

Technical Report/Relatório Técnico

**A COMPARISON AMONG FIVE LANGUAGES
FOR DIGITAL SYSTEMS SYNTHESIS**

P.B.Alves, O.A.C.Macedo, N.Marranghello, A.C.R.Silva

TR 2002-004-IN

A comparison among five languages for digital systems modeling

P. B. Alves, O. A. C. Macedo, N. Marranghello, A. C. R. Silva

1 Introduction

When developing a system one has to describe its behavior, and possibly some of its structures to serve as a reference to the subsequent design phases. For such a description a natural language is generally used. This means that no formalism, and possibly some ambiguity is present in the initial description. The major advantage of such an informal specification is its readability and understanding.

Nevertheless it is necessary to formalize the initial description because of, at least, two reasons: (a) the need to document the system in a standard and ambiguity-free language; (b) to allow for a computational treatment of the description.

Besides being a way of communication among designers, documentation is also used to store information for future use such as maintenance and upgrading.

The computational treatment consists in simulating the behavior of a system before its implementation. A formal, possibly mathematical, analysis of the properties of the system is done to prevent it from undesirable situations. System synthesis can also be done automatically with of computational tools.

A formal modeling language is chosen according to the characteristics inherent to the system. The purpose of this paper is to compare five languages used for system modeling. The comparison is based on parameters common to a wide variety of systems. The parameters are: sequential and concurrent behavior decomposition, state transitions, exception handling, sequential algorithms, behavioral completion, timing, hierarchical decomposition, description's modularity, and readability.

To show the potential of each language, we are going to informally describe a digital system (an automatic room temperature controller) and then formalize such description in all of the proposed languages.

We are going to analyze each language separately, highlighting the most relevant aspects. As our focus is on modeling digital systems, we are going to limit ourselves to the analysis of languages for Discrete Event Systems (DES) modeling.

2 Digital systems

In this section we present some concepts related to digital systems, which will be used in the following sections.

2.1 Reactive systems

According to the definition given by Harel and Pnuelli [1], reactive systems are those that continuously react to external or internal stimuli and are completely event driven, i.e., all the actions and state changes occur only in the presence of some foreseen events. However the system must be fault tolerant, and in the case of unforeseen events, the system must react in such a way to abort the current operation and change to another state, which is safer. This behavior is called exception handling.

Examples of reactive systems are increasingly more common in commercial, industrial, military, and scientific applications. The aircraft embedded systems, for example, are considered to be highly complex reactive systems, because they treat a relatively great number of variables, and further react to them in real time.

2.2 Embedded systems

Another class of digital systems is that of embedded systems. These systems belong to a greater system, controlling them and taking decisions, or at least helping the users to take decisions. They are becoming very popular. Examples are easily found as systems that supervise several aspects of a car; as aircraft control systems, which keep pilots informed about all the variables important to the flight; as controllers for microwave ovens, which are provided with instructions to control cooking processes, and so on.

2.3 Synthesis of digital systems

After the modeling of a digital system, and after the model simulation and optimization phases, several times it is desirable to synthesize the system, i.e., from an abstract model, to obtain an implementable description of the system.

There are three possibilities to this implementable description: the geometrical, structural and behavioral axis. Gajski and Kuhn introduced an Y-shaped diagram relating these three aspects of a system [2]. Each axis of the diagram represents a design aspect. The closer to the vertex of the diagram the lower is the abstraction level, e.g., at the behavioral axis, differential equations lie in the lower level, whereas algorithmic languages are at the highest level.

At the geometrical axis we have a description of the physical components of a system, its sizes and routes between the components, the so-called circuit layout. At the structural axis, the designer describes the structures, which will be used to compose the system, such as logic gates, transistors, registers, etc. At least at the behavioral axis we have a description about how the system reacts both quantitatively and qualitatively.

The languages considered in this paper are used mainly for the behavioral modeling of systems. System synthesis is usually done by traversing from the structural axis to the geometric one - often at the logic level - and from then on to the behavioral axis, at the manufacturing masks level. At this last phase, called *technology mapping*, one uses cell libraries to synthesize the system. Depending on each manufacturer, and its particular technology, a different library is used.

3 Description of the Temperature Controller

We are going to model a digital control system in all of the proposed languages. This will be used as a basis for the comparison among the languages. The system can be seen as having two distinct parts: a control unit, and a data path. The former is the focus of this paper, as the behavior of the system is implemented on it. However, we are also going to describe the data path, for a better understanding of the controller as a whole.

3.1 Behavior

The system is to measure room temperature at regular time intervals and display it to the user. To keep room temperature in the range $[t_{\min}, t_{\max}]$ the system uses heating and cooling devices.

3.2 Data path

A temperature sensor transforms its measurements into an analog voltage, which is converted to a digital equivalent through an A/D converter.

The digital value has no meaning to the user. Thus the system uses a look-up-table (LUT) to obtain the temperature value (e.g., in degrees Celsius). Figure 1 presents a schematic overview of the data path of the system. In this data path the inputs coming from the A/D towards the LUT are stored in the address register (AR), and the corresponding outputs from the LUT are stored in the data register (DR).

Then, it is necessary to display the data stored in DR, and compare it to t_{\min} and t_{\max} , using the two comparators COMP1, which outputs value 1 when $t_{\min} < t_{\text{env}}$ (t_{env} is the temperature of the environment)

and COMP2, which outputs value 1 when $t_{max} > t_{env}$. These two bits are used by the controller to figure out what should be turned on or off.

Note that some components receive input signals, like load, read, and ec. These are control signals provided by the control unit to coordinate the system.

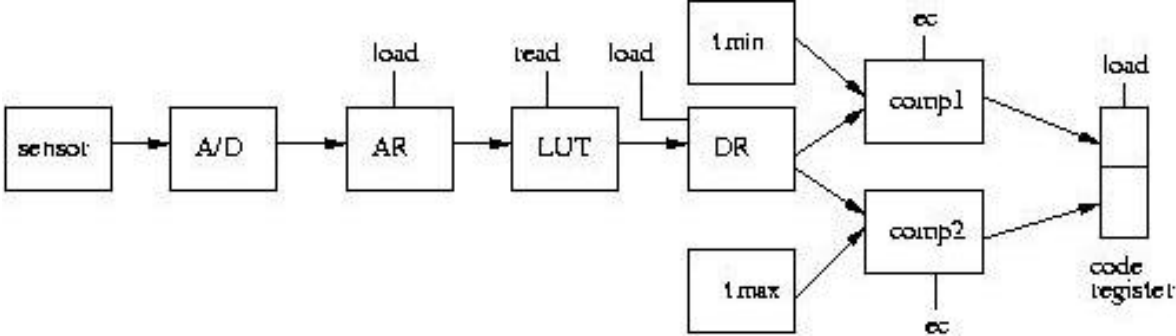


Figure 1: System data path

4 Statecharts

Created in the 80's by Prof. David Harel [3] while working for the Research and Development Division of the Israel Aircraft Industries, this language has been designed to model reactive systems.

Figure 2 illustrates a typical Statecharts model having a super-state “Y” composed by two other substates “A” and “D” concurrent to one another, each one having their own associated substates. The actions are taken according to holding of conditions related to each arc, which will determine the next state. For example, the occurrence of event y or event x leads the system from state B to C. Further, the little black square in the right upper corner of event A indicates the default transition, i.e., when the system reaches state A, automatically it is lead to state B. The black circle in state D has the same meaning.

4.1 Main features

Statecharts models support multi-level design methodology, i.e., designs that come from a description in a high level of abstraction, and are refined until they can be implemented. This approach is said to be a top-down design style. The bottom-up approach, in which a system is built from low-level building blocks, is more generally used for system analysis.

Another important characteristic of Statecharts is that is event driven. This means that states in the system can only change when an event occurs, and the conditions associated to it hold. There are also some events that can abort process execution. These are exceptions, which have to be dealt with.

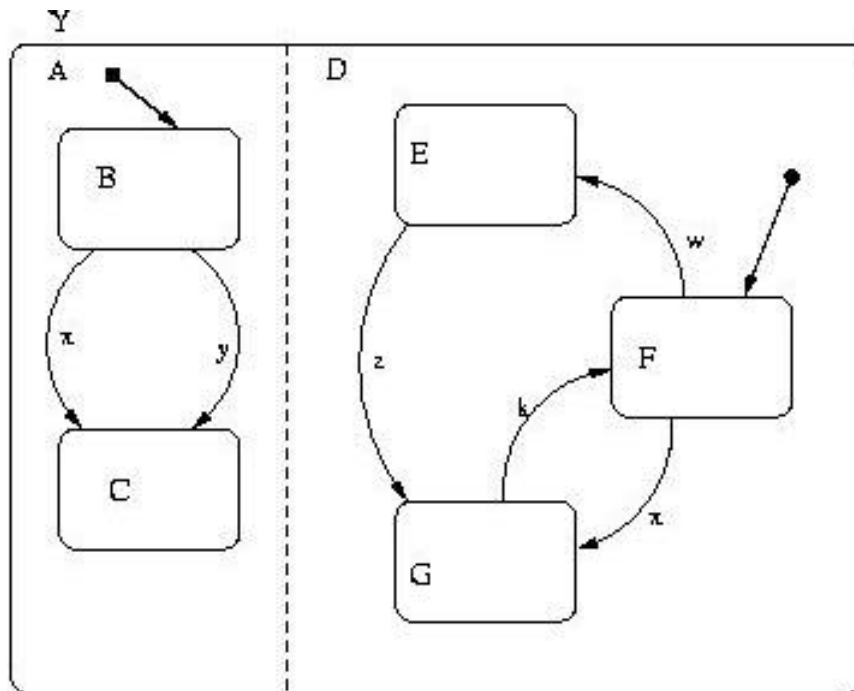


Figure 2: A Statecharts example

Several times it is also important that the system have some kind of memory, which stores the last state reached by the system within a hierarchical level of the model. In Statecharts models this memory is represented by a structure called *history*. Its work may be described in the following terms: when the system goes to a state that is in the immediately lower level, and it has to choose one among several states, if there is a history, it is checked. In this case, the system decides to change to the last state reached in that level.

As in most systems, there can be a concurrence relation among several activities, i.e., more than one process may be executing in the system at the same time. Relations like that can be modeled in Statecharts with simple structures.

One last feature of this language is its strong visual appeal. In most languages the graphical schemes are just an informal way of visualization, and an aid to the designer; the description is done in a textual language. In Statecharts diagram (which have a well-defined semantics) is the standard form of description, and a textual language is provided as an aid in case the designer finds it useful.

4.2 Tools

Taking advantage of the last feature pointed out in the previous section, we are going to discuss the computational tools used for description and analysis of Statecharts models. The main among such tools is possibly STATEMATE [4]. Their developers aimed at creating a friendly environment emphasizing the visual aspects of the models.

The first versions of STATEMATE dates back to the eighties possibly influenced by the then emerging graphic interfaces technology. Since its creation STATEMATE has been steadily updated.

STATEMATE is basically a simulator, to which a state and a set of conditions are given. In turn the program deducts a new state. The algorithm used in the implementation is as follows.

First stage: step preparation.

Add external events to the event list generated internally. This implies the execution of all actions, including changes in the given values, conditions and activities, but not in the states.

Second stage: step contents computation.

To the current state, the algorithm checks the set of enabled transitions, verifying the existence of conflicts, non-determinism, leaving to the user to solve an undesirable situation.

Third stage: step execution.

In this stage, state transitions according to changes in the previous stages.

4.3 Controller model

Figure 3 shows the temperature controller model in Statecharts. The initial state of the system is “reading”, whose default substate is “ready to read”, as indicated by the arrow leaving the little black circle. Note that from this state, the system waits for 100 ns before going to the next state. setting to 1 the value of signal “ldar”. The reason to choose a time event to the state change is the fact that Statecharts does not support directly behavioral completion and event sequence concepts. In this case, assuming a time interval to the state changes, one forces a sequence to happen.

Another remarkable feature is the default transitions at each level. They are fired whenever the system reaches the level they belong to. Because of this, they do not need firing conditions; however they provide a change in the system signals.

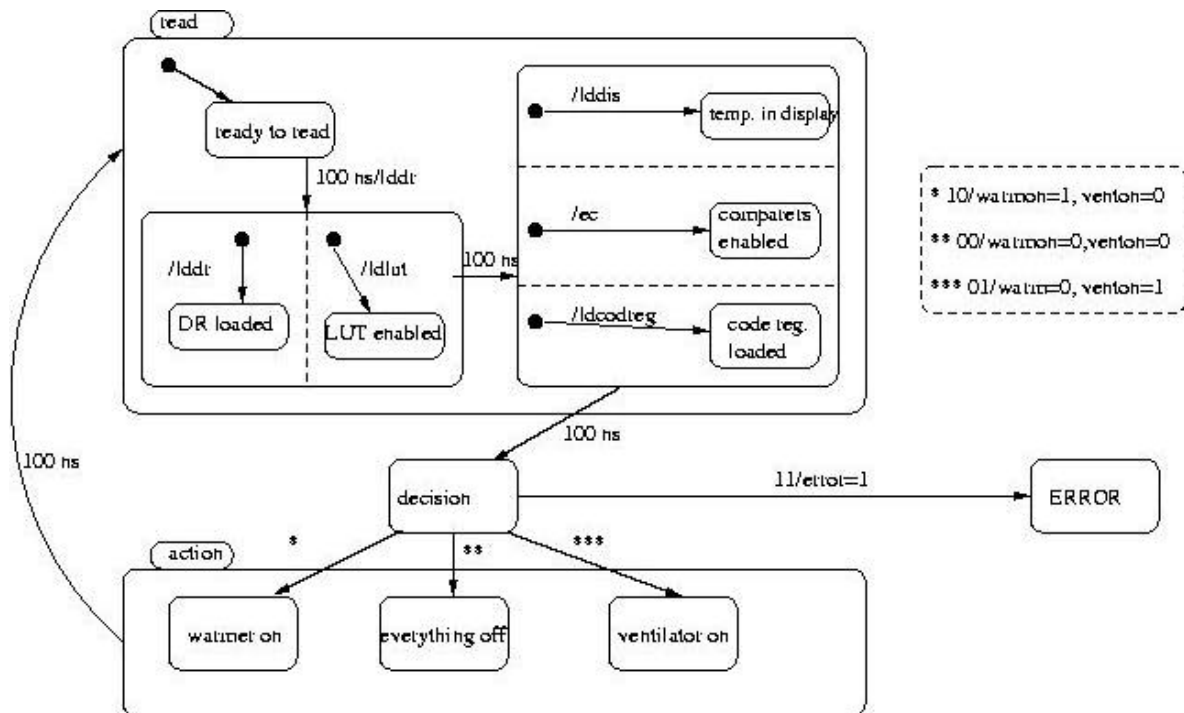


Figure 3: temperature controller model in Statecharts

5 VHDL

Among all languages presented in this paper, VHDL is the one having the most singular features. It is a hardware description language originated from a U.S. Department of Defense project called Very High-Speed Integrated Circuits [5]. In 1986 it became IEEE standard 1076, and nowadays it is widely used in academic and industrial environments.

5.1 Features

5.1.1 Behavioral modeling

Designing in VHDL is like programming. A source code describing the system is produced by the designer, and compiled. The compiler generates a *Component Library*. This is then submitted to a *VHDL simulator*, which analyses the system's behavior according to test sets provided by the designer, along with expected results. The simulator analyses the model taking into account time constraints. It is also possible to do logic synthesis, based on a behavioral description.

The VHDL programming the same paradigm of imperative high-level languages (it resembles Ada programming language). However, this is only the highest level of design as the description can be done in several levels of abstraction, as follows:

- algorithmic (imperative);
- register transfer level (RTL);

- gate level with unit delay; and
- gate level with detailed time.

5.1.2 Modularity

In VHDL, there is a quite clear separation between system interface and its behavior. For system interface it is used a programming structure called *entity*, in which only the input and output signals of the system are defined. To model system behavior there is another programming structure called *architectures*, in which information about how the outputs will be given as a function of the inputs, and possibly, of an internal state are provided. It is important to note that as entities and architectures are modeled separately it is possible to create several architectures describing the same entity. Each architecture can be separately tested before the best one is chosen. This allows version control, and model reuse.

5.1.3 Concurrency and hierarchy

Digital systems are predominantly concurrent. Signal value changes on gates and wires can occur simultaneously and independently. Thus, a designer can create his/her model in VHDL, describing concurrent activities through *processes*.

Timing is also an important feature of VHDL. There are two timing models in this language one assigning a unit delay to all gates in the considered system, and the other assigning individual gate delays. As exact gate delay values are dependent on actual transistors characteristics, the timing models are usually theoretical forecasts of such delays.

The most important feature of this language is perhaps that it allows modeling at a high level of abstraction without losing a hardware perspective. In other words, the designer can always keep a precise idea of the behavior of the system signals, even when describing it at the algorithmic level.

5.2 Tools

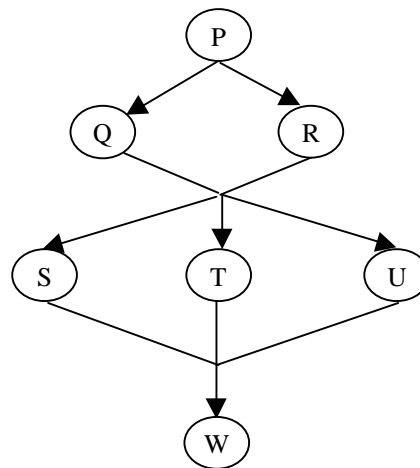
Several products for design and analysis of systems have been developed as integrated environments with tools for text editing, waveform editing, graphic editing, symbolic editing, model analysis, design rule checking, logic devices programming, technology mapping, and so on. The interfaces for such environments are quite user-friendly, and some of them even support other hardware description languages than VHDL, such as Verilog HDL.

5.3 Controller model

Figure 4.a is the transcription of a sample of the VHDL model of the temperature controller. During the modeling phase we took into consideration the fact that VHDL does not support behavioral decomposition. To avoid such a drawback our description has been partitioned in several specific processes. To synchronize them a classic mechanism of process communication through control signals has been used. A sample of this is presented in lines 24 and 25 of Figure 4.a. This communication can be better understood if we consider lines 45 and 51 in Figure 4.a. Line 45 shows the activation of signal `QR_activate` allowing the concurrent execution of processes Q and R, which are respectively responsible for loading the data register and enabling the LUT. Line 51 presents the deactivation of this signal, preventing processes Q and R from resuming execution.

```
1 entity contr_temp is
  ...
20 begin - behavioral
  ...
24 signal Q_activate: boolean:=false;
25 signal R_activate: boolean:=false;
  ...
47 QR_activate <= true;
  ...
51 QR_activate <= false;
  ...
110 W: process(cod);
  ...
126 and behavioral;
```

(a)



(b)

Figure 4: VHDL model of the temperature controller

(a) sample code; (b) process graph

Figure 4.b displays a schematic model of process concurrency. Processes S, T, and U, behave similarly to processes Q and R, as explained above. Process W (line 110 – Figure 4.a) chooses whether or not to activate the heating/cooling devices according to bit vector `cod`.

6 SpecCharts

SpecCharts is an extension to VHDL proposed specifically to model embedded systems [6]. Its main goal is to provide the user with a natural modeling tool. SpecCharts code is processed by a front-end, which translates it to standard VHDL.

6.1 SpecCharts versus VHDL

According to Vahid et al. [6], many factors contributed to the choice of creating a *front-end* to an existing language rather than producing a brand new software. Among these factors is the time required for the development and integration of compilers, simulators, and synthesis tools. Yet another important point has been the fact that VHDL has become kind of an international standard language for hardware description, and translators have been developed between VHDL and many other languages.

The most visible contribution of SpecCharts is its capability to model three features not supported by VHDL, namely: state transitions, exception handling, and behavioral completion.

6.2 Program-State Machines

SpecCharts models follow a basic concept called Program State Machine (PSM), which is a convergence between the ideas of algorithms and Finite State Machines. PSMs are defined as state machines in which each node contains a section of code that is executed whenever the system reaches that node. This conceptual model provides a way to synthesize part of a system in hardware and part in software through co-design, exploring the best technological features available.

6.3 Tools

The main computational tool created to deal with PSMs and SpecCharts is SpecSyn, which supports a design methodology called specify-explore-refine [7]. The three steps of this methodology can be briefly explained as follows:

Specification: Using PSM concept, the designer models state transitions according to the expected inputs, and associates to each one an arbitrarily complex program. Execution time constraints and transmission baud rates can be described. Other technology dependent constraints such as component sizes and I/O limitations can also be defined.

Exploration: the user to the program must define System components such as processors, memory, and buses. Having the specification of the components it is necessary to partition the system to decide which components will be responsible for each part of the functional specification. Afterwards, the program produces forecasts of system parameters such as global performance, hardware size, and software complexity.

Refinement: The third step consists in producing an optimized partitioned specification of the system taking into account the parameters estimated in the previous step. The refined specification must be in a user readable format, and has to be ready for simulation.

6.4 Controller model

SpecCharts description of the controller displays interesting characteristics regarding behavioral decomposition. There can be observed many similarities between SpecCharts and VHDL models, e.g., entity declaration is the same. The main difference between these languages is how easy it is to model behavioral decomposition. While VHDL does not completely support such a decomposition, in SpecCharts it is enough to define one of three available types (sequential behavior, concurrent behavior, or leaf) within the `behavior` construction.

Other considerable aspect in SpecCharts models is the use of arcs connecting processes. For instance, lets consider the following sample code of the controller model.

```
read: (TOC, true, convert);
convert: (TOC, true, compare);
compare: (TOC, true, decide);
decide: (TOC, true, read);
```

These four lines of code define behaviors (`read`, `convert`, `compare`, and `decide`). In each line there is a reserved word `TOC` (standing for transition-on-completion), which means that upon completion of the current behavior the control of the program will be immediately transferred to the behavior indicated as the last parameter of the command line, e. g., upon completion of behavior `read`, control is transferred to behavior `convert`.

7 Grafcet and Grafchart

As seen in section 4 graphical languages are important due to their visual appeal. This characteristic makes the modeling of systems an easier task, and serves as an aid to documentation. Grafcet and Grafchart are two of such languages. Grafcet has been introduced in 1977, in France, as a formal specification method for logic controllers. Grafchart is a higher level language based on Grafcet, Petri nets (discussed in section 8), and in imperative programming languages [8].

In Figure 5 we present a portion of a Grafcet model where three states (5, 6, and 7) and two transitions (a, and b) are shown. The table in Figure 6 indicates the active states (those to the left of the table) as well as those to become active after the execution of each transition (those circled in the table), which are determined according to a given set of conditions.

7.1 Characteristics

7.1.1 Receptivity

An important Grafcet concept is that of receptivity. It indicates when a change of state occurs. As in Statecharts this situation is determined by the occurrence of an event and (when applicable) the validation of a condition.

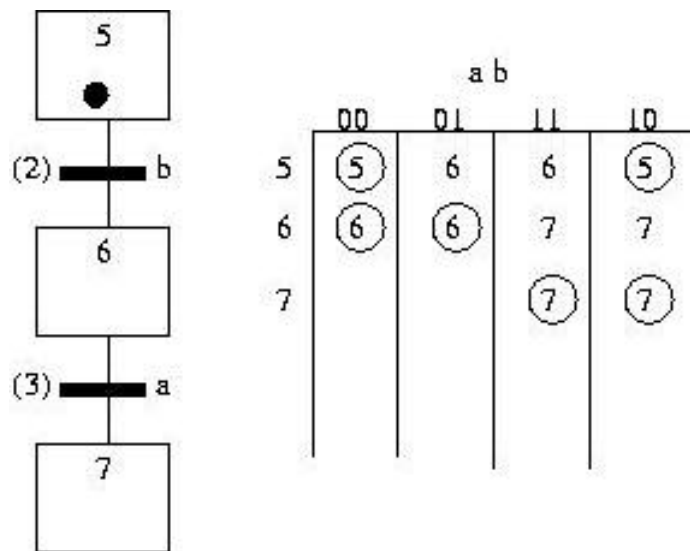


Figure 5: Grafcet language example

The concept also exists in Grafchart. However, it is slightly different, i. e., the condition is not only boolean, but any allowed G2 language structure (the language in which Grafchart is implemented), as numbers, strings, and vectors.

7.1.2 Description and Implementation

Grafcet has a graphical syntax that makes it very expressive. However, it is not an implementation driven language. Unlike Grafchart, which is aimed at system description and implementation, Grafcet is mainly concerned with system description.

7.1.3 More Complex structures

Grafchart supports more complex structures than does Grafcet. These include processes and procedures, which are executed as function calls in general purpose programming languages. These structures do not exist in Grafcet, and have to be specially modeled when needed.

7.1.4 Concurrency and state transitions

Grafcet and Grafchart are powerful enough to model concurrency, specially regarding state transitions. Sequential behavior is also an important characteristic in that they support procedure calls. Timing characteristics can only be indirectly modeled with Grafchart. Grafcet does not support timed models at all.

7.2 Tools

There are several easily found tools that support Grafcet models. A set of such tools is available for download at <http://www.-mips.unice.fr/~gaffe/tools.html>. For instance, after downloading a software package from this site models can be graphically described with a tool called eg7. The models can then be compiled using g2sc, which generates an intermediate code in Esterel. The user can also choose other intermediate languages such as luster, VHDL, and C. Otherwise, a logic equation description format (BLIF) can be chosen too. Finally, the output from g2sc can be read and processed using sg7 simulator, which generates the corresponding results to the user.

7.3 Controller model

Figure 6 illustrates the model of the temperature controller in Grafchart. The dashed box in the figure shows the meaning of actions associated to each step. Transitions in Figure 6 are associated to guards that are conditions to be satisfied in order for them to be triggered. It can be observed that behavioral decomposition is easily modeled in Grafchart. This is done with AND-divergence (graphically represented by two parallel bars), and OR-divergence (represented by a single bar that splits itself into several other transitions). AND-divergence represents the execution of concurrent processes, and OR-divergence represents the choice among different processes.

8 Petri Nets

Presented in 1962 in the thesis of Carl Adam Petri, *Communication with Automata*, Petri nets became one of the most important tools for system modeling due to both its mathematical correctness, and its

visual appeal. Under the generic name of Petri nets, there is an entire set of languages that can be divided into two broad categories: ordinary, and high-level nets. On top of these one can have different timing models [9].

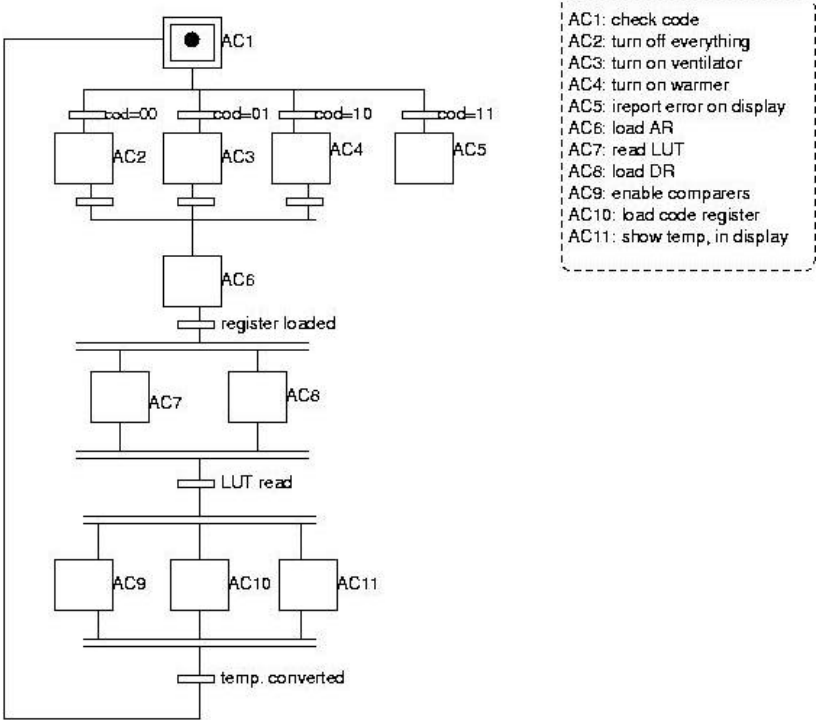


Figure 6: Grafchart model of the controller

Figure 7 illustrates a typical Petri net model consisting of three state (P1, P2, and P3), and three transitions (T1, T2, and T3). Transition T3 is said to be enabled because all its pre-conditions are satisfied, and its post-condition is not. This is represented by the existence of a black dot in states P1 and P2, and by state P3 being empty. Depending on the extension considered transitions T1 and T2 may be enabled or not. However, this discussion is outside the scope of this paper.

8.1 Main features

An important feature of Petri nets is its capability for modeling states changes. These changes are asynchronous, and driven by either internal or external events [10]. Many systems, such as communication networks, robotic and traffic systems present this type of behavior. Among all languages considered in this paper, Petri net is the one most suitable for modeling concurrent events, as several transitions can be enabled at the same time independently of one another. This property is known as persistence.

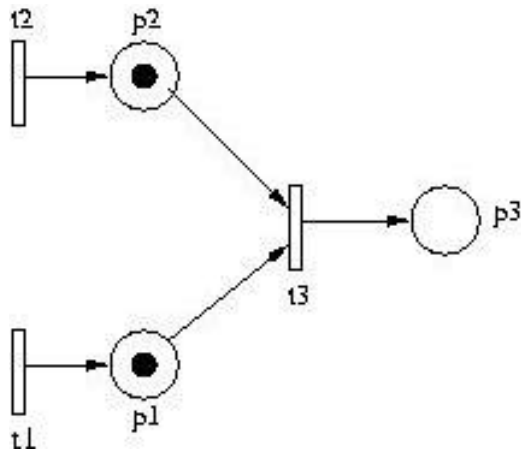


Figure 7: Typical Petri net model

Hierarchy is another important functionality incorporated in Petri nets several years ago. More recently, object oriented Petri nets are also showing up [11]. Temporization is yet another important feature when considering practical applications. Knowing critical points of the system it is possible to optimize it. In real time systems this characteristic is particularly important.

From a mathematical stand point Petri nets can be treated by algebraic techniques using state equations and state graphs. Reachability graphs are the main method of analysis of the dynamic properties of a Petri net. To analyze structural properties the invariant method can be used [12]. All of these methods of analysis are important to verify inconsistencies in the system, as the possibility of occurrence of deadlocks, for example.

8.2 Tools

There are many Petri net tools for system analysis. One important tool is Design/CPN, which was developed in a partnership between the Colored Petri Net group at the University of Aarhus, Denmark, and Metasoft Corp., U.S.A., to support colored Petri net model analysis. The software has three main parts: a graphics editor, a simulator, and an occurrence graph generator. For many years Design/CPN has been developed and supported by the CPN group at the University of Aarhus. Design/CPN supports models with complex data types (color sets), and complex data manipulation (guard functions) – both specified in the functional language Standard ML. The package also supports hierarchic models with well-defined interfaces.

Another important Petri net based tool is Petrify [13], developed for digital system synthesis. Besides synthesizing digital systems from Petri net descriptions, Petrify can also simplify a given net.

Simplification is processed as follows: from the supplied net the corresponding reachability graph is generated. Considering the theory of regions [14] it is possible to identify regions in the graph that correspond to states in the net.

8.3 Controller Model

Figure 8 presents a Petri net model of the temperature controller, where the dashed box contains the meanings of the states and transitions. State changes follow Moore model as the actions (outputs) are connected to inputs and the next state. This may be the most distinctive point between Petri nets and Grafcharts. Concurrency is modeled by a transition with several output states such as t5 and t8. Between these two transitions there are two sequences of concurrent behaviors represented by {p4, t6, p6} and {p5, t7, p7} in Figure 8. The same applies to the three sequences between t8 and t12.

In high level nets the transitions can also be associated to guards as is the case of transitions t1, t2, t3 and t4 in Figure 8. These transitions will fire depending on the value of the bit vector cod, which indicates the correct temperature range. Transition firing time is not taken into account as all delays are assumed to be equal. Since Petri nets support event sequencing, firing time turns out as a low-level detail.

9 Comparison among the languages

Table 1 displays the overall comparison of the five languages considered in this paper. To compare the languages the parameters presented in section 1 were used. For the comparison we used four levels of excellence, which can be defined as follows:

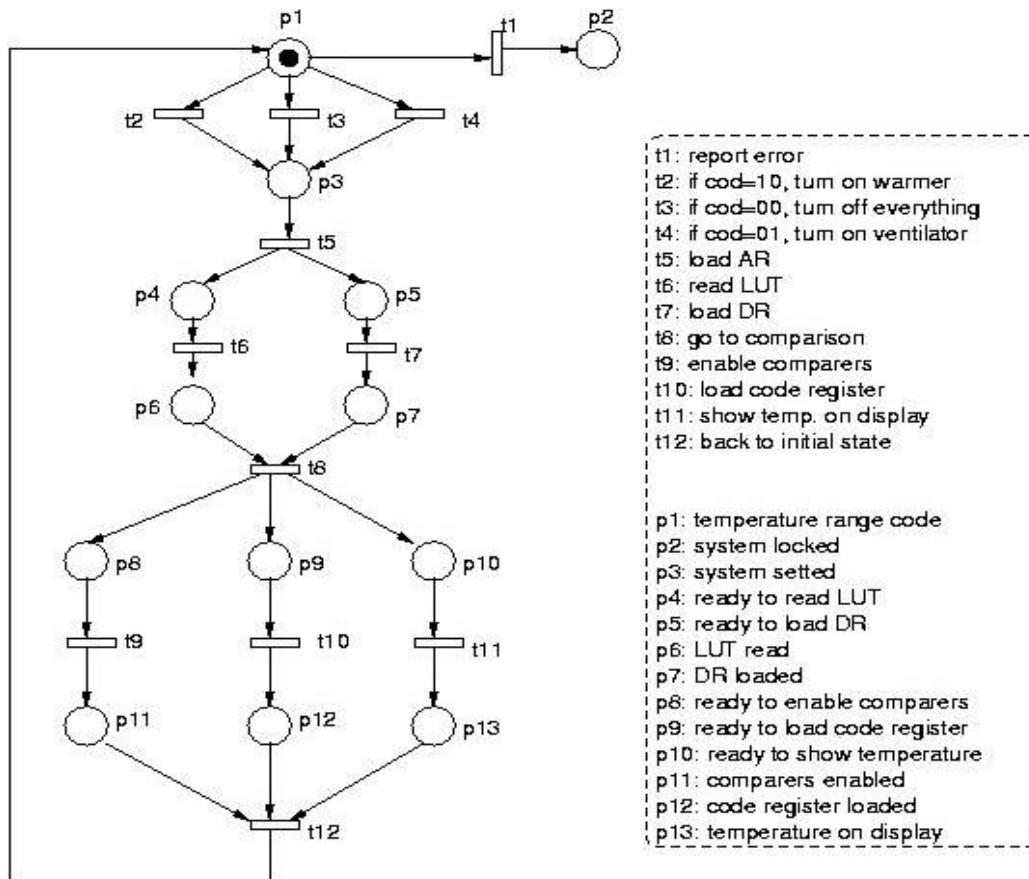


Figure 8: controller model in Petri nets

Poor: the language barely supports the considered characteristic, or does not support it at all.

Fair: the language mildly supports the considered characteristic.

Good: the language normally supports the considered characteristic.

Excellent: the language not only supports the considered characteristic, but also has related features that make its use more comfortable to the user.

| Table 1 – Comparison among the languages | | | | | | |
|--|----------|-----------|-----------|------------|---------|-------------|
| Parameter | Language | VHDL | Petri Net | SpecCharts | Grafcet | Statecharts |
| Algorithms | | Excellent | Poor | Excellent | Poor | Poor |
| Exceptions | | Fair | Poor | Good | Poor | Excellent |
| Behavioral decomposition | | Fair | Excellent | Excellent | Good | Fair |
| Hierarchy | | Poor | Good | Excellent | Good | Good |
| Behavioral completion | | Good | Excellent | Good | Good | Poor |
| Modularity | | Good | Fair | Excellent | Fair | Good |
| Readability | | Poor | Good | Poor | Good | Excellent |
| State transition | | Fair | Excellent | Good | Good | Good |

Each of the parameters presented in Table 1 is commented in the following:

Algorithms: The feature that differs VHDL from the other languages is its imperative programming style.

Exceptions: Statecharts allow recursive exception handling, i. e., exceptions handled at one level are reflected to lower levels.

Behavioral decomposition: This feature is present in all the languages. However, Petri nets allows a clearer distinction between sequential and concurrent behaviors.

Hierarchy: SpecCharts is the language that best describes hierarchy levels, because the designer has to explicitly declare the type of each level. Furthermore, different levels do not directly communicate to each other as arcs can only connect events at the same level.

Behavioral completion: In Petri nets it is natural to model completion of a process as there is no need to specify conditions for state changes after the conclusion of the steps belonging to some process.

Modularity: Module interfaces in SpecCharts is well defined, what allows great flexibility and model reuse.

Readability: Statecharts visual appeal allows faster modeling, and results in easier to understand models.

State transitions: The languages presented in this paper were derived from the same conceptual model, based on conditions and events. The exception is VHDL, which is predominantly based on the concept of algorithms, in which the actions are controlled by means of programming structures (“if”, “while”, etc). However, Petri nets provide the best support for state transitions because of the many features associated with transitions, such as guard functions, input and output functions, and timing.

10 Conclusions

One can observe that there is a pattern followed by almost every language: their main feature is state transitions, which occurs as a function of external (or internal) events and/or boolean conditions. The other characteristics are represented as a function of this one. The exception to this is VHDL, which follows a paradigm closer to programming languages.

None of the languages considered in this paper adequately support all the proposed characteristics. Petri net is the one, which comprises more features, being the most flexible, and resulting in more expressive models.

References

- [1] D.Harel, and A.Pnuelli, *On the development of reactive systems*, Technical Report, The Weizmann Institute of Science, Rehovot, Israel, 1984.
- [2] D.D.Gajski, and R.H.Kuhn, *New VLSI Tools*, Guest Editors' Introduction, IEEE Computer Magazine, 16(12)11-14, 1983.
- [3] D.Harel, *Statecharts: a visual formalism for complex systems*, Technical Report, The Weizmann Institute of Science, Rehovot, Israel, 1987.
- [4] D.Harel, and A.Naamad, *The STATEMATE semantics of Statecharts*, ACM Transactions on Software Engineering and Methodology, 5(4)293-333, 1996.
- [5] S.Mazor, and P.Langstraat, *A guide to VHDL*, Kluwer Academic Publishers, 2nd Edition, 1997.
- [6] F.Vahid, S.Narayan, and D.Gajski, *SpecCharts: A VHDL front-end for embedded systems*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 14(6)694-706, 1995.
- [7] D.Gajski, F.Vahid, S.Narayan, and J.Gong, *System level exploration with SPECSYN*, Design Automation Conference Proceedings, pp.812-817, 1998.
- [8] C.Johnsson, and K.-E.Årzén, *Grafchart and Grafcet: a comparison between two graphical languages aimed for sequential control applications*, Preprints of the 14th World Congress of IFAC, vol.A, pp.19-24, July 1999.
- [9] J.Wang, *Timed Petri Nets: Theory and Applications*, Kluwer Academic Publishers, 1998.
- [10] L.E.Pinzón, H.-M.Hanish, M.A.Jafari, and T.Boucher, *A comparative study of synthesis methods for discrete event controllers*, Formal Methods in System Design: An International Journal, vol.15, pp.123-167, September 1999.
- [11] A.Perkusich, and J.C.A. de Figueiredo, *G-Nets: A Petri net based approach for logical and timing analysis of complex software systems*, The Journal of Systems and Software, vol.39, pp.38-79, October 1997.
- [12] T.Murata, *Petri nets: properties, analysis, and applications*, Proceedings of the IEEE, 77(10)541-580, 1989.
- [13] J.Cortadella, M.Kishinevsky, A.Kondratyev, L.Lavagno, and A.Yakovlev, *Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers*, IEICE Transactions on Information and Systems, vol.E80-D, no.3, pp.315-325, 1997.
- [14] A.S.Yakovlev, and A.M.Koelmans, *Petri nets and digital hardware design*, Lectures in Petri Nets II: Applications (W.Reisig and G.Rozenberg, eds.), in Lecture Notes in Computer Science, vol.1492, pp.154-236, Springer Verlag, 1998.