PREMIÈRE PARTIE

MARTE

Chapitre 1

Model based analysis

1.1. Introduction

Nowadays embedded computer systems have become more and more widespread among the automatic devices in our daily lives. We find them in various transport systems : automobiles (GPS, braking system, motor control, etc.), aviation (navigation system and flight control system, pressure control and air conditioning, etc.), in medical equipment (monitoring patients, controlling the functioning of devices such as the pacemaker, etc.), in construction building (fire safety, elevator control, power management system, etc.).

This type of software ensures automatic functions, crucial functions even, because sometimes, human lives, as well as highly important economic stakes are at risk. It is thus very important to ensure the well functioning of embedded computer systems, that is, to make sure that there are no errors or bugs when compared with their functional specs. As a result, finding the bugs and ultimately fixing the system in accordance with the system specs have always been two very significant phases in the software engineering cycle, ever since software systems have been first industrialized.

In the past, as long as embedded computer systems remained relatively simple (that is, as long as their inner working, as well as the environment they were meant to control, remained simple), the task of finding the bug could be done manually. The main body of systems that had to be validated was, in this case, often designed and implemented as a sequential and deterministic software that would run in a loop when interacting

---

Chapitre rédigé par Frédéric BONIOL, Philippe DHAUSSY, Luka LE ROUX and Jean-Charles ROGER.

with an environment sample. The validation phase was carried out through a series of tests or simulations. However, the fact that the functionality of these systems has improved gradually and also that they have become more and more widespread has lead, on the contrary, to an irreversible increase in their complexity. These complexities can be divided into two categories.

Firstly, there is an external complexity connected to the environment that the embedded system must control. This environment is indeed more and more often made of several distinct yet interdependent physical entities, entities that must be controlled coherently and simultaneously. This is, for instance, the case for flight control systems that need to manage more than twenty control surfaces at the same time, being thus connected to more than thirty sensors.

Secondly, there is an internal complexity that is connected to the software architecture of the embedded system.

This architecture is also more and more often composed of several software processes that interact with each other at the same time.

In both cases (internal and external), the simultaneity that characterizes the entities or the processes leads to an explosion in the number of possible behaviors ; this, in turn, leads to an explosion of test scenarios that must be looked into before validating the system. To test this explosion, it suffices to consider a system made of ten actors that interact simultaneously, each of them carrying out a simple sequence of nine actions. Supposing an extreme scenario where the actors are completely asynchronous (i.e. there is no synchronicity between the actions of each actor), then the number of possible combinations between the 90 actions, and therefore the number of possible cases of running the system in such a way as to validate it, becomes of the order of 3.10^{82} - which is almost equal to the number of atoms in the universe.

Obviously, totally asynchronous systems or environments do not in fact exist, since the actors and their actions are often at least partially in synch. It remains however true that real systems are often made up of different processes (of the order of tens) and they execute a large number of actions (often much more than nine), and also that the environment they have to control can also be composed of several entities (usually, of several tens of entities). The number of potential behaviors to be validated thus becomes enormous. This phenomenon is known as a "‘ combinatorial explosion "’. Its immediate consequence is that manual testing methods and simulations are no longer adequate.

As a result of this observation, researchers have explored several techniques ; among these, the family of formal methods that have, over the years, added to the already existing body of competent and accurate solutions for helping designers think up non

faulty systems. As we have seen in chapter **??**, in this field, *model-checking* techniques [QUE 82, CLA 86] have been strongly promoted because of their ability to automatically carry out property tests on the software models. To this end, several tools (*model-checkers*) have been developed [FER 96, HOL 97, LAR 97, BER 04].

Broadly, the functioning of these tools consists in trying to model in a compact and abstract fashion the set of possible behaviors of the environment as well as of the system to be validated ; exhaustively carrying out each such scenario and finally, as a result, deciding if the set of possible executions is faultless. The quickness with which this entire process is carried out depends on the compaction degree of the set of behaviors. A great deal of research has been carried out in this respect [VAL 91, MC. 92, GOD 96, EME 97, ALU 97, PEL 98, BOS 05, PAR 06]. However, given the enormous size of the sets considered (often several orders of size higher than the number of atoms in the universe), the real progress of monitoring tools does not yet allow us to process the real, industrial-sized systems.

In this chapter we will lay out a complementary path, one that relies on *model-checking*, allowing us to push forward the limits of combinatorial explosion. This path seeks to process the two sources of complexity : external complexity (i.e. that of the environment) and internal complexity (i.e. that of the system). More precisely, this approach seeks to reduce the space of possible behaviors by considering an explicit model of the system's environment. In other words, we seek to "'close"' the system with the set of use cases relative to the environment it is supposed to respond to, and of this environment only. The reduction is based on a formal description of these use cases that the systems interact with, hereby called $contexts$.

The objective is thus to guide the model-checker in focusing its efforts not on exploring the entire space of behaviors–which may be enormous–but on validly restricting it in order to check for specific properties. This property check therefore becomes possible in the state space thus reduced. This method is founded on the sensors' knowledge and on their expertise, which allows them to explicitly specify the environment of the system.

As with every formal method, the approach we present here rests on a set of formal languages, a set of links between one language and the other, as well as on exploring tools and monitoring tools. In what follows, we will develop the principles of this approach by using two particular languages : Fiacre [FAR 08], for describing the model system that needs to be validated, and CDL (*Context Description Language*), for describing the use cases of the environment considered for validation.

We will then use three tools coupled with these two formalisms. The first one is the TINA SELT model-checker [1] [BER 04]. This is a model-checker that is particularly

1. Developed at LAAS, www.laas.fr/tina.

well adapted to the verification of the exigences that express invariants and that are formalized in temporal logic.

The second one is an OBP Explorer[2] that is an explorer of a Fiacre model coupled with an accessibility analyzer. OBP Explorer is more adapted to the property checks that can be expressed via monitors. Both of these tools are connected upstream to a third tool, OBP[3] ; OBP applies the method of behavior space reduction presented in this chapter.

This method and these tools are illustrated in the case study of the UML MARTE-described Pacemaker, that was presented in chapter **??**. The case study is converted in Fiacre (the input language of the two explorers/*model-checkers*), and then the performances of the two tools are being studied over two types of exigences : an invariant (expressed in temporal logic) and a monitor (expressed in CDL language).

We can then show that the two INA SELT tools and the OBP Explorer, if used without the context reduction method, cannot overcome the phenomenon of combinatorial explosion that is inherent to the case study of the pacemaker.

We will then show that considering use cases of the pacemaker's environment, formally modeled in CDL, allows us to considerably reduce this combinatorial explosion, using TINA SELT as our OBP Explorer. More precisely, we will show that breaking down the environment into separate use cases completely covering the set of environment behaviors (i.e. partitioning it, forming a partition in the mathematical sense using OBP) – allows us to push further the limits of the combinatorial explosion. We will then show that this endeavor of partitioning the environment can start automatically, each time the barrier of the combinatorial explosion is reached, thus allowing for the property check to take place on large scaled models.

The chapter is organized as follows. In section **??**, we will describe that part of the UML-MARTE model that will be considered for the transformation in Fiacre language, language whose syntax we will briefly present. We will choose two exigences that we wish to check on the model.

Section 1.3 describes the principle of exigence checking by considering use cases that correspond to the programming models of the pacemaker. Test results are shown for both tools : TINA SELT and OBP Explorer. The technique of reduction via context exploitation, using OBP, is described in section 1.4 as well as its implementing in CDL language. We present the checking methods using both tools, TINA and OBP Explorer. The chapter finishes with an assessment and a conclusion in 1.5.

---

2. Developed at the ENSTA in Brittany www.obpcdl.org.

3. From a practical point of view, OBP integrates the OBP Explorer tool.

1.2. Model and requirements to be verified

In this section we will present the part of the pacemaker model translated in Fiacre starting from a UML-MARTE model. We will also broadly present the Fiacre language and the translation principles from UMP to Fiacre. We will select for presentation two exigences from the specifications of the pacemaker that will be affected by the check.

1.2.1. *The UML-MARTE model that needs to be translated in Fiacre*

The UML model, introduced in the previous chapter, is interpreted as a set of processes (see figure 1.1) whose behaviors are described with the help of state machines. These processes communicate via FIFO type links (*First In First Out*) or via shared variables (registers).

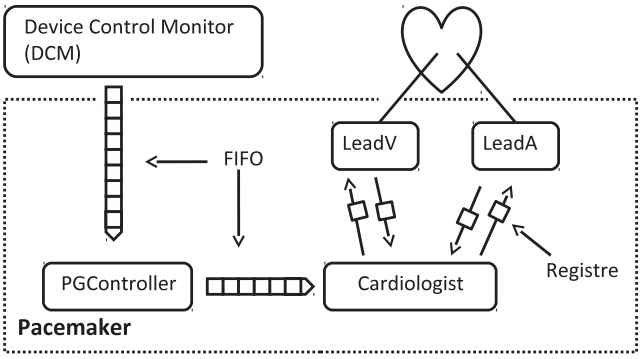

Figure 1.1 – Architectural model of the pacemaker.

The environment is composed of `DCM` (*Device Control Monitor*) and the heart. The pacemaker is composed of the `PGController` that receives the instructions from `DCM` (modes and values of the parameters), the `Cardiologist` that receives signals from the sensors `LeadA` and from the `LeadV`. These are stimulated by the heart. The `Cardiologist` takes up the responsibility of acting upon `LeadA` and `LeadV` according to the modes and parameters received from `DCM` via the `PGController`. The `DCM`, manipulated by the physician, sends parametering messages to `PGController`. The behavior of the `PGController`, `LeadA` and `LeadV` are modeled by the automatons in figure 1.2 and that of the `Cardiologist` process corresponds to the automaton described in chapter **??** and in figure **??**.

The process `PGController` receives the messages from `DCM`, processes them before sending them out to the `Cardiologist`. It ensures the coherence of the messages

(making sure there are no negative parameters or incoherent values), and sends the parameters over to the `Cardiologist`. In order to apply them to the `Cardiologist`, it first requires that the currently running mode stops, sends out the new parameters and finally starts the new mode.

In order to apply the new parameters, the `Cardiologist` process must go back to the idle mode called $Waiting$. Once the mode is configured, the `Cardiologist` process oversees the pulse ($Pulse$) of the heart and applies a stimulation policy. The `Cardiologist` is capable of counting the time that separates the two heartbeats (a cycle) and to react in accordance with the functioning parameters.

The `Leads` transform the heartbeats in messages that are comprehensible for the `Cardiologist`. They are also in charge with applying the stimulation of the heart (signal $Pace$) at the command of the `Cardiologist`.

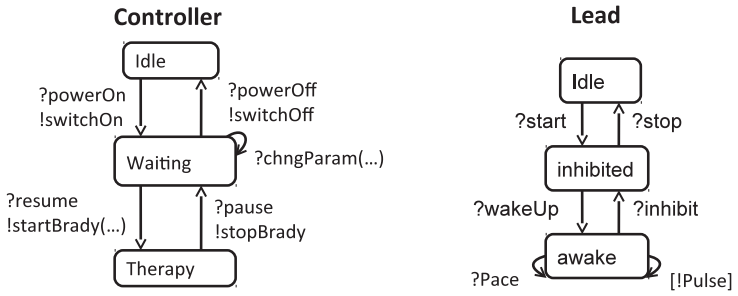

Figure 1.2 – Automatons of the PGController and Lead

In this model, we seek to model the behaviors by considering realistic scenarios, taking into account the behavior of the `DCM`and of the heart, as indicated in section 1.3.1.

1.2.2. *Fiacre language*

Fiacre language (Intermediate Format for the Architectures of Distributed Embedded Components) [FAR 08] has been developed within the TOPCASED project as a key language linking high level formalisms such as UML, AADL, SDL with formal analysis tools. Using an intermediary formal language has the advantage of reducing the semantic gap between the high level formalisms and the descriptions internally manipulated by verification tools such as Petri networks, process algebras or timed automatons. Fiacre can be considered as a language disposing of a formal semantic that serves as input language for different checking tools.

Fiacre is a formal specification language that describes the behavioral and timed aspects of real time systems. It integrates the notions of process and components :

– the automatons (or processes) are described by a set of states, a list of transitions between these states with classical constraints (variable allocations, *if-elsif-else*, *while*, sequence compositions), non-deterministic constructions and communications done via ports and via shared variables ;

– the components describe the compositions of the processes. A system is built as a parallel composition made of components and/or processes that can communicate between them via ports or shared variables. The priorities and the time constraints are associated with communication operations.

The Fiacre processes can be synchronized with or without value passage via the ports. They can also exchange data between them via asynchronous communication threads implanted by shared variables.

This is the communication mode that we use in the Fiacre model. The expression of time in Fiacre language is based on the semantic of timed transition systems (TTS) [HEN 91]. Every transition is associated with time constraints (a minimum time and a maximum time). These constraints ensure that the transitions can be done within the time intervals defined (not earlier nor later).

1.2.3. *The translation principles of the UML model in Fiacre*

These are the translation principles of the UML model in Fiacre language.

The active objects of the UML model are translated by instances of the Fiacre process. They communicate with each other via shared variables. They enable the modeling of shared registers or of message threads for an asynchronous communication in FIFO mode.

According to the indications in chapter **??**, the objects `PGController`, Cardiologist, `LeadV` and `LeadA` are implanted by the Fiacre process with the same name. On the other hand, the objects `accelerometer` `battery` and `logger` do not bear any influence on the behavior of previous objects. Therefore they have not been considered in the Fiacre modeling for the analysis of the model.

The Fiacre model has a $Pacemaker$ component (listing 1.) that reunites the instances of the processes running at the same time (operator $||$) : `PGController`, `Cardiologist` and two instances of the `Lead` process. We will add to the component an instance of the `DCM` process in order to simulate its behavior. These instances communicate via shared variables of `t_fifo` type for the message threads or of `t_registre` type for the registers. Thus, the instances of `DCM` and `PGController` processes share the variable $DCMToController$ declared as a message thread. The `PGController` and `Cardiologist` instances share the message thread $ControllerToCardio$. The `Cardiologist` instance shares with the first (respectively with the second) `Lead` instance the shared

variables $CardioToLeadA$ and $LeadAToCardio$, (the variables $CardioToLeadV$ and $LeadVToCardio$ respectively).

```
component Pacemaker is
var
DCMToController, ControllerToCardio : t_fifo,
CardioToLeadA, CardioToLeadV,
LeadAToCardio, LeadVToCardio : t_registre,
par
    DCM        (& DCMToController)
  ||  PGController (& DCMToController, & ControllerToCardio)
  ||  Cardiologist (& ControllerToCardio, & LeadAToCardio,
                  & CardioToLeadA, & LeadVToCardio,
                  & CardioToLeadV)
  ||  Lead       (& CardioToLeadA, & LeadAToCardio)    // instance leadA
  ||  Lead       (& CardioToLeadV, & LeadVToCardio)    // instance leadV
end
```

Listing 1 : Fiacre declaration of the $Pacemaker$ component

The behavior of each process is modeled by an automation. The states and transitions of UML *state-charts* are translated by Fiacre states and translations. Listing 2 partially showcases the programming of the process (`PGController`).

```
process PGController (& inputFromDCM : t_fifo_DCMToController,
                      & outputToCardio : t_fifo_ControllerToCardio) is
states
  Idle, Waiting, Therapy
var
parameters : t_parameters,   // An array of parameters (DCM level)
modification : t_modification, // A parameter index and its new value
  mode : t_Mode          // An array of  parameters (Cardio level)
init
      to Idle

from Idle
if not (empty inputFromDCM) then
  case first inputFromDCM of
    PowerOn ->
     inputFromDCM := dequeue inputFromDCM;
     outputToCardio := enqueue (outputToCardio, SwitchOn);
     to Waiting
      | any -> null // Should not happen
  end case
end if;
  loop

from Waiting
     ...

from Therapy
     ...
```

Listing 2 : Fiacre declaration of the PGController process

For the asynchronous communications, the operations $first$, $dequeue$ and $enqueue$ respectively allow us to read a message at the beginning of a thread, to delete a message at the beginning of the thread and to write a message at the end of the thread. Every process assigns for itself a unique entry thread. Writing a message in this thread ($enqueue$) corresponds to sending the message.

For this translation, the UML model is interpreted using the following semantic : the ordering of simultaneous processes (processes that take place at the same time) rests on the atomic and non-deterministic intertwining of the transitions of each processes ("'atomic"' meaning that a transition cannot be paused in the middle of its execution). In other words, for all processes that have retrievable transitions, a transition may be chosen in a non-deterministic fashion and executed atomically. Then another transition is chosen from the set of retrievable transitions. This ordering policy is in accordance with the semantic hypotheses followed in chapter **??** throughout the UML modeling of the system of the pacemaker, and corresponds to the semantic of Fiacre programmes.

As we have previously seen, the representation of time in Fiacre is symbolic, based on intervals. We shall see in section 1.2.4 that we wish to check the exigences that refer to periods of time between two signals or during a cycle.

Given the requirements that we need to check, we must be able to manipulate the values corresponding to the time periods between the events or the actions carried out by a process. We choose to manipulate variables or meters that can measure the time by breaking it down into discrete units. The time unit chosen can be parametered.

1.2.4. *Exigences*

This section will be concerned with two exigences, which have been introduced in chapter **??**, exigences that we wish to check using the Fiacre model. These exigences regard the parameters specified with the help of `DCM`.

The first exigence (*Exigence 1*) regards the tracking mode (`VDD`). In the behavior of the pacemaker, an auricular signal is followed by a ventricular signal. The parameter *Atrial Ventricular Delay* (`AVDelay`) defines the maximum time interval that may pass between these two events. In the absence of the response expected from the ventricle, the pacemaker must stimulate it. Having received an auricular signal in tracking mode (`VDD`), the pacemaker need not "'wait"' for more `AVDelay` time units. In this model, we chose milliseconds as time units. The exigence is expressed thus :

EXIGENCE 1 In `VDD` mode, the time between an auricular signal and a response or a stimulus coming from the ventricle cannot be higher than the value of the parameter `AVDelay`.

The second exigence (*Exigence 2*) describes the maximum duration of a pulse cycle that must be considered.

A `LowerRateLimit` parameter (LRL) indicates the minimum period of a pulse below which the heart must be stimulated by the pacemaker. This means that a cycle must not last longer than the interval indicated by the LRL parameter. It is expressed thus :

EXIGENCE 2 The period of a cycle must not be higher than the LRL interval.

1.3. *Model-checking* of the exigences

In order to establish the verification of a set of exigences on a model that must be validated, we need to use a model that can be simulated, as well as of exigences that are formulated, for example, by logical formulae (LTL, CTL) or by automatic monitors (see chapter **??**). We must also model the behaviors of the environment that interacts with the model we must validate. This environment corresponds to different use cases of the system that we must consider for exigence checking.

1.3.1. *Use case*

In order to showcase the analysis carried out on the model, in this chapter we will concern ourselves with the analysis of relevant exigences when the pacemaker is in nominal mode, i.e. in walking mode. In what follows we shall consider this particular context of use. Given the specs provided, the relevant parameters that we must take into account are : the choice among the seven modes ($mode$) and the different values ARP, VRP, $AVDelay$ and LRL.

The `DCM` interacts with the pacemaker by sending it the values for the different parameters, and their potential values can be multiple. In this context, the `DCM` sends out the values for the five parameters : $mode$, ARP, VRP, $AVDelay$ and LRL in any order. In our modeling, the sending out of the values for the `DCM` to the pacemaker is done by communicating a $chgParam$ message to the destination of the $PGController$ process. This message has two arguments : the name of the parameter that must be changed, and the new value. For instance, in order to put the pacemaker in mode VOO, the message sent is : *chgParam (mode, VOO)*. Figure 1.3 showcases an interaction that implements the five messages for the mentioned parameters.

The different values that these parameters can take and that we have considered in our Fiacre modeling are :

– parameter mode : seven possible modes : VOO, AOO, VVI, AAI, VVT, AAT, VDD ;

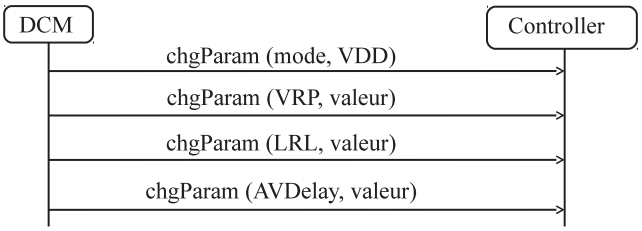

Figure 1.3 – An example of an interaction scenario (in mode VDD) betwen DCM and PGController.

– ARP parameter : solely in modes AAI and AAT with 36 possible values in each of them (from 150 to 500 ms at a pace of 10 ms) ;

– VRP parameter : solely in modes VVI, VVT and VDD with 36 possible values in each of them (from 150 to 500 ms at a pace of 10 ms) ;

– AVDelay parameter : solely in the VDD mode with 24 possible values (from 70 to 300 ms at a pace of 10 ms) ;

– LRL parameter : in all the modes, except AOO and VOO, with 62 possible values (from 30 to 175 ppm, at a pace of 5 ppm up to 50 ppm, then a 1 ppm pace up to 90 ppm, then a 5 ppm pace up to 175 ppm).

Therefore, the number of values that the parameters can take on depends on the modes. This is indicated in the table 1.1. The total number of possible combinations, that is of possible instances of the scenario presented in figure 1.3, is thus 62 498.

Modes	AOO	VOO	AAI	AAT	VVI	VVT	VDD
AVDelay values	–	–	–	–	–	–	24
ARP values	–	–	36	36	–	–	–
VRP values	–	–	–	–	36	36	36
LRL values	–	–	62	62	62	62	62
No of combinations	1	1	2 232	2 232	2 232	2 232	53 568

Tableau 1.1 – Number of different values for the parameters

1.3.2. *Properties*

Exigence 1 can be expressed under the form of an invariant of the system. In `VDD` mode, the `Cardiologist` process enters the state *TrackedPacing_WaitingVPulse* when it receives an auricular signal and waits for a ventricular signal. The variable `timeCount` of the `Cardiologist` then measures in units the time that has passed in that state. This allows us to transform the expression of the exigence in the following invariant :

INVARIANT 1 The state `TrackedPacing _ WaitingVPulse` of the `Cardiologist` process implies that the value of the `timeCount` is lower than or equal to `AVDelay`.

This invariant is written in SELT linear logic for the *model-checker* TINA SELT :

```
[] ( cardio_1_sTrackedPacing__WaitingVPulse
     => {cardio_1_vtheMode.AVDelay} >= cardio_1_vtimeCount
   );
```

Exigence 2 could equally be expressed in the form of an invariant of the model. However, this would require that we adapt the model by adding extra variables that could be referenced by the invariant. Without these variables, the time that has passed in the current cycle would not be known by our system. We will thus choose an alternative that consists in encoding the exigence using a monitoring automaton (*Obs 2*) [HAL 93] illustrated in figure 1.4. Let us explain its principle.

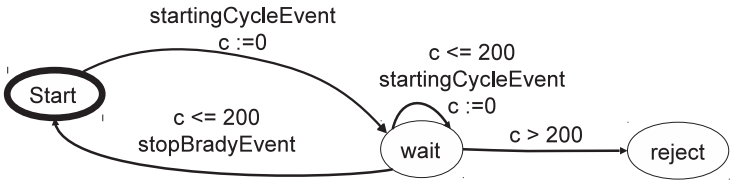

Figure 1.4 – Obs 2 : timed monitor that encodes exigence 2.

A monitoring automaton is an automaton that is sensitive to events taking place throughout the exploration of the mode and of the environment, events such as sending and receiving messages, changing the state of the processes and the variable values. With each execution of a transition in the model or the environment, the monitor executes a transition (if it exists) that corresponds to the event that just took place. The monitor has an error state $reject$. Access to the $reject$ state means that the property encoded by the automaton has been falsified. An accessibility analysis consists of researching possible scenarios that have been observed during the exploration of the system model and its environment, leading to the $reject$ state of the monitor. Several monitors can thus be defined by the same exploration of the model. Regarding the monitors, we can thus encode certainty-type properties as well as limited vitality-type properties. The coding interest on the part of the monitors is to be able to express the properties that are more difficult to express in temporal logic such as LTL or CTL. The monitors cannot be managed in the current version of TINA SELT, so we will use the OBP Explorer tool to check them.

As regards exigence 2, in order to be able to measure the time that separates two beginnings of a cycle, we will time the monitor. The timing must be done with the

help of a clock whose value increases in synch with the time of the simulation of the model. The clock can be reset to zero when a transition has been executed. The clocks of the monitoring automatons are being managed within the OBP Explorer tool in accordance with the semantic of the TTS (for *Timed Transition Systems*).

In the example in figure 1.4 that encodes exigence 2, the monitor goes into $wait$ mode as soon as the event occurs `startingCycleEvent`. This event is defined starting from a predicate `startingCycle` that we will specify as follows :

$$event\ startingCycleEvent\ is\ \{\ startingCycle\ becomes\ true\}$$

This specification corresponds to the description format of the predicates that we use in CDL language, described in section 1.4.3. The predicate `startingCycle` is thus defined starting from the states $TrackedPacing\_StartingPulse$, $Asynchronous\text{-}Pacing$, $InhibitedPacing\_StartingPulse$, $TriggeredPacing\_WaitingPulse$ of the `Cardiologist` automaton and of the `timeCount` variable :

$$\begin{array}{ll} predicate & startingCycle\ is\ \{ \\ & (\ \{cardiologist\}1@TrackedPacing\_StartingPulse\ or \\ & \ \ \{cardiologist\}1@AsynchronousPacing\ or \\ & \ \ \{cardiologist\}1@InhibitedPacing\_StartingPulse\ or \\ & \ \ \{cardiologist\}1@TriggeredPacing\_WaitingPulse\ ) \\ & and\ \{cardiologist\}1 : timeCount\ =\ 0\} \end{array}$$

A clock `c` is associated with the monitor. The event `stopBradyEvent` is detected when the stimulation is paused, that is when the message $StopBrady$ sent by the `PGController` is received. This is specified thus :

$$\begin{array}{ll} event & stopBradyEvent\ is \\ & \{\ receive\ StopBrady\ from\{PGController\}1\ to\ \{cardiologist\}1\} \end{array}$$

When this event is detected, the monitor goes back to its initial state, waiting for a new beginning of the cycle. The $reject$ node of the monitor is reached if the value of the clock goes higher than the LRL value. The clock is reset to zero with each detection of `startingCycleEvent`. The value of the clock `c` thus represents the time that has passed within the same cycle.

1.3.3. *Property check*

In order to verify the properties on the model, the latter must be composed with the use case that describe the environment of the system, then simulated and explored

by a checking tool. The exploration generates a transition system (SdT). This represents all the behaviors of the model in its environment under the form of a graph of configurations and transitions. On this SdT, we may carry out a property check, either (figure 1.5 (a), by applying, as in the case of the TINA SELT tool *model-checking* algorithms on logical formulae, or, as in the case of the OBP Explorer tool (figure 1.5 (b), an accessibility analysis of $reject$ nodes of the monitors. The difficulty with this technique is the production of the SdT that can have a large size, larger than the size of the available memory (combinatorial explosion). Should the available memory be insufficient, the verification is not usually completed.

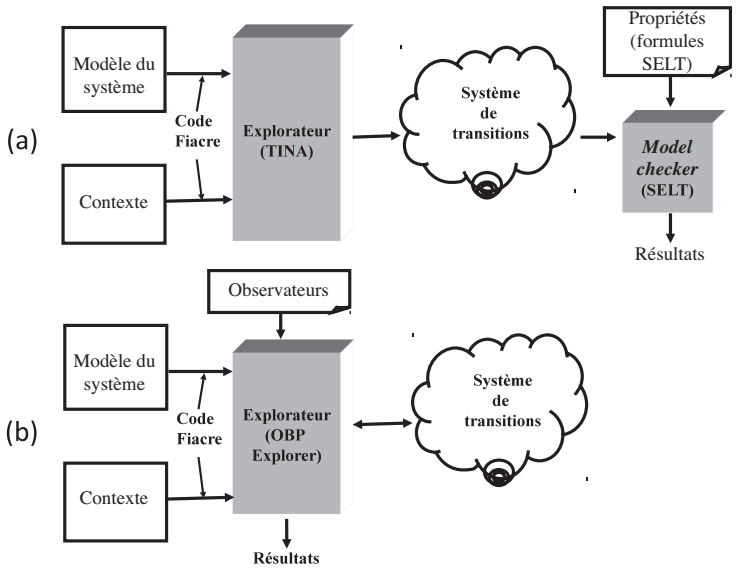

Figure 1.5 – Tested verification tools : TINA SELT and OBP Explorer

In order to be able to check the two properties mentioned, we must integrate, in the Fiacre model, the behavior of the process `DCM` in order to simulate sending the values for the five parameters to the `PGController` process. We will use the both tools mentioned previously [4] : TINA SELT for checking invariant 1 and the OBP Explorer tool for the verification of monitor 2. in figure 1.4. Both these tools differ in the moment where the actual verification takes place. In the case of the TINA SELT chain, the TINA explorer explores the system composed with its environment, generates a transition system (the graph of global behaviors), and then in a second stage the SELT

4. In both cases, the verifications are carried out on a PC, with two gigabytes of memory

model-checker checks the property on that graph. Conversely, the OBP Explorer generates the behavior graph by taking into account the properties that must be verified (coded as monitors) and carries out the verification "'on the move"', that is, at the same time as the the behavior graph is being generated. The verification, in this case, is reduced to searching for the *reject* nodes in the graph.

The table 1.2 presents the results of the verification of invariant 1, with the TINA SELT tool, for different numbers of possible combinations of value exchange between the five parameters.

The verification must be carried out on the entire set of combinations (53 568 for the VDD mode). But the verification does not end because of a combinatorial explosion. We will thus test several combinations in order to evaluate the complexity that can be processed with our tools. We note there has been an explosion for 2 592 combinations. We have only been able to explore less than 4 % of the space of these combinations, and thus only 4 % of the space of the behaviors. Therefore we cannot deduce a result for our verification. The table gives us, for each number of possible combinations, the number of configurations (states of the transition system) and of transitions explored as well as of the *model-checking* time for this mode VDD.

Number of combinations	Number of explored configurations	Number of explored transitions	Time (sec)
288	321 170	748 218	12
490	541 192	1 260 682	22
768	839 834	1 956 308	33
1 134	1 233 560	2 873 322	52
1 600	1 734 954	4 041 026	219
2 592	Explosion	–	–

Tableau 1.2 – Verification with TINA SELT of the invariant 1

The table 1.3 presents the results for the verification of exigence 2, for all the modes, with the OBP Explorer tool. All of the modes involve the application of a set of 62 498 combinations. The combinatorial explosion takes place from 352 combinations onwards (or less than 0,6 % of the space of the combinations), which stops us from being able to check the properties beyond this complexity.

Let us note that here the explosion takes place for a number of combinations (352) that is much lower than in the case of the check for invariant 1 with TINA (i.e. 2 592). This is explained by the fact that the introduction of a monitor in the exploration and accessibility analysis adds a component to the composition of the model, which in turn increases the size of the generated transition system.

Number of combinations	Number of explored configurations	Number of explored transitions	Time (sec)
7	40 455	48 781	2
42	299 740	364 391	10
110	744 224	909 908	25
226	1 783 438	2 238 305	64
352	Explosion	–	–

Tableau 1.3 – Verification with an OBP Explorer of the monitor that corresponds to exigence 2

1.3.4. *First assessment*

As a consequence of these checks, we note that the combinatorial explosion of the size of the SdT's stops us from verifying the exigences beyond a complexity that is quickly reached. The number of feasible configurations in the model explored quickly becomes too large to be memorized because these configurations are generated on the set of the model's behaviors. And this despite considering the environment of the system and its encoding in a Fiacre model composed with the model of the system. A first explanation comes from the phenomenon already observed when we consider the monitors in the exploration of the system model and its environment using an OBP Explorer tool. Considering a new actor increases the size of the space of generated states and thus reinforces the combinatorial explosion.

Generalizing from this observation, we may note that, if considering the environment in Fiacre models is indeed necessary, then considering all the behaviors of the environment at once can only lead to a more rapid explosion. In order to overcome this difficulty, and to check the properties on a number of more significant combinations, we will implement a technique whose goal is to limit the complexity of the SdT's by breaking down the environment in several use cases.

The objective is to verify the exigences, not only in a single exploration, but on several explorations. Each exploration corresponds to a sub-set of behaviors of the `DCM` in such a way that the exploration becomes possible on state spaces that are smaller and smaller in size. We will describe this technique in the following section.

1.4. Context exploitation

When we do the classical implementation of the *model-checking*, the model explored includes the use scenarios. The use scenarios describe the interactions between the components of the environment and the model. Considering the model with the behavior of the environment leads to the necessity of exploring a state space that is, most of the times, very large (figure 1.5).

Here we choose to specify the environment not only as a single global process, but also as a set of explicit and separate scenarios called *contexts*. In this approach, the environment is described as the union between all of the contexts. Each context allows us to activate sub-sets of behaviors of the model. However, the exigences that we must verify are certainty or accessibility properties, and they are satisfied if and only if they are satisfied in all contexts taken separately [ROG 06]. This enables us, with the help of *model-checking*, to no longer need to explore a "'large"' state space but several tinier spaces (as many as there are contexts) (figure 1.6). This "'divide and conquer"' approach also allows us to carry out the verifications on systems of significant size. We specify here the principle implemented with the OBP tool.

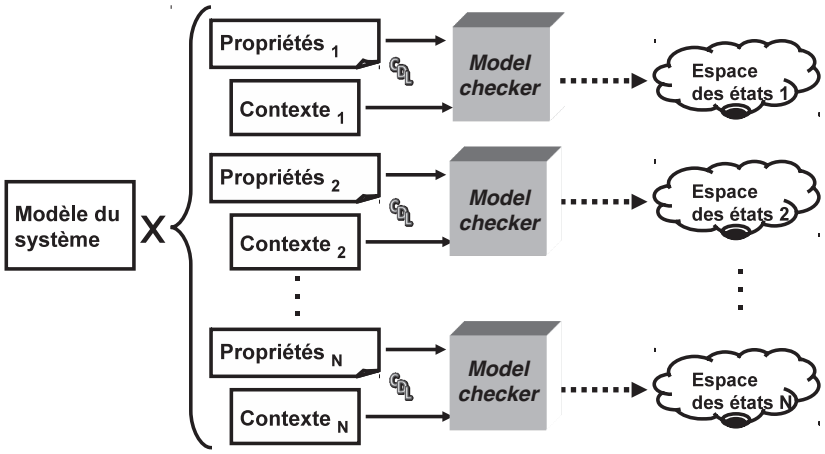

Figure 1.6 – Exploration with identification of separate contexts

1.4.1. *Identifying the context scenarios*

In order to identify the contexts and to be ale to formally specify them, the user relies on the knowledge they have of the environment of the system. They generally correspond to use modes of the modeled component. In the context of reactive embedded systems, the environment of each component of a system is often well known. It is thus more efficient to identify this environment than to seek to reduce the space of the model configurations of the system that is to be explored.

The objective is to have a description of the sub-sets of behaviors displayed by the actors in the environment (Contexte$_i$, i $\in$ [1..N] in figure 1.6) and of the sub-sets of properties (Propriétés$_i$) associated with these behaviors. This identification of the contexts can already allow us to bypass the explosion when exploring the model.

In order for this approach to be well grounded, the process of system development must include a stage where we specify the environment, thus allowing us to accurately identify the sets of concluded and completed behaviors.

The strong hypothesis underlying the implementation of this methodological process is that the designer is capable of identifying all of the possible interactions between the system and its environment. We will also assume that each context expressed in the beginning is also finished, namely that the scenarios described by this context do not have infinite iterative behaviors.

Particularly in the field of embedded systems, we base this hypothesis on the fact that the designer of a software component must exactly and fully comprehend the perimeter of its use (i.e. constraints, conditions) in order to be able to develop them correctly. It will be necessary to carry out a formal study of the validity of this work hypothesis according to targeted applications. We will not approach this aspect that requires a specific methodological work to be carried out. However, in our current approach, we must include the infinite behaviors of certain entities of the environment in our system model.

1.4.2. *Automatic partitioning of the context graphs*

When the previously described restriction of the behaviors of a model does not suffice, namely when one of the Contexte$_i$ leads to an exploding state space, we implement a second lever meant to reduce the space of these states. Each context that leads to an explosion is automatically and recursively partitioned in a set of ever more reduced sub-contexts (figure 1.7).

In order to do this, we will implement a recursive partitioning (or splitting) algorithm in our OBP tool. Figure 1.7 illustrates the `explore_mc()` function for the exploration of a $model$, with a $context$ and the verification for a set of properties pty. The context is represented by an acyclic graph. This graph is composed with the model throughout the exploration. In case of explosion, this context is automatically split into several sub-graphs (including the parameter d, which specifies the depth of the graph where the splitting starts). This recursive processing is carried out until all of the explorations have been implemented without leading to an explosion.

In fact, for every Context$_i$, we transform a global verification in tinier K_i verifications, where K_i is the number of sub-contexts obtained after partitioning the Context$_i$. In the end, this means for the set of contexts that we must transform the N verifications (as many verifications as there are contexts) in $N' = \sum_{i=1}^{N} K_i$ smaller verifications. We must thus note that the implemented partitioning technique observes the following principle : for a given context, the union of the executions – described by the set of sub-contexts generated by the partitioning of the context – includes the executions

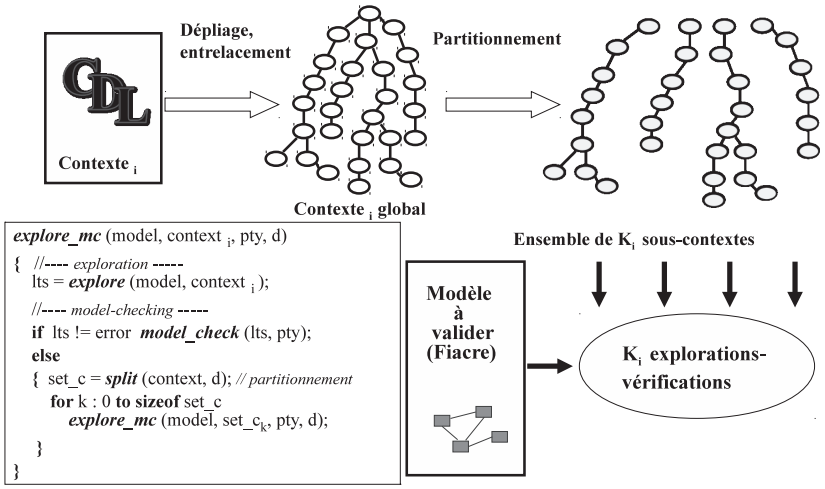

Figure 1.7 – Partitioning a context and checking each partition

described by this initial context. The properties are thus preserved by the partitioning of the context as shown in [ROG 06].

1.4.3. *CDL language*

CDL [DHA 09] is a DSL[5] drawing from *Use Case Chart* by [WHI 06] based on activity diagrams and sequence diagrams. The objective of CDL is to allow the formal modeling of the contexts of the systems (i.e. the Contextes$_i$ mentioned in the previous sections). This formalism enables us to describe the behavior of several entities (called $actors$) that make up the environment. They are executed at the same time and interact with the model of the system.

Equipped with a graphic and textual syntax, a CDL model describes, on the one hand, a scenario with the activity diagrams and the sequence diagrams. On the other hand, it describes the properties we must verify by relying on property defining patterns. A CDL meta-model was defined, as well as a syntax and a formal semantic that are described [DHA 11a] in terms of traces[6], drawing from the works of [HAU 05] and [WHI 06].

A CDL model is structured in a hierarchical fashion. A first level specifies the set of actors that make up the environment of the system and that evolve at the same time. At

5. *Domain Specific Language.*

6. For the detailed syntax, see also [DHA 11b] available on www.obpcdl.org

this level, a CDL context can be represented (in a simplified manner) by a construction of type $A_1 \parallel A_2 \parallel \ldots \parallel A_n$ where each A_i represents an actor of the environment. Each A_i is then detailed in an activity diagram, namely as a composition of MSC or activity sub-diagrams. The possible combinations at this level being the intertwining between two or more branches (i.e. the sequence) or the choice between two or more branches (i.e. the alternative). Then each MSC is described as a simplified sequence diagram of UML2.0 type [ITU 96], that describes the interactions between the actor and the system. Formally, a CDL model can be considered as a set of MSC's composed between them with the help of two operators : the sequence (seq), the parallel (par) and the alternative (alt).

When we compile a CDL model, the diagrams that correspond to each actor are laid out (examined from the angle of each finished loop) then intertwined according to Fiacre semantic and at the same time with UML-MARTE semantic. The intertwining of the set of MSC's, describing the behavior of the context, generates a graph that represents all of the executions of the actors in the environment considered. This graph is then split in such a way as to generate a set of sub-graphs that correspond to sub-contexts, as mentioned in 1.4.2. When the monitor explores the environment, each sub-graph is composed (figure 1.7) with the model that needs to be validated, and the properties are checked against the result of this composition.

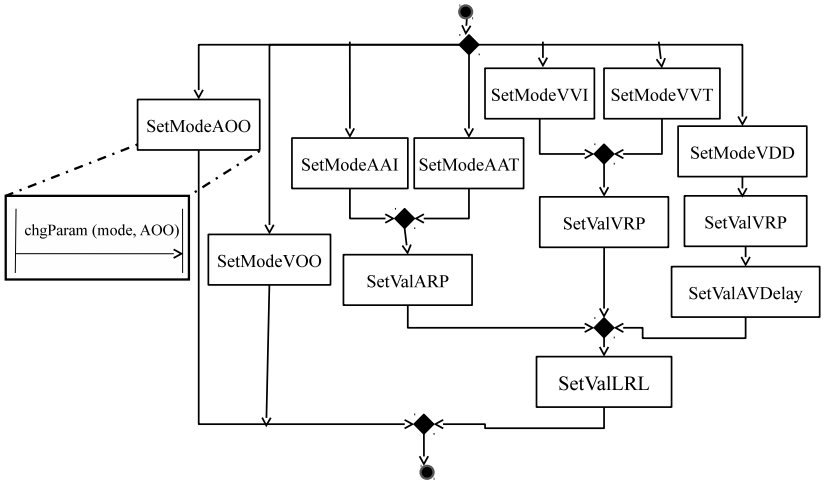

Figure 1.8 – CDL model of the DCM

Figure 1.8 graphically represents a CDL model of the DCM (which is the only actor in the pacemaker's environment). In this context, the DCM can run one of the seven MSC's $SetModeX$ (the operation « alternative » is graphically represented by

a diamond) . Each MSC $SetModeX$ models the sending of an instruction of changing the mode of the DCM towards the pacemaker. For example $SetModeAOO$ is an MSC containing the unique interaction « chgParam (mode, AOO) » between the DCM and the PGController (see figure 1.3). This mode changing having been executed, the DCM continues by executing zero, one, two, or three MSC's $SetValX$, depending on the branch caught in the first alternative. Each $SetValX$ is an MSC that models all the possible alternatives for the allocation of a value at the parameter X, or X is ARP, VRP, $AVDelay$ or LRL. For example, the MSC $setValARP$ (figure 1.9) models an alternative between 36 possible allocations of the parameter ARP. The CDL model in figure 1.8 thus represents the set of 62 498 possible combinations at the input of the pacemaker.

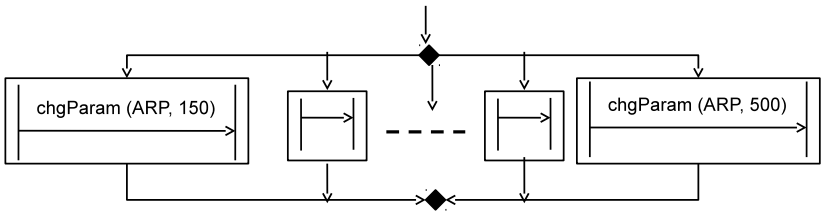

Figure 1.9 – SetValARP : value exchange for the ARP parameter

Finally, let us note that CDL thus enables us to specify the properties starting from definition patterns that define the properties in such a way as to help the user formalize them. We will not, however, approach this aspect in this chapter.

1.4.4. *CDL model exploitation in a model-checker*

The OBP tool takes as an input the CDL models, in order to convert them in two ways, as showcased in figure 1.10. Let OBP translate the CDL diagrams in Fiacre programs so that they are later submitted to TINA, and let OBP generate the data for the OBP Explorer, in itself integrated in OBP.

The Fiacre programs, or the data generated by OBP, describe in both cases a set of acyclic context graphs. They represent the set of possible interactions between the environment and the model. In order to validate the latter, it is both necessary and sufficient to compose them with each graph [ROG 06]. The properties must thus be verified against the result of each of these compositions. In the case of TINA, the SELT formulae are verified by *model-checking*. In the case of OBP Explorer, an accessibility analysis is carried out on the result of the composition between a generated graph, a set of monitors and the model. In these two cases, OBP recovers the results provided by SELT or by OBP Explorer and formats them to make them comprehensible for the user. The splitting of the context by the OBP in a set of graphs allows us to reach,

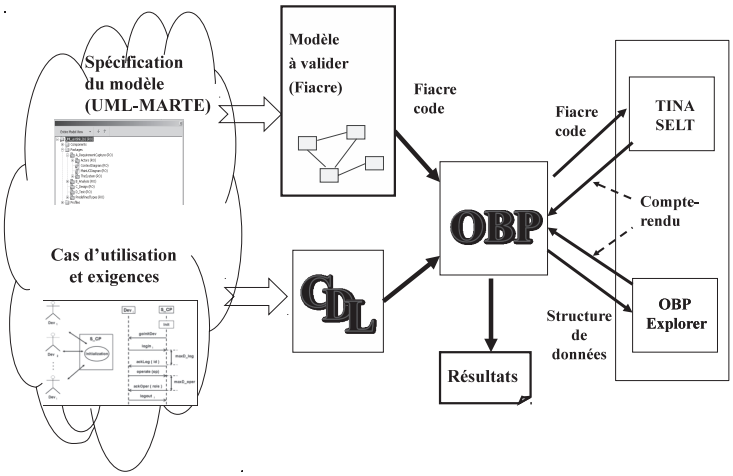

Figure 1.10 – Transformation of a CDL model with OBP

within the composition, limited-sized transition systems thus ensuring the accessibility analysis in the case of an OBP Explorer or the *model-checking* in the case of TINA.

1.4.5. *Description of a CDL context*

For the case study in question, the CDL context describes the interactions between the `DCM` and the `PGController` process. It thus includes only one CDL actor. The interactions are described as alternatives and sequences that model all of the envisaged scenarios, namely the combinations of values provided to the `PGController`. One of these scenarios, as illustrated in figure 1.8, describes sending out the mode value to the `PGController` process and then sending a value for each of the parameters ARP, VRP, LRL and $AVDelay$. The textual version of the set of scenarios illustrated by figure 1.8 is named listing 3. The operator « [] » specifies the alternative and the « ; » – the sequence. The $allModes$ scenario references $refractoryPeriodProperty$. We are concerned with the property that corresponds with the monitor *Obs 2* to be checked.

```
cdl allModes is {
properties refractoryPeriodProperty;
{
   { set_mode_AOO [] set_mode_VOO }
   []
   {
      {
          { { set_mode_AAI [] set_mode_AAT }; set_val_ARP }
        []
          { { set_mode_VVI [] set_mode_VVT }; set_val_VRP }
```

```
        []
          { set_mode_VDD; set_val_VRP; set_val_AVDelay }
      };
      set_val_LRL
   }
}
```

Listing 3 : CDL model (text version) for all the modes.

With :

– set_val_ARP equivalent to : $setARP\_150$ [] ... [] $setARP\_500$;

– set_val_VRP equivalent to : $setVRP\_150$ [] ... [] $setVRP\_500$;

– $set\_val\_AVDelay$ equivalent to : $setAVDelay\_70$ [] ... [] $setAVDelay\_300$;

– set_val_LRL equivalent to : $setLRL\_30$ [] ... [] $setLRL\_175$.

1.4.6. *Results*

We will present the results of the verification of the pacemaker by implementing the CDL contexts and the partitioning technique of the context graphs.

The table 1.4 presents the results for the verification of exigence 1 for the mode VDD with the TINA SELT tool for the exploitation of the contexts. Contrary to the experimentation shown in section 1.3.3, the verification ends in around half an hour including the full check of the DCM, namely offering the 53568 possible combinations of the VDD mode (last line of the table). The verification shows that the exigence 1. is satisfied by the model of the pacemaker.

Number of combinations	Number of generated sub-contexts	(Cumulated) number of explored configurations	(Cumulated) number of explored transitions	Exploration time (sec)
1 600	10	1 735 440	4 042 232	72
2 592	12	2 807 562	6 538 731	125
6 144	16	6 598 944	15 367 920	268
12 000	20	12 790 310	29 786 725	508
20 736	24	22 059 672	51 371 796	855
32 928	588	34 932 422	81 367 573	1 499
47 616	768	50 791 728	118 294 920	2 017
53 568	864	57 148 944	133 100 760	2 270

Tableau 1.4 – Verification of the invariant 1 (mode VDD) with TINA SELT and the exploitation of contexts.

In order to illustrate the evolution of the monitoring time, we have also tested the monitoring method on the sub-sets of DCM going from 1 600 possible combinations (from this context configuration the classical method of section 1.3.3 explodes) to the totality of combinations. We can then note that the monitoring time evolves fairly exponentially, by remaining reasonable (maximum half an hour) for a space of really large states (up to 57 millions of configurations).

The table 1.5 showcases the results for the verification of the monitor *Obs 2* corresponding to exigence 2, for all the modes, with the exploitation of the CDL model of the DCM by OBP. Here as well, we note that the verification finishes in approximately six hours, including the complete model of the DCM offering 62 498 possible combinations (last line of the table). The space of the explored states contains around 535 million configurations. As in the previous case, the time evolution of the verification is shown by a succession of experiments carried out on ever larger contexts (comprising 352 combinations in total). Here as well, the time evolution is exponential, but remains feasible in practice (six hours approximately).

Number of combinations	Number of generated sub-contexts	(Cumulated) number of explored configurations	(Cumulated) number of explored transitions	Time of exploration (sec)
352	7	2 680 017	3 362 387	92
578	12	4 519 385	5 729 766	238
884	13	7 033 893	8 987 094	241
1 282	14	10 179 272	13 069 412	437
3 746	72	30 869 430	40 072 535	1 686
8 194	228	67 893 326	88 639 720	3 963
15 202	344	126 267 775	16 5504 813	6 355
25 346	484	21 2075 376	27 8725 276	9 679
39 202	648	329 522 688	434 009 683	14 226
55 554	636	476 083 965	628 001 017	19 779
62 498	940	535 871 149	706 896 539	22 238

Tableau 1.5 – Monitoring of exigence 2 with OBP Explorer and the exploitation of contexts.

Let us note once again that the size of the generated transition system is more important than in the case of exploration with TINA. For the same number of combinations, its size is here ten times higher. As explained in section **??**, introducing the monitor throughout the exploration and the accessibility analysis increases the complexity of the generated transition system.

1.5. Assessment

The identification of CDL contexts and the partitioning technique allow us to restrict the execution of models and enables us to carry out the explorations till the end. In the case of TINA or of the analyzer internal to the OBP, the exploration of all the behaviors is impossible without implementing a context model as well as the partitioning technique of this model. We have shown that the application of the set of 62 498 combinations of the values of the parameters associated to the chosen modes can lead to a verification of the two exigences considered. What is more, the input of CDL models is to offer a formal framework for describing the exigences that can be automatically translated into monitoring automatons.

In industrial applications, the description of contexts that interact with the validating model is often informal, and sometimes incomplete. The approach allows the user to formalize this environment and to specify, in a set of CDL models, the use cases of the developed component. This formalization can only improve the design even if the user does not exploit it by using it for formal analyses. This formalization is based on activity diagrams and sequence diagrams, which are more easily accessible for an engineer. Conceptually, the CDL principles can thus be implanted without further, more standardized formalisms, such as activity diagrams or UML sequences.

In our illustration, we have considered the DCM to be the sole actor in the environment. Because we only have a unique actor in this case, the scope of the automatic partitioning may seem limited. However, this is not the case as the environment is made up of several actors. OBP builds from the behavior of the actors, from a graph of the set of actors' behaviors considering the intertwining of these behaviors. This graph is then split using the technique previously described in this chapter.

We could carry out verifications on more complex models, namely on a much more significant number of combinations. The exploration as well as the actual check would last significantly longer. In order to accelerate the verification we should increase the performance of the equipment by, firstly, increasing its memory so that we can limit the operations of context partitioning and secondly, by carrying out the processing of contexts on a network of machines [DHA 11a].

1.6. Conclusion

The objective of this chapter has been to present a verification technique (by *model-cheking*) to check the properties on the UML-MARTE model described in chapter **??**.

The approach described seeks to reduce the space of possible behaviors by considering an explicit model of the environment as a union of contexts. This model is described with the help of CDL language that allows us to describe the interactions between the environment and the formalized model that needs to be validated.

A context thus formalized can be exploited by the OBP tool and split into sub-contexts that are composed with the model that needs to be validated. This partitioning allows us to reduce, with every verification, the number of behaviors of the model to be explored, thus also reducing the combinatorial explosion. Throughout the explorations, we implement and verify the monitors. We have illustrated this technique for a set of interactions between the DCM and the pacemaker controller. This technique is implemented on the TINA *model-checker* and the OBP Explorer analyzer, which is internal to the OBP.

Bibliographie

[ALU 97] ALUR R., BRAYTON R. K., HENZINGER T. A., QADEER S., RAJAMANI S. K., « Partial-Order Reduction in Symbolic State Space Exploration », *Computer Aided Verification*, p. 340-351, 1997.

[BER 04] BERTHOMIEU B., RIBET P., VERDANAT F., « The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », *International Journal of Production Research*, vol. 42, 2004.

[BOS 05] BOSNACKI D., HOLZMANN G. J., « Improving Spin's Partial-Order Reduction for Breadth-First Search », *SPIN*, p. 91-105, 2005.

[CLA 86] CLARKE E., EMERSON E., SISTLA A., « Automatic verification of finite-state concurrent systems using temporal logic specifications », *ACM Trans. Program. Lang. Syst.*, vol. 8, n° 2, p. 244-263, ACM, 1986.

[DHA 09] DHAUSSY P., PILLAIN P.-Y., CREFF S., RAJI A., TRAON Y. L., BAUDRY B., « Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation », *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, vol. LNCS 5795, Springer-Verlag, p. 438-452, 2009.

[DHA 11a] DHAUSSY P., BONIOL F., ROGER J.-C., « Reducing State Explosion with Context Modeling for Model-Checking », *13th IEEE International High Assurance Systems Engineering Symposium (Hase'11)*, Boca Raton, Etats-Unis, 2011.

[DHA 11b] DHAUSSY P., ROGER J.-C., CDL (Context Description Language) : Syntaxe et sémantique, Rapport, ENSTA-Bretagne, 2011.

[EME 97] EMERSON E., JHA S., PELED D., « Combining Partial Order and Symmetry Reductions », *Tools and Algorithms for the Construction and Analysis of Systems*, Enschede, Pays-Bas, Springer Verlag, LNCS 1217, p. 19-34, 1997.

[FAR 08] FARAIL P., GAUFILLET P., PERES F., BODEVEIX J.-P., FILALI M., BERTHOMIEU B., RODRIGO S., VERNADAT F., GARAVEL H., LANG F., « FIACRE : an intermediate language for model verification in the TOPCASED environment », *European Congress on Embedded Real-Time Software (ERTS)*, SEE, janvier 2008.

[FER 96] FERNANDEZ J.-C., GARAVEL H., KERBRAT A., MOUNIER L., MATEESCU R., SIGHIREANU M., « CADP : A Protocol Validation and Verification Toolbox », *CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, London, UK, Springer-Verlag, p. 437-440, 1996.

[GOD 96] GODEFROID P., PELED D., STASKAUSKAS M. G., « Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs », *International Symposium on Software Testing and Analysis*, p. 261-269, 1996.

[HAL 93] HALBWACHS N., LAGNIER F., RAYMOND P., « Synchronous observers and the verification of reactive systems », NIVAT M., RATTRAY C., RUS T., SCOLLO G., Eds., *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, Workshops in Computing, Springer Verlag, June 1993.

[HAU 05] HAUGEN O., HUSA K. E., RUNDE R. K., STOLEN K., « STAIRS towards formal design with sequence diagrams. », *Software and System Modeling*, vol. 4, n° 4, p. 355-357, 2005.

[HEN 91] HENZINGER T., MANNA Z., PNUELI A., « Timed Transition Systems », *Proceedings of the 1991 REX Workshop*, 1991.

[HOL 97] HOLZMANN G., « The Model Checker SPIN », *Software Engineering*, vol. 23, n° 5, p. 279-295, 1997.

[ITU 96] ITU, « Message Sequence Chart (MSC) », *ITU-T Recommendation Z.120*, Geneva, 1996.

[LAR 97] LARSEN K. G., PETTERSSON P., YI W., « UPPAAL in a Nutshell », *International Journal on Software Tools for Technology Transfer*, vol. 1, n° 1-2, p. 134-152, 1997.

[MC. 92] MC.MILLAN K. L., PROBST D. K., « A Technique of State Space Search Based on Unfolding », *Formal Methods in System Design*, p. 45–65, 1992.

[PAR 06] PARK S., KWON G., « Avoidance of State Explosion Using Dependency Analysis in Model Checking Control Flow Model », *ICCSA (5)*, p. 905-911, 2006.

[PEL 98] PELED D., « Ten Years of Partial Order Reduction », *CAV '98 : Proceedings of the 10th International Conference on Computer Aided Verification*, Springer-Verlag, p. 17–28, 1998.

[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*, Londres, Royaume-Uni, Springer-Verlag, p. 337–351, 1982.

[ROG 06] ROGER J.-C., Exploitation de contextes et d'observateurs pour la validation formelle de modèles, PhD thesis, ENSIETA, Univ. of Rennes I., décembre 2006.

[VAL 91] VALMARI A., « Stubborn sets for reduced state space generation », *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, London, UK, Springer-Verlag, p. 491–515, 1991.

[WHI 06] WHITTLE J., « Specifying precise use cases with use case charts », *MoDELS'06, Satellite Events*, p. 290-301, 2006.