible and therefore easily translated into executable automata APA.

5.2.3 Proving of consistency of the component design

In Arts'Codes we use Temporal Logic for consistency proving of the various components.

The three main problems in building an embedded system are:

1. Design methodology,
2. A-priori validation process (prove that the ultimate goal of the system is satisfied),

3. On-line correctness of the execution.

In this thesis, Arts'Codes gives an answer to the first and last problem (the design methodology and the correct execution). Here we want to answer also the second problem (the a-priori validation process) by introducing a formal description of the goals and the behavior of these components. Only from this formal specification a validation diagnosis can be deduced.

For embedded systems, the validation process problem is particularly important, but also particularly hard, since real time constraints, distribution and safety are crucial issues. In these cases, temporal logic has shown to be a powerful tool for specification and for validation [73,74]. However, temporal logic specifications are hard to make, they have been used for large systems (such as avionics, train design etc.) and could be used even more by the industrial engineers.

Arts'Codes, allowing the composition of the system in modular hierarchic components, facilitates the building of the whole system gradually; and can then also facilitate its validation.

In order to build a new component, an engineer can use a set of validated existing components (in a catalog); he can also add some new components and build a whole system with them.

Associated to the components, the engineer will introduce a set of temporal formulas which express the goal of the component in addition to its behavior (its Manager).

The a-priori validation will consist in showing that:

\( (B) \text{ and } (SCG) \implies (G) \), where:

\( B \) is the temporal logic formula expressing the behavior of the component.

\( G \) is the temporal logic formula expressing the desired properties (Goals) of this component.

\( SCG \) is the conjunction of all the Goals of its sub-components (already validated, stored in the catalog).

If we want such an a-priori proof to be carried out, goals and behaviors must be expressed in the same logical notation (here it will be in Propositional Temporal Logic (PTL) [73][74]).

The conditions can be based on internal boolean flags or external valued signals exchanged between components, or exchanged between some components and the outside world. Actions are started, and possibly stopped by the manager through the use of signals or flags; they can also induce the emission of other
signals or the setting of other flags. This assumption allows us to follow the synchronous model [12].
These properties can be written into formulas of Temporal Logic which can be accepted by an automatic prover such as the Stanford STEP prover[74].

**Definition of components’ goals**
Let us add to the Arts’ Codes component the definition of a goal, so that for validation purpose, each component X can be express by:

```
COMPONENT X
{
  ATTRIBUTES : ...shared and local variables
  SUBCOMPONENTS: ...list
  GOALS : ...PTL formula
  MANAGER : ...PTL formula
  ASSERTIONS/GUARDS : ...PTL formula
}
```

The **GOALS** formula describes the Aims of the component, meaning the properties that the component must fulfill at the end of its work, here specified in linear PTL (Propositional Temporal Logic notation):
"Conditions \lor signals \implies Property \land \neg signals", meaning:
(Conditions or signals true) imply that (Property and signals are true).

**Definition of the components’ behavior**
The component behavior is defined separately for the Manager and the Guard:
**MANAGER:** The component behavioral Manager describes the ordering of its operations and the activation of different subcomponents; it describes in fact the logical controller properties.
In Arts’codes they are described using a CoSt diagram translated in an executable automaton. They can also be translated in linear PTL (Propositional Temporal Logic notation):
"Conditions \lor signals \implies Actions \land \neg signals"

**GUARD:** This is the watch-dog of the Component which ensures the correct-working properties that the component must satisfy during all its execution, and the reactions it has to do in case of ill-functioning (exception).
This corresponds in Propositional Temporal Logic to:
(\neg Exception\_Condition \lor signal \implies repair\_Actions \lor \neg exit\_signal) \lor
Consistency of the components design

In order to check the a-priori consistence of the components design, let us check that the Managers/Guards behaviors are consistent with the goals. This can be done by computing the formula (brought above):

\[(B) \land (SCG) \Rightarrow (G),\]

where:

- **B** (the temporal logic formula expressing the behavior of the main Components' manager and guard merged together),
- and all **SCG** (the conjunction of all the goals of its sub-components)

imply that **G** (the temporal logic formula expressing the desired property (goals) of this component, is satisfied).

For this, we shall suppose that all the Goals of the SCG sub-components (taken from the catalog) have already been validated and that we can directly use their **Goals** (their behaviors having been already proved).

Example

Let us take the example of the soda factory example (seen in section 1.1.9), and apply to it this Temporal logic validation process, using the STEP linear PTL language [74].

Let us say that in Arts'Codes we have here 3 sub-components: **Filler**, **Sealer**, **Belt**; and a **main_system** component controlling them. The subcomponents' Managers (behavior) have already been validated, so we need to use only their goals for the proof of the **main_system** component controlling them. But for the **main_system** component, we need to define now both the goal and the behavior.

All component **goals** or **behaviors** will be described as macros in this PTL language.

The general **goal** of the whole machine is that when a bottle enters the belt, it has to exit filled and sealed, or in PTL notation:

\[ (e_{BottleEnter} \Rightarrow <> (P_{exit} \land P_{Full} \land P_{Sealed}) ) \]

meaning that:

- **if** **e_BottleEnter** occurs this must **imply** that **later** the properties **P_exit**, **P_Full** and **P_Sealed** will be **satisfied**

Here are all the definitions in the PTL language:

```plaintext
// signals
e_BottleEnter, e_UnderFilling, e_Full, e_UnderClosing,
e_Closed, e_EmptyBelt: bool Flexible
```
// flags
p_Full, p_Sealed, p_exit: bool const

// actions
a_MoveBelt, a_StopBelt, a_OpenFilling, a_CloseFilling, a_DoClose: bool Flexible

**macro Filler** _Goal_: bool where
Filler_Goal =
((a_OpenFilling \<=> a_CloseFilling ==»<> e_Full \<=>e_full) \(\&\) p_full)

// Meaning: Filler_Goal is true
if a_OpenFilling and later a_CloseFilling occur implying that later e_Full will occur
and so p_full will be permanently true

**macro Sealer** _Goal_: bool where
Sealer_Goal =
((a_DoClose ==»<> e_Closed \<=> e_Closed ==»[] p_Sealed))

// Meaning: Sealer_Goal is true
if a_DoClose occurs implying that later e_Closed will occur
and so p_Sealed will be permanently true

**macro Belt** _Goal_: bool where
Belt_Goal =
((e_BottleEnter ==»<> e_UnderFilling) \/
(e_Full ==»<> e_UnderClosing) \/
(e_Closed ==»<> e_EmptyBelt) \/
(e_EmptyBelt ==» p_exit))

// Meaning: Belt_Goal is true
if e_BottleEnter occurs implying that later e_UnderFilling will occur
and if e_Full occurs implying that later e_UnderClosing will occur
and if e_Closed occurs implying that later e_EmptyBelt will occur
so p_exit will be permanently true

**macro System** _Behavior_: bool where
System_Behavior =
((e_BottleEnter ==» a_MoveBelt) \/
(a_MoveBelt \& e_UnderFilling ==» a_StopBelt \& a_OpenFilling) \/
(a_OpenFilling \& e_Full ==» a_CloseFilling \& a_MoveBelt) \/
(a_MoveBelt \& e_UnderClosing -> a_StopBelt \& a_DoClose) \/
(a_DoClose \& e_Closed -> a_MoveBelt) \/

corresponding to the system Behavior automata (seen in the above section 1.1.9):

1. isEmptyBelt, e_BottleEnter \rightarrow WaitingToFilling, MoveBelt
2. WaitingToFilling, e_UnderFilling \rightarrow isFilling, StopBelt
3. isFilling, e_Full \rightarrow WaitingToClosing, MoveBelt
4. WaitingToClosing, e_UnderClosing \rightarrow isClosing, StopBelt
5. isClosing, e_Closed \rightarrow isMovingOut, MoveBelt
6. isMovingOut, e_EmptyBelt \rightarrow isEmptyBelt, StopBelt

**macro System\_Goal**: bool where 
System\_Goal = (e_BottleEnter ==> <> ( P_exit \land P_Full \land P_Sealed) )
// Meaning: System\_Goal is true
if e_BottleEnter occurs implying that later
the properties p_exit, p_Full and p_Sealed will be satisfied

The global Checking formula for the Stanford's STEP linear PTL validity checker [74] is:
Filler\_Goal \land Sealer\_Goal \land Belt\_Goal \land System\_Behavior ==> System\_Goal

In order to prove his system consistency, the engineer has to describe by himself the goals formulae of the components, according to the specifications of the system that he is building. But in the part IV, we shall study the way of generating automatically the behaviors PTL formulae from direct translations of the Arts'Codes Managers and Guards design.
6 Evaluation of the Design by users

6.1 Background
In order to evaluate the Arts’Codes design method a questionnaire was redacted and presented to a student group, after they finished their exercise using the Arts’Codes method. The questionnaire was analyzed, and according to the questionnaire results many improvements were proposed.

6.2 The Cognitive Dimensions (CD)
The questionnaire is based on A. Blackwell and T. Green, according to their paper "A Cognitive Dimensions Questionnaire Optimized for Users" [22]. Blackwell & Green present a method for diagrams evaluation based on the Cognitive Dimensions of notations. In this questionnaire they require to explain the Cognitive Dimensions very clearly in simple terms such that they can be understood easily. The following is a partial list of the Cognitive Dimensions. This list was copied as is from the original paper [19]:
Note that the same CDs were defined also in [23]. Both papers together give a full image description of the CDs.

Abstraction gradient: An abstraction is a grouping of elements to be treated as one entity, whether just for convenience or to change the conceptual structure. What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?

Closeness of mapping: What ‘programming games’ need to be learned? Programming requires a mapping between a problem world and a program world. The closer the programming world is to the problem world, the easier the problem-solving ought to be.

Consistency: When some of the language structure has been learned, how much of the rest can be inferred successfully?
**Diffuseness:** How many symbols or graphic entities are required to express a meaning? Some notations use a lot of symbols or a lot of space to achieve the results that other notations achieve more compactly.

**Error-proneness:** Does the design of the notation induce ‘careless mistakes’? Does it make them hard to find once they have occurred?

**Hard mental operations:** Are there places where the user needs to resort to fingers or pencil annotation to keep track of what's happening?

**Hidden dependencies:** A hidden dependency is a relationship between two components such that one of them is dependent on the other, but that the dependency is not fully visible. Is every dependency overtly indicated in both directions?

**Premature commitment:** Do programmers have to make decisions before they have the information they need?

**Progressive evaluation:** Can a partially-complete program be executed to obtain feedback on ‘How am I doing?’ The ability to evaluate their own problem-solving progress is essential for novices and desirable even for experts.

**Role-expressiveness:** Can the reader see how each component of a program relates to the whole? Role-expressiveness is enhanced by meaningful identifiers, by well-structured modularity, and by the presence of ‘beacons' that signify certain code structures.

**Secondary notation:** Can programmers use layout, choice of naming conventions, grouping of related statements, color, and other cues to convey extra meaning, above and beyond the ‘official’ semantics of the language?

**Viscosity:** How much effort is required to perform a single change? One standard example of viscosity is having to make a global change by hand because the environment contains no global update tools.

**Visibility:** Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will?

### 6.3 The questionnaire preparation

In addition to the Cognitive Dimensions notation we took advises from Dr. Y. Badihi a lecturer JCT lecturer in psychology, and the help of Y. Blushtein, a student who helped me in this evaluation stage.

The questionnaire had much iteration of versions until we achieved the final one.

There are four types of questions:

1. Knowledge: we have to be sure that the student understood the item.
2. Evaluation: we want to know the item evaluation by the students.
3. Guidance: a question who guides the student to a specific weakness.
4. Collection: a question that collects new ideas/proposals from the students.

The evaluation has five options 1 to 5. In each question the numbers were explained briefly to their signification according to the question.

The evaluation questions were redacted as "to what degree ..." and according to this redaction the following explanation was presented for the five values:

1: not at all
2: small degree
3: medium degree
4: large degree
5: very large degree

A total of 21 questions were redacted and presented.

6.4 The population

The questionnaire was presented to a group of 54 students of the Computer Science department who are finishing the 4th year (the last engineering year). The questionnaire was a continuation of their exercise based on the Arts’Codes method which was previously explained. The exercise was based on the Steam Boiler Controller case-study [33], which was designed according the Arts’Codes principles.

According to the knowledge questions all students who have a mark under 60/100 were filtered, and a total of 50 students remained for the evaluation.
The questionnaire was individually filled-in by the students; with no help one from the other.

6.5 The questions

Each question tests a specific Cognitive Dimensions which appears near each question.

The following six questions are composed of knowledge and evaluation:

1. 
knowledge: What are the two principal diagrams in this method ?
Evaluation: To what degree splitting the design into two diagrams contributes to the comprehension ?
Cognitive Dimension: Hard mental operations
**knowledge**: When is a sub-component active?

**Evaluation**: To what degree the sub-component activation is clear?

**Cognitive Dimension**: Role-expressiveness

3.

**knowledge**: How components concurrency is expressed?

**Evaluation**: To what degree the component concurrency is clear?

**Cognitive Dimension**: Role-expressiveness

4.

**knowledge**: What is a Gate?

**Evaluation**: To what degree using Gates helps to the design comprehension?

**Cognitive Dimension**: Hard mental operations

5.

**knowledge**: How strayed behaviors are handled in this method?

**Evaluation**: To what degree this strayed behavior handling method, improve the design comprehension?

**Cognitive Dimension**: Abstraction gradient

6.

**knowledge**: What is an Assertion?

**knowledge**: When an Assertion is active?

**knowledge**: How an Assertion connects with the Guard?

**Evaluation**: To what degree the use of the Assertion is efficient?

**Cognitive Dimension**: Abstraction gradient

The following five questions are for evaluation with guidance to a specific weakness:

7.

**Evaluation**: To what degree is easy to find a specific item in the diagram?

**Guidance**: what is the item which is harder to find?

**Cognitive Dimension**: Visibility

9.

**Evaluation**: When you have to make a change to a diagram, to what degree is easy to make it?

**Guidance**: what are the changes which are harder to make?
**Cognitive Dimension**: Viscosity

10.  
**Evaluation**: To what degree the design time is reasonable?  
**Guidance**: what is the design part who take the biggest efforts?  
**Cognitive Dimension**: Diffuseness

11.  
**Evaluation**: To what degree some parts of the design are hard to understand?  
**Guidance**: which part particularly?  
**Cognitive Dimension**: Role-expressiveness

15.  
**Evaluation**: To what degree there is an item in the design, which seems to you that describes the system strangely?  
**Guidance**: which item?  
**Cognitive Dimension**: Role-expressiveness

The six following questions are only for evaluation:

8.  
**Evaluation**: If you have to compare two different design items, to what degree you can see both at the same time?  
**Cognitive Dimension**: Visibility

13.  
**Evaluation**: to what degree this design method describes accurately the system structure?  
**Cognitive Dimension**: Closeness of mapping

14.  
**Evaluation**: to what degree this design method describes accurately the system behavior?  
**Cognitive Dimension**: Closeness of mapping

16.  
**Evaluation**: When you read the diagrams to what degree each item role is understandable in a global view?  
**Cognitive Dimension**: Role-expressiveness
17. 
**Evaluation:** If you have to connect items, where a change of one of them influences the other, to what degree this connection is clear?

**Cognitive Dimension:** Hidden dependencies

20. 
**Evaluation:** To which kind of system this method is suitable?

The following four questions roles are for collecting new ideas/information:

12. 
**Collection:** Which common errors seems to be frequent in this method?

**Cognitive Dimension:** Error-proneness

18. 
**Collection:** Is this method dictates some design order?

**Collection:** If the previous answer is positive, which order?

**Cognitive Dimension:** Premature commitment

19. 
**Collection:** On your opinion is there any need of comments adding in some design parts?

**Collection:** Where?

**Cognitive Dimension:** Secondary notation

21. 
**Collection:** Do you have any suggestion to improve the Arts’Codes method?

**6.6 Questionnaire analysis**

The following table (see Table 2) presents the evaluation’s scores given by the students. The columns are the question number, and each line represents a specific student.

The last line is the columns average.
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>?</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>?</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Questionnaire results
In the following table (see Table 3) the question averages are grouped by the Cognitive Dimensions. Near each CD appears the list of questions that the CD contains and the average of their scores. The table is sorted in an descending order according to the CD score.

<table>
<thead>
<tr>
<th>Cognitive Dimension</th>
<th>List of questions</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closeness of mapping</td>
<td>13, 14</td>
<td>4.3</td>
</tr>
<tr>
<td>Hard mental operations</td>
<td>1, 4</td>
<td>4.2</td>
</tr>
<tr>
<td>Abstraction gradient</td>
<td>5, 6</td>
<td>4</td>
</tr>
<tr>
<td>Role-expressiveness</td>
<td>2, 3, 11, 15, 16</td>
<td>3.8</td>
</tr>
<tr>
<td>Visibility</td>
<td>7, 8</td>
<td>3.6</td>
</tr>
<tr>
<td>Hidden dependencies</td>
<td>17</td>
<td>3.5</td>
</tr>
<tr>
<td>Viscosity</td>
<td>9</td>
<td>3.1</td>
</tr>
<tr>
<td>Diffuseness</td>
<td>10</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3: Questionnaire results grouped by CDs

6.7 Conclusions
As we can see in Table 3, Closeness of Mapping, Hard mental operations and abstraction are the stronger Cognitive factors (all between 4 - 5), all the other are not bad (between 3 – 4). This results points to accuracy in problem design and not too much cognitive efforts comparing diagrams, but concerning to modifications and looking for some design details in the diagram it is good but not enough.

6.8 Second Questionnaire
6.8.1 Description
A second questionnaire was presented a year after to a second group of students (a total of 20 students). They received an UML prepared design of a bottle filling factory, and they had to design this problem in Arts’Codes. Six students had a score in the knowledge questions lower than 60/100 and were filtered.

6.8.2 Results
Table 4 presents the second questionnaire results:
As in the first questionnaire, we present in Table 5 the questions results grouped by the different Cognitive Dimensions.

<table>
<thead>
<tr>
<th>Cognitive Dimension</th>
<th>List of questions</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closeness of mapping</td>
<td>13, 14</td>
<td>4.3</td>
</tr>
<tr>
<td>Hard mental operations</td>
<td>1, 4</td>
<td>4.3</td>
</tr>
<tr>
<td>Abstraction gradient</td>
<td>5, 6</td>
<td>4.0</td>
</tr>
<tr>
<td>Role-expressiveness</td>
<td>2, 3, 11, 15, 16</td>
<td>3.9</td>
</tr>
<tr>
<td>Visibility</td>
<td>7, 8</td>
<td>3.9</td>
</tr>
<tr>
<td>Hidden dependencies</td>
<td>17</td>
<td>3.6</td>
</tr>
<tr>
<td>Viscosity</td>
<td>9</td>
<td>3.7</td>
</tr>
<tr>
<td>Diffuseness</td>
<td>10</td>
<td>3.5</td>
</tr>
</tbody>
</table>

**Table 5: Questionnaire No. 2 results grouped by CDs**

The following table (see Table 6) presents a comparison between the two questionnaires results:
<table>
<thead>
<tr>
<th>Cognitive Dimension</th>
<th>Questionnaire 1</th>
<th>Questionnaire 2</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closeness of mapping</td>
<td>4.3</td>
<td>4.3</td>
<td>No difference</td>
</tr>
<tr>
<td>Hard mental operations</td>
<td>4.2</td>
<td>4.3</td>
<td>+ 0.1</td>
</tr>
<tr>
<td>Abstraction gradient</td>
<td>4</td>
<td>4</td>
<td>No difference</td>
</tr>
<tr>
<td>Role-expressiveness</td>
<td>3.8</td>
<td>3.9</td>
<td>+0.1</td>
</tr>
<tr>
<td>Visibility</td>
<td>3.6</td>
<td>3.9</td>
<td>+0.3</td>
</tr>
<tr>
<td>Hidden dependencies</td>
<td>3.5</td>
<td>3.6</td>
<td>+0.1</td>
</tr>
<tr>
<td>Viscosity</td>
<td>3.1</td>
<td>3.7</td>
<td>+0.6</td>
</tr>
<tr>
<td>Diffuseness</td>
<td>3</td>
<td>3.5</td>
<td>+0.5</td>
</tr>
</tbody>
</table>

Table 6: Questionnaires comparison

6.8.3 Conclusions
As we can see in Table 6, Closeness of Mapping, Hard mental operations and abstraction are always the stronger Cognitive factors (all between 4 - 5), the other entire are still lower but better (high medium 3.5 - 4). This result confirms the previous questionnaire results: accuracy in problem design and not too much cognitive efforts in comparing diagrams, but concerning to modifications and looking for some design details in the diagram it is good but not the best. The reason for the better scores at the second questionnaire may be a consequence of methods comparison (UML versus Arts’Codes). The second questionnaire was presented after an exercise based on the two methods. May be that these scores are relative to the UML method and not absolute scores. In other words: if UML looks so, then Arts’Codes can receive more points!
Summary of Design contribution

This section summarize the Arts'Codes design method by: presenting the methodological contribution, defining the essential point of the approach and listing the Arts'Codes innovations.

7.1 Arts'Codes contribution

The natural Arts'Codes approach, where the cognitive efforts are taken with care, achieved its goals, restricting all the design into two types of interleaved diagrams.

The readability (of the various behaviors) and accuracy are strong features. This can be seen at the evaluation test, where Closeness of Mapping, Hard mental operations and abstraction were detected as the stronger cognitive factors.

The main topic that has to be researched and enhanced is the modification feasibility, but it seems to be related to the student's experience. More training may overcome this lack.

Another approach to overcome the modification feasibilities is to build design patterns, studying the different forms of expressiveness; giving a broad catalog of different solutions.

Arts'Codes gives emphasis to the cognitive efforts factor, eliminating even powerful notations elements, preferring simplicity and clear design.

Arts'Codes also provides clear and natural phases of development, where the program structure is used as an aid for the mental image.

Arts'Codes also attempts to clarify the design in order to achieve that "what you see is what you get".

Therefore Arts'Codes supplies:

- Strong isolation between the environment and the application.
- Connections between components are fully described and traced.
- Association and composition links have well-defined bounds.
- Normal and excepted behaviors are separated.
- Priorities are not attached to events, but they are attached locally to a specific transition or component.
- Concurrency and composition have a strong cognitive notation.
Strong and well-define connection between design and execution.

7.2 Arts'Codes Innovations
The following list contains the design features innovated in this thesis:

§ Sub-component activation: sub-components are scheduled in a FSM.
§ Gates: FSM hierarchy model is connected via gates, forming a unified well-modularized structure.
§ Manager/Guard: split of normal and excepted behavior.
§ Sockets: a graphical form of instantiation.
§ Assertion: well-defined and focused exception handling.
§ Composition definition: a new composition definition.
Part III

Real-Time Execution Platform

In this part the execution platform is presented. The concept execution platform merges a set of capabilities which enables to run the application. The nature of this execution platform handles with concurrency, communication and synchronization as implicit mechanisms, which are derived from the specific graphical design and from the model definition. In this part we present the various kinds of platforms proposed in the literature, show what we think should be enhanced; and describe our solution of execution platform as is, with its characteristics and capabilities for Arts’Codes. In the next part (part IV and V) we shall show how a specific application design is compiled and run on such a platform.

Chapters for this part:

8 Literature Review on Execution
9 Proposal for the Execution
10 Summary of contributions for the Execution
8

Literature Review on Execution

8.1 Multi-threading platforms

8.1.1 Background

Nowadays real-time developers prefer interrupt-oriented and multi-threading platform platforms for real-time applications. For them, it is a natural passageway from regular applications. Multi-threading platforms approach split the application in different concurrent modules, synchronized between them. In Java (C++ or C#) for example, classes may inherit from the predefined Thread class, and becomes a thread. This class can be instantiated many times, producing in such a way a set of similar concurrent threads. The scheduler is preemptive and responsible of the CPU sharing between the threads, usually according to by-priority algorithms. This approach is obviously non-deterministic, and has also to pay for overhead fees concerning to context-switch handling by the scheduler. A real-time application that adopts the multi-threading approach uses usually a Real Time Operating System (RTOS).

8.1.2 Real Time Operating System (RTOS)

Real time operating systems are operating systems that support a list of features such as:

- **Fast interrupt handling:** the response time to a hardware interrupt must be as short as possible, in order to reduce the overhead on interrupt handling and context-switch (a special case of a timer interrupt handling) and in order to try to foresee the timings.

- **Hardware memory protection:** The memories of different threads are isolated. The memory protection must be done by hardware; otherwise the overhead of secure handling will be too expensive. The deletion of data or code from memory after there use, is either forbidden or strictly controlled in order to prevent loosing time.

- **Preemption:** A running thread can be preempted and replaced by another one, whenever the RTOS requests.
Minimized overhead of system calls: system calls requests must have take a time as small as possible.

Multiple priorities levels: different families of priorities are defined, assuring the correct order of execution for concurrent threads.

Priority inheritance: when a resource is assigned to a low-priority thread, needed by a high-priority one; then temporarily the low-priority thread inherits the high-priority until it released the requested resource.

Light and customized kernel: usually real-time applications are run on embedded micro-controllers with restricted memory size. Therefore the RTOS must support a small kernel. This restriction obligates RTOS designers to modularize the RTOS, in order to enable a customization of the RTOS parts that will be downloaded to the small embedded device.

Predictable response time: response time is pre-known for reactions which appear in an interrupt handler and in system calls; but concerning for reactions in a thread the response time is unknown, and it depends on the higher priorities threads execution.

### 8.1.3 Advantages & Disadvantages

RTOS solution gives familiar skills of software developments, which causes programmers select multi-threading platforms as a framework for their real-time products.

But concerning the real-time performance, it seems to be poor. The follow list presents the main disadvantages of multi-threading platforms for real-time applications:

- **Non-determinism**: the same input series may produce different outputs, according to the different status of scheduling when the series come.

- **Unpredictable response-time**: The interrupts or system calls latency can be known for the best RTOSs, but the reaction latency for a specific event appearing in a thread depends on the ready-queue size and the different priorities of the waiting threads, provoking unpredictable response-time.

- **Racing conditions**: Concurrency in the multi-threading pattern, by definition, promotes simultaneous access to shared data. This can be solved by synchronization tools, but has to be handled separately and requires additional execution time.

- **Interrupt nesting**: Interrupts with high-priority are handled in middle of lower-priorities handling, provoking interrupt nesting. Nesting is an unstable situation and difficult to debug, and therefore undesirable. Different forms and combination can occur.
Unstable Preemption: Threads are preempted only on policy request of the scheduler, no matter in which phase of execution they are. Certainly this situation can be solved by atomic transactions, but it is not an implicit feature of the RTOS platform.

In summary, RTOS operating systems, based on the multi-threading platform, were developed to be a basis for real-time systems; giving the basic tools for these applications time restrictions. But in terms of determinism they trend to be unpredictable. This lack is originated from the scheduling algorithm which depends on the currently states of each threads. The order of the thread scheduling not only depends on the input set, but also on the whole operating system status that differs in different running. Also the multi-threading method pays context-switch latency overhead and lacks from the race-condition problem due to their concurrency and data sharing.

8.2 Synchronous platforms

Reactive systems
A reactive system has to react to incoming inputs by emitting suitable outputs. The following pretty Berry’s description illustrates a well-defined image of a reactive system and its environment interaction:
"The clearest fact about a reactive system is that there is an alternation between environment moves and program moves, like in a two-player game. The environment chooses the inputs of the program, the program replies by computing the outputs. The game is asymmetrical in the sense that the environment drives it by choosing the inputs and the timing. The reactive program is in a slave position and it must be always ready to accept any input. Therefore, the global input can be characterized as being an environment-provided sequence of input events" [35].

Reactive systems trade-offs
The simple way to apply this concept is by reacting to each input when it comes, handling the incoming events by an event-driven loop. A problem of this simple approach is that it can conduct to a non-deterministic system, because the reading rate of the inputs and the reactions order influence to the final output results (the reading and reaction of the next event depends on the reaction time of the previous one).
Another disadvantage is that urgent events are treated no faster than regulars ones, delaying critical outputs to be emitted at the required time. Therefore various synchronous platforms were developed to try order assuring determinism and clearness, building in that way robust applications and improving the clearness of the design.

**Berry’s optimal model**

The following are the properties for an optimal model of programming proposed by Berry [35]. According to his opinion the programming model has to be:

- **Simple**: "simple and intuitive to help user understanding"
- **Accurate**: "accurate enough to describe the physical reality one deals in a sensible way"
- **Semantic**: "mathematically efficient to be useful for defining the semantics of programs and for program analysis, optimization, and verification"
- **Realizable**: "general enough to cope with different ways of realizing systems"

**8.2.1 Berry's diagnosis**

The famous Berry's Synchrony hypothesis comes to assure the design of reactive systems. Berry diagnoses two essential natural properties of reactive systems which complicates the development:

- **Reaction nesting**: A reaction to an event can be nested by a new event which preempts the current reaction. This phenomenon may be repeated many times causing a set of competed reactions and even recursion. This nature of input behavior makes a foggy picture of possible system scenarios and makes difficult the system design.

- **Unstable Environment image**: The environment image is composed by the different inputs. The inputs are dynamic values which changes their values with the time advancing. Since the dynamic on the input values, two reactions will see different environment image and then will react in a different manner causing an unpredictable system behavior.

**8.2.2 Berry's solution: Synchrony Hypothesis**

The magic solution is assuming just for the design and verification model that we have hyper-fast machine:
"All the above problems disappear when one adopts the Synchrony hypothesis: each reaction is assumed to be instantaneous – and therefore atomic in any possible sense. Synchrony amounts to saying that the underlying execution machine takes no time to execute the operations" [34].

This approach partially asides the time dimension from the system simplifying the implementation:

"time is define externally to programs by the flow of inputs, and that program internal bookkeeping is done in zero-delay with respect to all external time units. The only instructions that take time are those explicitly required to do so" [36].

The simplicity improvement of the zero-delay (synchronous) model pop ups when concurrency is adopted, giving a constant and uniform environment image to all system components:

"At reaction $n$, processes $p$ and $q$ see the same inputs on the signals they share, their common outputs are merged, and each of $p$ and $q$ sees the signals emitted by the other process. A parallel statement performs its own bookkeeping in zero-delay, and all control handling operators should have the same property" [36].

Unfortunately the Synchrony Hypothesis has problems at the implementation stage, where we see that we have not a hyper-fast machine:

"There we do not require exact simultaneity of $I_n$ and $O_n$" [36].

At this stage the application has to be configured suitable to the requested platform, by calibrating the cycle duration:

"This is well known for digital circuits: the design is done at zero-delay level, which makes expression, simulation, verification, and optimization much easier. Then, for each target technology, accurate timing analysis is performed on the optimized, placed and routed circuit to compute the minimal clock cycle time, i.e. the time physically required to obtain $O_n$ from $I_n$ in all possible situations"[36].

The ESTEREL synchronous language which adopts the Synchrony hypothesis is based on the separation of concerns principle, using two basic models:

§ **Semantic:** in this model the programmer deals with the program behavior in a logical way "explaining why things happen but neglecting how they are actually realized. In particular, we neglect the time it takes to emit signals or to propagate the control" [35].

§ **Implementation:** Applied by a compiler, who moves the logical model to a more detailed platform, realizing the real system. Inter-model consistency specially timing analysis has to be checked. "Accurate timing analysis is much harder for software implantation... however, that problem is not particular to Esterel... the only things we propose for Esterel are to develop good optimizers" [35].
8.2.3 Synchrony hypothesis Implementation
As mentioned before, when coming to the implementation stage, the zero-delay model crashes because the reactions really take time. This restriction can be deal, by assuming that the reaction is done before the next input arrives. In this manner the machine is relatively fast, but not infinitely fast.
The mentioned approach can be applied by the running platform simulating a synchronous circuit:
"The software analogue of synchronous circuits is cycle-based reaction, a very common model in software process control. The implementation cyclically repeats a sequence of three actions: reading the inputs, computing the reaction, and producing the corresponding outputs" [38].
The reaction is done really before the next input arrives, because artificially inputs are delayed:
"Inputs events occurring during a reaction are queued for the next reaction, which makes the reaction atomic and deterministic" [38].

8.2.4 Synchrony Hypothesis interpretations
The synchrony hypothesis is interpreted in the literature by several authors in different ways, each one emphasizing other property.
One emphasizes the separation between computation and communication:
"The synchronous assumption implies a total order of events and leads to a clean separation between computation and communication and gives a solid base for formal methods" [39].
Few tends to emphasize the instant signal broadcasting advantage who gives a unified global image of the system: "Signals are instantaneously broadcast: this leads to a global perception of the system" [42].
Others tend to emphasize the global clock and zero-delay: "makes two fundamental assumptions: the existence of a global clock abstraction to discretize computation over instances, and computation takes no time within each instance" [40].
Harel opine that the perfect synchrony hypothesis asserts "that external events are responded to immediately" [45].
Pnueli also emphasize the system response time "the response to an external stimulus is always generated in the same step that the stimulus is introduced ... the synchrony hypothesis is an abstraction that limits the interference that may occur in the time period separating the stimulus from the response and , hence, provides a guaranteed response as a primitive construct" [55].
The order of input and outputs is emphasized by Bhattacharjee "perceive all the external events in a suitable order and produces the outputs reactions before reacting to a new input event set" [54].

A similar opinion is brought by Halbwachs sustaining that the Synchronous paradigm essence is the atomic reaction: "such reactions may involve many computations, which, from the automaton point view, are considered atomic (i.e., input changes are only taken into account between two reactions). This is the essence of the synchronous paradigm, where such a reaction is often said to take no time. An atomic reaction is called an instant (logical time) and all events occurring during such a reaction are considered simultaneous" [56].

Also Boussinot and DeSimone highlights "that the environment does not interfere with the program during reactions. For example, an input event used by a program is not allowed to change before the end of the current reaction. From the logical point of view, it is equivalent to consider that the program are always ready to accept new input events, or in other words, that reactions 'take no time'" [57].

A good summary is brought by Gunzert:

he list "the following abstractions:

- The computer is infinitely fast.
- Each reaction is instantaneous and atomic, dividing time into a sequence of discrete instants. Different reactions cannot interfere with one another.
- A system’s reaction to an input appears at the same instant as the input.

A real system can behave synchronously if it is fast enough. It must always finish its computations before more events arrive from the environment. This requires knowing both:

§ the minimum inter-event time as well as
§ the worst-case execution time.

The synchrony hypothesis is a generalization of the synchronous model used for digital circuits where each reaction must be finished in one clock cycle. The synchronous model of time simplifies the design of correct systems. Temporal details are hidden during specification and so the behavior of the system is also simplified. Nondeterministic behavior caused by the interference of parallel actions cannot occur. Deterministic systems are one order of magnitude easier to specify, analyze and test as non-deterministic ones" [41].

8.2.5 Synchronous versus Asynchronous models
The synchronous model is called also timed-triggered model, because it is the base system clock which triggers the reactions. The opposite model is the asynchronous or event-triggered where the reactions are triggered by the event coming without a global synchronizing clock. Gunzer discuss the advantage of the synchrony (time-triggered) model versus the event-triggered one:

"The advantage of an event-triggered architecture is flexibility, the disadvantage, however, is non-determinism. For safety-critical systems, this is not acceptable. In a time-triggered system all activities are driven by the progression of a global time base. All tasks and communication actions are periodic" [41].

Halbwachs highlight the timed-triggered architecture advantages, concerning to determinism: "This instantaneous' communication is called the synchronous broadcast. The important point is that, in contrast with the asynchronous concurrency considered in asynchronous languages like Ada, this synchronous product can preserve determinism, a highly desirable feature in reactive systems design" [43].

In Harel's Statemate system [45] both models are supported, but even in the asynchrony model the zero-delay is supported. One of the most significant differences between the two models (in Statemate) is that in the Synchronous model the time increments automatically between the cycles, in contrast to the asynchronous model where the time must be incremented explicitly. Moreover, the asynchronous model allows "several steps to take place within a single point in time" [45].

In Selic's ROOM method [28] the asynchronous model was adopted:

"We have chosen the run-to-completion programming model for the behavior of actors. In this model an actor is normally in a receiving mode during which it awaits incoming events. If an event occurs, the actor responds by performing some activity appropriate to that event and then returns to the receiving mode to await further events. If a new event occurs while an actor is still busy processing the previous one, the new event is queued by the receiving end port and will be automatically resubmitted when the actor returns to receiving mode" [28]. The events are ordered in their queue by the priority assigned to each event.

**8.2.6 Synchrony Hypothesis Properties**

The synchronous system improvements to real-time application's development, can be listed by:
§ Having **timeliness** "since this implementation is a finite state machine, the maximum amount of time taken by any reaction can be accurately bounded if the execution times of the transitions are known" [44].

§ Having **clearness** because "this scheme is attractive because it simplifies formal treatment of the language, yet allows a programmer to write programs whose behavior is much clearer to the human reader" [44].

§ Being **deterministic** "Along with the synchrony hypothesis, both communication and pre-emption preserve determinism" [44]

§ Having **less overhead** like in multithreading systems "does not incur any run-time" [44].

### 8.2.7 Preemption

"Process preemption deals with controlling the life and death of concurrent processes" [47]. Well defined preemption mechanism is critical for reactive systems.

A typical preemption is a process abortion or suspension.

Berry [47] distinguishes between two interpretations of preemption: "may preempt" and "must preempt".

"May preemp" it doesn't assures the preemption immediately. This lack may be critical in a real-time system. The "must preempt" in contrast assures an immediate preemption which can be implemented only in a time-based system.

Berry introduced the "must preemption" as a built-in feature in synchronous languages.

This implicit feature is naturally included in synchronous platforms, because all inputs are checked at each step, and all concurrent threads shared the same "frozen" input, and therefore rapid "must" reactions can be provided (which is not the case in multi-threading platforms).

It is optional to give to the process the option of performing its "last wills" before a preemption is optional. In weak abortion the option is given, but in strong ones not.

These kinds of abortion can be delayed or immediate, where delayed means that current signals values are not related: "a trigger waits for a strictly future occurrence" [49].

In summary four types of possible abortion are provided [47]:

1. **Delayed weak abortion**: ignores current signals value at starting time, and gives the "last wills" before abortion.
2. **Immediate weak abortion**: relates to current signals value at starting time, and gives the "last wills" before abortion.

3. **Delayed strong abortion**: ignores current signals value at starting time, and doesn't give the "last wills" before abortion.

4. **Immediate strong abortion**: relates to current signals value at starting time, and doesn't give the "last wills" before abortion.

Suspension and resuming are also supported by Berry, who introduced all these preemptions as operators in ESTEREL. All these preemptions are performed basically in the Synchrony hypothesis in order "to be able to abort or suspend any statement at any time" [47].

### 8.2.8 ESTEREL Language

"Esterel is an imperative concurrent language specifically dedicated to control-dominated (parts of) reactive programs" [35]. It is a language which provides parallel activities that communicates with broadcast signals. At each instant a signal has only one presence status, and it can be present or absent. This feature allows data coherency for all listening partners [57].

The following Esterel presentation is for the Pure Esterel sub-language where the inputs and outputs are restricted to pure signals. Pure signals are simple variables with only two optional values: presence or absence status.

Esterel is organized in modules with input/output interface and an executable body. The module syntax is:

```plaintext
module M:
    input names;
    output names;
    statement
end module
```

The statement role is to react to the inputs signal, by computing all outputs signals. This reaction is instantaneous also called instant.

"A statement can start in some instant; it then remains active within the instant and possibly for some further instants until it relinquishes control, either by terminating or by exiting a trap. The only way for a statement to stay active from one instant to the next one is to explicitly execute a `pause` statement, which pauses for exactly one instant" [58].

It has a state which is implicitly encoded, and the reaction is according to the current input history state.
Esterel statements can be kernel-statements or derived-statements. In this section we will concern only to the first statement set.

The following are the kernel-statements:

- \( \text{nothing} \): nothing is executed.
- \( \text{emit } S \): local or output signals are "determined on a per-instant basis".
  
  By default is absent, and is present if an \( \text{emit} \) statement is executed.
- \( \text{pause} \): it pauses a statement and continues it in the next instant.
- \( \text{present } S \text{ then } p \text{ else } q \text{ end} \): starts body \( p \) or \( q \) according to \( S \) status.
- \( \text{suspend } p \text{ when } S \): \( p \) is guarded by \( S \) except for the first instant.
- \( p; q \): executes \( p \) and then \( q \).
- \( \text{loop } p \text{ end} \): execute body \( p \) repeatedly. It can be stopped by a \( \text{trap} \) and \( \text{exit} \).
- \( p \mid q \): starts \( p \) and \( q \) in parallel.
- \( \text{trap } T \text{ in } p \text{ end} \): defines an exit point which starts the body \( p \).
- \( \text{exit } T \): exits the trap \( T \).
- \( \text{signal } S \text{ in } p \text{ end} \): starts the body \( p \) with a fresh signal \( S \).

**Causality**

"The principle of causality requires that there is a clear causal ordering among the transitions taken in a step, such that no transition \( t \) relies for its activation on events generated by transitions appearing later than \( t \) in the causal ordering" [55].

"The intuitive semantics specifies what should happen when executing a program, but it does not guarantee that an execution actually exist and is unique. Indeed, we shall need extra criteria for this to happen". Berry's Esterel language supposed to be robust, but unfortunately sometimes leads to non-determinism or even incorrectness.

The causality defects in Esterel (which cause to the constructive approach amend), are a consequence of the instantaneous emission of signals (effect) in the same point of time of the inputs arrive (cause).

Examples of these defects are brought in the next section.

**Logical Correctness**

The logical coherence law says "a signal is present in an instant if and only if an \( \text{emit } S \) statement is executed in this instant" [58].

When instantaneous reactions (the input and the output are in the same point of time) are adopted this definition may produce paradoxical statements between control and signals.
The following is an inputless module ("inputless program reacts on empty input events, i.e. on 'clock ticks'" [58]) which represents a logically incorrect program [58]:

```plaintext
module P
output O
  present O else emit O end
end module
```

Here no logical coherent assumption can be made:
1. Assuming that O presents is not true because then the sentence emit O will be not executed.
2. In contrary assuming that O absents is also not true because then the sentence emit O will be executed.

The following is a nondeterministic program [58]:

```plaintext
module P
output O
  present O then emit O end
end module
```

Here all logical coherent assumptions can be made:
1. Assuming that O presents is justified because emit O is then executed.
2. In contrary assuming that O absents is also justified because emit O is then not executed.

**The Constructive Approach**

The Constructive approach came to solve the Causality defects in Esterel. Logical correctness is not enough for programming. The program has to be in accordance also to the intuitive semantics of the language, which is suitable to the programmer's intention.

In a present statement when we write present S then p end, "we obviously mean 'first test the status of S, then execute p if S present', assuming that the status of S should not depend on what p might do" [58]. It is not that the cause must precede the effect in meaning of time "since everything is conceptually instantaneous, but that of sequential causality" [58].

Note that aside of the concurrency operator '||' all statements are sequential. The following example illustrates an intuition contradiction of sequential execution:
Present O then nothing end;

Emit O

It is logical coherent but the status of O flows backward contradicting the sequential operator ';'.

The approach of the constructive semantics is to add restriction to the logical coherence rule as follows:

§ "A signal is declared present if and only if it must be emitted"
§ "A signal is declared absent if and only if it cannot be emitted." [58].

A signal then has three possible statuses:

1. Yet unknown (⊥)
2. Known to be absent (-)
3. Known to be present (+)

According to the three-state signals, two approaches are proposed:

Behavioral Approach:
The approach is to analysis a way that all signals have a known status. During the test no speculative assumptions are made. The analysis restarts whenever found a new known signal status. When beginning all signals are unknown. The program is constructive only if all signals are known

Operational Approach:
This approach runs the program, and at each unknown signal status in a present statement it suspends the thread until the signal status clarification. Again, the program is constructive only if all signals are known.

8.2.9 Andre’s SyncCharts (or SSM - Safe State Machine)

SyncCharts is a new model derived from Statechart and ARGOS (ARGOS is a Synchronous language derived from Esterel). It interleaves graphic design with synchronous execution.

SSM is integrated in SCADE Suite™ and Esterel Studio™. SSM is the commercial version of SyncCharts [53].

There are several innovations that were introduced in SyncCharts (as presented in 2003 [49]) which were close to some of the innovations of our thesis (but not a component oriented context as ours), without knowing each other. Note that Arts’Codes was also publicized in the same year [18] [50] [51].

Graphic design
The structure of SyncCharts is composed by states and macrostates which "contains other states. A simple state does not".

"A State-Transition Graph (STG) is a connected set of states. A STG must have an initial state" and "also have final states, denoted by double circles".

"STGs are necessarily contained in a macrostate", and when "several STGs are in the same macrostate, they are separated by dashed lines, and they are said to be concurrent".

"SyncCharts definitely surpass Statecharts: they make a clear distinction between various kinds of preemption (abortion or suspension, weak or strong)" "they promote preemption as a first class concept, so that normal as well as abnormal behaviors can be easily specified" [46].

The state transitions are defined in three categories:

"There are three types of transitions: strong abortion, weak abortion, and normal termination transitions" [49] (see Fig. 30).

Also a restriction is made when connecting the transition to its source and target states:

"Contrary to Statecharts, SyncCharts respect a strict containment policy: there is no inter-level transition" [49].

**Synchronous Execution**

The reactions are executed cyclically as other synchronous models:

"An evolution cycle is as follows:

1. **Read the inputs**: in our restricted presentation, this means "get the presence status of each input signal, yielding an input image".

2. **Compute the reaction**: according to the internal state of the syncChart and the input image, compute the new internal state and the output image (i.e., find for each output signal its new presence status).

3. **Perform the outputs** (i.e., effectively deliver output signals to the environment)."
It applies the super-step instant, built from several micro-steps:
"Thus, a reaction appears as a sequence of instantaneous transitions or microsteps driven by causality relationship" [49].

An execution tracing is brought by Andre, showing the SyncChart behavior of Fig. 31:

"Suppose that states wA, wB, and idle are active in syncChart ABSync. If A is present, then the transition from state wA to state dA is taken. The associated effect (emission of local signal arm) is executed. Now, this signal is instantaneously broadcast. State idle was waiting for the presence of arm. The
presence of arm triggers the transition from state idle to macrostate Timer. Entering macrostate Timer, causes the activation of the initial state of the STG in Timer. Further evolution is no longer possible in this reaction [49].

**Abortions Types**
The strong Abortion differs from the weak one in the option to finish the microstep's evolution:
A "strong abortion ... is exited without any prior internal evolution" [49]. while "a weak abortion is performed differently: before exiting the macrostate origin of the weak abortion transition, the internal evolutions of the macrostate are executed" [49].
This distinguishing was taken from ESTEREL (and adopted also by Arts'Codes using negative and positives priorities).
"A normal termination transition has no explicit trigger".
The order of transitions testing is determinate by their priority: "strong abortions have priority over weak abortions, which have priority over a normal termination" [49].
Besides of this priority rule, the programmer can attach a priority number to a transition (Arts'Codes supports priorities attaching to transitions and even to a component, which is crucial for determinism).
Another kind of preemption is used by Andre: the suspension. The suspension notation looks like a lollipop which frozen the FSM. During the FSM suspension signals are ignored and no transitions are executed. Note that suspension is attached only to macrostates.
When a hierarchy of STG is defined, the order of execution is:
1. Strong abortion
2. body: inner STGs
3. suspension
4. weak abortion
This means that no body execution is made when a strong abortion is active. By a weak abortion, the transition is made (as mentioned) after the body execution.

The usual behavior in SyncChart is that when entering a state current signals values are not related: "a trigger waits for a strictly future occurrence". The default "strictly future occurrence" (or delayed at ESTEREL) can be modified using the '#' sign which converts it to an immediate preemption (Statemate differs not supporting the immediate preemption). As in ESTEREL immediate and delayed preemption can be combined with strong and weak preemptions, producing four.
kinds of abortion. It is denoted that an immediate strong abortion can bypass a state.

**The Reactive Cell**

The following definition of a Reactive Cells, is the simple way for the execution of a FSM's hierarchical structure (this mechanism is similar to the Arts'Codes one and it will be discussed later, see section 9.4.3).

Andre defines Reactive Cell as a "state with its outgoing transitions. A reactive cell can be active (alive) or idle (doing nothing at all)" [49].

The FSM hierarchy is handled as follows:

"An active reactive cell is permanently testing the triggers associated with its transitions. As soon as a transition can be taken, the reactive cell is deactivated and the reactive cell target of the transition is activated. A reaction now appears as a propagation of activations/deactivations among a collection of cells" [49].

This execution model was been inspired from Boussinot’s reactive objects [52].

**Signals**

There is a differentiation between input signals coming from the environment and those which are locally emitted:

"When computing a reaction, only input signals, which are imposed by the environment, have a definite status. The presence status of all other signals must be determined. Like in the ESTEREL language, we assume that a non-input signal is present if and only if it is emitted during the instant" [53].

Signals may be pure signals or valued signals. Pure signals can present (+), absent (-) or undefined (⊥).

Valued signals have the mentioned 3 options, with an attached value.

A trigger that have an undefined signal, it is delayed until the signal is defined.

The pure signals are not "persistent", which means that they are reset between the steps.

Pure signals have a life time of only one step.

**Other topics**

The method supports multi-threading.

It seems that this method does not support the history feature like in Statecharts. This method supports macro-state reference which is similar to the Component type in Arts'Codes. This feature enables the creation of several instances of the macrostate type. Even this feature the static structure is not treated enough and it is really poor.
Reincarnation (something like recursion) by a normal termination produces new instances of local signals. This phenomenon may produce confusion and it must be take with care.

There are no static reactions.

All reactions leave the state (in Arts' Codes loops transitions are static reactions).

**The Execution Platform Algorithm**

The SSM executes its several hierarchical levels recursively. The algorithm starts form the top (highest level or root) macro-state (see Fig. 32), "the reaction of a macrostate relies on the reactions of its STGs. The reaction of an STG relies on the reaction of its reactive cells. The reaction of a reactive-cell relies on the reaction of its body (a macrostate or a simple-state). Of course, this approach is recursive and has to be applied down to the leaves of the tree which are simple-states" [53].

At any level of the hierarchy each macro-state or simple state acts as a reactive-cell, i.e. has a behavior that reacts to signals (or variables). Fig. 32 illustrates globally the recursive/parallel algorithm.
The following text boxes contain the various functions which perform the recursive execution algorithm. The titles are colored with yellow, and recursive calls are grayed.

**Compute the SSM**
- Read external inputs
- Set all outputs to **unknown**
- **Compute the macro-state** (top macro-state)
1 Scans all transitions in decreasing order priority, and for each unknown signal waits until the tested signal status is fixed (by another concurrent thread) and then the transition trigger can be tested.
8.2.10 Harel’s Statemate [45]

Abstract
Statemate is the system which implements the Statecharts semantics. The well-known Statechart design (see section 4.3) method is based on hierarchical state-diagrams which can be combined with state-diagrams concurrency. Other features were added like history connectors which remember the current state before a high-level transition - for future resuming, entry and exit actions which defines the behavior when entering/exiting a state; and static transitions defining the inner behavior of the state. It also supports events (signals) with life time of only one step.

Delayed changes
The Statemate semantics adopt also the synchronous approach (synchrony hypothesis), and it slices the time in steps. In this method changes that occur in a step can be sensed only after the step is completed. In contrary to SyncChart this rule is adopted not only for externals stimuli but also for internals ones. According to this changes-delay, "the system cannot enter and then exit a given state in the same step" [45]. This restriction may provoke misleading at the programmer design. The following Harel's example points out the state transition misleading:

![Figure 33: Misleading example with state-transitions](image)

\[
\textbf{Compute the simple-state}(S:\text{ simple-state})
\]

\[
\begin{align*}
\text{\$} & \quad \text{If } S \text{ is a final-state then} \\
& \quad \text{Return Dead} \\
\text{Else} & \quad \text{emit effect (if is it assigned)} \\
& \quad \text{Return Pause}
\end{align*}
\]
In the Fig. 33, standing at state \( A \) at the beginning of the step, with events \( e \) and \( f \) generated, only \( t1 \) is performed because the arrived of state \( B \) is sensed only at the next step. Moreover, even at the next step \( t2 \) will be not performed because the events have only one step life-time.

Another restriction is that the evaluation of arithmetic expressions, are based on the variables value at the beginning of the step, even this value was changed. Following the original Harel's example is brought:

"the action \( act \) is defined as:
\[
X:=X+1; \; Y:=X*5; \; \text{if} \; X=5 \; \text{then} \; \text{act1} \; \text{else} \; \text{act2} \; \text{end if}
\]

The value of \( X \) used in evaluating the arithmetic expression \( X*5 \) and the Boolean expression \( X=5 \) is the value had at the beginning of the step (before it was incremented). If the value of \( X \) at the beginning of the step was 4, then \( \text{act2} \) (and not \( \text{act1} \)) is executed" [45].

This feature means that commands separated with semicolons in the same action are executed in parallel, which means "do this too" and not "and then do".

**History Connector**

Statemate supports two kinds of history: regular and deep. The history feature enables state registration of the deactivated automaton in order to be resumed in the future.

In the case of deep history, the state registration is made also in the inner hierarchy levels. The history connector as a transition target, is applied only if the system was in the past at the history connector host state.

The following is an Harel's example where the Statemate will behave not as the programmer expectation. Let design the following Statechart (see Fig. 34):

![Statechart](image)

**Figure 34: Misleading example with history connector** [45]
The current state is the shadowed one and the event e is present. The reaction is "hc!(S)" (hc! is an abbreviation of history-clear). According to Statemate rules, changes are delayed to the next step and then the history clear will be applied not for the current step, which in consequence the target state will be the history state (which it may was not the programmer intention). This behavior is not graphically clear, and then will provoke a misleading (these aspects of cognitive problems were taken in care at the Arts'Codes improving the design method and platform execution – cooperation).

**Inter-level transition**

A transitions arrow can cross the boundaries of nested states (which is forbidden in SyncCharts). This feature points to one of the challenges of Statemate semantics definition: "to accurately define the states that have been exited by taking the transition and those that are entered" [45].

This rule may provoke possible illegal/conflicted transitions, for example two concurrent automata may reach two different or-states (two states in the same automaton). This option is checked by Statemate (In Arts'Codes we avoid it by abstracting concurrency in different components, what solved this problem implicitly by definition).

**Transitions Priority**

In order to solve the transitions conflict, where "there is some common state that would be exited if any one of them were to be taken" [45] or "they depart from the same state, and then, at most one of them can be taken at given step" [55], transitions priority has to be handled.

In Statemate transitions of upper levels have higher priority than those from lower levels (while in SyncChart, the transition priority is higher for a strong-abortion and lower for a weak-abortion). Transitions priorities of the same level are considered.

Static reactions (reactions performed in the state, without leaving it), are executed only if during the step the state is not leaved. This rule gives them a lower priority in comparison to the outgoing transitions.

**Activities**

Activities are "sub-components" (similar to Arts'Codes sub-components) which are textually activated/deactivated by the Statechart.

A transitions must be completed during the step, without any "stuck" in any connector, i.e. it has to reach a target state (this rule is applied also in Arts'Codes
beside of a heavy transition, by transiting the Manager into a "pseudo-stuck" during the heavy reaction, see section 9.2.5).

**The Execution Platform Algorithm**
Statemate is a Visual Programming Language (VPL), which means that the design stage is also the coding one. In order to run the designed application, the VPL must supply an algorithm which dictates the semantics of the execution platform. The Statemate's algorithm is described here, partially abstracted, in order to present only its basics.
In contrary to SSM, Statemate has not local variables. Its algorithm also doesn't need to work with recursion.
System status is defined by the following parameters:

§ List of current states
§ Current time
§ List of current active activities
§ List of internal events and variables with their current values
§ Information about history connectors
§ External events (environment changes)

In base of the system-status, the algorithm will perform the various active transitions: executing their reactions and arriving new states. This provokes a new system status, which will be the reference of the next algorithm performance, at the next step.
The following is a briefly description of the algorithm of execution:

§ Handle with transition conflict (this includes also static-reactions) and decide a deterministic (if is it possible, because no transitions priority is used, see Arts'Codes approach in section 9.4.5.2) set of transitions.
§ Execute all static-reactions
§ Execute the compound-transitions in the following order:
  o Remember the source-states for the history connector
  o Remove the source states from the system-status
  o Execute the suitable exit-actions
  o Execute the transition action
  o Execute the suitable enter-actions
  o Add to the system-status the target-states

**Note:**
§ All event and variable changes during the step, are not updated in their original address, they are only assigned to a variable-copy and updated at the next step. In this way the system is insensitive to the order of the
actions execution, using the last step's values. (In Arts' Codes the priority order of the transitions is used for transition conflict and also for the transition's execution order which one transition can affect the next one, because no variable replication).

Write-write condition is also checked when to actions in the same step, intents to change a variable value.

An activity which is activated in the current step can activate another sub-activity only in the next step.

**Time Dynamics**

One of the main competing in a real-time system is the time restrictions. Various transitions' triggers may be influenced by a time condition; which means that the time advance must be treated with care, in order to react as soon as the time is come.

In Statemate two models of time are proposed: Synchronous and Asynchronous.

**Synchronous model:**

In this model the time is incremented by one time unit between the steps. During the step one scan is made over the transitions list and the outputs are delayed to the next step.

Here the perfect synchrony hypothesis is not hold, because "the time it takes to react depends on the number of steps needed to complete the system reaction" [45] and also "it is possible that non-zero time passes between the instant in which the external event occurs and the time the system reacts to it" [45].

**Asynchronous model:**

In this model the system reacts to the external event whenever it comes, allowing the execution of several consecutives steps, one provoked by its predecessor. This steps series is called a super-step, which during it the time is not advanced, and the environment has no any effect "internal events are indeed sensed within a super-step, but externally generated ones are not" [45] (this model adopts by Harel "the perfect synchrony hypothesis").

A super-step main loop can be defined as follows:
"repeatedly execute one step until the system is in a stable state" [45], a stable state is reached when no more transitions are enabled. In the synchronous model instead only one step is performed per time unit.

**Racing conditions**

A Racing Condition may appear when two actions executed in the same step writes to the same variable. A less acute type of racing is where a variable is
changed and then tested in the same step. In this case the value of the variable is correct but the programmer may expect to another outcome (because the new value is sensed only in the next step).

More accurately a racing condition occurs when a series of transitions performed in two different legal orders gives different value to a specific variable. Or using the Harel's definition: "A race situation one in which, had we executed the enabled transitions in a different order (yet legal according to the above criteria), we might have obtained different results in one or more data-items or conditions" [45].

Racing Conditions are detected and reported by the system.

**Static Structure**
Not supported.

**8.2.11 Boussinot's Reactive Objects**
This model is based on synchronized concurrent objects which encapsulate data, shared by concurrent methods.

The objects "run in parallel and share the same temporal reference" [61]. The execution of the objects methods are divided in temporal instants, giving them the option to be run only once at each instant.

The execution of an object is terminated for the current instant when all its methods finishes. A new instant takes place when all objects finish their methods execution.

A method call is asynchronous, enabling the caller to continue its task without waiting for the called method return.

Objects are dynamically created and even methods can be dynamically added to objects.

The following is a paragraph took from Boussinot's paper [61], which gives a global sight to the execution scenario of the method:

"There exist an initial object, with only one method; this method is continuously executed by the system and each execution of it defines a new global instant. During one instant, objects are created and methods are called; execution of called methods can cause creation of new objects and calls of new methods, that will be executed in the same way. A new phase is issued when all methods either are suspended, or have finished to execute for the current instant. The current instant is terminated when all called methods have terminated their execution for that instant; then, the initial method is called another time, defining a new instant, and so on forever" [61].
8.2.12 Other Synchronous languages

Other synchronous languages are brought here for the literature review completion.

**SL** [57]
SL is a synchronous language which is in fact a restriction of Esterel. "The guideline is to forbid (unlike Esterel) the possibility of deciding that a given signal is absent during one instant, before the end of that instant" [57]. According to this restriction only weak preemptions are allowed. This approach avoids causality cycles and by definition uses only delayed signals: "the only moment one can decide that a signal is absent is the end of the current instant, as before that moment, the signal could always be later emitted. Thus, reaction to a signal absence is necessarily postponed to the next instant. In other words, unlike Esterel, SL forbids immediate reactions to signal absences" [57].

**Argos** [59]
It offers graphical and textual syntaxes. It is based on Statechart, but with inter-level restriction. The synchronous approach is strictly applied.

**Signal** [37]
A Synchronous data flow language similar to Lustre (see next section). It has no universal clock and it is determined at compiler time. The program is defined by a set of equations based on signals. The inputs and outputs are discrete values series: "Inputs and outputs are sequence of values of some type, each value of the sequence being present at some instant".

The system abstracts the time simplifying the design: "A Signal program does not deal with the exact duration between two signals. It only knows the relative order of signals, with the fact they are simultaneous, or not". Also the "zero-delay" model is applied: "An other hypothesis is that computation takes a null time. So we consider that new values are produced at the same logical instant as data used in their computation".

Note that time can be modified by modifying the external time input signal, coming from the environment.

**Lustre** [59][60]
A synchronous data flow language similar to Signal. Its formalism is very similar to temporal logic.
It is also declarative because it is compound by a set of equations that must be always verified by the program variables.

8.3 Parnas’s pre-run scheduling [62]

Real-time Operating Systems (RTOS, see section 8.1) supplies a multithreading platform for real-time applications, based on priority scheduling, which is adapted to the real-time applications time constraints. This multithreading platform makes easy the application programming, and the design stage. But the main problem with the RTOS proposed solution is the lack of determinism and overhead context switch handling time.

Parnas is one of the well-known researchers that alerted the academic population to prefer pre-run-time scheduling when it is possible. He wrote in its short abstract for his paper titled "Priority Scheduling Versus Pre-Run-Time Scheduling", preceding the research conclusions: "Builders of real-time systems often use priority scheduling in their systems without considering alternatives. This paper examines one alternative, pre-run-scheduling, and show that when it can be applied it has significant advantages when compared to priority scheduling schemes" [62].

Parnas conclusion is that comparing to the off-line approach the on-line priority scheduling:

1. Is difficult to handle "complex applications constraints" with it.
2. "Achieves lower processor utilization".
3. Run-time behavior "is much more difficult to analyze and predict".
4. "Provides less flexibility in designing and modifying the system to meet changing application requirements".

The on-line priority scheduling disadvantages are mainly a consequence of:

1. "Rigid hierarchy of priorities"
2. schedules "are computed on-line"

In summary the main Parnas suggestion is to pre-run determine the scheduling order as much as is possible (see also section 9.4.5).

8.4 Analysis and discussion
In this section we present the analysis and discussion of these different platforms approaches, separately for each one; and accordingly we deduce the list of features that we desire to adopt for Arts'Codes.

8.4.1 Simple Approach
The simple approach propose to handle each event when it arrives, and queuing all other which came in middle of the handling. In Real-time applications, priorities should be are attached to events, similar to the run-to-completion ROOM method.
This approach, although its simplicity:

1. Aspires to non-determinism: its reading rate of the inputs (influenced by the reaction duration) may influence to the final output results.
2. This reading rate may also produce a convoy-effect, promoting unpredictable response time of critical events.

8.4.2 Multi-threading
Multi-threading platforms besides of parallel prioritized-thread execution, applies also interrupts handlers, which are also prioritized. The programming style is certainly familiar and therefore comfortable, but:

1. Trends to non-determinism: originated from the scheduling algorithm; the order of the thread scheduling not only depends on the input set, but also on the whole operating system status that differs in different running.
2. In consequence to its non-determinism property, is much more difficult to analyze and predict” [62].
3. Pays context-switch latency overhead fees.
4. Lacks from the race-condition problem due to their concurrency and data sharing.
5. Threads are preempted no matter in which phase of execution they are.

8.4.3 Pre-run scheduling
The pre-run-time scheduling approach it adopts a rigid degree of determinism, but "requires that the major characteristics of the processes in the system be known in advance" [62]. This approach fits to predetermined applications (applications in which we want real-time exact prevision). But in fact it fits to specific applications types, like periodic processes, where their release and deadline time in addition to precedence relations and exclusions are supplied. But "there are situations where a 'pure' pre-run-time scheduling approach is not practical" [62].
8.4.4 Synchronous

Really, the synchronous approach avoids event nesting, providing also a stable environment image which is the input for the whole application modules. But it has many disadvantages.

Causality

The main problem of the synchronous approach is the consequences of the principle of causality. "The principle of causality requires that there is a clear causal ordering among the transitions taken in a step, such that no transition \( t \) relies for its activation on events generated by transitions appearing later than \( t \) in the causal ordering" [55].

This principle in fact defines a simultaneous incoming of inputs and emanating of outputs in the same step of time, which produces a deep confusion on cause and effect relation: an effect is really an effect or it is also the cause?

When taking out the notion of the time, cause and effect concepts trends to be unclear, and then amends like the constructive approach have to be imposed. Furthermore the constructive amend claims time overhead, and also not each application is acceptable, e.g. "The causality chain may be cyclic: its instantaneous execution is obviously unacceptable. In such a case, the syncChart is rejected." [49]

Inter-level transitions

Inter-level transitions are a powerful tool but hard to implement; SyncCharts for instance refuses this feature. In Statemate it exists but it may provoke possible illegal/conflicted transitions, for example two concurrent automata may reach two different OR-states (two states in the same automaton).

Delayed changes

In this method changes that occur in a step can be sensed only after the step is completed. This restriction trends to programmer's misleading, where e.g. a variable is changed and then tested in the same step. In this case the value of the variable is correct but the programmer may expect to another outcome (because the new value is sensed only in the next step).

Asynchronous model

The Asynchronous model is more reactive when the system is idle, but if the previous super-step was too long, the time-reaction of the following external
events will be too long accordingly, which put in doubt Harel's claim that the asynchronous model holds "the perfect synchrony hypothesis". In the synchronous model the reaction is immediately in resolution of the basic time unit, even the event was not completely handled. Also the asynchronous model may lose a time based condition if its range passed during the super-step duration. Another disadvantage is that the Asynchronous model may produce an infinitive loop, because it "repeatedly execute one step until the system is in a stable state" [45].

8.4.5 Adopted features
According to this analysis, the following existing features were adopted and implemented in Arts'Codes approach:

- Partial synchrony hypothesis (see section 9.4.2)
- Partial SSM execution algorithm (see section 9.4.3)
- Partial pre-run scheduling (see section 9.4.5)
- Partial multi-threading platform (see section 9.4.2.4)
- Logical thread representation (see section 9.4.4)
9.1 Abstract
Our execution platform is based on hierarchic parallel automata, which reacts in a double discrete time domain. The various automata (one automaton for each component) are concurrent but their reactions are (rigid) ordered by priorities and executed sequentially in separate granules parts of time. Two separate time domain exist: one for the *external* environment which is portioned in low resolution cycles called *step*, and the other is the *internal* system clock (which is hardware dependent) with higher resolution.

The system runs on a tri-processor platform model:

1. **Environment Kernel** (EK): responsible of the environment interactions.
2. **Reaction Kernel** (RK): responsible of the fast reaction computation.
3. **Synchrony Co-Processor** (SC): which is attached to synchronization tasks, and responsible of the background processing (heavy reactions) and of the host communication.

Each processor has a different scheduler algorithm.

The input system set is composed of all inputs arrived during the recent step, and their output will appear immediately at the end of the current one, achieving a reaction time of two external steps, in the worst case.

9.2 Time domain
The nature of a reactive system is to put the system in a slave position reacting to inputs dictated by the environment, the master. The inputs may come in different order and timing; and the reactive system, the slave, must "dance" according to the environment "pace". As described by Berry [35] an asymmetric game with alternates moves first by the environment and secondly by the system.

9.2.1 Complexity
The nature of a slave-reacting system provokes three specific problems:
1. **Reaction nesting**: A reaction to an event can be nested by a new event which preempts the current reaction. This phenomenon may be repeated many times causing a set of competed reactions and even recursion. This nature of input behavior makes a foggy picture of possible system scenarios and makes difficult the system design.

2. **Unstable environment image**: The environment image is composed by the different inputs. The inputs are dynamic values which changes their values with the time advancing. Since the dynamic on the input values, two reactions will see different environment image and then will react in a different manner causing an unpredictable system behavior.

3. **Skipping of time conditions**: Some time conditions may be tested late, loosing their time range.

### 9.2.2 The time model

In order to solve these problems, Berry introduced the Synchrony Hypothesis. The Synchrony Hypothesis makes order in the input sequences and really gives a robust solution to the slave-reacting system. In addition to its talents, the Synchrony Hypothesis introduces the causality's paradox, a consequence of the time abstraction. The cause and the effect take place in the same instant, producing the undesirable philosophy question: *may an effect produce the cause?*

This approach is abstract and theoretical, but gives its solution although the causality paradox invasion.

Our approach takes the *synchrony hypothesis* main idea, but without making time abstraction.

The time domain is glanced from two different views or clocks:

1. One view for the *external* environment which is supposed that their changes occur only between rigid equal cycles, named *steps*.

2. The second is the *internal execution platform* time advance, in *instant* units, which is finest and it is hardware dependent.

The *external* environment time is sliced in equal cycles, defining a discrete time domain made of equal cycles (*steps*). The duration of the cycle is dictated by the nature of the input (the frequency of input changes) and the reaction time requirements (maximum response time).
The internal clock is also discrete, and its instant duration is dictated by the hardware oscillator (the processor speed). Obviously, the steps (external cycles) are longer than the internal cycles (instants).

Time conditions are based on the external cycles. The time measure is made by special objects that are platform independent (referenced in regular time units: seconds, milliseconds, etc), and are transparently adapted to the internal time domain.

The internal time domain is used for the priority sorting of the reactions and for the change of internal variables' value, which are executed sequentially in a specific order, one after the other in consecutives internal instants.

All environmental inputs arrived during the recent step are referenced with arrival-time of the current step.

All environmental inputs with arrival-time of the current step are reacted during the current step, and their environment outputs will be emitted at the beginning of the next step. All the active components have the opportunity to react during each step, to a stable environment image.

In the worst case, the suitable output for an environmental input, coming immediately after the beginning of a step; will be emitted to the environment with a response time of two steps (two external cycles).

Briefly we can say that environmental inputs including time, are measured according with the external time, in steps units. Reactions and internal variables behave according to the internal time, in instant units.

In both time domains, changes occurred during a cycle are sensed only at the beginning of the next cycle.

9.2.3 Rationality

This doubles the discrete time domain approach, and solves the three mentioned (see section 9.2.1) natural complexities of reactive systems:

1. Reaction nesting
2. Unstable environment image
3. Skipping of time conditions

Furthermore, this solution avoids the causality paradox.

Reaction nesting (1\textsuperscript{st} complexity solution) is avoided because the set of the referenced inputs is stable and constant during the step, what provokes to a stable environment image (2\textsuperscript{nd} complexity solution). This stable image gives the
opportunity to all components to react to a common environment image during the whole step.

It is true that the internal image is changing during the step because of the components’ reactions, and consequently it doesn’t give the opportunity to each component to react to the same internal image; nevertheless the order of the reactions were dictated by the programmer, and this is the requested behavior imposed by him.

The environment image is imposed externally, and to solve the input sequence complexity we freeze the image during the step, but the internal component outputs which are inputs to others are dominated by the programmer. Moreover because the reactions are serialized during the step, no new input can occur during the current reaction; there is no real concurrency during a step.

Finally, because the external discrete time is the basis of the time conditions, and during the step the external time is stable, no time conditions can be skipped (3rd complexity solution).

The environmental outputs are not immediately sent during the step, in order to give the opportunity to all the components to compute the output set. When all components have completed the output set, this set is flowed out at the beginning of the next step.

### 9.2.4 Restrictions

According to this model, all reactions have to finish their execution before the beginning of the next step; otherwise timing values will be unreliable, and response time unpredictable. This restriction limits the system to very short reactions, or to too long steps.

Both solutions are not desirable:

- Short reactions limit the system to control applications where the outputs are short commands (for instance reading/writing to external-devices communication ports).
- Too long steps induces a too long response time.

### 9.2.5 Heavy Reactions

In general fast reactions are done by the Reactive Kernel (RK). In order to achieve fast reaction time (for I/O for instance) despite long time of some other reactions (computations for instance), the Reactive Kernel (RK) is assisted by the Synchrony Co-Processor (SC).
The SC runs in its platform heavy-reactions, which are reactions with long time duration (more than a step). Each reaction that has a loop is defined as a heavy-reaction.

When a heavy-transition (a transition which reacts with a heavy-reaction) is detected and the heavy-reaction has to be executed, the heavy-reaction inputs are passed to the SC in order to run the heavy-reaction. The input passing is made between the steps.

When the heavy-reaction execution finishes in the SC, the outputs are passed to the RK immediately after the end of the current step.

During the heavy-reaction execution the heavy-transition in the RK is frozen, the source automaton-state is left, but the destination state is not reached. This is implemented by adding a pseudo state, with no transition or inner elements. During the execution, the automaton stays at this pseudo-state, which means that not all the automaton is frozen, but higher state levels continue reacting. In case of higher levels abortion, the heavy-reaction will stop at the end of the current step.

When the heavy-reaction finishes, its outputs are passed to the RK at the beginning of the next immediate step, the heavy-transition leaves the pseudo-state, and the automaton reaches the target state.

According to this solution:

§ The step can be configured to be short.

§ Long duration reactions can be executed in parallel (by the SC co-processor).

§ The system continues reacting without waiting for the heavy-reaction execution completion.

§ The system response time it is not damaged.

### 9.3 Arts execution platform

The Arts execution platform, as mentioned in the abstract section (section 9.1) is a based on a tri-processor platform model. In this section each processor (kernel) is explained separately by describing its internal algorithms. A detailed kernels' cooperation description can be found in section 11.3.

#### 9.3.1 Reactive Kernel (RK)

The Reactive Kernel runs a Parallel Automata model, which is a translation of the various components' automata. This kernel receives fresh inputs
from the Environment Kernel, at the beginning of each step. During the step it reacts to these input; and emits outputs, at the end of the step, these outputs are sent back to the Environment Kernel.
The Parallel Automata algorithm is supported by the enumerated principles in section 9.4. The detailed algorithm description is brought in section 11. The Parallel Automata is presented here by exhibiting its concurrency robustness and determinism properties; and showing the global algorithm of execution.

9.3.1.1 Concurrency robustness and determinism
The components run in parallel, but the reactions are consecutives in each step. At each step all the components receives the opportunity to react. The components reactions are executed according to their priorities each one after the other. The time is granulated permitting to each reaction to finish their execution without any disturb, which means that all reactions are atomic. This approach enables component concurrency, avoiding simultaneous dangerous operations. No racing conditions and no parallel reactions are allowed. There is no any problem of data coherency because in fact there is no parallel treatment of data. The fixed order of reactions which is imposed by their a-priori explicit priorities, enables a consistent and robust system with fully deterministic features. Every input series will output the same output set.
This approach facilitates also the execution with no need of scheduling handling, because the order of the reactions execution is predefined at the design stage.

9.3.1.2 Execution by priority
All transitions have a priority attached, in order to be sorted and inserted in its suitable place, when reacting in a step. The priorities values may be positive or negative. A positive value defines a transition that will be tested only after the inner state elements (sub-components and sub-states) executed. A negative priority value determines a transition that will be tested before the inner state elements execution.
Sub-components have also priorities, which fixes their execution order. The sub-components have a fixed priority value which was defined in the static diagram, and according to their value they are executed. Their priority has also positive and negative values, and they are executed after or before the sub-states correspondently.
A transition that its trigger is a sub-component exit-gate, has no priority, and is tested immediately after the sub-component execution.
The algorithm of execution can be summarized as follows (see also section 11.2.4):

**Do the State (S: State)**

1. Test all negative transitions according to their absolute priorities values, in an ascending order.
2. Execute all negative sub-components according to their absolute priorities values in an ascending order.
3. **Do the State (current sub-state)**
4. Execute all positive sub-components according to their priorities values in an ascending order.
5. Test all positive transitions according to their absolute priorities values, in an ascending order.
6. Each of the previous statement may abort and here the destination state is "returned"

If a transition is tested and executed, aborting the current state, the rest of the transitions, sub-components and sub-states will be not executed, and the destination state will start its execution in the same step.

A loop-transition (a transition where source-state = target-state) is similar to a static-transition in Statecharts and it doesn’t leave the state.

The execution of the component will finish when the component arrives a destination state for the second time. This approach avoids infinite loops, but gives the component the opportunity to advance its reaction (it is a hybrid model between the Synchronous and the Asynchronous time's model in Statemate, as explained in section 8.2.10).

Here is the component execution algorithm:

**Do the Component (C: Component)**

```plaintext
// S is the current component state
While first time in S do
  S = Do the State(S)
```

### 9.3.2 Environment Kernel (EK)

The Environment Kernel is responsible to update continuously the current environment image. Between the steps, the current environment image is copied
to the Reactive Kernel (this image is the EK output and the RK input), who will
react to the fresh image by computing its suitable outputs.
The Reactive Kernel doesn’t emanate out directly its outputs by itself, it only
exchanges its outputs with the Environment Kernel environment image input.
The Reactive Kernel flushes out the outputs at the beginning of the next step or
during it.
In summary the Environment Kernel task is:
   1. To acquire the environment image, which will be the next step Reactive
      Kernel input
   2. To emanate the outputs, which were produced by the Reactive Kernel in
      the previous step.
The EK is composed by an array of rudimental components, called virtual devices,
which have to execute several times in a step. The number of executions in a
step depends on the nature of the environment element, which this virtual-device
is responsible for. For example if the environment element is a button, the
virtual-device will be executed one time. For an Analog to Digital Converter for
example, the times of execution per step should be suitable to the digitizing rate.
The virtual-device components are executed by their constant priorities order,
determined by the programmer.
The scheduler algorithm for the EK generates a fixed pre-scheduled order, in
order to satisfy all virtual-device components requirements.
The virtual-device components are similar to the Reactive Kernel components
with the following restrictions:
   $ No components hierarchy
   $ No states hierarchy
   $ No Gates
   $ No Guard
Briefly, they can be defined as simple automaton with input, outputs and local
variables. This rudimentary structure enables fast execution and abilities to
acquire frequent environment changes.
The virtual-device components' input and outputs are copied to/from the
reactive-kernel-root component.

**9.3.3 The Synchrony Co-Processor (SC)**
The Synchrony Co-Processor is responsible on:
   1. Step pace making
   2. Heavy-Reactions execution
   3. Host communication
9.3.3.1 Step pace making

The SC has a hardware timer which measures the requested step duration. When the timer times out, the following is executed:

1. It tests if the RK inStepFlag is reset, which tells to the SC that the RK finishes its execution for the recent step. If the RK is still executing, the SC sets the stepOverflowFlag and waits until the RK finishes (inStepFlag reset). The stepOverflowFlag can be tested in RK as a pre-defined assertion. Also the various clocks in RK will advance according to the elapsed (more than one step) time.

2. It sets the EK pauseFlag, which tells to the EK to suspend its execution immediately after it finishes the current virtual-device component execution. The EK will then suspend and set the acknowledgeFlag, which will release the SC to execute the next statement.

3. The SC then will:
   a. Copy all RK reactive-kernel-root outputs to the corresponding EK virtual-devices.
   b. Copy all EK virtual-devices to the corresponding RK reactive-kernel-root.
   c. Heavy-reactions inputs are passed from the RK to the SC
   d. Heavy-reactions outputs are passed from the SC to the RK.

4. Now the SC synchronization task finishes and a new step can begin, for this the following is made:
   a. The EK pauseFlag is reset, and then the EK resumes its loop.
   b. The RK inStepFlag is set, which tells to the RK to begin the new step execution.
   c. The SC continues with the Heavy-Reactions execution and Host-communication.

The timer handling is preferred to be handled as a timer interrupt handler, but it can be executed also as a thread in the multithreading platform with high-priority.

9.3.3.2 Heavy-Reactions execution

Runs locally the heavy-reactions triggered by the Reactive Kernel, in a multithreading background mode.

9.3.3.3 Host communication
The embedded controller is all the time in connection with the host main computer, during the development and debugging period, in order to provide the following tools:

- Applications downloading
- Debugging

The communication tools run as a thread, concurrently with the multithreading heavy-reactions.

### 9.4 Arts Principles

Here we present the principles of the execution of Arts'Codes using Parallel Automata. This Parallel Automata model is applied only in the Reactive Kernel domain.

#### 9.4.1 Abstract

The approach of the Arts model is to skip undesirable heavy consumption of memory by tables handling, while holding in the memory only the necessary information.

The hierarchy states and component structure in addition to the transitions connections; are directly translated to code by rigid translations rules, creating a regular code program.

The model integrates several improved existing approaches, in addition to innovations yield by this research, in order to achieve robustness and determinism, creating also an economic (in terms of memory) and fast code.

**The eight principles of the model** are based on innovations or modifications/improvements of current approaches, they are:

1. Environmental Synchrony Hypothesis
2. Full isolation between environment-interaction and program-logics
3. Synchronous and Multi-threading platforms interleaving
4. Granulation
5. Concurrent and Sequential Run interleaving
6. Light reactive-cells
7. Leaf states flat automaton
8. Logical threads

**1st principle: Environmental Synchrony Hypothesis**
The synchrony hypothesis is implemented only for the environmental input/outputs. Time is also handled as an environment input (see section 9.4.2.3).

2\textsuperscript{nd} principle: Full isolation between environment-interaction and program-logics
During the steps, the interaction with the environment is handled separately and in parallel to the reaction logics (computation of outputs), in two different hardware-platforms; without sharing any resources (CPU, memory, time, etc). They exchange their data only between the steps (see section 9.3).

3\textsuperscript{rd} principle: Synchronous and Multi-threading platforms interleaving
The application is split, so that it can be run on two different cooperating platforms: one for the Synchronous execution, and one for the multi-threading (see section 9.4.2.4).

4\textsuperscript{th} principle: Granulation
All reactions are atomic actions, and should be defined as short as possible; permitting frequently free holes for secure "context-switch" (see section 9.4.5).

5\textsuperscript{th} principle: Concurrent and Sequential Run interleaving
At each step, the concurrency of components is achieved by a fixed order of atomic reaction (which belong to various components'); assuring in that way concurrency (all components run their reactions in each step) and determinism (the reactions are performed in a fixed order) (see section 9.4.5).

6\textsuperscript{th} principle: Light reactive-cells
The automaton is represented by separated cells, without conserving the automaton structure in a specific data-structure (see section 9.4.3).

7\textsuperscript{th} principle: Leaf states flat automaton
Introducing a new technique to handle a hierarchic automaton with inter-level transitions, as a simple flat one-level automaton (see section 9.4.3.4).

8\textsuperscript{th} principle: Logical threads
Each component has a method, known by the system, used for the concurrent component activation. This logical thread representation behaves according the Concurrent and Sequential Run interleaving principle (see section 9.4.4).
9.4.2 The Arts' Codes Synchrony Hypothesis version

Berry's Synchrony Hypothesis is the basis of the Arts platform, but was adopted partially. In the following section we remind the Synchrony Hypothesis concept, its problems, and the reasons why Arts adopted a partial Synchrony Hypothesis version (see also section 9.2).

9.4.2.1 Reminding the original Synchrony Hypothesis

The famous Berry's Synchrony hypothesis [12] comes to simplify the design of reactive systems avoiding:

§ Reaction nesting.
§ Unstable Environment image.

"All the above problems disappear when one adopts the Synchrony hypothesis: each reaction is assumed to be instantaneous – and therefore atomic in any possible sense" [34].

Time is out of programming scope: "time is defined externally to programs by the flow of inputs, and that program internal bookkeeping is done in zero-delay with respect to all external time units" [36].

The simplicity of Berry's theory pop ups when concurrency is adopted, giving a constant and uniform environment image to all the system elements:

"At reaction $n$, processes $p$ an $q$ see the same inputs on the signals they share, their common outputs are merged, and each of $p$ and $q$ sees the signals emitted by the other process. A parallel statement performs its own bookkeeping in zero-delay, and all control handling operators should have the same property" [36].

The problem is that hardware platform are not enough fast to apply the Synchrony Hypothesis, in fact all "Inputs events occurring during a reaction are queued for the next reaction, which makes the reaction atomic and deterministic" [38].

9.4.2.2 Complexity

The Synchrony Hypothesis makes order in the input sequences and really gives a robust solution for real-time applications. It gives a frozen environment image, which alleviates the programmer efforts, without need to think how to defense its code from possible "undesirable events attack".

In addition to its talents, the Synchrony Hypothesis introduces the causalities paradox, due the time abstraction. The cause and the effect takes place in the same instant, producing the undesirable philosophy question: may the effect produce the cause?
This problem emerges from the time abstraction; the input and output are instantaneous which introduces the causality paradox.

9.4.2.3 The Arts version
Our approach adopts the Synchrony Hypothesis main idea, but freezing only the environment from input or outputs flow, and even the environment time, during a step.
The Arts execution platform itself doesn’t apply the Synchrony Hypothesis approach for its internal code, it only applies it concerning the environment.
All inputs arrived during the previous \textit{step} (in the Environment platform) are reacted during the current \textit{step} (in the Reactive Kernel), and their outputs are emitted at the beginning of the next \textit{step} (in the Environment platform).
This approach gives a common environment view to all components, where each component can react to it (in the Reactive Kernel), and confirm a unified set of outputs without the environment disturbance.
It improves also the timeliness, no time condition will be skipped because the time is advancing only between the \textit{steps}.
All this advantages are wined from the Synchrony Hypothesis without the causality paradox disadvantage.

9.4.2.4 Synchronous and Multi-threading platforms interleaving
According to this model, all reactions have to be finished before the beginning of the next \textit{step}. This restriction limits the system to very short reactions, or needs to have too long steps (taking into account long reactions). In order to be able to run long reactions, the Reactive Kernel is assisted by the Synchrony Co-Processor, which can run heavy-reactions in its multi-threading platform. When a heavy-transition is tested and the heavy-reaction has to be executed, the heavy-reaction inputs are passed to the SC in order to run the heavy-reaction. The input passing is made between the \textit{steps}. When the execution finishes, the outputs are passed immediately after the end of the current \textit{step}.
In this way the \textit{step} can be configured to be short in the RK, long duration reactions can be executed in parallel in the SC, the system can continue reacting without waiting to the end of the heavy-reaction execution, forming in such a way a hybrid Synchronous and Multithreading platform.

9.4.3 The Arts reactive-cell version
Arts’Codes approach adopted the Andre’s reactive-cell main idea, which was improved as it will be described in this section.

**9.4.3.1 The Andre’s original reactive-cell**

Andre defines a reactive-cell (see section 8.2.9) as a "state with its outgoing transitions. A reactive cell can be active (alive) or idle (doing nothing at all)" [49]. The reactive-cell may be built in a hierarchy structure, where higher levels can influence to lowers ones. Influential transitions are treated by Andre in the following manner:

"An active reactive cell is permanently testing the triggers associated with its transitions. As soon as a transition can be taken, the reactive cell is deactivated and the reactive cell target of the transition is activated. A reaction now appears as a propagation of activations/deactivations among a collection of cells" [49].

The reactive-cell approach focuses the state as the principle actor, without referencing to the global image of the whole automaton. The observer can group all reactive-cell rules and sketching their interconnections, discovering in such a way the whole automaton behavior.

This approach is not new; it is the Divine approach in the universe creation. The Creator of the universe didn’t reveal us the global natural rules, He only revealed us the particular rules of the elements. The human observer can then investigate the particular laws of nature; this is the work of the scientists. Comparing each rule to the other, and discovering a unified global and coherent system of rules in the nature, the scientist will not have any doubt about the Divine presence who built this unified global and coherent system of rules.

This approach guided us also in the Arts’Codes model: the execution is made in the Arts platform by particular states (reactive-cells, similar to natural particular elements), which have logical connections in a mutual automaton (similar to the whole global natural rules of the creation). The logical global connections are "discovered" in the Codes design diagrams.

**9.4.3.2 Realization of the original approach**

The approach is realized by simple writing particular methods in a class component for each state (see also section 11.2.1). These methods have a series of conditions that represents the transitions conditions. The conditions are pre-sorted (at the compilation phase) according to the transitions priorities. At the execution phase, if the condition is true, the method finishes. In such a case the active sub-components hosted by the state (or hosted by lower states levels) are paused, and the method returns.
The inner automaton current method state is called according to its priority, in addition to the several sub-components *Run* methods (in the component main loop, see section 11.3.2). Only one state can be run at a time, because no concurrent states are permitted (concurrency is realized by activating two separate sub-components in the same state).

This approach is simple when no inter-level transitions are permitted like in Andre’s SyncCharts, but in Arts‘Codes the transitions realization is more complicated due to inter-level transitions option. In order to realize the inter-level transitions, the Andre’s reactive-cell was improved in this work, as described in the next sections.

### 9.4.3.3 Arts‘Codes Inter-level transitions complexity

**Intra-level transitions:**

When restricted connections are handled between states, without permitting transitions passing from lower to higher levels and backwards, each level is handled as a separate automaton, being activated or deactivated by higher levels.

In this case an automaton which resides in a lower level can’t influence higher levels; because transitions cannot exit its host state.

Then the implementation is simple: each method that represents a state, executes a series of conditions, calling also the inner automaton current state-method, and all sub-components *Run* method. The state-method can never be interrupted by a high or low level state. It can be obviously be aborted by a higher transition state level, but not in the middle of the method execution.

The component has only to hold one current state variable for each simple automaton, which is handled independently from other automaton influence. The hierarchic automaton is then handled as a rigid tree of various automata.

**Inter-level transitions:**

But when the bound levels are open, and a transition can pass to lower and even to higher state levels, a more sophisticated algorithm has to be implemented. Then, the logical structure can not be seen as a hierarchical automaton tree, because a low level can return backwards to higher levels. The state-method continuation is influenced by the called low level state-method, which can change the higher levels automaton current state, and even which may influence all its ancestors. This can occur for instance when the ancestors are in the middle of their execution waiting for the lower level methods return.
Now when opening the levels bounds, the risk is that this rigid automata tree structure is turned to be a plate of spaghetti. Not in vain Andre prohibited inter-level transitions.

9.4.3.4 The Arts'Codes improved reactive-cell

Our Manager is described as a hierarchic automaton, i.e. a set of states ordered in a tree structure (see section 5.2.1).
This automaton has at each instant a list of current (active) states, one state for each hierarchic state-level, because no state-concurrency (AND-states) is allowed.
The Arts'Codes improved model transforms a hierarchic automaton to a flat one-level state-diagram, where only the lowest level of states is considered.

Definitions

Let us define some structural tools which will help us building this "flattened hierarchic automaton"

- **Constant Current States List (CCSL):** is the list of current states (one state for each state-level).
- **Current Leaf State (CLF):** is the lowest state-level in CCSL.
- **Root Level State (CRT):** is the highest state-level in CCSL.
- **CCSL(CRT):** is the CCSL suitable to a specific CRT. By definition more than one CCSL exist for each CRT.
- **CCSL(CLF):** is the CCSL suitable to aspecific CLF. By definition only one CCSL exist for each CLF.
- **CCSL_i:** is the the current state in state-level; where $CCSL_0 = CRT$

The "flattened automaton"

Two algorithms were implemented:

1. The improved cell transforms the hierarchic automaton to a flat one-level state-diagram, where only the CLFs are considered. This simplification turns the complex multi-level automaton to a simple flat one level automaton, which transits from one CLF to another CLF; and according to the CLF, the CCSL can be reconstructed ($CCSL(CLF)$).
2. The execution of the CLF is made by calling the CRT-cell, where $CRT = (CCSL(CLF))_0$. The CRT-cell tests its conditions, calls it sub-components' Run method and invokes the $CCSL_{1,cell}$ (the next current state-level). The $CCSL_{1,cell}$ may return a true value which tells the CRT that it is not anymore active (an inter-level transition.
was transited); this will provoke the cell to be exited. This procedure is invoked recursively until it reaches the CLF. According to this improved reactive-cell, the current state is determined by the CLF which invokes the CRT (\(CRT = (CCSL(CLF))_o\)). The Run method (section 11.3.2) executes one CRT-cell after the other, in a simple loop.

9.4.4 The Arts "Java's Thread::Run" version

The well-known Thread Java class enables multithreading execution by generalizing this class. The piece of code that has to be run is described in the virtual method Thread::Run. The Java Virtual Machine will run this Run method as a separated thread in parallel to all existing ones. This idea, to define a class where one of its methods is the code that has to be run in parallel, and being this running function known by the Execution Platform as a concurrent code, was adopted by the Arts platform. Java's approach was improved in our model by giving a constant code to the Run method, with no need to define it as an abstract method.

Since the component behavior is described as an automaton, which is event-driven and is not all the time executing, passing from state to state, entering or exiting from/to gates; and being run in a synchronous platform giving the opportunity to be run only partially at each step; a fixed piece of code has been written which is suitable for all components realization. A detailed explanation is brought in section 11.3.2.

9.4.5 The Arts improved "Parnas's pre-run-time scheduling"

In order to achieve determinism, big efforts were made to fit a suitable scheduling algorithm to the Reactive Kernel domain. This algorithm adopts the Parnas pre-scheduling approach [62]. Reactions are then granulated, components stay concurrent, but reactions are pre-scheduled (at the compilation phase) and run sequentially in each step of time.

9.4.5.1 On-line versus off-line scheduling

Parnas is one of the well-known researchers that alerted the real-time programmers population to prefer pre-run-time scheduling when it is possible. The main Parnas suggestion is to pre-determine the scheduling order as much as is possible (see section 8.3).

9.4.5.2 Reactions Granulation
The pre-run-time scheduling approach, "requires that the major characteristics of the processes in the system be known in advance" [62]. This approach fits specific applications types, like periodic processes, where their release and deadline time in addition to precedence relations and exclusions are supplied. But "there are situations where a 'pure' pre-run-time scheduling approach is not practical" [62]. The combination of a synchronous platform, which runs a parallel automata model, together with Granulation approach enables pre-run-time scheduling for each typical real-time application.

Our research conclusion is that as much reaction as possible can be shorted or "granulated", the system will be more deterministic (which is not the case in multi-threading platforms where reactivity is achieved by decreasing determinism). Long reactions reduce also timeliness, and provoke the system to be less reactive. Short atomic reactions enable robustness, the reaction is not interrupted; it gives more flexibility for priority reactions sorting, and supplies also free holes for the system to be more reactive.

Short "granulated" reactions can be then ordered and executed in a step, in a fixed series pre-run ordered according to their priorities.

The Granulation concept directs the programmer to cut himself his code into safe short partitions (granules) which can’t be interrupted. These granules, which are in fact the automaton reactions, will determine the dynamic time-slice durations (each reaction has different time-slice duration). The CPU time is then distributed to the several reactions in a fixed hierarchic order. The reactions may belong to several components, but in fact they are sorted all together in one global fixed list.

In Arts'Codes all transitions, sub-states and sub-components priorities are determined in the design phase.

By the Granulation approach, pre-run scheduling can be achieved, together with atomic safe reactions, almost for each real-time application.

**9.4.5.3 Concurrent and Sequential Run interleaving**

In the Arts solution, while adopting the granulation approach for the reactions, the component concurrency principle is also applied. The several components are reacting in parallel (each one receiving many safe time-slices in a step), but in fact the reactions run sequentially (concurrent and sequential run interleaves). The sequential execution of reactions avoids race conditions, and other lacks which can be provoked by a parallel execution of reactions.

Finally we can say that Arts platform negates usual multithreading algorithms, it absolutely repulses over-step reaction duration (which is not the case in
SyncCharts); *heavy-reactions* were excluded from the synchronous platform, and included in the Synchrony Co-Processor domain. In the case of an activation or termination of a *heavy-reaction*, the Reactive Kernel handles them as two separate short reactions.

Despite of negating multi-threading model, in fact the same result is achieved; a thread is programmed as a hierarchic automaton running in parallel to other ones. Furthermore, the *Concurrent and Sequential Run interleaving* avoids context-switch overhead, and we win determinism by defining a fixed pre-scheduled order of atomic reactions.

### 9.4.6 Other Arts solutions

This section enumerates other techniques used for the execution platform implementation.

#### 9.4.6.1 Assertions and Guard

In Arts the normal and exception behavior are designed separately in each component. This separation is lightly implemented by simply defining one automaton, which combines both behaviors. In spite of that both behaviors are combined, the connection between them is made via *gates*.

This approach permits the suspension of the component from its normal behavior when it is not working well, but in the same time other components (which are not its descendent) are working in their normal behavior.

#### 9.4.6.2 Between-Steps Commands

*Arts* performs specific actions that are done only between the time *steps* such as: incrementing the timer, receiving signals which have one *step* of lifetime, etc.

This approach avoids condition’s test missing, because the variable on which the condition is based is changed out of the *step*.

#### 9.4.6.3 Timers

*Arts* has a built-in class for timers implementation, which is adapted to the system *step*.

Timers advance only between the *steps* (in the DoBetweenSteps method, see previous section), assuring in this way accurately response to each time-condition, without loosing any of them.

#### 9.4.6.4 Gates
Gates (see section 5.1.2) are well-defined links between components' Managers and their ancestors/descendants. They implement secure interfaces between components, including implicit predefined gates. By this approach mode of initialization are defined by entry-gates and exit modes by exit-gates.

This approach improves clearness in the debugging stage, because the components behavior links are clear and rigid.

### 9.4.6.5 Pause and Resume

Pause and resume are predefined gates, which acts like the history pseudo-state in Statechart. The difference is that they work only for the whole component, and there is no any history pseudo-state for automaton states. Actions may be executed before entering/exiting these gates.

Another difference is that in Arts the default initial entry-gate (similar to the default initial state in Statechart) works only for the first time. When returning to the component after an exit through the pause exit-gate, the component is entered through the resume entry-gate if no other explicit entry-gate was defined.

This approach improves the simplicity of the execution platform and the clearness because this feature is used based on a component unit; which have a well-define behavior and can be saved and returned to their previous state.

### 9.5 Arts hardware platform

This section presents a prototype of hardware platform fitted to the Arts execution platform. Thanks are due to B. Avramovsky and A. Milshtein, engineering students at the JCT electronic-department, who have participated in the implementation of this tri-processor hardware prototype.

#### 9.5.1 Abstract

The prototype platform is based on three ADUC812 evaluation boards, each one based on the Intel 8051 micro-processor (see Fig. 35).

Each board has its own memory which is accessed only by the host processor.

Each board runs one of the three Arts's kernels: EK, RK and SC.
9.5.2 Memory switcher

A memory switcher was built with a capacity of 2 KB. The Reactive Kernel and the Environment Kernel have each one a portion of memory where the inputs and outputs are stored. Once before each step (i.e. between the steps) the memories are switched in order to interchange the inputs and outputs. This approach is the faster way to copy memory between the kernels, without giving access of one of them to any other memory which is not owned by it.

9.5.3 Synchronization

The synchronization and the memory-switch control are made by the Synchrony Co-processor, which notifies and synchronizes the memory interchange in a secure slice of time between the steps. A step was defined as 200 milliseconds. The Synchrony Co-processor also alerts in case of step overflow, when the Reactive Kernel is not ready to memory interchange at the requested time. The EK and the RK are released of timing control; they are only concentrated on their own work, the timing responsibility is held by the SC.
9.5.4 The implemented application
I/O devices were added for analog to digital and digital to analog converters, in addition to digital I/O, in order to build a real application.
The application acquires audio sound (8 kHz/8 bits resolution) at the EK, and processes it in the RK, which outputs the data back to the EK (the buffer size is 160 bytes).
The output is an on-line processed sound.
The *heavy-reactions* were not applied because in this prototype the memory sharing between the Reactive Kernel and the Synchrony Co-processor was not implemented.

9.5.5 Conclusions
The system was built according to the principle that memories must be physically isolated in order to assure that the EK and The RK doesn't disturb each other.
Now, we are starting an attempt to build a tri-processor platform with shared memory, based on a FPGA circuit, the architecture and behavior being built in VHDL language.
In this section the execution platform is evaluated, according to the different features which have to form a real-time system.

10.1 Robustness
Arts fulfills the principles:
- Environmental Synchrony Hypothesis
- Full isolation between environment-interaction and program-logics
- Concurrent and Sequential Run interleaving

Environmental Synchrony Hypothesis
Reaction nesting is avoided because the set of the referenced inputs is stable and constant during the step, which induces a stable environment image. This stable image gives the opportunity to all components to react to a common environment image.
Even if the internal image is changing during the step (due to the RK reactions), nevertheless the order of the reactions were dictated by the programmer, and this is the requested behavior imposed by him.

Full isolation between environment-interaction and program-logics
The isolation between environment and the program-logic assures a true environmental image, because the environment interaction is not disturbed by the reactions; no events are missed.

Concurrent and Sequential Run interleaving in program-logics
This approach enables component concurrency, but avoids simultaneous dangerous operations. No racing conditions and no parallel reactions are allowed. There is no problem of data coherency because in fact there is no parallel treatment of data.

10.2 Correctness
The proposed addition of Goals to the components, using Temporal Logic equations, increases the correctness by assuring that the behavior are consistent
with these goals, so that assuring that the architecture is well-designed and coherent. (see section 5.2.3).
The compilation rules were defined to preserve the design structure without making essential transformations, assuring the homothetic approach. The homothetic approach increases the design/code suitableness, increasing in that way the correctness.

10.3 Determinism
According to the principle Concurrent and Sequential Run interleaving, the fixed order of reactions which is imposed by their explicit priorities, enables a consistent execution with fully deterministic features. Every input series will output the same output set.

10.4 Timeliness
According to the principles:
- Environmental Synchrony Hypothesis
- Synchronous and Multi-threading platforms interleaving
- Granulation
The timeliness is a built-in feature, being an implicit part of the system implementation. Because the external discrete time is the basis of the time conditions, and during the step the external time is stable; no time conditions can be skipped.

10.5 Responsiveness
According to the principles:
- Environmental Synchrony Hypothesis
- Synchronous and Multi-threading platforms interleaving
- Granulation
The maximum response-time in the worst case is two time steps.
The addition of the heavy-reactions solution provides that despite of long reactions:
- The step can be configured to be short.
- Long duration reactions can be executed in parallel in the SC processor.
The RK processor continues reacting without waiting for the heavy-reaction execution completion.

The system response time it is not damaged.

Also since to the Synchronous and Multi-threading combination, the system will not be "stuck" on infinite loops at any reaction. This is avoided by defining each reaction which contains a loop as a heavy-reaction, which means that will run in the multi-threading platform. Any heavy-reaction may "stick" only the heavy-transition, but not other components or other heavy-reactions.

10.6 Concurrency

According to the principles:

- Environmental Synchrony Hypothesis
- Full isolation between environment-interaction and program-logics
- Synchronous and Multi-threading platforms interleaving
- Granulation
- Concurrent and Sequential Run interleaving

By adopting the granulation approach, the concurrency principle is realized, several components are executed in parallel, but in fact their reactions run sequentially during the step (concurrent and sequential run interleaves).

This approach facilitates also the execution with no need of run-time scheduling handling, because the order of the reactions is predefined at the design stage. The tri-processor platform increases true concurrency, running the EK, SC and EK in different processors.

10.7 Compactness

According to principles:

- Light reactive-cells
- Leaf states flat automaton
- Logical threads

In this approach there is no any data structure which contains the states interconnections. Each state is represented by a simple method despite of the hierarchy and inter-level transitions complexity, so the executing code is light and rapid.

10.8 Scalability
Since there is no overhead Operating System that has to be installed and customized; by definition, the generated system fits to the exact space necessary for the execution, specifically for each application.
PART IV

From Design to Execution

In this part are presented, the rules for the compilation of the visual design into code, and the complete tri-platforms model of execution: detailed algorithms of execution, data structures, behavior representation etc.

Later the CHILD language is presented, which enables a textual translation of the Arts'Codes graphic diagrams, and which is a programming language suitable to describe the semantics of the design.

The last chapter describes rules of compilation in order to translate the CHILD language into Temporal Logic format for components' goals proving.

Chapters for this part:

11 Automatic bridge between Design and Execution
12 CHILD Language
13 Compiler Rules to provable Temporal Logic
11

Automatic bridge between Design and Execution

In this chapter, the data structures and rules that were defined for the compilation of Codes to Arts are presented. This compilation takes as input the Codes visual design (both static and dynamic diagrams), and translates them to a robust light code which will run according to the Arts execution specifications. The Arts compilation holds the homothetic approach of this method, without changing the model structure.

The robust light code is suitable to run in embedded platforms where the code is stand alone and can run by itself without superfluous overhead, and where memory size and CPU speed are critical.

The code is Object-Oriented, coded in C++ language, and uses different modes of polymorphism.

The chapter is divided into four sections:

1. Data structures: the construction where the static design is saved.
2. Behavior representation: the way that the dynamic design is represented in code.

11.1 Data structures

The following list of classes was defined in order to implement the representation of the design graphics items:

1. ComponentT: represents the component including all its items (static and dynamic).
2. VirtualDeviceT: represents the virtual-device including all its items (static and dynamic).
3. ClockT: offers timing services.
4. AssertionT: handles abnormal behavior.
5. HeavyReactionT: handles concurrent heavy-reactions executed in the Synchrony Co-Processor.
11.1.1 The ComponentT Class

This class represents the component including all its items. It has common implemented methods, and abstract methods which have to be implemented for each component, according their features in the application (see Fig. 36). Components that appear in the design are represented by inheriting the ComponentT class, creating a new subclass, and implementing abstract methods. In addition the new subclass has to add attributes and methods to include inputs, outputs, shared and local variables, states, etc.

The component class abstraction includes the static and dynamics item, which represents the structure and behavior.

The structure and the behavior are represented in a light manner, i.e. they have not heavy structures containing a formal description of the automata design; avoiding in such a way large automata tables handled by a slowing scanning overhead.

The Arts’Codes implementation approach is light, distributing the definitions in independent state-cells, connecting between them by just sharing variables, or by calling the suitable methods.

**Architectural Static implementation**

Static elements are: inputs/outputs, sub-components and interconnections.

Input and outputs are defined via the Set method which may have different method-type-signature (arguments fashion), and has to be added to the subclass which inherits the ComponentT class. This function is called by the host-component that aggregates an instance of this component definition. The Set method is used to interconnect the host with its sub-component.

Sub-components are defined by aggregating same/other components types creating new instances. These instances will be members of the ComponentT new subclass.

**Static interconnections implementation**

The sharing of input/outputs between components that are defined via the mentioned Set method, which is inspired from the electronic circuit metaphor.

In an electronic circuit the wire connects the data emitter to its receiver without time limitations, enabling on-line updating. This receiver may be connected to another wire and forwarded, all the nodes being permanently connected.

This approach is implemented in Arts by passing the input/output addresses through the Set method, by which they are saved in suitable class members.
**Dynamic Implementation** (brief presentation)

Dynamic elements are: Manager, Guard and Assertions. The Manager and the Guard includes: gates, states and transitions. The transition includes: the source-state, a condition, an action and the target-state.

---

### ComponentT

**Attributes:**

- `+ ComponentT()`
- `// + void Set(...) ; // for external connections set (i/o)`
- `# void Gates(); // specific gates definition`
- `# void Call(int id); // binds states -number to methods`
- `+ int Run(); // handles the component behavior`
- `+ void Paused(); // called by upper level to pause itself`
- `+ void SetEntryGate(int gate);`
- `+ void DoBetweenSteps(); // statement to do between cycles`
- `# ComponentStatus status; // NotActive, Active or Suspend`
- `# ComponentModes mode; // Manager or Guard`
- `# int currentState; // current leaf-state`
- `# int currentStateLevel[3]; // current states for each level`
- `# int history; // remembers the leaf-state when exiting`
- `# int entryGate;`
- `# int exitGate;`
- `# AssertionT *currentAssertion;`

**Methods:**

- `+ HeavyReactions heavyReactions;`
- `# void SetcurrentStateLevels(); // sets current states for all levels according to the current leaf-state`
- `# void PauseChilds(); // pause sub-components`
- `# bool CallLowerLevel(int level); // calls specific state level method`
- `# bool TestAssertion(AssertionT *AssertionP);`
- `# void DoPause(); // handles an exit through the pause gate`
- `# void DoResume(); // handles an entry through the resume gate`
- `# void DoTerminate(); // handles a normal component termination`
- `# void DoException(); // handles an exception exit`
- `# void ReturnToManagerEntryGate(int _entryGate);`
- `# bool HeavyReactionsNoMoreNeeds();`

---

**Figure 36: The ComponentT class**
There is no specific data structure to define the hierarchic state-diagrams of the Manager and Guard.
The states are represented as independent cells, which are implemented as ComponentT class methods. These methods will be called directly or indirectly by the Run method when they are active.
The transitions are conditions in the states methods ordered by their priority. The sub-components are also activated by the state-method, invoking the sub-component's Run method.
Sub-states are also activated by their super-state, which will called lower states levels.
Gates are implemented by the Gates abstract method, which enumerates (in a switch statement) the action that have to be taken when entering or exiting the component.
Assertions are aggregated by instancing the ArtifactT class (see section 11.1.4). They are tested in the state-methods as conditions.

11.1.2 The VirtualDeviceT class
This class is inherited for the virtual-device implementation. Virtual-devices are light components (see Fig. 37) which are handled in a more simple way just for I/O purpose. They have no gates, no Guard, no state-diagram hierarchy, and even no sub-components aggregations. They just behave as simple state-diagram. They have inputs, outputs, shared variables and states, which are represented in a similar way as in the ComponentT class.

<table>
<thead>
<tr>
<th>VirtualDeviceT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attributes:</strong></td>
</tr>
</tbody>
</table>
| # int currentState;  // current diagram state  
| # ComponentStatus status;// NotActive, Active or Suspend  
| # int timesOfExecPerSecond;   // times to be called per second  
| - unsigned int internalCycle;  
| - unsigned int internalTime;  |
| **Methods:** |
| + VirtualDeviceT(int timesOfExecPerSecond);  
| + void Run(); // handles the behavior  
| + void GetAndPut(); // I/O interchange with the reactive-kernel  
| # void Call(int id); //binds states-id-number to methods  |

Figure 37: The VirtualDeviceT class
States are represented as methods that include transitions with conditions.
The input and output dynamics are specified in the GetAndPut abstract method.

11.1.3 The ClockT class
As mentioned in the section 9.2, "the time measure is made by special objects that are platform independent (referenced in regular time units: seconds, milliseconds, etc), and are transparently adapted to the internal time domain". This transparent adaptation is made by the ClockT class (see Fig. 38).
Timers are implemented by an instantiation of this class.
The time unit is configured by the SetTimeUnit method. Then the timer is started with a specific initial value. The timer is incremented by calling the Advance method in the DoBetweenSteps method, which is called between the steps. A specific time-date can be tested by the Reached method.

```
ClockT
Attributes:
- TimeUnit timeUnit; // millisec, sec, min or hour
- unsigned long int counter; // current timer value
- bool enabled; // activation flag
- bool overflow;
Methods:
+ Clock();
+ void SetTimeUnit(TimeUnit _timeUnit);
+ void Start(unsigned long int t0); // inits a value and starts the clock
+ void Advance(); // called by DoBetweenSteps method
+ bool Reached(unsigned long int target); // tests a timeout
+ void Freeze(); // freezes the time counter
+ void Unfreeze(); // unfreezes the time counter
+ bool Overflowed();
- unsigned long int ConvertUnitToMillisecs(unsigned long int t);
```

Figure 38: The ClockT class

11.1.4 The AssertionT class
This class is inherited by the various assertions, defined in the design stage, creating a new subclass.
The subclass has to implement the Trigger method telling the frequency of test and to implement the Assertion method to tell which condition has to be tested. In addition the host-component has to insert the assertion in the suitable state-method.

In the component's initialization stage, the assertion's Set method is called by the host-component in order to set the initial values (critical, initial state and variables for condition test).

A wider explanation will be brought later in this chapter (see also Fig. 39).

| AssertionT |
| Attributes: |
| # bool critical; // correct or critical |
| # int initialState; // which state-Guard has to be reached first |
| Methods: |
| # bool Trigger(); // when it has to be to tested |
| # bool Assertion(); // the condition to test |
| + bool Test(); // performs the test at the requested time |
| # void DoBetweenSteps(); // time advance, etc |
| + void Restart(); |
| + int GetInitialState(); |

**Figure 39: The AssertionT class**

### 11.1.5 The HeavyReactionT class

This class enables heavy-reactions (see Fig. 40). Heavy-reactions are built to allow long reactions taking more than a time-step, without altering the synchronous platform timing (see section 9.4.2.4).

The heavy-reactions don't run in the Reactive Kernel processor as a regular reaction, but in the Synchrony Co-processor. The heavy-reactions are executed in a heavy-transition, which are long transitions that take more than one step. This is implemented by waiting in a pseudo-state until the heavy-reaction is finished.

According to this implementation the heavy-reaction has five status during its execution: Idle, Started, Running, Finished and OutputReported; which will be explained later.

In addition there is a class container named HeavyReactions which holds all defined reactions. The handling of the heavy-reactions is performed through this container class.
11.2 Behavior implementation

According to the Arts’Codes method, the Normal behavior is contained in the Manager, the exception handling in the Guard and the connection between the Manager and the Guard is done by Assertions which appear in Manager’s states and in Guard’s gates.

11.2.1 States

The Manager and the Guard are designed by a hierarchic automaton.

For the hierarchic automaton representation, an improvement of Andre’s reactive-cell method was adopted (see section 9.4.3). The reactive-cell approach focuses the state as the principle actor, without referencing to the global image of the whole automaton. The observer can group all reactive-cells’ rules and can sketch their interconnections, discovering in such a way the whole automaton behavior.

By this approach there is no data structure which represents the whole state’s inter-connections. Each state is represented by a simple method which is a member of the subclass (which inherits ComponentT).

---

**Figure 40: The HeavyReactions class**

<table>
<thead>
<tr>
<th>HeavyReactionT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attributes:</strong></td>
</tr>
<tr>
<td>+ HeavyReactionStatus status;</td>
</tr>
<tr>
<td>* LPTHREAD_START_ROUTINE lpStartAddress;</td>
</tr>
<tr>
<td>- HANDLE hThread;</td>
</tr>
<tr>
<td><strong>Methods:</strong></td>
</tr>
<tr>
<td>+ HeavyReactionT();</td>
</tr>
<tr>
<td>+ void GetInput() = 0; // get specifics input at the beginning</td>
</tr>
<tr>
<td>+ void WriteOutput() = 0; // writes specific outputs by the end</td>
</tr>
<tr>
<td>+ DWORD WINAPI Run(LPVOID dummy) = 0; // the specific reaction</td>
</tr>
<tr>
<td>+ void Start(); // starts the heavy-reaction</td>
</tr>
<tr>
<td>+ void Terminate(); // kills the heavy-reaction, called by the owner comp.</td>
</tr>
<tr>
<td>+ void Handle(); // handles this reaction, called by the Synch Co-proc</td>
</tr>
<tr>
<td>+ bool CanReachTargetState(); // test for ending the heavy transition</td>
</tr>
<tr>
<td># bool RunHeavyReaction();</td>
</tr>
<tr>
<td>// # void Set() = 0; // for external connections set (I/O)</td>
</tr>
</tbody>
</table>
This method handles triggers, actions, assertions, sub-components' activation and sub-states calling.

For example the *Centrifugation* state in Fig. 57, taken from the washing-machine example (see section 15.2.3) is translated as:

```cpp
1 void ProgCtrl::Centrifugation() {
2  try {
3    if (timer.Reached(centrifugationTime[*program])) {
4      timer.Freeze();
5      *process = "Idle";
6      exitGate = Terminate;
7      throw true;
8    }
9    Motor.Run();
10   Pump.Run();
11   if (TestAssertion(&pw))
12     throw true;
13  } catch(bool exception) {
14    Motor.Paused();
15    Pump.Paused();
16  }
17 }
18 }
```

Now we’ll explain the translated code using the line-number as reference:

1. The state was added as a method in the component class.
2. When *timer* reaches the programmed centrifugation time, actions at lines 4-5 are executed and the component leaves through its *terminate* gate (line 6).
3. Activate the *Motor* sub-component.
4. Activate the *Pump* sub-component.
5. Tests assertion *pw*.

Lines 14-17 suspends the sub-components which were activated in this state, when the state is left.

### 11.2.2 Transitions

For enabling inter-level transitions support, the Andre’s reactive-cell was improved in Arts’Codes (see a wide explanation in section 9.4.3). For this, two basic ideas were defined:
1. If the global automaton is hierarchic, then its current state is determined by the innerness automaton level current state, called in Arts' Codes the Current Leaf State (CLF). This assumption turns the complex multi-level automaton to a simple flat one level automaton, which transits from one CLF to the other.

2. In this hierarchic case, the execution of the CLF is made by calling the highest current state level, called in Arts' Codes the Current Root Level State (CRT), which appears in the CLF's CCSL (which is set by the SetCurrentStateLevels method). The CRT state tests its transitions' conditions, calls its sub-components (their Run method) and to its unique current sub-state method (the sub-state call is made by the CallLowerLevel method which uses the information supplied by the SetCurrentStateLevels method). This assumption produces a series of concatenated current states method calling, which may stop when a transition abandon the current-state at some level.

The first idea is implemented by the virtual method called SetCurrentStateLevels which builds up, for each CLF, the Constant Current States List (CCSL is the list of suitable current states active for a specific CLF). Each time that a transition state is performed, that means that the automaton has changed its current CLF; then the SetCurrentStateLevels is called and a new CCSL is built up.

The virtual SetCurrentStateLevels method is overridden by the component class and a switch statement is built according to each CLF ancestors.

The second idea implemented by calling the highest level in CCSL, which will call the next level and so on.

For example the Washing state in Fig. 58, taken from the washing-machine case-study (in section 15.2.3) is translated as:

```cpp
1 void ReactiveKernelRoot::Washing() {
2   try {
3     if (*e_start) {
4       PanelCtrl.SetEntryGate(1);
5       process = "Idle";
6       currentState = 1;
7       throw false;
8     }
9     if (CallLowerLevel(1))
10       throw false;
11   }
```
catch(bool exception) {
  ProgCtrl.Paused();
  PanelCtrlr.Paused();
}

In line 9 the Washing state which is the zero level (the root of the CCSL), calls the next state (low) level that is level one. This state level can make a transition which can affect its ancestor, then the function-call returns true and the Washing state abandons its method.

This series of calls is finished when the lowest state level CLF is reached, so it will return without calling to a lower level.

Here is an example of an overridden SetCurrentStateLevels method. This sample is for the ReactiveKerneRootl component, i.e. that is the Washing state host:

```cpp
void ReactiveKernelRoot::SetCurrentStateLevels() {
  ComponentT::SetCurrentStateLevels();
  switch (currentState) {
    case 3:
      currentStateLevel[0] = 2;
      currentStateLevel[1] = 3;
      break;
    case 4:
      currentStateLevel[0] = 2;
      currentStateLevel[1] = 4;
      break;
  }
}
```

In this implementation the Washing state is represented by the number 2, and DrClosed state as number 4.

In lines 8-10 Washing is set as level 0, and DrClosed as level 1, while DrClosed state is the Current State Leaf.

### 11.2.3 Gates

Gates are the tools for entering and exiting to/from a component Manager. They can execute an action when entering/exiting. When entering, an initial state has to be defined.

This is handled by the virtual Gate method, which is overridden by the component class, aided by a couple of switch statements. The Gates virtual method is called before the highest level current state in the Run method.
Here is the Gate’s implementation for the reactive-kernel-root component of the washing-machine case-study (see Fig. 58):

```cpp
1  void ReactiveKernelRoot::Gates() {
2   if (exitGate == Inactive) {
3     switch (entryGate) {
4       case Init:
5         tempRequired = 30;
6         program = 0;
7         process = "Idle";
8         PanelCtrl.SetEntryGate(1);
9         currentState = 1;
10        status = Active;
11         break;
12       case Resume:
13         DoResume();
14         break;
15     }
16     entryGate = Inactive;
17   }
18   switch (exitGate) {
19     case Pause:
20        DoPause();
21        break;
22     case Critical:
23        DoTerminate();
24        break;
25   }
26 }
```

Lines 3-16 implements the entry-gates, and line 18-25 the exit ones. For each gate an entry case entry is written including the action to perform (see for example lines 4-11 for the Init gate implementation). In addition an initial state is defined (line 9).

The entry-gate is determined by the component’s host. In line 8 the PanelCtrl entry-gate is set to 1, since ReactiveKernelRoot is its host.

The Resume entry-gate (lines 12-14) calls the DoResume method which handles the history pseudo-state.

Exit-gates are handled in a similar fashion.
11.2.4 Priors

Priorities are implemented by including the sentences in the suitable order, determined at the design stage.

The order is defined by:

1. The transition’s priorities that appears at the root of each transition.
2. The sub-component’s priorities which appears in the static diagram.
3. The Assertion’s position (pre/post).
4. The state’s hierarchy.

The following definitions define the order of the sentences in the state-method:

1. Pre-Assertions
2. Negative priorities transitions (in absolute ascending order)
3. Negative priorities sub-components (in absolute ascending order)
4. Next level current state-method (recursion)
5. Positive priorities sub-components
6. Positive priorities transitions
7. Post-Assertions

For example the ProgCtrl Wash state in Fig. 57, is implemented as:

```c++
1 void ProgCtrl::Wash() {
2   try {
3     if (TestAssertion(&cwl fh))
4       throw true;
5     if (TestAssertion(&hw))
6       throw true;
7     Thermos.Run();
8     Motor.Run();
9     if (timer.Reached(washTime[*program])) {
10    timer.Start(0);
11    *process = "Rinsing";
12    Filling.SetEntryGate(Init);
13    Pump.SetEntryGate(Init);
14    Motor.SetEntryGate(1);
15    i = 0;
16    currentState = 3;
17    throw false;
18  }
19  }
20 catch(bool exception) {
21    Thermostat.Paused();
```
Lines 3-6 are the pre-assertions, which are followed by thermostat sub-component which has negative priority and there no negative transition or sub-state. Then the Motor sub-component (line 8) followed by the positive transition (lines 9-18), and there is no post-assertion.

11.2.5 Heavy Transitions

Heavy-transitions are implemented by adding a pseudo-state for waiting for the end of the execution in the Synchrony Co-processor. When the trigger of the heavy-transition is true, then the heavy-reaction is started and the pseudo-state is reached.

The pseudo-state just checks the CanReachTargetState method for this heavy-reaction until the end of the execution; then it will enable the reaching of the target-state.

Taking the Rinse state of the ProgCtrl component (see Fig. 57) as an example the following pseudo-state is added as:

```c
1  void ProgCtrl::Rinse() {
2    try {
3      Motor.Run();
4      Filling.Run();
5      Pump.Run();
6      if (timer.Reached(rinseTime[*program])) {
7        if (cacp.CanReachTargetState()) {
8          timer.Start(0);
9          *process = "Centrifugation";
10         Pump.SetEntryGate(Init);
11         Motor.SetEntryGate(2);
12         pw.Restart();
13         currentState = 4;
14        }
15      }
16    } catch (exceptionType) {
17      #error exception thrown in Program
18    }
19  }
```

In line 2 the cacp heavy transition is tested for reaching the centrifugation state (state number 4).

The following code is for the Rinse state itself (see Fig. 57):

```c
1  void ProgCtrl::Rinse() {
2    try {
3      Motor.Run();
4      Filling.Run();
5      Pump.Run();
6      if (timer.Reached(rinseTime[*program])) {
```
In line 7 the `cacp` heavy reaction is started, and in line 5 a transition is made to the pseudo-state.

### 11.2.6 Hierarchic Component Activation

In Arts’Codes, sub-components are not active even they appear in the static diagram. A sub-component must appear in a state to be active. Then, when the component’s Manager reached this state, it calls the sub-component’s Run method.

The sub-component may appear in several states and then, a call to its Run method will appear in all relevant state-methods. An example to a sub-component call can be seen in the previous section in the *Rinse* state-method lines 8-10. In these lines the *Run* method of *Motor*, *Filling* and *Pump* sub-components are called.

This *Run* method call occurs at each *step* of the Reactive Kernel.

When the Manager leaves the current state the sub-component *Paused* method is called, which handles the pause *exit-gate*.

### 11.2.7 Exception Handling

The component’s behavior is split into normal behavior (Manager) and exceptions (Guard). Exceptions are detected by assertions which control the transition from the Manager to the Guard. After the exception handling is completed, the Guard returns to one of the Manager’s *entry-gates*.

*Assertions* are inserted in the method-state as conditions using the *TestAssertion* method. The *TestAssertion* method takes the *assertion* object as the method argument, and it returns a Boolean value with the test result.
Assertions are defined by inheriting the AssertionT class and overriding its virtual methods.
All states are handled in the same way without distinguishing between Manager or Guard.
The only difference is when transitions pass through a gate from a Manager state to a Guard state and back; in this case they are treated as follows:

$\$ From Manager to Guard: by the TestAssertion method.
$\$ From Guard to Manager: by the ReturnToManagerEntryGate method.

In the Centrifugation state-method of the washing-machine example in section 11.2.1, the TestAssertion method tests the pw assertion (in line 10), and if it is true then abandons the state. The initial state at the Guard was defined by in the Assertion object by the following sentence:

```cpp
pw.Set("PW", 6, WaterQuant);
```

which means that pw assertion called "PW" assertion will enter the Guard state number 6 (g_Pump), when the exception arises. WaterQuant is a variable needed for the assertion's test.

In the following g_Pump Guard's state-method we see a return to the Manager through the Critical Manager's entry-gate:

```cpp
void ProgCtrl::g_Pump() {
   *pumpLed = true;
   ReturnToManagerEntryGate(Critical);
}
```
The following are the AssertionT virtual methods overridden by the pw subclass:

```cpp
bool PW::Trigger() {
   return timer.Reached(5);
}
```

Trigger determines when the assertion is to be tested (frequency in this case).

```cpp
bool PW::Assertion() { // The Assertion is the condition to test
   return *WaterQuant<quantT0-5 || *WaterQuant<5;
}
```
The Assertion is the condition to test.

```cpp
void PW::Restart() {    // Restart resets the assertion testing
   quantT0 = *WaterQuant;
   timer.Start(0);
}
```

Restart resets the assertion testing.

```cpp
void PW::DoBetweenSteps(){    // DoBetweenSteps does the timing handling
   timer.Advance();
}
```
11.2.8 The between steps actions
In addition to the regular behavior which is handled during the steps, there are some tasks that must be handled before the step begins. This handling involves tasks like timers’ advance, signals’ reset (a signal is a flag with only one step lifetime), etc.
The request actions have to be executed before the step begins, has to be inserted in the virtual DoBetweenSteps method. This method has to be overridden by the subclass to include the requested actions.
The DoBetweenSteps method of the reactive-kernel-root component is called before the step begins. Inside of this method its sub-components' DoBetweenSteps methods are called, which themselves call their sub-components' DoBetweenSteps methods, and so on.
Also assertions which handle timers work in the same manner, i.e. they have to override the DoBetweenSteps method which is called by the host-component.

11.2.9 Virtual and Physical devices (VD & PD)
The virtual device behavior is defined by the state methods, which are very simple because VDs have no state’s hierarchy.
The Call virtual method has to be implemented for the state-number of the method to bind.
The GetAndPut method defines what are the inputs/outputs which have to be interchanged between the VDs and the reactive-kernel-root, before the step is beginning.
The DoBetweenSteps executes the actions that should be done only between the VD time micro-steps (the VDs pacemaker steps are shorter than the RK steps).
The following lines define a virtual-device that functions as a simple button (see Fig. 60):

```cpp
1 void ButtonT::Call(int id) {
2   switch (id) {
3     case 0:
4       None();
5       return;
6     case 1:
7       Released();
8       return;
}```
case 2:
    Pushed();
    return;
};

void ButtonT::GetAndPut() {
    *e_ButtonPressed = i_ButtonPressed;
    i_ButtonPressed = false;
}

void ButtonT::Released() {
    if (GetButtonStatus()) {
        i_ButtonPressed = true;
        currentState = 2;
    }
}

void ButtonT::Pushed() {
    if (!GetButtonStatus()) {
        currentState = 1;
    }
}

This button VD has two states: one when it is pressed (lines 24-28) and the second when it is released (lines 18-23).

The VD sends an event between the steps, which have only one step lifetime (lines 14-17), because the GetAndPut method is called once between the steps.

The GetButtonStatus (lines 19, 25) method is an abstract method which has to be implemented according to a specific physical device. This method is implemented by a PD (Physical Device). The PD inherits a VD and implements the abstract functions for a specific device.

The VD is the template, in this case of a button, and the PD binds the template to a specific hardware.

11.2.10 Events

Even the internal variables changes are immediately detected in the step, there is an option to use the event variable to implement delayed Boolean values, with
one step of time-life. This helps the programmer to assure, that a specific internal event is broadcast and seen by all the correspondent components. The event is implemented by a simple class which has two variables: one for the hided value and the other for the shown one. At each step (using the DoBetweenSteps method), the shown variable is assigned with the hided one, and the hided one is assigned to false. The sender assigns true to the hided variable, and the receiver tests the shown one.

11.3 Software Execution Model

11.3.1 Abstract

Arts’Codes execution platform is a tri-processor platform. In order to test the method, the model was built under a multithreading system, simulating processors as threads. Three threads were built each one simulating one processor. For each thread the corresponding algorithm was programmed. Each processor has a main-loop which iterates while the halt flag is false. In this section an explanation is presented separately for each processor.

The Reactive Kernel, which is the responsible to compute the outputs for the entering inputs according to the components Managers’ automata, it synchronizes its task with the Synchrony Co-processor by using the inStepFlag.

The Environment Kernel is responsible for collecting the environment’s inputs, and for emitting the RK’s computed outputs. The EK collects/emits continuously while the pauseFlag flag is false; otherwise the EK pauses giving the Synchrony Co-processor the opportunity to interchange the data between the RK and EK.

The Synchrony co-processor paces the system, marking accurately the steps. The SC uses the inStepFlag and pauseFlag flags to isolate secure portions of time between the steps, to perform the interchange of inputs/outputs between the Environment and Reactive kernels.

In Fig. 41 a global glance of the tri-processors system cooperation can be seen.

11.3.2 Reactive Kernel (RK)

The main-loop

The following is the main loop of the parallel-automata system. One iteration of this loop defines the famous step (the colored yellows’ lines), that we told about it in the previous sections.
DWORD WINAPI ReactiveKernel(LPVOID dummy) {
    reactiveKernelRoot.Set(shared input/outputs with the several VDs);
    reactiveKernelRoot.SetEntryGate(Init);

    while (!inStepFlag);
    while (!halt) {
        reactiveKernelRoot.Run();
        inStepFlag = false;
        while (!inStepFlag);
        reactiveKernelRoot.DoBetweenSteps();
    }
}

Initialize
Line 2 defines the interconnections between the root component and its surrounding virtual-devices.
Line 3 determines the entry-gate of the root component.
Line 5 waits for the instepFlag set by the Synchrony Co-processor. This mean of synchronization is used by the Synchrony Co-processor in order to pace the system. The inStepFlag flag will be set only at the beginning of the new step.

Main loop
Lines 6-11 are the main loop that executes the steps one after the other, as soon as the halt flag becomes true. The termination will happen only between steps.
In line 7 he RK calls the Run method of the reactive-kernel-root component, instance, giving in this way the possibility to all the components’ hierarchy to react.
Line 8 marks that all the suitable reactions were done, which tells the Synchrony Co-processor that the interchange of data between the RK and the EK should be done.
Line 9 waits for the beginning of the next step.
Line 10 executes the operations that must be done between the steps. This is the last sentence of the main loop, and the loop begins a new iteration.

Summary
After defining the interconnections between the virtual-devices and the reactive-kernel-root component, in addition to the definition of the initial state of reactive-kernel-root’s Manager; the main loop begins.
The main loop gives to the components’ hierarchy the opportunity to react, waits for the next step and executes the actions that should be done only between the steps.

**The Run method**

The *Run* method main idea was taken from the Java’s Thread class which enables multithreading execution when generalizing the class. The Java Virtual Machine runs this *Run* method as a separated thread in parallel to all existing ones. In Arts the Run method it is implemented as a fixed-code, which is suitable to all subclasses, with no need to define it as a virtual function.

Since the component behavior is described as an automaton, the *Run* method task is to behave according to the rules of the specified automaton.
Figure 41: The Arts tri-processor execution platform
The following lines are the fixed-code used to "Run" all components:

```cpp
int ComponentT::Run() {
    if (exitGate == Inactive) {
        if (entryGate == Inactive) {
            if (status == Suspend)
                entryGate = Resume;
        } else
            status = Active;
    } else
        visitedState[0] = false;

    while (!visitedState[currentState] || entryGate!=Inactive || exitGate!=Inactive) {
        Gates();
        lastExitGate = exitGate;
        exitGate = Inactive;
        SetCurrentStateLevels();
        Call(currentStateLevel[0]);
        visitedState[currentState] = true;
    }

    return lastExitGate;
}
```

Lines 2-9 (green colored) decide which *entry-gate* has to be entered. One of the options is the *Resume entry-gate*, when in its last activation the component left via the *Pause exit-gate*.

Lines 11-12 all states are marked as not visited.

Lines 13-20 are the main loop which iterates until the automaton transits to a state that was already visited by this loop, or if the Manager exits through some *exit-gate*.

Line 14 calls to the virtual *Gates* method, which was implemented by the subclass who inherits the ComponentT class. In this method the switch cases for the entry and exit *gates* are checked. In the *entry-gate* switch case, an initial *Current Leaf State* (CLF) should be selected, besides the suitable transition's action.
Line 17 defines the several current states for each level, according to the Current Leaf State, named the \textit{Constant Current States List (CCSL)} (see section 9.4.3.4). In line 18 the \textit{Current Root Level State (CRT)} method is called (i.e. the highest active state level), which will invoke the lower active states levels. This loop will execute the behavior defined by the Manager until a state is reached for the second time or the component is left.

\textit{Manager and Guard}

The Manager can exit due to an assertion, passing the control to the Guard. The passing form the Guard to the Manager is made by the \textit{DoException} method which remembers the history state, and enters the Guard state defined by the assertion.

The following lines are the \textit{DoException} method code:

```c
void ComponentT::DoException() {
    history = currentState;
    mode = Guard;
    currentState = currentAssertion->GetInitialState();
}
```

The returning is made by the \textit{ReturnToManagerEntryGate} method which enters the specified Manager entry-gate defined by the Guard state method.

The following lines are the \textit{ReturnToManagerEntryGate} method code:

```c
void ComponentT::ReturnToManagerEntryGate(int _entryGate) {
    entryGate = _entryGate;
    mode = Manager;
    if (entryGate == Resume)
        status = Suspend;
}
```

\textbf{11.3.3 Environment Kernel (EK)}

The EK is responsible of the permanent environment I/O image update. In this section the full algorithm of virtual-devices execution is presented.

\textit{The main loop}

The following lines are the algorithm used by the Environment Kernel.

```c
1 DWORD WINAPI EnvironmentKernel(LPVOID dummy) {
2 struct _timeb time0, timeb;
3 unsigned long int last, now;
4 int i;
```
5  _ftime(&time0);
6  last = time0.millitm;
7
8  while (!halt) {
9    for (i=0; i<TotalVirtualDevices; )
10       if (!pauseFlag) {
11          acknowledgeFlag = false;
12          virtualDevice[i++]->Run();
13       }
14       else {
15          acknowledgeFlag = true;
16          while (pauseFlag);
17       }
18
19       do {
20          Sleep(MicroStepInMillisecs/5);
21          _ftime(&timeb);
22          now = (timeb.time-time0.time)*1000+timeb.millitm;
23          acknowledgeFlag = pauseFlag;
24       } while (now < last+MicroStepInMillisecs);
25       last = now;
26
27    for (i=0; i<TotalVirtualDevices; i++)
28       virtualDevice[i]->DoBetweenSteps();
29  }
30}

Lines 8-29 are the main-loop of the EK, which iterates while the halt flag is false. The main-loop is split into three sections:

1. Lines 9-17 (the colored yellows’ lines) executes the virtual-devices Run method. If the pauseFlag is true it waits until it becomes false.
2. Lines 19-25 (the colored greens’ lines) waits for the remaining time to the next micro-step (the virtual-devices base step).
3. Lines 27-28 (the colored violets’ lines) execute the action that should be done only between the micro-steps.

**The Run method**
The virtual-device is executed once in a cycle. The cycle is defined for each virtual-device separately, and it is defined in micro-step units.

When a new cycle begins the VD automaton is executed until a state is reached for the second time.

In the following lines of the Run method we see in lines 7-10 the main loop, which executes a simple automaton, just calling the current state-method. There is no state hierarchy in virtual-devices.

Lines 2-4 determines if the VD has to be execute, or if the new cycle didn’t begun.

```cpp
1 void VirtualDeviceT::Run() {
2   internalTime += MicroStepInMillisecs;
3   if (internalTime >= internalCycle) {
4     internalTime -= internalCycle;
5     for (int i=0; i<MaxStates; i++)
6       visitedState[i] = false;
7     while (!visitedState[currentState]) {
8       Call(currentState);
9       visitedState[currentState] = true;
10     }
11   }
12}
```

11.3.4 Synchrony Co-Processor (SC)

The following lines describe the Synchrony Co-processor algorithm. The SC is responsible for the system pacing and heavy-transitions parallel execution.

```cpp
1 DWORD WINAPI SynchronyCoprocessor(LPVOID dummy) {
2   struct time0, timeb;
3   unsigned long int last, now;
4   int i;
5
6   _ftime(&time0);
7   last = time0.millitm;
8
9   while (!halt) {
10      while (inStepFlag)
11        stepOverflowFlag = true;
12      pauseFlag = true;
```
13  while (!pauseAcknowledgeFlag);
14
15  for (i=0; i<TotalVirtualDevices; i++)
16    virtualDevice[i]->GetAndPut();
17  reactiveKernelRoot.heavyReactions.Handle();
18
19  pauseFlag = false;
20  inStepFlag = true;
21
22  do {
23    Sleep(StepInMillisecs/5);
24    _ftime(&timeb);
25    now = (timeb.time-time0.time)*1000+timeb.millitm;
26  } while (now < last+StepInMillisecs);
27  last = now;
28  }
29}

The main loop of the SC is defined in lines 9-28. This main loop is split into three sections:

1. Lines 10-13 (colored yellows’ lines) waits for a secure time-slice between the steps, where both: the Reactive and the Environment kernels are paused in a secure state. In lines 19-20, the RK and EK are released to continue their work. Line 11 sets the stepOverflowFlag flag, to enable assertion’s arise and the suitable handling in the Guard.

2. Lines 15-17 describes the tasks to do in this secure time-slice. Two tasks must be done: inputs/outputs interchange between RK and EK (lines 15-16), and the heavy-reactions handling (line 17).

3. Lines 22-27 waits for the remaining time until the next step.

11.4 Rules of Compilation

The Arts’Codes is a VPL (Visual Programming Language) language, which is visually designed and can be directly compiled to code; according to well-defined rules.

Since the code structure is Object-Oriented, where data structured patterns are predefined (see section 11.1), therefore the rules are basically built into three stages:
1. Derivation of classes (inheritance).
3. Abstract method implementation.

Four classes were defined, modeling each one a basic element, which are:

1. Component
2. Virtual Device
3. Assertion
4. Heavy Reaction

These predefined classes are in fact four basic patterns, with which the code is built.

11.4.1 Derivation of classes
The first stage in the direct translation is to create a sub-class for each basic element. Therefore:

1. Each component which appears in the component's catalog of the design, is represented by a derived class of ComponentT base class.
2. Each virtual-device which appears in the virtual-device's catalog is represented by a derived class of VirtualDeviceT base class.
3. Each assertion is represented by a derived class of AssertionT base class.
4. Each heavy-reaction is represented by a derived class of HeavyReactionT base class.

11.4.2 Addition of members
The second stage is to include additional members inside the derived classes, according to the design.

Addition to ComponentT
Public members:

- The Set method is added as a member class. It defines all inputs/outputs as arguments receiving their addresses. Its aim is to:
  1. Save these addresses for later use. T
  2. To call the Set method of all its sub-components, in order to pass them the corresponding addresses. In this way the "wiring" is applied, implementing in such a way the input/output interconnections for each sub-component.
  3. To call the Set method of its assertions, in order to pass them the requested input data.
To register the *heavy-reactions* in addition to relevant input/output addresses.

- To initiate constant variable's values.

**Protected members:**

- Shared variables are added as attributes.
- Local variables are also added as attributes.

**Private members:**

- Sub-components are added as attributes, each one declared according to its type (subclasses derived from *ComponentT*).
- States (both Manager's and Guard's states) are added as methods, each one representing the state behavior.
- Pseudo-states are added for each *heavy-reaction*. This state is active during the *heavy-reaction* running.
- Assertions are added as attributes, each one declared according to its type.
- *Heavy-reactions* are added as attributes.

**Addition to VirtualDeviceT**

**Public members:**

- The constructor method is added including the suitable addresses as arguments, in order to connect to the *reactive-kernel-root*. It includes:
  - Saving of inputs/outputs attributes.
  - Several initializations.
  - Times per-second activation.

**Protected members:**

- Abstract methods are added, in order to be implemented later by the *physical-devices*. This technique enables *virtual-devices* patterns that can be reused for several *physicals-devices*.

**Private members:**

- Input and outputs are added as attributes holding their addresses; enabling inputs/outputs interchange (with the inputs/outputs of the *reactive-kernel-root*) between the steps.
- States are added as methods, each one representing the state behavior.

**Addition to AssertionT**

**Public members:**
§ The Set method is added as an attribute in order to receive the addresses of the relevant input data. In addition any initialization needs are executed in it.

Private members:
§ Data input addresses are added as attributes.

**Addition to HeavyReactionT**

Private members:
§ Data input addresses are added as attributes.
§ Local variables are added as attributes.

### 11.4.3 Abstract method implementation

Each of the four predefined patterns defines abstract methods, which are fixed methods that must appear in each derived class, but with different implementation. Therefore the third stage is to implements the abstract methods.

**Adding Abstract methods in ComponentT**

Public members:
§ The method DoBetweenSteps is implemented in order to include:
  o Timer advance
  o Calling to sub-component's DoBetweenSteps method.
  o Calling to Assertion's DoBetweenSteps method.
  o Events refreshing (events have only one step time-life).

Private members:
§ The Call method is implemented binding state's numbers to the suitable state's method invocation, using a simple switch statement.
§ The Gates method is implemented defining different gates translations. Entry and exit gates are handled separately by different switch statements. Each gate translation may include:
  o Statements representing the reaction.
  o Sub-component's entry-gates specification.
  o Assertion activation.
  o Initial state selection.
  o Any suitable predefined method, for some pre-defined gate handling such as:
    § DoPause
    § DoResume
DoTerminate
DoException

- Exit-gate activation.

The PauseChilds method is implemented by calling the Paused method of all sub-components. The PauseChilds is called by the DoPause method.

Adding Abstract methods in VirtualDeviceT

Public members:
- The method DoBetweenSteps is implemented in order to include any needed action between micro-steps.
- The GetAndPut method is implemented in order to interchange inputs and outputs with the reactive-kernel-root, between steps.

Private members:
- The Call method is implemented binding state's numbers to the suitable state's method calling, using a simple switch statement.

Adding Abstract methods in AssertionT

- The method Trigger is implemented by returning the suitable condition value.
- The method Assertion is implemented by returning the suitable condition value.
- The method Restart is implemented in order to include any initialization.
- The method DoBetweenSteps is implemented in order to include timer advancing.

Adding Abstract methods in HeavyReactionT

Public members:
- The GetInput method is implemented by copying all suitable inputs from the host-component.
- The WriteOutput method is implemented by passing the suitable outputs to the host-component.
- The method Run is implemented in order to execute the heavy-reaction. It calls the ResetWrapper method before the execution, and assigns a Finished status at the end.

11.4.4 States

The state translation is more complicated and it is explained here in a separate section.
States are represented as methods covering:

- Transitions firing
- Assertions testing
- Sub-component execution
- Heavy-reaction handling
- Sub-state calling

The order of the previous list handling should be sorted according to the priorities rules in section 11.2.4.

Each state is structured by a try-and-catch statement, assuring that when each method is left, it will pause any active sub-component in this state.

**Transitions**

A transition is made of:

- Trigger
- Reaction
- Target-state.

The trigger is tested in an *if* statement. The statements inside the sentence are the reaction's statements, including the *currentState* assignment with the new target-state number.

**Assertions**

Assertions are tested with the *TestAssertion* method. When it returns true it exits the method activating the *Exception* predefined *exit-gate*.

Each entering transition to the assertion's host-state, calls the *Restart* assertion's method; activating in such a way the *assertion* before entering the state.

**Sub-components**

Sub-components are executed by simply calling their *Run* method. This *Run* may return an *exit-gate*. This returned value is tested when an *exit-gate* was designed to trigger a transition.

**Heavy reactions**

When a *heavy-transition* (a transition which activates a *heavy-reaction*) is fired, the *currentState* attribute is assigned with a temporary pseudo-state, and the *Start* method is activated for this *heavy-reaction*.

The pseudo-state has only one condition statement testing the *CanReachTargetState* method. When this method returns a true value, the transition continues, and the target-state is activated.
**Sub-states**

The sub-state-method is called. If it returns true, it means that the sub-state broke its level bounds, causing its upper-level to leave. In this case the state-method should be exited.
12

CHILD Language

12.1 Abstract

In this thesis two solutions are presented for the translation of the Arts'Codes graphical design.

A first solution, which was tried, is to define a language named "CHILD" which is a textual translation of the Arts'Codes graphic design. "CHILD" stands for Component Hierarchic Interface Language for Design. This declarative language can be tested lexically, syntactically, semantically, and compiled into parallel automata tables and then run by an execution platform. Furthermore CHILD can also be translated into a Propositional Temporal Logic Language in order to allow the proving of the consistency of the components with their goals.

Another solution is that the Arts'Codes graphic design can be directly translated to a light Object-Oriented code and executed on the tri-processors platform seen in the previous sections. This second solution is preferred for efficient execution.

The first solution is heavier and more complicated but it has the advantages to allow more semantic and consistency testing. Big efforts were made on this approach, including the help of two engineering students Michael Goltsman and Shneur Zingerevitich, which programmed the compiler that runs according to the CHILD specification, this approach is briefly presented here.

The definitions taken for the testing of this approach are brought here, which includes the language syntax and an example. The syntax rules are brought in appendix C.

12.2 CHILD Language Syntax

12.2.1 Component declaration

The syntax of the CHILD language was built so that to be homothetic to the graphic design method: each graphic element correspond to a sentence of the CHILD language. For instance, Each graphic component is translated to a "component" phrase structure in CHILD.

The component is defined in the following manner:
component component-type-name : max-instantiation
{
    external {
    }
    Protected {
    }
    subcomponent {
    }
    manager (external-variable-list) {
        local:
            state:
            condition:
            action:
            transition:
            assertion:
        }
    guard {
        local:
        states:
        conditions:
        actions:
        transitions:
    }
}

As we see the component has internal sections to define:
1. External input/outputs
2. Shared (protected) variables
3. Subcomponents
4. Manager
5. Guard

The Manager itself has also internal sections which some of them appear also in the Guard:
1. local variables
2. state
3. condition
4. action
5. transition
6. assertion

12.2.2 Inputs and outputs declaration

Inputs and outputs are defined in the external section. The syntax is:

\[
\text{in|out : } \\ \ \text{*variable-type variable-name;}
\]

\[
\text{*Gate gate-name;}
\]

\[
\text{Gate Exception;}
\]

\[
\text{Gate Critical;}
\]

which means that the variable can be in, out or both if appears in both sections.

Gates are also defined like simple variables.

12.2.3 Sub-components declaration

The definition syntax is:

\[
\text{*component-type sub-component-object([variable-name | NC],)};
\]

It defines the sub-components of the host-component.

The definition is composed of the component-type (must be already defined earlier) and an object name.

Inside the parenthesis the interconnections with its parent are defined. The order of the "arguments" is the order of the variables defined in the external section.

Where NC is the acronym of no-connection.

12.2.4 Manager and Guard

The Manager (similar in Guard) has to define the following items:

- List of local variables
- States hierarchy
- List of conditions
- List of actions
- List of transitions grouped by states.

The state hierarchy is defined as follow:

"{state-name{ state-name, state-name }, state-name, state-name }".

The actions as:

[heavy]\*action-name() {

*statement;
}

}
Where the heavy keyword expresses that the action takes time more than one step to be executed.

The grouped transitions are described as:

1. *state: state-name {
2. set assertion: [ *assertion-name | cycle_overflow | no_value ];
3. set subcomponent: sub-component-object-list;
4. *when (gate|condition: name) then {
5. set action: *action-name;
6. set state: state-name.state-name;
7. set gate: terminate|pause|critical| gate-name |
   *sub-comp-object-name .[init|resume|continue|subcomp-gate-name];
8. set signal *signal-name;
9. }
10. }

Each state has its assertions and sub-components to be activated which appears in the following lines 2-3.

Lines 4-10 defines a transition: with its condition in line 4, action in line 5 and next state/gate at lines 6-7. The gate may be an entry-gate of a sub-component. The transition may emit a signal in line 8.

### 12.3 Examples

Following is the CHILD code for the Pump component (see Fig. 42) of the washing-machine case study (see section 15). The code is the automatic translation compiled by the ACE (see section 14) editor from the graphic design.

```
component Pump
{
    External {
        in:
            Gate I, R;
        out:
            Flag PumpOn;
            Gate T, P, E;
    }

    Protected {}
}
```
manager(PumpOn) {
    state: {PumpActive}
    condition:
        sdf=[k < 5;]
    action:
        TurnOn() {PumpOn=1;}
        TurnOff() {PumpOn=0;}
    transition:
        when (gate: Terminate) then { set action: None}
        when (gate: Pause) then {
            set action: TurnOff
            set state: Remember
        }
        when (gate: Exception) then {set action: None}
        when (gate: I) then{
            set action: TurnOn
            set state: PumpActive
        }
    state: PumpActive {} 
}
13

Compiler Rules to provable Temporal Logic

13.1 Abstract

In this chapter a solution is proposed to the problem of giving an apriori validation of the Arts’codes design, by introducing a formal description of the goals and the behavior of the components; so that we can verify (before execution) that the behavior of the components will satisfy the goals assigned by the specifier and the designer. Only from this formal specification an apriori design validation diagnosis can be deduced. As we have seen in the precedent chapters, Arts’Codes has also tools (assertions and guards) to verify the execution at the run-time level.

But if we want an apriori proof to be carried out, goals and behaviors must be expressed in the same logical notation (here it will be in Propositional Temporal Logic PTL [73,74]). Since Arts’codes is based on the principle of building a system by modular hierarchic components, this facilitates the building of the whole system gradually, components by components, and this can also facilitate its gradual validation on a component-basis.

We thought to deduce automatically the behaviors and the goals properties of the Arts’Codes components from the CHILD textual representation of the graphical diagrams. For this, a compiler was built (with the help of an engineering student A. Shechter) in order to generate the Goals and the behaviors formulas of Temporal Logic for all the components, from the CHILD text. Then these PTL formulas are fed in an automatic prover such as the Stanford STEP prover [74] and the goals-behaviors consistency is checked automatically.

13.1.1 Principle of the proving

In order to build a new component, an engineer can use a set of validated existing components (in a catalog). He can also add some new components and build a whole system with them. Associated to the components, he will introduce a set of temporal formulas which express the goals of the component in addition to its behavior (its manager).
The a-priori validation will consist in showing that:

\[(B) \text{ and } (SCG) \text{ implies } (G)\], where:

\(B\) is the temporal logic formula expressing the behavior of the component, \(G\) is the temporal logic formula expressing the desired properties (Goals) of this component, \(SCG\) is the conjunction of all the Goals of its sub-components (already validated, stored in the catalog).

### 13.2 Compiling rules

For the compiling purpose, the source code is the CHILD text and the target code is made of two equivalent forms of temporal logic languages: TCOM and PTL.

PTL [73] [74] is based on mathematical formulas using Boolean variable names and symbolic temporal logical operators. Examples of operators:

- `\&\&` means AND,
- `|` means OR,
- `=` means IS EQUAL TO,
- `<>` means ONCE LATER,
- `[ ]` means ALWAYS IN FUTURE,
- `-->` means IMPLY etc...

For instance, in PTL a formula is of the kind:

\[G= [\neg (E --> (X=1 \& F=1 \& S=1))\]

Formulas may also be grouped in "macros", i.e. sub-formulas which compute Boolean variables. We use these macros to compute the components goals or behaviors.

TCOM [75][76] is a component-oriented language where the Goals and Behaviors of the component are written in Temporal Logic using key words instead of symbolic logical operators, this a simple engineering dialect equivalent to PTL, but component-oriented.

For instance the previous PTL formula would be written in the TCOM dialect

GOAL: when(Enter) then (Exit and Full and Sealed);

### 13.2.1 Concordance Rules between CHILD and TCOM/PTL

Here are the rules of the translation:

1. Each component of the CHILD language is interpreted as a component of the TCOM specifications.
2. As the program written in CHILD must start with the main component named Controller, so the appropriate TCOM interpretation must start
with the component named MAINSystem. In addition, it (and all the rest components) can include the sub-components.

Variables part of CHILD are transformed into DECLARATIONS part of TCOM.

a. external \{in...\} is transformed into DECLARATIONS: IN...

and accordingly external \{out...\} is transformed into DECLARATIONS: OUT....

b. protected is transformed into LOCAL.

c. sub-component should be represented as INCLUDES.

manager part is interpreted as BEHAVIOR in TCOM.

a. local here includes the variables list, which in a fact is a part of the variables list of the whole component. So, it should be ignored as soon as it already appears in the DECLARATIONS part of the TCOM component.

b. state includes the hierarchical list of the states for the current component. TCOM does not support states so this list should be represented as a list of the Boolean flags, which will show what the current state of the program is. The hierarchy of the states can be reflected in the namespace of the flags. For example, the following states list: \{state-name \{state-name, state-name\}, state-name \{state-name \{state-name\}\}, state-name\} should be translated into the appropriate flags’ namespace: state11, state21, state22, state12, state23, state31, state13. The first part of the index is the level number and the second one – sequence number of the state on the level. Moreover, of course this list should be added to the LOCAL block of TCOM.

c. conditions are supposed to be transformed into the when/then block. The conditions block structure is as follows: condition-name = condition_expression. condition-name, when transformed will be a Boolean flag, telling if the condition-expression is true or false, and condition-expression will stay as it is. So, it will look like as follows: when(condition-expression) then(condition-name). It means that the condition-name must be added to the LOCAL list of TCOM as a Boolean flag.
d. actions are supposed to be transformed into the OPERATIONS block. The actions block structure is as follows: action-name(){*statement;}. action-name, when transformed will be a Boolean flag, telling if the action should be performed or not. The name of this flag should be defined in the following way: Do_action-name. This name should also be added to the LOCAL list of TCOM.

e. The transition block is some more complex. It consists of conditional blocks:

```
*when (condition_expression) then {
    set action: action-name;
    set state: state-name;
    set gate: gate-name;
    set signal: signal-name;
}
```

and of states blocks:

```
*state: state-name {
    set assertion: assertion-name;
    set sub-component: sub-component-object-list;
    when (gate|condition: name) then {
        set action: *action-name;
        set state: state-name.state-name;
        set gate: gate-name;
        set signal *signal-name;
    }
}
```

The conditional blocks’ translation is trivial: conditional_expression is copied, as it is, to the when_condition of the BEHAVIOR and all the actions (i.e. set action, set state, set gate and set signal) are translated into the then_expression of the conditional block. In the general case, it’ll look like this:

```
when(conditional_expression)
    then(  action-name AND
            state-name AND
            gate-name AND
            signal-name);
```
Of course, all the variables are added to the DECLARATIONS list:

- **action-name** is added as  
  ```plaintext
  boolean signal : S_do_action_name;
  ```

- **state-name** is added analogously as  
  ```plaintext
  boolean signal : S_state_name;
  ```

- **gate-name** is added as  
  ```plaintext
  boolean flag : F_gate_name;
  ```

- **signal-name** is added as  
  ```plaintext
  boolean signal : S_signal_name;
  ```

The states block differs from the conditional block by having some additional attributes: **state-name**, **assertion**, and **sub-component**. So the translation will look like the following:

- When (state-name)
  ```plaintext
  then (assertion-name AND
  sub-component-object-list);
  ```

- When (state-name AND condition-name)
  ```plaintext
  then (action-name AND
  state-name AND
  gate-name AND
  signal-name);
  ```

- The **assertions** part contributes not only the behavior but also the goals in our system. Its **type** attribute shows if the entry should be added to the BEHAVIOR (correct or critical value) or to the GOALS (goal value). The assertions part structure is performed bellow:
assertion: assertion-name {
    local {
        *variable-list;
    }
    type: [correct | critical | goal]
    position: [pre | post]
    trigger: trigger-conditional-expression;
    assertion: assertion-conditional-expression;
    action {
        *statement;
    }
}

As you can see, the position is not used here (there is no suitable translation for it in TCOM). The local part is skipped as soon as here can be the variables from the local part of the manager only.

5 guard part is interpreted similarly to the manager.

6 Add the whole BEHAVIOR block to the GOALS.

Assertions and transitions cannot be empty.

### 13.3 Compiler architecture

The compiler is double; it transforms the CHILD code into the PTL specifications in two main parts:

- CHILD to TCOM compilation
- TCOM to PTL compilation

Then the prover can be invoked for PTL specifications validation

#### 13.3.1 CHILD to TCOM compilation
Parser, as an input, gets the file with the CHILD specifications and if there was no error found it creates a “Report.ok” file containing the parse process information. Otherwise (i.e. if there are some errors), it creates a “Report.error” file with the description of the error and the line number where it was detected. Only having the positive report the user will be allowed to continue with compilation. This option becomes possible because of the step-by-step menu enabling. At program start, the user is not authorized to make any actions on the program but only to open the source file or to create a new one and to edit its contents. After the program was saved (or the file was opened), “Parse CHILD” and “Statistics” menus are enabled (see Fig. 44).

And so on – the user can access the next task option only by successfully fulfilling the previous one.

Clicking the “Parse CHILD” the “yy.exe” file is being run. As soon as this is done by “system” method call, the user can see the black command prompt window for an instance. The essence of this executable is CHILD grammar checking.
As it was mentioned above, the program has an error pointing ability, i.e. if there is any error detected, the line, containing this mistake is highlighted with red, and its text becomes selected. The message box with error description and line number is popped up. This note is also copied to the “Messages” section of the application with the type the message: information, warning, or error.

Here is the example of error notification. As it is defined in the CHILD grammar, the first component must be the main one and its name must be “controller”. If this rule (or any other) is not fulfilled, the user is being prompted with the error notification (see Fig. 45).

If the source code was parsed successfully, the next option (“Compile CHILD to TCOM”) in the “Tools” menu becomes enabled.

![Figure 45: CHILD 2 TCOM Compiler Error Notification]

### 13.3.2 TCOM to PTL compilation

This part, like the previous one, consists of two stages – TCOM parsing (i.e. checking the grammar of the generated code) and compilation of TCOM into PTL (see Fig. 46).

The TCOM parser (“TCOMparser.exe”) and compiler (“TCOM2PTL.exe”) are called by the system function (like “yy.exe”) and so there will appear command prompt window for an instance when these files are started.
As it was described in the CHILD specifications part, the component may have a few (or none) sub-components. However, TCOM to PTL compiler can resolve only one level hierarchy, i.e. only the main component can include the sub-component and this contradicts to the CHILD basis.

The solution proposed was to split the TCOM program into a few programs according to the hierarchy in such way that it will contain only one level heritage (hierarchy) and the component, containing the other one in this case is called main.

![Figure 46: TCOM to PTL Compilation Part Structure](image)

For example, the main component Main_System includes the sub-components a and b; a in its turn includes c and d, and b includes e and f (see Fig. 47).
The result of the hierarchy splitting will give three files with the following components in them (each file is named after the main component in it) (see Fig. 48 a-c):

**Figure 47: Example of TCOM Program’s Hierarchical Structure**

**Figure 48(a): Result Of the Hierarchy splitting: “a.tcom”**

**Figure 48(b): Result Of the Hierarchy splitting: “b.tcom”**

**Figure 48(c): Result Of the Hierarchy splitting: “main.tcom”**
The algorithm is rather simple – at first, all the leaves are taken with their parents, and then the parents are taken with their own parents and so on...
The consistent compilation and validation of these files should give the proper answer, as if you would have had compiled and tried to validate this program as it is.

13.3.3 PTL specifications validation
If the both previous parts of the project were implemented as the windows application (MFC), this snippet is a Unix based one.
PTL specifications in our case can be checked for validity by the means of STeP - The Stanford Temporal Prover. This system consists of three main components:

§ **The Top-Level Prover** (This is where the top-level proof search is conducted, by applying simplification steps to the current goal. Tactics are also available here, to execute sequences of proof commands automatically).

§ **The Interactive Prover** (The interactive prover is based on a Gentzen-style sequent calculus, specialized to linear-time temporal logic. The proof search proceeds as usual, with logical rules that reduce the current goal to a number of sub-goals, and simplification steps that close or simplify the current goal. Tactics are available for automatically executing sequences of proof steps. A goal and asserted axioms are entered from STeP's top-level proof environment via Undo/Redo).

§ **Verification Diagrams and the Diagram Editor** (Verification diagrams are a visual representation of a proof; a well-formed verification diagram represents a set of verification conditions which are sufficient to establish a given formula. The diagrams can be constructed, loaded, edited, transformed, verified, used in a proof, saved, etc. by means of Diagram Editor).

13.4 Example
Using the same *soda factory* example seen in the previous parts:

13.4.1 CHILD Specification (translated from the Arts'Codes Diagrams)

```plaintext
component controller {
  external{
    in:  flagged signal BatF, BatS, EnterB, ExitB, Full, Sealed;
    out: flagged signal BleaveF, LeaveS, Fill, Seal;
      flagged signal Full;
```
sub-component: Fill_station:  
  StartFillStation();
  Seal_station: StartSealStation();
  Belt: StartBelt();

manager {
  local:
  state:
  conditions: Fill = BatF;
    BLeaveF = Full;
  Seal = BatS;
    BLeaveS = Sealed;

  transitions:
  assertions:
    Check_EnterB {
      Local []
      type: goal;
      position: pre;
      trigger: EnterB;
      assertion: EnterB;

      action {
        ExitB=1;
        Full=1;
        Sealed=1;
      }
    }
}

component Fill_station {
  external{
    in: flagged signal Fill;
    out: flag Full;
    flagged signal Full;
  }

  sub-component: 
  manager {
    local:
    state:
  conditions: doFillingFunction Until Full = Fill;
    always Full = Full;
  actions: FillingFunction() {}
    SetFull() { Full = 1;}

  transitions:
    when(gate:Full) then {
      set action: SetFull();
      set state: full;
    }

    state: ful l{
      set assertion: no_value;
      when (condition : Full == Fill) then {
        set action: FillingFunction();
      }
    }

  assertions:
Check_Fill {
    local{
        type:    goal;
        position: pre;
        trigger: Fill;
        assertion: Fill;
        action{
            later Full;
        }
    }
}

Check_Full{
    local{
        type:    goal;
        position: pre;
        trigger: Full;
        assertion: Full;
        action{
            always Full;
        }
    }
}

component Seal_station {
    external{
        in:    flagged signal Seal;
        out:    flag Sealed;
                flagged signal Sealed;
    }
    sub-component:
    manager {
        local:
        state:
        conditions:
        actions: SealingFunction() {}
                   SetSealed() {Sealed=1;}
        transitions:
        when(gate: Seal) then {
            set action: SetSealed();
            set state: sealed;
        }

        state: sealed {
            set assertion: no_value;
            when (condition: Sealed == Seal) then {
                set action: SealingFunction();
            }
        }

        state: Not_sealed {
            set assertion: no_value;
            when (condition: Sealed!=Seal) then{
                set action: SealingFunction() Until Sealed!=Seal;
                set state: sealed;
            }
        }
    }
}
assertions:
Check_Seal {  
  local{}  
  type: goal;  
  position: pre;  
  trigger: Seal;  
  assertion: Seal;  
  
  action{  
    later Sealed;  
  }  
}

Check_Sealed {  
  Loca l{}  
  type: goal;  
  position: pre;  
  trigger: Sealed;  
  assertion: Sealed;  
  action{  
    always Sealed;  
  }  
}
}

component Belt {  
  external{  
    in: flagged signal EnterB, BLeaveF, BLeaveS, Full, Sealed;  
    out: flagged signal BatF, BatS, ExitB;  
  }  
  protected: flag oneStep;  
  sub-component:  
  manage r{  
    local:  
    state:  
      conditions: Move Until BatF = EnterB;  
      Move Until BatS = BLeaveF;  
      Move Until ExitB= BleaveS;  
    actions:  
      Move() {}  
      SetBatF() {BatF=1;};  
      SetBatS() {BatS=1;};  
      SetExitB() {ExitB=1;};  
    transitions:  
      when(gate: EnterB) then {  
        set action: SetBatF();  
        set state: enteredFill;  
      }  
      when(gate: BleaveF) then {  
        set action: SetBatS();  
        set state: leftFill;  
      }  
      when(gate: BleaveS) then {  
        set action: SetExitB();  
        set state: exited;  
      }  
  }
}
state: enteredFill {
    set assertion: no_value;
    when (condition: BatF == EnterB) then {
        set action: Move();
    }
}

state: leftFill {
    set assertion: no_value;
    when (condition: BatS == BleaveF) then {
        set action: Move();
    }
}

state: exited {
    set assertion: no_value;
    when (condition: ExitB == BleaveS) then {
        set action: Move();
    }
}

assertions:
    Check_EnterB { local {}
        type: goal;
        position: pre;
        trigger: EnterB;
        assertion: EnterB;
        action {
            later BatF;
        }
    }

    Check_BLeaveF { local {}
        type: goal;
        position: pre;
        trigger: BLeaveF;
        assertion: BLeaveF;
        action {
            later BatS;
        }
    }

    Check_BLeaveS { local {}
        type: goal;
        position: pre;
        trigger: BLeaveS;
        assertion: BLeaveS;
        action {
            later ExitB;
        }
    }
}

13.4.2 TCOM Translation

COMPONENT Fill_station {

DECLARATIONS:
IN   boolean signal:  S_Fill;
    boolean signal:  S_Full;
    boolean signal:  S_do_FillingFunction;
OUT  boolean flag:  F_Full;
    boolean signal:  S_Full;
LOCAL boolean:  Full;
    boolean:  do_FillingFunction;
    boolean:  Fill;

OPERATIONS:
do_GetCurrentState:  GetCurrentState();
do_FillingFunction:  FillingFunction();
do_SetFull:  SetFull();

GOALS:
  when(S_Fill)
    then(later S_Full);
  when(S_Full)
    then(always S_Full);

BEHAVIOR:
  when(Full = Fill)
    then(do_FillingFunction);
  when(Fill)
    then(doFillingFunction Until Full);
  when( Full)
    then(always Full);
}

COMPONENT Seal_station  {
  DECLARATIONS:
  IN   boolean signal:  S_Seal;
    boolean signal:  S_sealed;
    boolean signal:  S_Not_sealed;
    boolean signal:  S_do_SealingFunction;
  OUT  boolean flag:  F_Sealed;
    boolean signal:  S_Sealed;
  LOCAL boolean:  sealed;
    boolean:  Not_sealed;
    boolean:  do_SealingFunction;

  OPERATIONS:
do_GetCurrentState:  GetCurrentState();
do_SealingFunction:  SealingFunction();
do_SetSealed:  SetSealed();

  GOALS:
    when(S_Seal)
      then(later F_Sealed);
    when(F_Sealed)
      then(always F_Sealed);

  BEHAVIOR:
    when(Sealed = Seal)
      then(do_SealingFunction);
    when(Sealed! = Seal)
      then(do_SealingFunction and sealed);
}

COMPONENT Belt {
  DECLARATIONS:
  IN boolean signal:  S_EnterB,
boolean signal: S_enteredFill;
boolean signal: S_do_Move;
boolean signal: S_leftFill;
boolean signal: S_exited;
boolean flag: F_oneStep;
boolean signal: S_EnterB,
S_Move,
S_BatF,
S_BleaveF,
S_BatS,
S_BleaveS,
S_Exit;

OUT boolean signal: S_BatF,
S_BatS,
S_Exit;

LOCAL boolean: enteredFill;
boolean: do_Move;
boolean: leftFill;
boolean: exited;
boolean: oneStep;
boolean: EnterB,
Move,
BatF,
BleaveF,
BatS,
BleaveS,
Move,
Exit;

OPERATIONS:
do_GetCurrentState: GetCurrentState();
do_Move: Move();
do_SetBatF: SetBatF();

GOALS:
when(S_EnterB) then(later S_BatF);
when(S_BLeaveF) then(later S_BatS);
when(S_BLeaveS) then(later S_Exit);

BEHAVIOR:
when(BatF = EnterB) then(do_Move);
when(BatS = BleaveF) then(do_Move);
when(ExitB= BleaveS) then(do_Move);
when(EnterB) then(Move Until BatF);
when(BleaveF) then(Move Until BatS);
when(BleaveS) then(Move Until Exit);
COMPONENT MAIN_MAINSystem { 
    DECLARATIONS: 
    IN boolean signal: S_BatF, 
        S_BatS, 
        S_EnterB, 
        S_ExitB, 
        S_Full, 
        S_Sealed; 
    OUT boolean signal: S_BleaveF, 
        S_LeaveS, 
        S_Fill, 
        S_Seal; 
    boolean signal: S_Full; 
    INCLUDES Fill_station, 
        Seal_station, 
        Belt; 
    GOALS: 
        when(S_EnterB) 
            then(S_ExitB=1 and S_Full=1 and S_Sealed=1); 
    BEHAVIOR: 
        when(S_EnterB) 
            then(S_ExitB=1 and S_Full=1 and S_Sealed=1); 
} 

13.4.3 PTL translation

SPEC

variable F_Full, F_Sealed, F_oneStep :bool Rigid
variable S_Fill, S_full, S_do_FillingFunction, S_doFillingFunction, S_Full, S_Seal, 
    S_sealed, S_do_SealingFunction, S_Not_sealed, S_Sealed, S_EnterB, S_BleaveF, 
    S_BleaveS, S_enteredFill, S_do_Move, S_leftFill, SExited, S_Move, S_BatF, 
    S_BleaveF, S_BatS, S_BleaveS, S_Exit, S_ExitB, S_LeaveS :bool Flexible

macro Fill_station_Goal :bool where Fill_station_Goal = 
    ( 
        []((S_Fill) --> (<>S_Full)) /
        []((S_Full) --> (<>S_Full)) 
    )

macro Seal_station_Goal :bool where Seal_station_Goal = 
    ( 
        []((S_Seal) --> (<>F_Sealed)) /
        []((F_Sealed) --> (<>F_Sealed)) 
    )

macro Belt_Goal :bool where Belt_Goal = 
    ( 
        []((S_EnterB) --> (<>S_BatF)) /
        []((S_BleaveF) --> (<>S_BatS)) /
        []((S_BleaveS) --> (<>S_ExitB)) 
    )

macro Main_MAINSystem_Behavior :bool where Main_MAINSystem_Behavior = 
    ( 
        []((S_EnterB) --> (S_ExitB=1 /
            S_Full=1 /
            S_Sealed=1)) 
    )
macro Main_MAINSystem_Goal :bool where Main_MAINSystem_Goal =
(
    []( S_EnterB) --> (S_ExitB=1 \ S_Full=1 \ S_Sealed=1 )
)

PROPERTY:
(Fill_station_Goal \ Seal_station_Goal \ Belt_Goal \ Main_MAINSystem_Behavior) ===> Main_MAINSystem_Goal
This part presents a case study in order to show the whole methodology's potentials. First the ACE editor, which was developed for this thesis, is presented, followed by the Statement of the needs of the "washing machine" problem, the example design and the compiled code. In order to shorten the presentation, only essential parts of the design and the executable code were brought.

Chapters for this part:

14 Arts’Codes Components Editor (ACE)
15 The washing machine case study
In order to implement the method and to facilitate the design, a JCT (Jerusalem College of Technology) grant was utilized for the developing of the Arts’Codes Components Editor (ACE). This is a graphical interactive editor. This development took three years incorporating many revisions and improvements. Two software engineers continue to working on the final refinements.

The editor defines three catalog lists: one for the components, the second for the virtual-devices and the third for the physical-devices.

Applications are defined as a set of component's socket and virtual-devices' sockets, while connections are traced between the virtual-devices and the component socket.

Components and virtual-devices are dragged graphically from their catalog lists straight to the suitable sockets.

Physical-devices are dragged straight into already inserted virtual-devices.

By pressing the corresponding button, the graphical design is translated to textual C++ code, which can be compiled and executed. This option exists also for the compilation to the CHILD textual language.

There are three different graphic windows for the component editing:

- Editing of the Static Architecture
- Editing of the dynamic Manager
- Editing of the dynamic Guard

Assertions are edited textually in a dialog window.

Virtual-devices editing is made in the same way as components.

Restrictions on graphic editing were defined in order to avoid errors and undesirable code, e.g. only one arrow can reach a component output, etc.

The ACE editor helped much in the definition of the methodology.

Screens views of the ACE editor can be seen in appendix B.
The washing machine case study

15.1 Statement of needs (SON)
The SON details the needs and the required technical characteristics for a Control Application (CA) of a specific Washing Machine (WM).

15.1.1 The needs
The CA will be able to execute a list of WP, using a series of processes The user selects the details of the WP at the Control Panel (CP).
Sets of small screens are supplied for the status display of the system.
The list of input/output signals, the internal devices control, and the access to the data registers are supplied in the Washing Machine Data Sheet (WMDS) document.

15.1.2 The Washing Programs
The CA will be able to execute a list of WP, which are composed by a series of processes.
The washing programs available for the user selection are:

§ Pre-Wash
§ Wash
§ Delicate.

The participating Processes are:

§ Filling
§ Washing
§ Rinsing
§ Centrifugation

Processes Description

§ Filling: the Tap fills the Drum until the Drum is full.
§ Washing: the motor rotates the Drum clockwise and against at slow speed. Each rotation takes 5 seconds. The Heater is turned on/off depending on the water temperature.
§ Rinsing: The Pump is activated; the Motor rotates as in the Washing process, and Tap fills the Drum as in the Filling process.
§ Centrifugation: The Pump is activated and the Motor rotates in clockwise direction at fast velocity.

**Warning:** Heater activation in an empty drum might burn the heater device.

**Program Description**

§ Pre-Wash: Filling until the Drum is full, Washing during 180 seconds, Rinsing 60 seconds and centrifugation for 120 seconds.
§ Wash: Filling until the Drum is full, Washing during 90 seconds, Rinsing 60 seconds and centrifugation for 120 seconds.
§ Delicate: Filling until the Drum is full, Washing during 60 seconds, Rinsing 30 seconds and centrifugation for 10 seconds.

**Note:** When the Door is open the current process pauses. The execution will be resumed after closing the door.

15.1.3 Control Panel and display

The control panel provides a set of two buttons for water temperature selection. The buttons increase/decrease the temperature value at steps of 5 centigrades, and shows the temperature's value at the temperature's display.

Another set of increase/decrease buttons is provided for Washing Program selection. The display may show the selected WP.

A single display is supplied for WP advancement, by displaying the current process.

The WP starts when the Start button is pressed.

The CP is locked during the WP execution.

15.1.4 Physical definitions

**Ranges**

Drum volume = 5 liters.
Thermostat Range: 30° - 95°.

**Processor and Devices interface**

The internal devices are connected to the processor by a connector of 21pins. The pins are used for input/output signals and for read/write of data.
Each change at any input pin produces a hardware interrupt at the processor, and it is handled by an interrupt subroutine.

**Note:** Signals are assigned to a single pin. For data R/W the address must be specified at the address’s pins, before R/W through the value pins.

**Figure 49: The washing-machine physical devices**
15.2 Design
This section presents the design of the washing-machine case-study, according to the defined four phases of the process development (see section 5.1.5).

15.2.1 1st phase: Environment wrapping
The first design stage is to define the application's environment, and how the environment connects to the application. In Fig. 50 all external devices were represented by the suitable virtual-devices (VD). Note that some of the virtual-devices, have oblique lines inside; this represents that no physical-device have implemented the virtual-device abstract methods (see section 5.1.5). Also connections were made between the reactive-kernel-root component and the surrounding VDs.

15.2.2 2nd phase: Hierarchic building of the component structure
Now we start building the root-component WRMK (in a top-down approach), which is so-called the reactive-kernel-root (see Fig. 51). The inputs and outputs are suitable to the environment.
The responsibilities of this component are split: some of them take the dynamic Manager, and the other responsibilities are taken by the two defined sub-components ProgCtrl and PanelCtrl. The inputs/outputs are divided between the Manager and the sub-components according to their responsibilities:

1. The ProgCtrl that is responsible for the washing programs execution receives the WaterQuant and WaterTemp inputs; and is responsible for the control of the TapLED, HeaterLED, PumpLED, TapOn, HeaterOn, PumpOn, MotorOn, Direction, Speed and Calibration outputs.

2. The PanelCtrl which is responsible for the user interface, receives and controls all inputs/outputs concerning to the buttons and displays.

3. The Manager receives only the e_start and DoorOpen inputs for high-level application control. These two inputs enable macro-control for starting, stopping, pausing and resuming the washing programs.

Shared variables were also defined in order to connect data between the Manager and the sub-components. E.g. TempRequired a PanelCtrl output, is connected to ProgCtrl as an input.

![Figure 51: The reactive-kernel-root component](image)
Then the architecture (structure) definition of the *reactive-kernel-root* is decomposed, and the two sub-components are designed (see Fig. 52 and 53) where each of them split their responsibilities to other sub-components, defining in such way a third level for the components hierarchy. Here we see that the designer has chosen the following priorities of execution inside the root component: P#0 (highest) for the Manager, P#1 for the ProgCtrl subcomponent and P#2 for the PanelCtrl component.

**Figure 52: The ProgCtrl component**

In Fig. 51 we can see only inputs and outputs which were delegated to this component. This component splits its inputs/outputs to its Manager and to four sub-components:

1. The *Filling* sub-component is responsible to assure the suitable water-level in the drum. For this it receives the *WaterQuant* and the *DrumCapacity* (which was internally defined) as inputs; and the control of the *TapOn* output.
2. The *Thermos* sub-component is responsible of the water temperature, and is responsible for *WaterTemp*, *TempRequired* and *HeaterOn*.

3. The *Pump* sub-component it just responsible for the *PumpOn* output.

4. The *MotorCtr* has to control the motor by the *MotorOn*, *Direction* and *Speed* outputs.

5. The Manager is responsible for the four processes of the washing programs, therefore it receives the user's selected washing program, and controls the rest of the outputs.

Inside of each sub-component appear the name, type and priority. Note that Filling and Thermos sub-components are defined with the same type of component, the *Regulatr*. The *Regulatr* is thus reused and being inserted twice in different sockets (The *Regulatr* component definition is presented in Fig. 54a and 54b).

![Figure 53: The PanelCtrl component](image-url)
In Fig. 53 the PanelCtrl is split in two more sub-components, one is responsible for the input-buttons and the second for the output-displays.

In Fig. 54 two of the third level components are presented: the *MotorCtr* and the *Regulatr*.

![Diagram](image)

**Figure 54: Third level components**

### 15.2.3 3rd phase: Behavior Molding

This phase has to define the Manager, which describes the normal behavior of each component, using a component-oriented Statechart (named CoST). The bottom-up approach may be used with no compromise.

In Fig. 55 the general *Regulatr* type behavior is presented which is instantiated for two subcomponents: *Filling* and *Thermos* (see Fig. 52).

Figures 54(b) and 55 represents the static and the dynamic view of the *Regulatr* type component, therefore both figures have to be referenced when considering this component instantiation.

The Manager has two states: the *Reached* state represents that the *Quantity* (input) is greater or equal than the *Reference* (input), and the *NotYet* state represents the opposite. The *Enough* and *Lack* conditions test the inputs, and the *Activate* and *Deactivate* reactions set/reset the *Activate* output. The "I" gate, which is the default *entry-gate*, selects the *Reached* state as the default state.
When the component is left via the Pause exit-gate the Deactivate action is performed. The Resume gate is not enabled because it has no transitions.

Figure 55: The Regulatr Manager

Figure 56: The Motor Manager
The Motor component in Fig. 56 is more complicated; it has two entry-gates defined by the programmer for to different modes of operation: "W" for the Washing and Rinsing processes and "F" for the Centrifugation process. Each of these entry-gates enters different states in order to perform different actions according the requested mode.

Local variables are defined in the top-left corner.

When the component is left via the Pause exit-gate the status is saved and then the motor is deactivated (StoreState reaction). The motor status is restored via the Resume entry-gate after a delay. This delay is expressed by connecting the "R" gate to the Delay4 state, which transits to the History pseudo-state after a timer condition (performing the RestoreState action before reaching the History pseudo-state).

The ProgCtrl component Manager in Fig. 57, expresses the powerful tool of parallel sub-component activation. It has four states corresponding to the four processes of the washing programs.

The first process (FillDrum state) has just to fill the drum, and this is done by activating the Filling sub-component until the drum is full (EnoughWater condition).
The second process (Wash state) has to activate the motor in wash mode, this is performed by activating the Motor sub-component through the "W" gate; and in parallel the Thermos sub-component is also activated in order to acquire the requested water temperature. This process finishes when the timer rises (EnoughWash condition) and then it transits to the next state.

Normal termination of the Manager exits through the "T" gate, and critical termination exits through the "K" gate.

In Fig. 58 the root component Manager is shown. In this Manager the powerful possibilities of hierarchy are shown in automata, in addition to inter-level transitions.

The priorities of the transitions are chosen by the designer and indicated by a number on the arrow beginning. Note that the priority of the transition that returns from Washing to Idle is negative (hard to see, but it is), which means that it has a high-priority. Note also that all target-transitions are leaf-states, but their sources may be any state; the Washing state for example that is not a leaf-state, serves only as a transition's source-state but not as a target-state.

When the door is opened (condition [OpenDoor] true) the two sub-components in DrClosed state are paused because the automaton transits to DoorOpen state which it doesn't activate any sub-component. This transition provokes these sub-components: PanelCtrl and ProgCtrl to be paused via their Pause exit-gate, and consequently also all their sub-components which are descendent of them.

Figure 58: The reactive-kernel-root Manager
After the return to the DrClosed state through the condition [CloseDoor], all the sub-components will resume according to their Resume entry-gate.

The normal termination of ProgCtrl triggers a transition back to the Idle state; and an abnormal termination exits the component via the "K" exit-gate.

15.2.4 4th phase: Exception Handling

The exceptions handling is made to assure correct execution, or to alert when it is not achievable.

In Fig. 57 in state Wash two assertions were inserted, one to assure correct behavior (blue) and the second (red) to alert for an abnormal exception. Both, when arising enters their suitable entry-gates in the Guard (see Fig. 59). They return to the Manager after their handling, to a suitable Manager entry-gate ("I", "R" or "K"; in the right side of Fig. 59).

For example the critical (red) assertion CWLFH (Critical Water Level For Heating) rises when the water-level is not enough for the heater activation, which can then be damaged. The Guard, passing to the FillDrum state attempts to fill it again, and to resume ("R" gate) the washing program. After few attempts it enters the Manager via the Critical ("K") gate which terminates the component in an abnormal status (see Fig. 57).

![Figure 59: The ProgCtrl Guard](image-url)
15.2.5 Virtual and Physical devices

The Virtual-devices are treated similarly as components defining inputs/outputs and behavior. But the virtual-devices inputs/outputs concern only the reactive-kernel-root. The real original I/O are performed via abstract conditions/actions which are implemented by physical-devices (see section 5.1.5).

The button virtual-device is shown in Fig. 60. It has one output which is the event sent when the button is pressed. This event is sent by the Report reaction in the VD automaton. ButtonPressed and ButtonReleased conditions are abstract, and they must be implemented by a specific physical-device.

![Figure 60: Button Virtual Device (static and dynamic views)]

15.3 Code generation

Here we bring the direct translation of the ReactiveKernelRoot component: statically and dynamically.

15.3.1 The ReactiveKernelRoot structural translation

The following is the translation of the static diagram of Fig. 51 to code.

class ReactiveKernelRoot: public ComponentT {
  //static section
  protected:
    //external inputs list
    int* WaterQuant;
    int* WaterTemp;
bool* DoorOpen;
bool* e_start;
bool* e_TempInc;
bool* e_TempDecr;
bool* e_ProgInc;
bool* e_ProgDecr;

//external outputs list
bool* TapLED;
bool* HeaterLED;
bool* PumpLED;
bool* TapOn;
bool* HeaterOn;
bool* PumpOn;
int* Calibration;
bool* MotorOn;
int* Direction;
int* Speed;
String* TempTxt;
String* ProgTxt;
String* ProcTxt;

//shared variables list
int TempRequired;
int Program;
String Process;

class ReactiveKernelRootT {
public:
  ReactiveKernelRootT();
  void Set(int*_WaterQuant, int*_WaterTemp, bool*_DoorOpen,
           bool*_e_start, bool*_e_TempInc, bool*_e_TempDecr,
           bool*_e_ProgInc, bool*_e_ProgDecr,
           bool*_TapLED,
           bool*_HeaterLED, bool*_PumpLED, bool*_TapOn,
           bool*_HeaterOn, bool*_PumpOn, int*Calibration,
           bool*_MotorOn, int*_Direction, int*_Speed,
           String*_TempTxt, String*_ProgTxt, String*_ProcTxt);

protected:
  void PauseChilds();

private:
  void Call(int id);

  //sub-components list - internal connections
  ProgCtrl ProgCtrl;
  PanelCtrl PanelCtrl;

  //dynamic section
public:
  void Idle();
  void Washing();
  void DoorOpen();
  void DrClosed();
}
};
15.3.2 The ReactiveKernelRoot behavior translation

The following is the translation of the dynamicm Manager of Fig. 51 to code.

```cpp
void ReactiveKernelRoot::Set(int *_WaterQuant, int *_WaterTemp, bool *
*DoorOpen,
    bool *e_start, bool *e_TempInc, bool *e_TempDecr,
    bool *e_ProgInc, bool *e_ProgDecr,
    bool *TapLED,
    bool *HeaterLED, bool *PumpLED, bool *TapOn,
    bool *HeaterOn, bool *PumpOn, int *Calibration,
    bool *MotorOn, int *Direction, int *Speed,
    String *TempTxt, String *ProgTxt, String *ProcTxt)
{
    WaterQuant = _WaterQuant;
    WaterTemp = _WaterTemp;
    DoorOpen = _DoorOpen;
    e_start = _e_start;
    e_TempInc = _e_TempInc;
    e_TempDecr = _e_TempDecr;
    e_ProgInc = _e_ProgInc;
    e_ProgDecr = _e_ProgDecr;
    TapLED = _TapLED;
    HeaterLED = _HeaterLED;
    PumpLED = _PumpLED;
    TapOn = _TapOn;
    HeaterOn = _HeaterOn;
    PumpOn = _PumpOn;
    Calibration = _Calibration;
    MotorOn = _MotorOn;
    Direction = _Direction;
    Speed = _Speed;
    TempTxt = _TempTxt;
    ProgTxt = _ProgTxt;
    ProcTxt = _ProcTxt;
    ProgCtrl.Set(WaterQuant, WaterTemp, Speed, Direction, MotorOn, PumpOn,
                 HeaterOn, TapOn, TapLED, HeaterLED, PumpLED, &Process, &TempRequired,
                 &Program, Calibration);
    PanelCtrl.Set(e_ProgInc, e_TempDecr, e_TempInc, &Process, ProcTxt, ProgTxt,
                  TempTxt, e_ProgDecr, &TempRequired, &Program);
}
```

```cpp
void ReactiveKernelRoot::Call(int id)
{
    switch (id)
    {
    case 0:
        None();
        return;
    case 1:
        Idle();
        return;
    case 2:
        Washing();
        return;
    case 3:
        DoorOpen;
```
return;
case 4:
  DrClosed();
return;
}

void ReactiveKernelRoot::SetCurrentStateLevels()
{
  ComponentT::SetCurrentStateLevels();
  switch (currentState) {
    case DoorOpen:
      currentStateLevel[0] = ReactiveKernelRootState.Washing;
      currentStateLevel[1] = ReactiveKernelRootState.DoorOpen;
      break;
    case DrClosed:
      currentStateLevel[0] = ReactiveKernelRootState.Washing;
      currentStateLevel[1] = ReactiveKernelRootState.DrClosed;
      break;
  }
}

void ReactiveKernelRoot::PauseChilds()
{
  ProgCtrl.Paused();
  PanelCtr.Paused();
}

void ReactiveKernelRoot::DoBetweenSteps()
{
  ProgCtrl.DoBetweenSteps();
  PanelCtr.DoBetweenSteps();
}

void ReactiveKernelRoot::Gates()
{
  if (exitGate == Inactive)
  {
    switch (entryGate)
    {
      case Init:
        DoInit();
        PanelCtr.SetEntryGate(PanelCtrGate.U);
        currentState = ReactiveKernelRootState.Idle;
        status = Active;
        break;
      case Resume:
        DoResume();
        break;
    }
    entryGate = Inactive;
  }
  switch (exitGate)
  {
    case Terminate:
      DoTerminate();
      break;
  }
case Pause:
  DoPause();
  break;
+case Exception:
  DoException();
  break;
+case Critical:
  DoTerminate();
  break;
}

void ReactiveKernelRoot::Idle()
{
 try
  {PanelCtr.Run();
   if (*e_Start & & !*DoorOpen)
     {ProgCtrl.SetEntryGate(ProgCtrlGate.Init);
       PanelCtr.SetEntryGate(PanelliCtrGate.L);
       currentState = ReactiveKernelRootState.DrClosed;
       throw false;
     }
  } catch(bool exception)
    {PanelCtr.Paused();
    }
}

void ReactiveKernelRoot::Washing()
{
 try
  {
   if (*e_Start)
     {*
     *Process = "Idle";
     PanelCtr.SetEntryGate(PanelliCtrGate.U);
     currentState = ReactiveKernelRootState.Idle;
     throw false;
     }
   if(CallLowerLevel(1))
     throw false;
  } catch(bool exception)
    {PanelCtr.Paused();
     ProgCtrl.Paused();
    }
}

void ReactiveKernelRoot::DoorOpen()
{
 try
  {if (*DoorOpen == false)
   {

219
currentState = ReactiveKernelRootState.DrClosed;
PanelCtr.SetEntryGate(PanelCtrGate.L);
throw false;
}
} 
catch(bool exception)
{
}
} 
void ReactiveKernelRoot::DrClosed()
{
try
{
switch (ProgCtrl.Run()) {
    case ProgCtrlGate.Terminate:
        PanelCtr.SetEntryGate(PanelCtrU);
        currentState = ReactiveKernelRootState.Idle;
        throw false;
    case ProgCtrlGate.Critical:
        exitGate = ReactiveKernelRootGate.Critical;
        throw false;
    PanelCtr.Run();
    if (*DoorOpen == true)
    {
        currentState = ReactiveKernelRootState.DoorOpen;
        throw false;
    }
} 
catch(bool exception)
{
    PanelCtr.Paused();
    ProgCtrl.Paused();
} 
}

15.4 Execution
In order to run the application three different threads (as described in section
11.3) must run simultaneously, one on each processor (another option is to run
all the threads in the same processor). These concurrent threads are
synchronized by the middle processor, the Synchrony Co-Processor (see fig. 41).
Therefore, each of these threads are downloaded to their suitable processor.
In order to execute the application, these threads use the data-structures
programmed in C++ language (see previous section), which were compiled and
downloaded to the shared memory.
Even the case study presented is a control system application, for which
Arts'Codes was developed, but really Arts'codes is suitable for a broad range of
embedded systems applications.
Part VI

Summary

This part discusses the research contribution, and powerful features for further researches.

Chapters for this part:

16  Conclusions
17  Main Contributions
18  Further Researches
Conclusions

The main goal of this thesis was to propose "a clear and robust embedded real-time application methodology for the design and the code generation, and to propose a light-code execution platform" (see section 3). This main goal was split into six sub-goals. In this section we present the thesis conclusions according to the predefined sub-goals, each one in a separate section, prefacing the section with the sub-goal definition in italics font-style.

16.1 Natural and formal design

Finding a formal method to design real time applications in a clear and simple way, based on a natural cognitive human's approach, including modularity and encapsulation.

The natural Arts'Codes approach, where the cognitive efforts are taken with care, achieved its goals, restricting all the design only into two types of interleaved diagrams. The readability and accuracy of the static architecture and dynamic behaviors design are strong features. This can be seen at the evaluation test, where Closeness of Mapping, Hard mental operations and abstraction were detected as the stronger cognitive factors (see sections 6.7 and 6.8.3).

The main topic that has to be researched and enhanced, in this method, is the easy modification feasibility.

Arts'Codes gives emphasis to the cognitive efforts factor, eliminating even powerful notations elements, preferring simplicity, fewer notation and clear design.

Arts'Codes also provides clear and natural phases of development, where the diagrams structure is used as an aid for the mental image.

The component approach enhanced the abstraction and is used also for the concurrency notation, when appearing e.g. in the same automaton state.

The following are features which contributed to the readability and simplicity of the design model:

§ Strong isolation between the environment and the application.

§ Connections between components are fully described and traced.
Association and composition links have well-defined bounds.

Normal and excepted behaviors are separated.

Priorities are not attached to events, but they are attached locally to a specific transition or component.

Concurreny and composition have a strong cognitive notation.

Strong and well-defined connection between design and execution.

16.2 Visual Programming Language abilities

Defining translation rules, to automatically compile the design diagrams into a validation formal language and into a light but robust code.

A formal representation was provided for the diagrams, including the translation into the CHILD textual language which allows lexical and semantical testing, and translation into PTL Temporal Logic equations which allow proving the consistency of the components' architecture, goals and behavior.

Clear rules were defined for the design diagrams compilation into the light OO Arts code. The model of the design was preserved in the executable code, so that it assures the correctness.

16.3 Full and robust executable code

Finding a type of code enabling expression of: parallelism, synchronization, events, conditions and clocks, with emphasis on robustness.

This code is based on the parallel-automata concepts which is a faithful image of the design diagrams and allows a sure execution. Furthermore the component-oriented approach together with the time-granulation principle and the restricted synchrony hypothesis implementation of the execution platform, fulfill this sub-goal.

16.4 Full and robust execution platform

Building a new type of execution platform, which allows executing the mentioned code. This platform must support implicit synchronization and robust mechanism (transparent to the programmer), in order to enhance the robustness with a minimum programmer effort.

Two versions were proposed: one in software (parallel automata and threads oriented) and one in hardware (based on a tri-processor model).

The software version can be run in a wide list of embedded platforms.

The following principles (in building these platforms) contributed to fulfill this sub-goal:
§ Environmental Synchrony Hypothesis
§ Full isolation between environment-interaction and program-logics
§ Concurrent and Sequential Run interleaving
§ Use of a strong formal logic execution based on the parallel automata model

16.5 Fast and known response time

Optimizing the execution platforms to assure a known and reliable response time.

The maximum response time is two steps of the synchronous clock. This was assured by the hybrid synchronous and multi-threading platform. Also because of the Synchronous and multi-threading combination, the system will not be "stuck" on infinite loops at any reaction. This is avoided by defining each reaction which contains a loop as a heavy-reaction, which means that it will run in the multi-threading platform. Any heavy-reaction may "stick" only the heavy-transition itself, but not other components or other heavy-reactions.

16.6 Evaluation

Presenting clear metrics to evaluate the proposed methodology.

The design model was evaluated twice, using academic tools, with clear trends for the different cognitive-factors. The results in both tests were consistent and suitable, receiving similar results.

For the execution platform, the first attempt of a hardware platform was built, giving primary results.

For this thesis continuation, a second and more complex hardware platform is now in the way of being built, and several case-studies are being designed to be run on the software and hardware platforms.
Main Contributions

17.1 Design Contribution: Codes (Component Oriented DESign)

The following list contains the component oriented design features proposed in this thesis:

§ The global design approach is to fit the human cognitive model, and not as the popular design methods which describe the system suitable to the platform/language model.

§ Homothetic design: as in nature, all the levels are designed in a similar format, with similar rules and notation and this gives a new kind of composition definition.

§ Explicit Interleaving of architecture and behavior of components: Like in the creation, at each level of architecture we see only independent behaviors (with their various states: implemented as reactive-cells in our system) which are in fact connected to a global unified model/hierarchy.

§ Re-usable types of components: using the Sockets model, we have a clear graphical form of instantiation.

§ Sub-component activation: sub-components are activated in the Managers in the Component oriented Statechart (CoSt) formally represented in a powerful model: the parallel automata FSM.

§ Sub-component connections through Gates: this FSM-based hierarchy model is connected via gates, forming a unified well-modularized structure.

§ Behavioral model using Manager/Guard: this allows normal and exception handling of behavior; this assures the robustness of the execution.

§ In-state Assertions: this allows a well-defined and focused exception handling.

17.2 Execution model Contribution: Arts (Automata-based Real-Time System)

§ Possibility of Automatic validation of the components design consistency
§ Automatic translation of the graphic design diagrams in executable light code based on a parallel automata FSM model and improved reactive-cells.
§ Introducing a restricted Synchronous hypothesis of execution which avoids the causality problem.
§ Full isolation between environment-interaction and behavioral-logics: using a tri-processors model.
§ Synchronous and Multi-threading platforms interleaving: using a hybrid platform.
§ Execution platform with built-in features assuring the correctness and robustness of execution and avoiding conflicts.
§ Light code: no platform overhead, almost just compiled code.
§ Use of a Leaf states flat implementation of the automaton in the light-code: which simplifies the handling of hierarchic automata.
§ Improved Reliability: the reliability is improved by adding a predefined assertion for step overflow; and updating the clocks according to this overflow.

17.3 Arts’Codes Components Editor
In order to implement the method and to facilitate the design, the Arts’Codes Components Editor (ACE) was developed (see chapter 14). This is a graphical interactive editor. When the design is ready, by pressing the corresponding button, the graphical design is translated to textual C++ code, which can be compiled, downloaded to the tri-processor platform and executed.
Further Researches

The Arts'Codes method opens wide options of extensions. This work asserts the principles and the rules of the game, it just gave some tools, the research just begun.

A list of innovations/aids was proposed after finishing this final report, in order to continue the research.

18.1 Components polymorphism

Components represent some pattern of a black-box which has an internal structure and a behavior. This can be reused by inserting it in different sockets, instantiating it to work in the same manner for different inputs/outputs.

But it seems that polymorphism can be researched concerning components, creating a wide form of instantiation using Object-Oriented ideas like inheritance or templates.

Inheritance is already implemented in this work for virtual-devices, in order to adapt the virtual-device pattern to a specific physical device. This is by implementing abstract reactions. This idea could be used also in components. Inheritance could support also the automaton extension, like in ROOM [28], and not just reaction's implementations.

In summary component's polymorphism is really a wide field for future research.

18.2 Arts'Codes pre-defined design patterns

The art of reuse demands predefined solutions for several common modules. This approach promotes agile development based on reliable pieces of code.

Also in Arts'Codes this approach must be adopted. Arts'Codes is reusable according to its definition, and therefore increasing the reuse degree will exhibit its feasibility.

The research has to detect the common modules, define their input and outputs and implement them.

The polymorphism research results have to be implemented on these patterns.
18.3 Tri-processor platform

The conclusions of the developed hardware platform were taken into account and a new specification list of requirements was redacted for the next hardware platform generation based on FPGA.
This platform will be implemented using VHDL language, and it will be faster and support bigger programs.
The development is on the way.
The new specification includes:
- 3 processors with 10 MIPS speed,
- having a block of 64-128 kb shared memory,
- each one 256-512 kb private memory,
- non-volatile memory,
- analog and digital I/O, and
- 3 timers connected each one to the processor's interrupt pin.

18.4 Specific C++ features translation to ANSI C

As we stated it in section 11 the design is compiled to Object-Oriented based code, in C++. Unfortunately embedded systems almost don't support C++.
Converting OO code into the well-known ANSI C standard is crucial in order to enable a wide range of processors to be suitable for the Arts'Codes execution.
This conversion is not trivial, but obviously applicable and needed.
Bibliography

[1] whatis.techtarget.com/definition
[14] ai-depot.com/FiniteStateMachines/FSM-Background.html#1


  http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm
  http://bdn.borland.com/article/0,1410,31863,00.html
[71] wwwilogix.com
[74] Z. Manna and the STeP group,"STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems", 8th Intern. Conf. on Computer-Aided
http://www-step.stanford.edu/


Appendices

Appendix A  Execution Platform Classes
Appendix B  ACE screen views
Appendix C  CHILD syntax rules
const int MaxStates = 15;
const int TotalHeavyReactions = 10;

enum ComponentStatus {NotActive, Active, Suspend};
enum ComponentModes {Manager, Guard};
enum PredefinesGates {Inactive=-7, Init, Resume, Pause, Terminate, Exception, Critical};
enum TimeUnit {millisec, sec, min, hour};
enum HeavyReactionStatus {Idle, Started, Running, Finished, OutputReported};

#define Null 0
#define none 0

const unsigned int StepInMillisecs = 200;
const unsigned int MicroStepInMillisecs = 100;

class String {
    char str[80];
public:
    String();
    String(char *str);
    String operator=(String _s);
    String operator=(char *str);
    String operator=(int _n);
    bool operator==(char *str);
    bool operator==(String _s);
    char *GetString();
};

class Clock {
private:
    TimeUnit timeUnit;
    unsigned long int counter;
    bool enabled;
    bool overflow;
    unsigned long int ConvertUnitToMillisecs(unsigned long int t);

public:
    Clock();
    void SetTimeUnit(TimeUnit _timeUnit);
    void Start(unsigned long int t0);
    void Advance();
    bool Reached(unsigned long int target);
    void Freeze();
    void Unfreeze();
    bool Overflowed();
};
class AssertionT { //interface
protected:
    bool critical;
    int initialState;
    String name;
    virtual bool Trigger();
    virtual bool Assertion() = 0;
    virtual void DoBetweenSteps();
public:
    virtual void Restart();
    bool Test();
    int GetInitialState();
    char *GetName();
    // virtual void Set(...);
};

class HeavyReactionT {
private:
    HANDLE hThread;
public:
    HeavyReactionStatus status;
    HeavyReactionT();
    void Start();
    void Terminate();
    void Handle();
    bool CanReachTargetState();
protected:
    void ResetWrapper();
    bool RunHeavyReaction();
    LPTHREAD_START_ROUTINE lpStartAddress;
    // virtual void Set() = 0;
public:
    virtual void GetInput() = 0;
    virtual void WriteOutput() = 0;
    virtual DWORD WINAPI Run(LPVOID dummy) = 0;
};

class HeavyReactions {
    HeavyReactionT *heavyReaction[TotalHeavyReactions];
    int free;
public:
    HeavyReactions();
    void Add(HeavyReactionT * _heavyReaction);
    void Handle();
};

class ComponentT { // attributes
protected:
    int currentState, currentStateLevel[3];
    int history;
    AssertionT *currentAssertion;
    int entryGate;
int exitGate, lastExitGate;
ComponentStatus status;
ComponentModes mode;
private:
  bool visitedState[MaxStates];
  // methods
protected:
  virtual void Gates()=0;
  virtual void SetCurrentStateLevels();
  virtual void PauseChilds();
  virtual void Call(int id)=0;
  bool CallLowerLevel(int level);
  void None();
  bool TestAssertion(AssertionT *AssertionP);
  void DoPause();
  void DoResume();
  void DoTerminate();
  void DoException();
  void ReturnToManagerEntryGate(int _entryGate);
  bool HeavyReactionsNoMoreNeeds();
public:
  ComponentT();
  int Run();
  void Paused();
  void SetEntryGate(int gate);
  virtual void DoBetweenSteps(); // timer advance, events reset, etc.
static HeavyReactions heavyReactions;
  // virtual void Set(...) = 0;
};

class VirtualDeviceT {
    // attributes
private:
  bool visitedState[MaxStates];
  unsigned int internalCycle;
  unsigned int internalTime;
protected:
  int currentState;
  ComponentStatus status;
  int timesOfExecPerSecond;

    // methods
protected:
  virtual void Call(int id)=0;
  void None();
public:
  VirtualDeviceT(int timesOfExecPerSecond);
  void Run();
  virtual void DoBetweenSteps(); // timer advance, events reset, etc.
  virtual void GetAndPut()=0;
    // virtual void Set(...) = 0;
};
Appendix B

ACE screen views
Appendix C

CHILD syntax rules

http://www.cc.jct.ac.il/~a_hay/CHILDSyntaxRules.pdf