PART 1

General Concepts

Chapter 1

Elements for the Design of Embedded Computer Systems

1.1. Introduction

The development of embedded systems is usually done using low-level languages: namely, assembly language and oftentimes C. The main reason for this is the need to elaborate programs that have a small memory footprint because they are usually deployed on very compact architectures[1]. More costly development techniques have been gradually created. These development techniques are mainly based on an in-depth assessment of the requirements, intensive tests as well as very strict development procedures, which ensure a safety standard satisfying the expectations of the general public.

However, these systems that often accomplished critical missions frequently involved very expensive developing strategies, thereby being limited to a specific usage such as space travel, aeronautics, nuclear use and railroad transportation. Once these systems emerged in more "mainstream" industries, the approaches in development had to evolve in a cost-reducing direction. The economic revolution toward offshore development does not facilitate these aspects of viability/safety, but these new development approaches could, in the long run, become very competitive.

Chapter written by Fabrice KORDON, Jérôme HUGUES, Agusti CANALS and Alain DOHET.

1 For example, the probe Pioneer 10, launched in 1972, had only six Kbytes of memory [FIM 74], which did not stop it from transmitting an impressive amount of scientific data regarding Jupiter and its satellites, before becoming the first man-made object to have left the solar system.

The mere use of high-level programming languages is not, in and by itself, the solution. Embedded systems usually contain sophisticated mechanisms and runtime libraries. Because the latter cannot be certified, they are very difficult to use. Take the Ada language as an example of this. It was elaborated in the 1970s with the aim of developing low-cost embedded systems. Some of its features (usually the management of parallelism) have seen limited usage in certain fields, largely due to the code compiled having to be associated with limited runtimes (for instance, not being able to support the parallelism or the dynamic memory allocation).

One solution to maximising software viability and minimizing the cost of embedded computer systems is through the use of model-driven engineering (MDE). It indeed facilitates better interactions between the different languages and models used throughout the design/development cycle.

More particularly, it allows us to rely on dedicated "models", which sometimes facilitate the reasoning process. Thus, engineers can predict certain behavioral aspects of their programs. The transformation techniques that have been developed by this community ensure the link between the different development stages.

These new approaches will not suddenly replace the existing procedures. The actors behind the development of embedded systems (particularly when they carry out critical missions) cannot afford to take any risks. However, these actors have gradually taken an interest in these new techniques and their application in adequate methodological frameworks.

In the long run, the objective is to reach high-performance industrial processes capable of ensuring trustable software, by providing, for instance, co-simulation and formal verification and allowing the target code generation to be validated right from the beginning of the modeling.

It should not surprise us that the three notations presented in this book – systems modeling language (SysML), unified modeling language/modeling and analysis of real-time and embedded systems (UML/MARTE), and architecture analysis and design language (AADL) – are heavily reliant on model engineering. Model engineering is clearly a "hot topic" in the community at the moment.

Chapter outline. This chapter discusses several elements that are important in the development of embedded systems in the context we have just touched upon. We pay particular attention to:

– the modeling activity (section 1.2);

– the presentation of the UML, which serves as the foundation for two of the three notations presented in this book (section 1.3);

– the presentation of the MDE (section 1.4);

– an overview of the analysis techniques and, in particular, those used in this book (section 1.5);

– the methodological aspects of system development (section 1.6).

1.2. System modeling

The modeling operation has been a widespread practice in the history of humanity. It seeks to explain the behavior of a complex system via a model that abstracts it. We can cite the different models designed for explaining celestial movements: the system of the epicycles, the Ptolemaic system, the Tycho Brahe system, the Copernican system, etc. Their objective was to predict the evolution of the planets' position. They represent a process of ongoing fine-tuning of the understanding of a field, a new system replacing the old system when new evidence crops up showing that the old system does not correspond to reality. Modeling has, therefore, been an indispensable tool in experimental sciences for a long time.

In computer science, the big difference lies in the fact that the model does not reproduce a system we are trying to observe in order to understand its behavior. The model is placed at an earlier stage, and allows us to realize whether a solution, which is in the process of being discovered, will respect the properties expected.

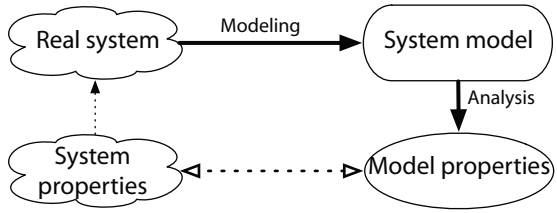

Figure 1.1. *Relations between the system, its model and their properties*

Figure 1.1 represents the relation between different entities of a system at the core of the modeling operation. The engineer models the system while he or she is in the process of designing it. The analysis of this model (via simulation or via more formal methods, see section 1.5.1) allows us to deduce system properties. However, the relation between the properties of the model and those of the system is by no means a trivial issue.

First, the expression of the properties may vary greatly in the two cases. Usually the property of the system will be expressed in a natural language, whereas the property of the model will have the shape of an invariant or will reference quite

precisely an entity of the analysis (whose meaning must be known to the users if they wish to understand it). We must thus anticipate elements of traceability between the two.

Second, the necessary abstraction of certain system elements during the modeling process (for instance, changing from discrete types to continuous types) can, when done poorly, result in a model that does not faithfully represent the system. If it is a superset, then a property that has not been verified on the model can be verified on the system; if it is a subset, then a property that has been verified on the model may not be true for the system.

Modeling is thus a challenging operation that must be carried out carefully. More particularly, the modeling choices must be documented properly. In the field of embedded systems, we will introduce several important notions regarding the concept of a model.

Structural or behavioral modeling. The former defines the structure of a system (i.e. the class diagrams or the UML instances) whereas the latter describes its behavior. When examining problematic behaviors, or identifying emerging behaviors, we must reach the second level, which generally presupposes that the modeling of the system structure already exists (at least the identification of the interacting actors).

Open or closed models. An open model describes a system. A closed model describes the system and the environment it interacts with. We can, thus, consider that the open model is "included" in the closed model.

The reason we need to distinguish between the two is that we are able to subject the model of a system to different conditions of execution. Each of them is thus characterized by a dedicated specification of the environment that "closes" the specification. This notion is particularly useful when we study different execution modes: a nominal mode, modes that have been degraded as a consequence of certain conditions, etc.

Notations used in this book. This book deals with embedded systems and only uses the notations that enable us to describe those systems. Out of these, we have selected three. Of the three, SysML [OMG 12a] and MARTE [OMG 12b] are UML profiles, and AADL [SAE 09] is a dedicated notation that integrates, at the same time, the description of the software part and that of the hardware part of a system.

1.3. A brief presentation of UML

The first version of the UML was version 2.8 of the "unified method", which was written by G. Booch and J. Rumbaugh. At the time, in 1995, Y. Jacobson was not yet part of the adventure, and the letter "M" stood for "method".

A year later, the first versions of the UML, namely versions 0.9 and 0.91, were published, which incorporated the work of Y. Jacobson. This time, the "M" stood for "modeling". Indeed, these three "fabulous evangelists", having not agreed upon one standard method, had instead focused on the notation.

UML 1.0 and 1.1 were proposed to the Object Management Group (OMG) in 1997. Then came version 1.2 in 1998, 1.3 (a very significant revision) in 1999, 1.4 in 2001 and, finally, 1.5 in 2002. This sequence of different versions is usually called "UML1.X". This was followed by UML 2.X, whose first stable version (2.0), appeared in 2003. This is a major revision now involving 13 diagrams instead of nine previously. The metamodel was also considerably modified (which is of relevance to tool designers). The notion of the profile was formalized, which has allowed, for example, the emergence of "variants", such as SysML and MARTE, which we will detail in this chapter.

Let us note that the transition from 1.X to 2.X, which was supposed to facilitate the use and efficiency of the language, has given mixed results. The industry and the toolmakers had put a significant amount of time into fully supporting UML 1.X and the required adaptation has delayed the operational use of UML 2.X, which has only recently begun to be used effectively in industry.

We will introduce the main UML characteristics (in its current version in 2012, the 2.4.1 version). The 16 UML diagrams are divided into two classes: static and dynamic diagrams.

Among these 13 diagrams, the four "main" diagrams are considered to be *classes*, *sequence*, *use cases* and *state machines*. They are used in the parts dedicated to SySML and to MARTE but also, systematically, by all the UML users.

UML is a notation. To use it well, we also need a method. Thus, each company has created its own method usually relying on the "unified" method (UM) or the *rational unified process* (RUP) (the two are pretty close). These methods describe when and how to use this or that diagram, how to organize the model as well as the documents and the codes that are generated while still respecting the specification/design processes that are enforced throughout the enterprise such as traceability, configuration management and quality (rules for a correct usage, coding rules, etc.).

1.3.1. *The UML static diagrams*

These diagrams describe the static aspects of a system (relative to their organization). In this category, we may find the following diagrams: *classes*, *composites*, *components*, *deployment*, *object* and *package*.

Class diagrams. They allow us to model the domain (for the sake of simplification, the data and/or the concepts manipulated by the application), and then the application itself. We will thus have the classes of the domain, then the application classes (analysis and design). An average model generally brings about approximately 100.

Composite diagrams. They allow us to decompose a class (in general an application class) into smaller parts. This decomposition enables us to show, for example, which part implements which operation, or how the different parts communicate with each other.

Let us note that there is a second type of composite diagram, which allows us to model *patterns*. Once these patterns are modeled, they can be used in class diagrams.

Components diagrams. They allow us to describe a system via its components and the interactions between components through their interfaces. A component can also be decomposed into subcomponents (just like the classes decompose into smaller parts) by means of a composite diagram.

The difference between a class and a component is the subject of a wide debate. However, our viewpoint can be summed up as follows:

– A class is the basic building block of a software.

– A component is also a building block but of a different level of abstraction; it generally groups together a set of classes, but can also group together other artifacts such as configuration files.

– Classes are interlinked via different relations (heritage, association and composition) and they propose various interfaces.

– Components are interlinked via interfaces.

– Classes can be decomposed into smaller parts.

– Components can only be decomposed into other components (i.e. components having a smaller granularity).

Deployment diagrams. They allow us to model the physical architecture (architectural components) of the application: machines, processes, communication modes, etc. In general, a process is made up of components (which are composed of classes and other artifacts) and distributed to one up to n machines.

Object diagrams. They instantiate a class diagram. A class diagram is a generic model (for instance, the description of the organization of an enterprise) whereas an

object diagram is a specific model (for instance, the description of the organization of an enterprise).

In general, we start out by drawing class diagrams and we make sure they concur with the object diagrams. However, several designers prefer to start with the objects (of the specific domain) and then generalize in order to find the classes. The final result is often similar.

Package diagrams. They allow us to organize the model. A *package* may contain *packages*, classes, objects and diagrams. In general, a model is organized in three *packages*: needs analysis (*use cases*), logical architecture (classes) and physical architecture (machines, processes and components). Depending on the method applied in the enterprise, each of these *packages* can be further decomposed.

It is worth noting that a *package* is not a component (neither software nor hardware) but rather a model structuring unit.

1.3.2. *The UML dynamic diagrams*

These diagrams describe the dynamic aspects of a system (i.e. relative to their execution). In this category, we may find the following diagrams: activity, interactions (*sequence*, *communication*, *overview* and *timing*), *use cases* and *state machines*.

Activity diagram. This type of diagram (which includes the concepts of parallelism) facilitates the modeling of an algorithm via concatenation of activities/events. It proposes an action language allowing us to specify in more detail all of the algorithmical processing.

Interaction diagrams. (sequence, communication, overview and timing) The sequence diagrams and communication diagrams allow us to model the collaboration between the objects (class instances). These two diagrams are almost equivalent even if they present us with several specificities. The main difference is a visual one: the sequence diagram shows a sequential view, whereas the communication diagram shows a spatial one.

The *overview* diagram allows us to show a concatenation of diagrams in the shape of an algorithm that is similar to the activity diagram. However, the activities and actions are in themselves diagrams. This gives us better readability throughout the specification of complex concatenations.

The *timing* diagram (derived from electronic engineering) facilitates the modeling of behaviors that are sequenced by time events (for instance, the time constraints between different states of several objects).

Use case. They facilitate the description of the system from an external point of view: the actors (roles) using the system as well as the services (*use cases*) offered by the system. The *use cases* are, in general, fine-tuned by activity diagrams and/or sequence diagrams filled in with natural language.

State machines. They model the behavior of an active class (asynchronous events, communication protocols, etc.). They can equally model the behavior of the system (its different states, the transition conditions from one state to another, etc.).

We call an "active class" a class that has an autonomous behavior.

1.4. Model-driven development approaches

The UML 2.0 version appeared in the early years of the 21st century; this version has been progressively integrated in development workshops, which, since then, have started to provide a very rich array of modeling tools. These tools have been developed in accordance with norms that synthesize a number of experiences and industrial expectations in the field of system engineering, thus grouping them together for the first time in years.

Around the same time, the notion of MDE appeared.

Three predominant approaches were born from this concept:

– the OMG approach: model-driven architecture (MDA), based on UML and Meta Object Facility (MOF) (the language that allows us to write metamodels in the OMG world);

– the ECLIPSE approach: *Eclipse Modeling Framework* (EMF) based on ECORE (the language that allows us to write metamodels in the ECLIPSE world);

– the Microsoft approach: tools and concepts based on *domain specific language* (DSL).

Let us note that the approach proposed by the OMG is not well equipped, contrary to the other two. Therefore, it is up to the user to make his or her workshop.

1.4.1. *The concepts*

The MDE mainly consists of using the models in the different phases of the developing cycle of an application. There are three levels that we will consider:

– the requirements or *computation independent model* (CIM);

– the analysis and the design or *platform independent model* (PIM);

– the "pre-code" or *platform specific model* (PSM).

The main objective of the MDE is the elaboration of models that are independent of the technical details of target platforms. This independence must engender, in the long run, the automatic generation of (via model transformation) a large part of the code of the applications, besides spurring a gain in productivity.

Another principle that is highly significant in MDE is the allowing of the transformation of the existing models into the target representation. Thus, whatever the technology used, it is possible to pass very easily from one technology to another, provided that we have the necessary transformation tools. These are based on transformation languages that must follow the *query/view/transformation* (QVT) norm proposed by the OMG.

1.4.2. *The technologies*

It is crucial to know which technology to use. Indeed, for a given domain of application, there are at least two options:

1) writing a metamodel of the domain of application, and then running adapted equipment:

2) writing an UML profile of the domain of application before using an existing "profile" tool.

We will illustrate the advantages and disadvantages of these two techniques using the "model editor" as an example.

In the first case, there are no modeling tools for the chosen technology. The approach consists of writing the metamodel of the technology (i.e. a DSL) and then generating an UML modeling tool (in general 60% of automatic generation with the EMF tools). Once the UML modeling tool is generated, the engineers create a "specific" tool, allowing us to carry out technology models.

In the second case, the choice is made to use an existing UML modeling tool. The approach consists of writing the profile of the technology. Once this profile is created, the engineers create a profiled generic UML tool, allowing us to carry out technology models.

In the first case, the graphic range of the editor proposes concepts such as "buffer" and "task". In the second case, the graphic range of the editor proposes the usual UML concepts such as the "class", but the class can only be stereotyped as a "buffer" or a "task".

The two techniques have their advantages and disadvantages. In the first case, the job engineers have an editor that gives them a job-specific vocabulary, which is an

advantage. In contrast, in the second case, they must pass through the notion of "class" before introducing the job concepts, which can be misleading.

In order for these two techniques to be equivalent, we should be able to configure the UML modelers so they present the job concepts without passing through the UML concepts, as we could observe it with SysML or the famous *block*, which is, in fact, a class stereotyped as *block*.

For the time being, the two techniques sit along side-by-side in the industry, belonging to opposite camps, each of them arguing in favor of its advantages.

1.4.3. *The context of the wider field*

The major objective of the models is to facilitate mutual understanding, exchanges and the communal work done by the actors of a project. Their construction and representation must therefore observe certain conventions and rules, which are turned into several norms and standards.

The modeling language determines the manipulated concepts, their semantic and their representation under a textual and graphic form. The variety, of the preoccupations both in the early design/development stages in the domains of application and in the specialties involved (safety, reliability, human factors...) has lead to so many languages that it is impossible to enumerate them all. They can, however, be classified into three large categories:

– The languages with a generic aim: the main language is UML, SysML and MARTE being seen as its derivatives for system engineering and real-time embedded systems.

– The more narrow, specialized languages, which are associated with formal verification methods (automata, Petri nets, B, Lustre, etc.) and allow a mathematical verification of certain expected properties.

– The non-formal specialized languages, connected to certain domains of application, specialties or specific preoccupations, whose terminology they integrate in their construction, as well as incorporating their rules and concepts. The DSL's come from this category. *Workflow* languages, such as the *business process modeling language* (BPML), will be useful in describing the needs for interaction between human beings and the systems.

In the case of embedded computer systems integrated in wider systems, it is necessary to use dedicated architectural frameworks to master both a global consistency and a potential evolution in the future. These frameworks structure and specify how to define the architecture of a system (or the way it will be used by the end user organization). To do that, they define the different required viewpoints for

its description, as well as the metamodel necessary to ensure the consistency between the different viewpoints and for carrying out impact analyses.

In comparison to the diagrams prescribed by languages such as UML, the views of the architecture frameworks distinguish themselves through the larger spectrum that they cover (they consider the dimensions that are necessary for the overall governance: enterprise organization and processes, evolution of the capacities, synchronization of the projects, etc.) and they are limited to specifying the type of information that must be provided, this giving a lot of freedom for the actual manner of representation. The NATO Architecture Framework (NAF) is one of the most recent and most advanced architecture frameworks. This is why, it is used beyond its military scope.

Name of the standard	Role of the standard
Fundamental concepts for modeling	
ISO/IEC/IEEE 42010 (2011): System and software engineering – Description of the architecture	Defines the principles that must be respected and the fundamental notions (viewpoints, architecture frameworks, architecture description languages, etc.)
IEEE 1471 (2000): IEEE Recommended Practice for Architectural Description of Software-Intensive Systems	As a reminder: replaced by ISO/IEC/IEEE 42010
Architectural modeling languages	
OMG UML: Unified Modeling Language V2 UML: ISO/IEC 19505 (V2.4.1 - 2012): Information technologies – Unified modeling language	
OMG SysML: System Modeling Language Specification (V1.3 - OMG, 2012/06)	Extension of the UML to system engineering
UML/SPT: Profile for Schedulability, Performance and Time (OMG, 2005)	Extension of the UML for the modeling of real-time systems. Replaced by the MARTE profile
UML/MARTE: Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE 1.1 – OMG, 2011/06)	Extension of the UML for the modeling of real-time and embedded systems. Replaces the SPT profile
UML/QFTP: Profile for Modeling Quality of Service and Fault Tolerance Characteristics & Mechanisms (OMG, 2008)	Extension of the UML in order to facilitate the modeling of aspects such as the quality of service and the tolerance to application faults
UML Testing Profile (UTP) (V1.1 – OMG, 2012)	Extension of the UML with concepts relative to the tests
SAE AS 5506 rev. A (2009): Architecture Analysis & Design Language (AADL)	Language for the description of the architecture (software + execution platform) of the real-time embedded systems with critical performance
IEEE Std 1320.2 (1998) – IEEE Standard for Conceptual Modeling Language – Syntax and Semantics for IDEF1X97 (IDEFobject)	IDEF is a family of modeling languages for software and system engineering
OMG BPMN 3.0 (2011): Business Process Modeling and Notation	Graphic notation standard serving to describe the processes of the enterprise (as well as the man–system interaction)
The Open Group – ArchiMate 1.0 Specification (2009)	Enterprise architecture description language
Architectural frameworks	
NATO AC/322-D 0048 (2007): NATO Architecture Framework V3 (NAF V3)	Architecture framework for large (military) systems
ISO 15704, Industrial automation systems – Requirements for enterprise-reference architectures and methodologies	
The Open Group Architecture Framework (TOGAF)	Enterprise architecture
ISO/IEC 10746 (1998): Information technology – Open distributed processing – Reference model (RM-ODP)	Description of distributed information systems

Table 1.1. *The main standards for system modeling*

Table 1.1 gives an overview of the main standards used for system modeling. The main organizations that devise the rules and standards regarding architecture modeling of embedded systems are:

– the International Organization for Standardization (ISO);

– the Institute of Electrical and Electronics Engineers (IEEE), which is an American professional organization; that publishes its own standards via the *IEEE Standards Association*;

– the OMG, an American association whose objective is to standardize and promote the object model in all its forms.

To this list, we may add some so-called "sectoral" actors:

– the International Society of Automotive Engineers (SAE International), which is a world association covering the aerospatial domain as well as the one of terrestrial vehicles, having the aerospace standard (AS) standards and ground vehicle (GV) standards;

– NATO, mainly in the military domain;

– *The Open Group*, for the enterprise information systems;

– the Electronic Industries Alliance (EIA), which stopped its activity in 2011 and has been replaced by five more specialized associations, among which the Governmental Electronics and Information Technology Association (GEIA);

– the European Cooperation for Space Standardization (ECSS), which holds a set of norms dedicated to spatial projects management, organized in three branches: projects management, product insurance and system engineering;

– the Requirements and Technical Concepts for Aviation (RTCA).

1.5. System analysis

For a long time, systems have been validated via tests and simulations whenever the executable models were available. However, this exploratory approach does not guarantee that the set of possible executions is well covered. So-called "shadow areas" within the execution of the system can still exist, and they can hide various faults. This is in particular true in the case of embedded systems. Non-functional constraints (such as energetic consumption and performance) can, indeed, be added to the functional constraints that are already difficult to verify.

To pass beyond the so-called "horizon", which is the limit of traditional simulation and test approaches, we must use formal methods, as shown by numerous

studies. Only formal methods guarantee that a property is verified because they rely on mathematical models' existance and enable us to reason on the basis of the specification.

In this domain, there are two families of formal verification techniques: theorem proving and model-checking. Let us briefly recall the principle of the first family that will not be further explained in this chapter because we will not use it. In the following, we present in more detail the model-checking approach, which is used in Chapter 8.

1.5.1. *Formal verification via proving*

The tools and emblematic languages of this family of formal techniques are Coq [AFF 08], Z [ISO 02] or B [ABR 96]. They all have their own success stories such as the behavior verification of metro line 14 in Paris, using method B.

The basic principle is to axomitize the system to be verified using a mathematic notation. Then, properties are theorems to be demonstrated from these axioms, possibly by means of intermediary intermediate lemmas or theorems. A "proof assistant" is used to help engineers to explore the "demonstration space". However, this assistant cannot perform the proof automatically.

The theorem prooving approach allows us to demonstrate that the system respects the desired properties. In addition, we know the validity conditions for these proofs, such as the parameters of the initial state, required for verifying the properties. They, therefore, are extremely useful data for the system designers.

However, this technique has several drawbacks. The first one is that it is difficult to put into practice. This approach requires highly qualified engineers mastering both the considered application domain and the demonstration techniques. Another difficulty refers to the absence of a diagnosis demonstration that cannot be performed (i.e. the exploration carried out by the proof assistant fails). Only an extremely high degree of expertise permits, in certain cases, to understand whether the absence of proof is due to the system itself or due to a modeling error.

1.5.2. *Formal verification by model-checking*

The *model-checking* principle [QUE 82, CLA 86, CLA 00] is very simple. The system is described using an executable formal notation, where a state is generally represented as a vector if values (e.g., the state of the variables of a process). This system is then simulated exhaustively, which can be done because the nature of the state allows us to tell, for each explored system configuration, whether it has been encountered before or not. Thus, if the system is finite, we may explore its state space exhaustively and look for the properties we wish to check (there are also model-checking techniques dedicated to infinite systems [DEM 11]).

The main advantage of this technique is that it is completely automatic; its use does not require the engineers to have any specific knowledge (only the specification language must be well known). Furthermore, in most cases, the response is either "yes, the property is verified", or "the property is not verified, but here is a counter-example that leads to the violation of the property". This information is a useful diagnosis, which can be used directly by an engineer on the basis of his knowledge of the system only.

Model-checking also suffers from a number of disadvantages. The first one is the combinatorial explosion of the number of states in complex systems [VAL 98]. This is particularly true when we introduce parallelism or when we wish to analyze time-based constraints. To avoid this problem, we must develop specific techniques that only function in certain cases. The engineer must then adapt their specification or use certain tools rather than others. Such cases temper the automatic use of model checking since, a deeper understanding of the underlying techniques is required to operate them.

The other problem is that the system cannot be verified in a parameterized way, as in the case of a proof, but only for given an initial configuration. This can make it difficult to identify the conditions that prompt a system to observe the desired properties.

Finally even if the counter example is small compared to the system complexity, the user may recover a trace of the 10^8 steps that are indeed difficult to analyze. Despite this, it is a good downsizing factor in comparison with a system that comprises, say, 10^{80} states.

In the rest of this section, we will look into the conditions that enable model-checking to verify systems.

1.5.3. *The languages to express specifications*

Expressing the specifications is a crucial problem because the verification algorithms function on the basis of the expression of the specification. In general, we can distinguish two parts in the system specification: the system itself and the properties it must respect. Let us note that, in general, the expression of the properties is harder than it appears.

1.5.3.1. *System modeling*

To be useful for the verification, we need a language with a formally defined operational semantic. Thus, the language must not only be executable but the notion of progression also needs to be formalized. These languages usually require two elements: the notion of state and the notion of transition between two states. More

particularly, in order to apply certain algorithms, we need the transition between two states to be reversible (which is not always the case in the "real world").

Thus, it is difficult to directly apply model-checking to a programming language whose semantic is too sophisticated (let us note that there are works aiming at directly verifying the programs, however this will be tackled later). For example, memory allocation is typically a complex problem that can be addressed using model-checking. *A priori*, automata are the basic language for model-checking, given that the state space is nothing other than an automaton whose nodes are the configurations (the states of the system – the vector we mentioned above) and the transitions and the relations between these states.

Thus, we obtain the state space of a system from a product between the automata that represent its components. However, given that automata are not necessarily the easiest model to use, it is often necessary to use "automata generators", such as Petri nets [DIA 09], or languages with possibilities restricted to a well-defined behavioral model such as PROMELA [HOL 97, HOL 04], CSP [HOA 85] and FIACRE [BER 08].

Finally, recent studies concern the transformation of high-level languages into formal languages. Let us mention:

– the transformation of C [ZAK 08, JIA 09] programs or Java programs [VIS 05] (in general, a language subset) into PROMELA;

– the transformation of some UML diagrams into Petri nets [KOR 10];

– the transformation of AADL specifications into Petri nets [REN 09] or FIACRE [COR 10].

1.5.3.2. *The expression of properties*

Once we know how to express the state of a system, the properties can be specified as logical formulas expressing constraints on the components of the vector that describes a state. We thus obtain an atomic expression allowing us to characterize a state pattern such as the formula below, which implies three variables V_1, V_2 and V_3:

$$V_1 < 4 \ \wedge (V_2 > 5 \ \vee \ V_3 = 5)$$

This formula can be assessed throughout the exploration of the state space of the system. We call these safety or reachability properties because they aim at identifying a state that observes the given pattern. When such a state is reached, the algorithm stops the construction and looks for a path between the initial state and the state that is

being characterized: this is the counter example. If the state space is explored without encountering the pattern, then the property is not verified. We can thus verify the absence of the undesired states in the system.

However, atomic expressions cannot express causal relationship between states. For example:

$$M_{received} = request \Rightarrow in\ the\ future, M_{sent} = response$$

This formula relates all of the states that correspond to the reception of a request to the fact that in the future, the server will necessarily send a response. These are temporal formulas (in the causal sense of the term). The atomic expressions are connected to this kind of formulas via temporal logic operators [WIK 12]. There are several classes of temporal logic, the most well known being the *linear temporal logic* (LTL) and *computation tree logic* (CTL).

For the management of time properties (i.e. involving time), engineers have developed TCTL or TLTL, which are extensions of CTL and LTL. CTL (and LTL) operators are then annotated with time intervals. Thus, the previous request, when timed, becomes, for example:

$$M_{received} = request \Rightarrow in\ less\ than\ 10\ time\ units, M_{sent} = response$$

The formula will not be verified unless the sending of the response follows the reception of the request in less than 10 time units.

A logic standardized by the ISO was elaborated in the 2000s: property specification language (PSL) [EIS 06, IEE 10]. It integrates notions coming from classic temporal logic and allows for time management or probability management.

There are much more than one algorithm required for assessing different types of formulas which varies in complexity. The reachability is the simplest (complexity in the size of the state space), then follow the algorithms for temporal logic formulas and, finally, those concerning timed temporal logic or probabilistic temporal logic. In certain cases (for instance, for timed systems), properties are undecidable: there is no algorithm that can systematically solve the problem.

The main difficulty in carrying out the model-checking approach lies in the capacity of the engineers to easily express the requirements that must be verified on a system. Temporal logic classes have a large power of expression and allow for a rigorous expression. However, in practice, they are difficult to handle in an industrial context because they demand great expertise from engineers. Indeed, a requirement

can reference numerous events, which are connected to the execution of the model or of the environment, and it depends on an execution history that must be considered when at verification time.

One solution to this problem is the use of dedicated languages, which allow us to express properties and abstract certain details, at the cost of reducing the expressiveness. Numerous authors have made this observation and some of them [DWY 99, SMI 02, KON 05] have proposed to formulate properties with the help of definition patterns. A pattern is a textual syntactic structure that allows a mode of expression that is closer to the languages used by engineers.

Another way of simplifying the expression of the requirements comes from the fact that, in the requirement documents, they are often expressed in a given context of the system execution. The requirements are associated with specific phases of the system execution. In [KON 05], the authors have proposed to identify the scope of a property by enabling the user to specify the temporal context of the property with the help of the operators (*global*, *before*, *after*, *between*, *after-until*). These allow us to associate the requirements to a particular temporal context of the execution of the model to be validated. The scope indicates if the property must be considered, for example, during all of the execution of the model, before, after or between some occurrence of events. The analysis presented in Chapter 8 (Part 3) is inspired from this notion.

Let us note another technique adapted to temporal properties involving the detection of evens: the approach based on observation models [HAL 93]. The main idea is to overload the model with elements that are mere observers (non-intrusive), which observe some particular states of the system. It has been proved that with such observers, we can express these formulas in the form of an accessibility property [KUP 99]; therefore, a property can be verified by simpler algorithms. However, this needs a modification which is sometimes complex in terms of specification. We must also ensure that this has no side effect on the behavior of the system (i.e. that the observer must remain neutral and should not hamper certain behaviors).

This procedure has been popularized with the LUSTRE [HAL 91] language and has been reused by the UPPAAL tool [UPP 12]. The expression of an invariant property by means of an observer is simpler, yet its complexity is correlated to the complexity of the initial property (it can be up to 20 states for realistic observers).

1.5.4. *The actual limits of formal approaches*

The formal approaches are gradually becoming more present in several industrial sectors, becoming more and more indispensable for ensuring higher reliability. An

indicator is the adoption, in the aeronautic standard DO178C, of formal techniques for system design. However, several questions yet remain open.

The first one considers the connection between the system and the model. Verification is carried out on a specification and not on the system in itself, often expressed in inappropriate terms or being too complex. Therefore, we must ensure consistency between the two, or else properties demonstrated on the model may not be true on the real system. This means that we need to use a very rigorous methodology.

To ensure this consistency between the system and the model, model engineering proposes approaches that involve transformations (several approaches are described in this book). The formal specification is thus generated from a high-level model that also serves to produce the code of the final application. This raises two issues:

– The approaches via transformations must preserve the execution semantic between the source model and the target model, which is difficult to demonstrate.

– The formal specification therefore become too complex: again we meet the problem of a combinatorial explosion, which is specific to model-checking approaches (or to their equivalent for proof-based approaches).

Finally, the use of formal methods requiring complex software tools raises the issue of their certification when they contribute to the elaboration of certified programs. One such successful experience has been carried out with the SCADE code generator, which is certified, and produces code that does not require any further certification efforts. The high licensing cost involved in certifying the code generator for SCADE, however, makes it rarely used. Specialists are debating about potentially using methodologies that allow for the use of uncertified tools in the development of certified software, an analysis of the validity of the results of these tools thus becoming indispensable. For a code generator, a simplified procedure for certifying generated programs, therefore, needs to be maintained.

These elements, along with the need for highly qualified engineers, are obstacles to the adoption of formal methods on a wider scale. It does not, however, stop progression of formal method use in computer science thanks to the outcome of research on the one hand and the increasing demand for system reliability, on the other hand.

1.6. Methodological aspects of the development of embedded computer systems

Since World War II, and then with the big space programs of the 1960s, the willingness to master the development of systems has grown considerably. To do this, we must have the various teams as well as the various specialties involved to

collaborate more efficiently in the design and in the production of these complex systems. Thus, the actors concerned had to formalize the nature of the activities required to pass from more or less well defined stakeholders needs to a real system.

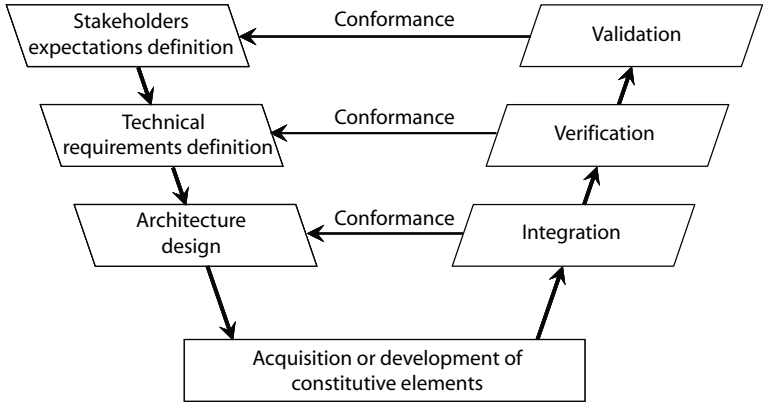

Figure 1.2. *V development cycle as proposed by the AFIS*

One of the most famous examples is the V development cycle (see Figure 1.2). This has become a standard in the industry since the 1980s. It distinguishes between:

– a downward specification/design branch, which groups together all the activities starting from the "need" to the "on paper" definition of the technical solution;

– an upward branch that covers the integration, verification and validation of the complete system;

– between the two, a horizontal passage corresponding to the acquisition or the development of constitutive elements (with, if necessary, other overlapped V cycles; the procedure being iterative).

This vision is of course simplified because it presents a perfectly sequential succession of activities. In reality, it also goes backward and iterations are inevitable. They can come from certain gaps or from evolutions of the requirements (certain aspects will only be emphasized via questions that arise during the design) as well as from interactions between specialities and/or from the search for better compromises, not to mention the problems detected during the integration or during tests.

This has led to more sophisticated development lifecycles (incremental, spiral, evolving, product lines, etc.), in particular for the software intensive systems. Nowadays, all of standards in engineering implement the following distinction between two dimensions:

– the *phases*, which punctuate the project, relying on an adapted development lifecycle model and facilitating the understanding of the entire process (for instance, the specification phase of the system, which results in supplying the technical specification);

– the *process* (set of connected activities), which indicates the activities that must be carried out; they are, essentially, independent of the type of system and the considered application domain.

The same type of activity may concern several phases. Thus, design activities may be necessary very early in the process, for instance for designing prototypes to illustrate and elaborate the requirements. Similarly, the verification and validation processes obviously apply to the final system, but they can also regard the results of other phases in order to detect problems or drifts as soon as possible (usually forgotten requirements/constraints or the impact of an undesirable event). The objective is to limit the impact of these problems on the production costs (money, time) of the system. It is the responsibility of the project management to select, organize and plan the activities that must be carried out.

1.6.1. *The main technical processes*

The main technical processes considered in engineering standards are, notwithstanding close variations in scope and terminology, based on the same pattern. Let us note that these processes indicate what needs to be done, but they do not suggest how. This is the objective of the methods, which are closely related to application domains as well as to specialties. In the processes usually considered, we find the following elements:

– The stakeholder requirement definition: this involves the collection, analysis and formalization, with the involved stakeholders (clients, users, support), of what the future system will have to do.

– The requirements analysis: this involves the translation of the stakeholders' requirements needs into measurable technical terms, without considering any possible implementation. The system is then seen as a "black box" whose interaction with its environment is analyzed. It is also in this stage that "non-functional" requirements (safety, security, reliability, availability, etc.) as well as any potential specific constraints (regulatory, environmental, etc.) are introduced.

– The functional design: this stage defines the functional architecture of the system (independent of any technological constraint), the allocation of requirements to functions as well as the description of their characteristics and behavior.

– The physical design: this stage partitions functions of the system (we speak of logical architecture), defines the physical architecture of the solution and allocates

functions to the respective elements (hardware, software and, finally, human elements). We specify here the components to be developed along with their interfaces.

– The system integration: this involves assembling in one single system the set of components that had been developed and acquired separately.

– The system verification[2]: this stage concerns the verification actions to be done by the system designer/developer. They enable the system designer to ensure that his or her product is "well-made", with respect to development rules or standards, etc. (he or she has made the system right).

– The system validation: this involves actions that enable the client and the end user to ensure that the product responds correctly to the stakeholders' needs and requirements identified in the first stage (they have the right system).

1.6.2. *The importance of the models*

System development requires the collaboration of multiple actors and teams that are sometimes important and very varied specialties. We need, therefore, means of support, ensuring a smooth communication among all parties involved. This is the role of the models, which are abstract and partial representations of the system from one point of view and with a level of granularity that facilitates the study of some of its characteristics (properties, behavior, etc.). Each model is, therefore, designed with one precise objective in mind; however, no model is enough by itself to translate the complexity of a system.

There are several modeling languages, more or less specialized. This book deals with the languages adapted to the description and analysis of embedded systems, which are characterized by strong expectations in terms of response time, safety and security:

– SysML is a graphic modeling language used for capturing the structural, functional and behavioral aspects of the systems incorporating hardware, software, people, and procedures. It can be used in support of specification activities, analysis activities, design and verification-validation activities.

– UML/MARTE and AADL are modeling and analysis languages dedicated to real-time embedded systems. They allow us to consider non-functional properties (time constraints, safety constraints, security constraints, etc.) and to verify properties such as scheduling, good transmission of messages and the right sizing of the hardware. Their use takes place rather at the end of the design activities. They are neither methodology nor tools.

---

2 Do not mistake with "formal verification" in the sense of section 1.5.

Assigning a main role to models, model-driven engineering aims to guarantee the consistency of the elements manipulated by the different stakeholders throughout the system development lifecycle. The main idea is to gradually refine the models throughout the requirement definition and analysis, as well as design analysis and to rely on model transformation techniques in order to guarantee that a global consistency is maintained and their properties are preserved throughout each stage.

The semantic differences between the modeling languages specific to certain domains and the limited interoperability between the tools are still significant obstacles in the way of achieving this aim.

1.7. Conclusion

In this chapter, we have sketched an overview of the modeling and the analysis of embedded computer systems. Far from being exhaustive, this overview seeks to show, the richness of these activities: the variety of the description methods, the analysis techniques and the need to combine them via a rationalized process in connection with the respective constraints (i.e. domain constraints, normative and regulatory constraints).

This chapter enables the reader to better contextualize each of the notations presented. The book is devised around the same plan: the presentation of the notation, its use for representing a complex problem: a pacemaker, analysis and code generation. Reading these chapters will also enable the reader to understand when and how to use the different notations.

At first sight, we could think that SysML is useful in the early phases of the modeling process. Being a notation for system engineering, we will have to complement SysML with a notation that is closer to implementation considerations. We must therefore chose between UML/MARTE and AADL:

– UML/MARTE is an obvious candidate because it is derived from UML, as SysML is UML/MARTE which allows us to follow the modeling of the system by explicitizing some aspects of the system. However, it has various limitations connected with UML: a lack of a clear process, too much room for interpretation and a semantic that is too large, so that in certain cases we will have to use a subset of it.

– AADL, allows for a more precise modeling, with a semantic that is restricted to critical embedded systems. Furthermore, it is already connected to several analysis tools. However, since it is outside of the UML framework, we must build a more complex traceability between SysML and AADL.

We can thus imagine several combinations: UML/MARTE for allowing us to model the PIM and then the PSM of the system considered, then AADL in order to

have a coherent view of all the elements, paving the way for the integration effort, as well as the verification and validation phases. It is better, if we limit ourselves to only one notation, UML/MARTE or AADL, to model the system and perform analysis and/or code generation.

As we have seen, choosing a notation is not a simple matter, and it is, above all, a choice that must be dictated by the problem we need to solve, by the tools that are available as well as by how familiar the engineer is with the notation.

This is the overarching goal of this book, enabling you, the reader, to make such a choice, by analyzing the same case study using these three notations.

1.8. Bibliography

[ABR 96] ABRIAL J.-R., *The B Book – Assigning Programs to Meanings*, Cambridge University Press, 1996.

[AFF 08] AFFELDT R., KOBAYASHI N., "A Coq library for verification of concurrent programs", *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 199, pp. 17–32, 2008.

[BER 08] BERTHOMIEU B., BODEVEIX J.-P., CHAUDET C., *et al.*, "Verifying dynamic properties of industrial critical systems using TOPCASED/FIACRE", *ERCIM NEWS, Special Issue on Safety-Critical Software*, September 2008.

[CLA 86] CLARKE E., EMERSON E., SISTLA A., "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[CLA 00] CLARKE E., GRUMBERG O., PELED D., *Model Checking*, MIT Press, MA, 2000.

[COR 10] CORREA T., BECKER L.B., FARINES J.-M., *et al.*, "Supporting the design of safety critical systems using AADL", *15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE Computer Society, pp. 331–336, 2010.

[DEM 11] DEMRI S., POITRENAUD D., "Verification of infinite-state systems", HADDAD S., KORDON F., PAUTET L., PETRUCCI L., (eds), *Models and Analysis in Distributed Systems*, ISTE Ltd, London and John Wiley & Sons, New York, pp. 221–269, 2011.

[DIA 09] DIAZ M., (ed.), *Petri Nets, Fundamental Models, Verification and Applications*, ISTE Ltd, London and John Wiley & Sons, New York, 2009.

[DWY 99] DWYER M.B., AVRUNIN G.S., CORBETT J.C., "Patterns in property specifications for finite-state verification", *21st International Conference on Software Engineering*, IEEE Computer Society Press, pp. 411–420, 1999.

[EIS 06] EISNER C., FISMAN D., *A Practical Introduction to PSL* (Integrated Circuits and Systems), Springer, 2006.

[FIM 74] FIMMEL R.O., SWINDELL W., BURGESS E., SP-349/396 PIONEER ODYSSEY, 1974. Available at history.nasa.gov/SP-349/contents.htm.

[HAL 91] HALBWACHS N., CASPI P., RAYMOND P. *et al.*, "The synchronous dataflow programming language LUSTRE", *Proceedings of the IEEE*, pp. 1305–1320, 1991.

[HAL 93] HALBWACHS N., LAGNIER F., RAYMOND P., "Synchronous observers and the verification of reactive systems", in NIVAT M., RATTRAY C., RUS T., SCOLLO G. (eds), *Third International Conference on Algebraic Methodology and Software Technology, AMAST'93*, Twente, Workshops in Computing, Springer-Verlag, June 1993.

[HOA 85] HOARE C., *Communicating Sequential Processes*, Prentice Hall, 1985.

[HOL 97] HOLZMANN G., "The model checker SPIN", *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[HOL 04] HOLZMANN G., "An overview of PROMELA", *The SPIN Model Checker*, Addison-Wesley, pp. 33–72, 2004.

[IEE 10] IEEE 1850, IEEE Standard for Property Specification Language (PSL), 2010.

[ISO 02] ISO/IEC 13568, Z formal specification notation – syntax, type system and semantics, 2002.

[JIA 09] JIANG K., JONSSON B., "Using SPIN to model check concurrent algorithms, using a translation from C to Promela", *2nd Swedish Workshop on Multi-Core Computing*, Uppsala, 2009.

[KON 05] KONRAD S., CHENG B., "Real-time specification patterns", *27th International Conference on Software Engineering (ICSE05)*, St Louis, MO, 2005.

[KOR 10] KORDON F., THIERRY-MIEG Y., "Experiences in model driven verification of behavior with UML", *Foundations of Computer Software, Future Trends and Techniques for Development, 15th Monterey Workshop 2008*, Budapest, Revised Selected Papers, Lecture Notes in Computer Science, vol. 6028, Springer, pp. 181–200, 2010.

[KUP 99] KUPFERMAN O., VARDI M., "Model checking of safety properties", *International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, vol. 1633, Springer, pp. 685–685, 1999.

[OMG 12a] OMG, The Official OMG SysML site, 2012. Available at www.omgsysml.org.

[OMG 12b] OMG, UML profile for MARTE modeling and analysis of real-time embedded systems, 2012. Available at www.omg.org/spec/MARTE.

[QUE 82] QUEILLE J.-P., SIFAKIS J., "Specification and verification of concurrent systems in CESAR", *Proceedings of the 5th Colloquium on International Symposium on Programming*, Springer-Verlag, London, UK, pp. 337–351, 1982.

[REN 09] RENAULT X., KORDON F., HUGUES J., "Adapting models to model checkers, a case study: analysing AADL using time or colored petri nets", *Proceedings of the 20th International Symposium on Rapid System Prototyping*, IEEE Computer Society, Paris, pp. 26–33, June 2009.

[SAE 09] SAE, Architecture analysis & design language V2 (AS5506A), 2009. Available at www.sae.org.

[SMI 02] SMITH R., AVRUNIN G., CLARKE L. *et al*., "Propel: an approach supporting property elucidation", *24th International Conference on Software Engineering (ICSE'02)*, ACM Press, St Louis, MO, pp. 11–21, 2002.

[UPP 12] UPPSALA AND AALBORG UNIVERSITIES, UPPAAL Home, 2012. Available at www.uppaal.org.

[VAL 98] VALMARI A., "The state explosion problem", REISIG W., ROZENBERG G. (eds), *Lectures on Petri Nets 1: Basic Models*, Lecture Notes in Computer Science, vol. 1491, Springer-Verlag, pp. 429–528, 1998.

[VIS 05] VISSER W., MEHLITZ P.C., "Model checking programs with Java Pathfinder", *Model Checking Software, 12th International SPIN Workshop*, Lecture Notes in Computer Science, vol. 3639, Springer, p. 27, 2005.

[WIK 12] WIKIPEDIA, Temporal Logic, 2012. Available at en.wikipedia.org/wiki/Temporal_logic.

[ZAK 08] ZAKS A., JOSHI R., "Verifying multi-threaded C programs with SPIN", *15th International SPIN Workshop*, Lecture Notes in Computer Science, vol. 5156, Springer, pp. 325–342, 2008.