Model Checking Ariane-5 Flight Program*

Marius Bozga, Laurent Mounier
VERIMAG

David Lesens
EADS Launch Vehicles

June 28, 2001

Abstract

This paper reports a verification experiment carried out on a re-engineered description of a part of Ariane-5 Flight Program. This is the embedded software which solely controls the Ariane-5 launcher during its flight, from the ground, through the atmosphere and up to the final orbit.

In this case study, the SDL language was used to describe the main functional behavior of the flight program including the most relevant actions and their associated timing constraints, which are necessary to ensure the correct operation of the launcher. This description abstracts away both complex functionalities such as navigation and control algorithms and also implementation details, such as specific hardware and operating system dependencies.

Several properties could then be verified on this specification using the IF toolbox, an open validation platform developed at VERIMAG for real-time asynchronous systems. The results obtained confirm that model-checking and complementary techniques (such as static analysis or abstraction), combined within a set of methodological guidelines, could be successfully applied to the validation of large real-time embedded systems.

1 Introduction

It is now well admitted that the increasing importance of software within critical embedded systems will necessarily influence the techniques used to produce such systems. In particular, embedded systems tend to become more and more complex, economic constraints always impose shorter development times, and classical testing procedures are clearly not exhaustive enough to guarantee a sufficient level of reliability at a reasonable cost. All these factors motivate the introduction of new techniques within the software development process, able to better take into account the semantic aspects of the application from the earliest design stages.

Formal methods are an example of such techniques. They consist in producing some intermediate descriptions (or *specifications*) of the system under development using well-defined

*Ariane-5 is an European Space Agency Project delegated to CNES (Centre National d'Etudes Spatiales) and unambiguous formalisms. Then from specifications it becomes automatically possible either to verify at early stages some expected high-level properties, or to generate executable code, or even to produce test sequences to be executed on the final system implementation. In fact, although limited in the past to small academic examples, formal methods seem now mature enough to be used within an industrial context, even for large scale applications. In particular, this evolution is facilitated by two important factors:

- the existence of well accepted *specification formalisms*, some of them being based on international standards like Lotos [16], Sdl [17] or Uml [19];
- the support of *validation tools*, either commercial (e.g, Tau [1], ObjectGeode [20] or StateMate [15]) or academic ones (e.g, Spin [14], Smv [18], Cadp [10]) able to handle large applications and providing many useful validation facilities (from interactive simulation to exhaustive verification and automatic test case generation).

In this context, this work was initiated by Eads Launch Vehicles to better evaluate the maturity and applicability of existing formal validation techniques, both from the description language and from the validation tools point of view. More precisely, it consisted in formally specifying some parts of an existing software, on a re-engineering basis, and to try to verify some critical properties on this specification. The software that has been chosen is the Ariane-5 Flight Program. This is the embedded software which solely controls the Ariane-5 launcher during its flight, from the ground, through the atmosphere, and up to the final orbit. The experiment was carried out using IF [7], a validation environment developed at Verimag. This environment relies on a general intermediate format for timed asynchronous systems, allowing to connect the ObjectGeode Sdl commercial toolset to several academic verification tools, including Cadp and Spin. Together, these tools offer efficient verification facilities such as on-the-fly model-checking, partial order reductions, or static analysis optimizations.

Nevertheless, even if recent researches have considerably improved the tool efficiency, it is still difficult to apply them on concrete case studies. For example, our Ariane-5 Flight Program specification is about 4000 lines of Sdl, which makes clearly impossible to verify it following a strict push-button approach. From this point of view, this experiment give us some hints for a *verification methodology* of large Sdl systems, taking into account most of advanced tools functionalities. The results obtained illustrate the increasing maturity of model-checking techniques to face industrial applications.

The paper is structured as follows. First, we briefly present in section 2 the IF validation environment. In section 3 we describe the Ariane-5 Flight Program. We begin with an informal overview of the program, then we continue with some deeper insights about the formal Sdl specification, as well as its surrounding environment and functional requirements. Finally, in section 4 we point out the concrete verification results obtained, following a set of general methodological guidelines. Some concluding remarks and perspectives are given in section 5.

2 IF Toolbox

The verification tools we used during this experiment are connected through the IF *validation environment* [7], which is developed at Verimag. This environment relies on a general *intermediate format* for timed asynchronous systems, the IF *language* [8], and integrates several components operating at different levels of abstraction.

2.1 The IF language

In IF, a system is expressed as a set of parallel processes communicating either asynchronously through a set of buffers, or synchronously through a set of gates. Processes are based on timed automata with urgencies [4], extended with discrete variables. Process transitions are guarded commands, triggered by synchronous/asynchronous inputs, and performing asynchronous outputs, variable assignments, or clock settings. Communication buffers have various queuing policies (fifo, stack, bag, *etc.*), they can be bounded or unbounded, reliable or lossy, and delayed or not.

An important feature of the IF language is to provide a well-defined *real-time semantics* of asynchronous systems by means of transition urgencies. More precisely, an urgency attribute is associated to each process transition in order to define its priority over *time progress* during simulation:

- *eager* transitions are assumed to be executed as soon as possible: time does not progress as long as an eager transition is enabled;
- *lazy* transitions are never urgent: enabled lazy transitions do not disable time progress;
- *delayable* transitions are a combination of eager and lazy transitions: they are enabled within a time interval, time *may* progress within this interval, and the transition becomes urgent when its upper bound is reached.

By offering a precise control of time during system simulation, the urgency mechanism provides a flexible way to specify the real-time constraints associated with each action performed. In particular the use of eager transitions allows to guarantee an immediate response of the system to some critical events (like a timer expiration), whereas lazy and delayable transitions allow to introduce some time non-determinism in the handling of partially constrained events.

2.2 The IF validation environment

The IF validation environment provides a complete verification chain consisting in several components organized into three levels of system representation (see figure 1).

The specification level components. This level corresponds to the initial program description, expressed for instance in a high-level specification language. The formalism we considered here is SDL, and we used the TELELOGIC OBJECTGEODE [20] environment to edit and maintain our specification. This environment also provides an interactive simulator allowing to visualize execution scenarios by means of MSC. It offers some verification facilities like deadlock detection or model-checking properties expressed by GOAL observers [2].

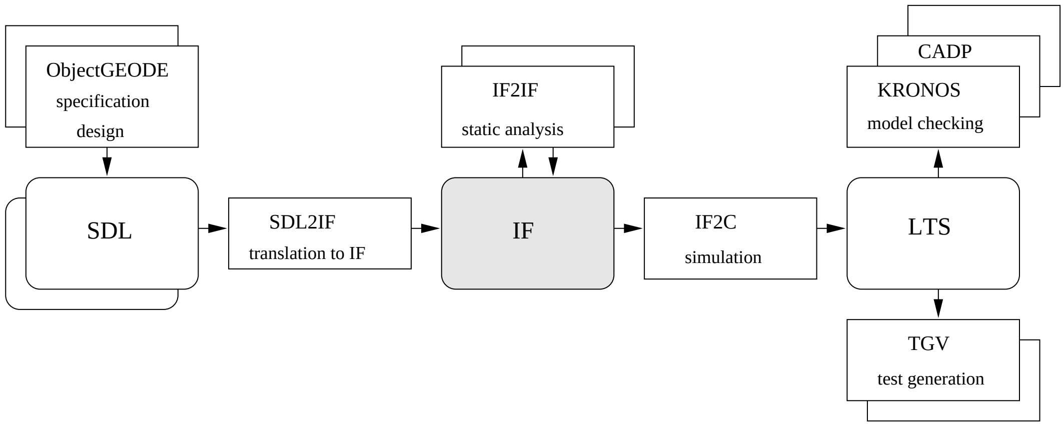

Figure 1: IF Toolbox Architecture.

The IF **level components.** At this intermediate level the initial specification has been (automatically) translated in IF using the SDL2IF translator. One of the important component available at this level is IF2IF, an optimisation tool based on static analysis techniques providing *dead variable resetting* [6], *clock reduction* and *program slicing* [21]. All these optimisations transform the initial IF specification into a "simpler" one (from the verification point of view) while preserving a particular set of properties (see section 4).

The semantic level components. This level gives access to the labeled transition system (LTS) representing the exhaustive behaviour of the IF system. This LTS is obtained by running a *simulation program*, generated by the IF2C component. The simulation program integrates several advanced verification techniques like *partial order reduction* and *on-the-fly model-checking*. The resulting LTS can be used within CADP [10], a verification toolset developed by the VASY team of INRIA Rhône-Alpes and VERIMAG. In particular we intensively used in this experiment two components of this toolset, ALDEBARAN, a bisimulation based minimisation/comparison tool, and EVALUATOR, an alternating-free μ-calculus model-checker. Each of these tools are able to compute diagnostic sequences at the LTS level that can be translated back into MSC to be observed at the specification level. Other components like KRONOS [22] (a Timed-CTL model-checker) and TGV [11] are also applicable at this level.

3 Ariane 5 Flight Program

In order to understand the functionalities of the flight program, we begin the presentation with a short informal overview of the whole Ariane-5 flight[1]. Then, we present how the flight program was formalised using SDL. We detail the main design choices and the abstraction of the environment. Meantime, we try to illustrate both the benefits and the limitations of using SDL as a description language for this kind of systems. We end the section by presenting the set of safety requirements needed in order to ensure the well-functioning of the flight program.

[1] the description was taken from the ESA - European Space Agency - web page: `http://www.esa.int/`

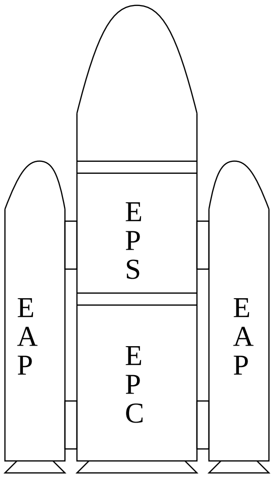

Figure 2: Ariane-5 launcher.

3.1 Overview

An Ariane-5 launch begins with ignition of the main stage engine (EPC - *Etage Principal Cryotechnique*). Upon confirmation that it is operating properly, the two solid booster stages (EAP - *Etage Accélérateur à Poudre*) are ignited to achieve lift-off.

After burn-out, the two solid boosters (EAP) are jettisoned and Ariane-5 continues its flight through the upper atmosphere propelled only by the cryogenic main stage (EPC). The fairing is jettisoned too, as soon as the atmosphere is thin enough for the satellites not to need protection. The main stage is rendered inert immediately upon shut-down. The launch trajectory is designed to ensure that the stages fall back safely into the ocean.

The storable propellant stage (EPS - *Etage à Propergol Stockable*) takes over to place the geostationary satellites in orbit. Payload separation and attitudinal positioning begin as soon as the launcher's upper section reaches the corresponding orbit. Ariane-5's missions ends 40 minutes after the first ignition command.

A final task remains to be performed - that of passivation. This essentially involves emptying the tanks completely to prevent an explosion that would break the propellant stage into pieces.

The flight program entirely controls the launcher, without any human interaction, beginning 6 minutes 30 seconds before lift-off, and ending 40 minutes later, when the launcher terminates its mission.

3.2 Formal specification

In order to build a formal specification, the SDL [17] language was preferred for this case study among other formalisms for several reasons. First of all, it is based on asynchronous communicating finite-state machines. Thus, it is particularly adequate to describe, at some level of abstraction, the whole flight program as a collection of processes asynchronously interacting each other. In addition, SDL provides an explicit notion of time and some corresponding real-time primitives. Since there exists an automatic translation from SDL to IF, it becomes possible to enforce the SDL time semantics using urgencies in order to express all the required timing constraints associated with the flight program components. Finally, SDL is currently supported by several integrated environments such as OBJECTGEODE [20] and TAU [1], which makes it very attractive from a development point of view.

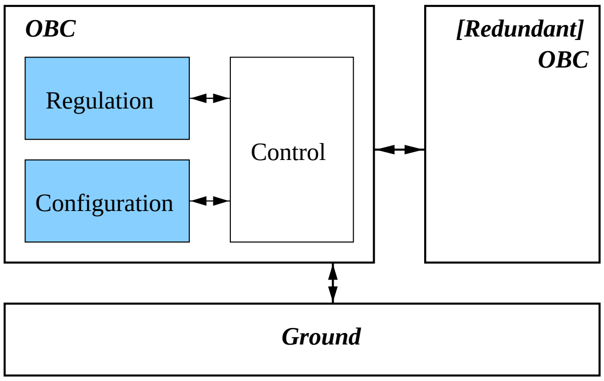

Figure 3: Flight program architecture.

As explained in the informal description, the main functionalities of the flight program are the following:

- *flight control*, which consists in navigation, guidance and control algorithms,
- *flight regulation*, which consists in observation and control of various components of the propulsion stages (engines ignition and extinction, boosters ignition, etc),
- *flight configuration*, which consists in managing changes of launcher components (stage separation, payload separation, etc).

The specification we treated in this case study focuses on regulation and configuration parts.

The flight regulation part is modeled by six SDL processes. With few exceptions, we have two loosely coupled SDL processes for each stage: one describing the *firing* and the other the *extinction* of the stage. In general, they work as follows. Both the firing and the extinction process receive as input the firing date, provided by the flight control part. Then, the firing process executes the firing sequence i.e, the set of actions to be done, with the right deadlines, in order to properly fire the stage at the given date. If some malfunctioning is detected during this sequence, the extinction process takes over and attempts to stop the firing, using an adequate stop sequence. Otherwise, the extinction will occur later, eventually prior to the moment when the stage is dropped out.

Example 3.1 *An over-simplified firing process is illustrated in figure 4. The informal actions* action_1*, …* action_n *must be executed precisely at time* T0-d1*, …* T0-dn *respectively, where* T0 *is a parameter received by the process and* d1*, …* dn *are constant values. The informal actions abstract external commands which have to be initiated by the process (external sensors reading, opening or closing engine valves, etc.) at the right moments in time.*

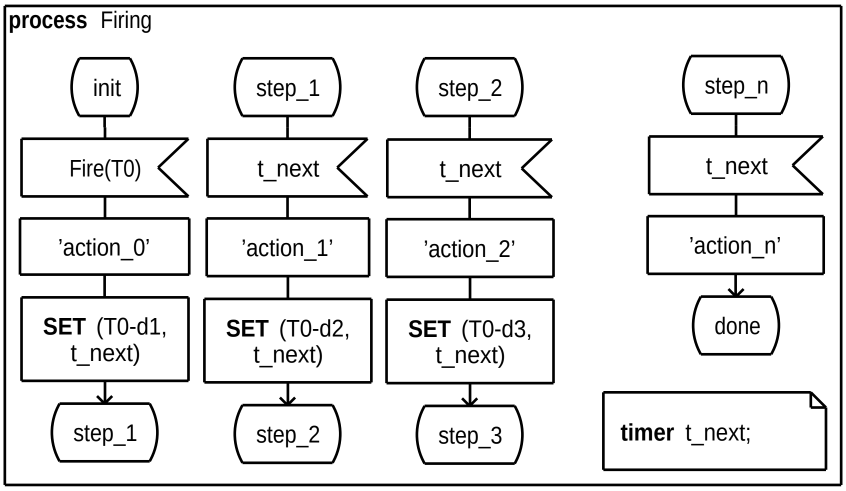

Figure 4: A firing process.

The flight configuration part contains seven SDL processes. Each process implements some particular configuration task: EAP separation, EPC separation, payload separation, etc. In their case too, the separation dates are provided by the control part, based on the current flight evolution.

Example 3.2 *A simplified configuration process is presented in figure 5. Here, the* opening *informal action must be executed on the reception of the* open *signal, eventually in the interval defined by timers* t_early *and* t_late. *Thus, if the* open *signal arrives too early, it must be saved or, if the signal does never arrive, the action has to be executed at the end of the interval.*

The translation of regulation and configuration processes into IF is straightforward. Nevertheless, an implicit assumption about the time progress was made in all of them: timeout-driven transitions are *urgent* and must be executed as soon as they are enabled i.e, exactly at the expiration time. This assumption is conflicting with the standard semantics of SDL, which consider that all the transitions are lazy. Fortunately, at IF level we could explicitly define timeout-driven transitions as *eager*, thus modeling exactly the intended behaviour.

3.3 Environment

In order to obtain a realistic functional model of the flight program, we have to take into account its surrounding environment. For example, precise human interactions are expected to initiate the launch procedure. Furthermore, both regulation and configuration parts were designed to closely interact with the control part of the program during the flight. Some minimal coordination must be ensured here e.g, that all the components received the same firing date otherwise no meaningful verification could be done.

To handle all the assumptions on the system environment, the solution we adopted is to "close" our SDL specification by adding external processes abstracting the actual behaviour of the control part, the redundant program and the ground:

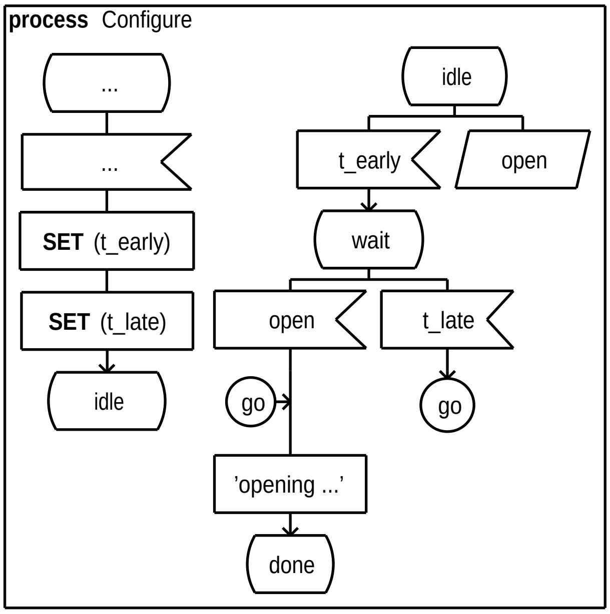

Figure 5: A configuration process.

- the *flight control* is over simplified. It consists on several processes which describe a nominal behavior: they are supposed to send, with some controlled degree of uncertainty, the right flight commands, with the right parameters at the right moments in time[2]. Nevertheless, here again the time semantics has to be considered with particular attention in order to obtain the intended behaviour (see example 3);

- the *redundant* program: at the initialisation time, the main program requests the status of the *redundant* one. Hence, the later was abstracted by a simple non-deterministic process, which could respond either positively or negatively to the main's request;

- the *ground* part implements the nominal behavior of the launch protocol on ground side. Progressively, it pass the control of the launcher to the on board flight program, by providing the launch date and all the other confirmations needed for launching. On the other hand, it remains ready to take back the control, completely, if some malfunctioning is detected during the launch procedure.

Example 3.3 *In the nominal case, the control part eventually sends the* extinction *command of the vulcain engine within some given time interval* $[L, U]$*. Such behavior could be sketched by the* SDL *process from figure 6. Nevertheless, its meaning is quite unclear: following the standard* SDL *semantics it is possible that the process stays forever at the* init *state. Following the* OBJECTGEODE *semantics, the transition is executed as soon as it becomes enabled, so when* now *equals* min*. Unfortunately, none of these interpretations is the intended one. The right solution is achieved only at* IF *level by defining the transition as* delayable*: it will be eventually executed at some time within the interval.*

[2]The data used here correspond to the flight no. 503 of Ariane-5 launcher

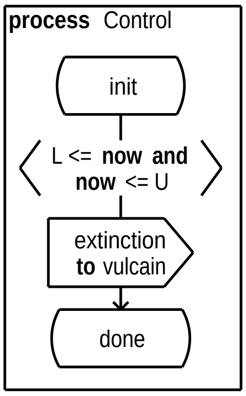

Figure 6: Control process.

3.4 Requirements

With the help of EADS experts, we identified a set of about twenty safety functional requirements ensuring the right service of the flight program. The requirements were classified into three classes, as follows:

- *general requirements*, which are not necessarily specific to the flight program but in general, to all critical real-time systems. They include basic untimed properties such as the absence of deadlocks, livelocks or signal looses, and basic timed properties such as the absence of timelocks, Zeno behaviors or deadlines missed;
- *overall system requirements*, which are specific to the flight program and concern its whole behavior. For example, we mention here the global order for the flight phases (e.g, ground, vulcain ignition, booster ignition, etc...), or the vulcain engine extinction in the presence of anomalies;
- *local component requirements*, are also specific to the flight program and concern the functionality of some of its parts. In this category, we consider for example checking the occurrence of some actions in some component (e.g, payload separation occurs eventually during an attitudinal positioning phase, or the stop sequence no. 3 could happens only after lift-off, or the state of engine valves conforms to the flight phase, etc.)

Initially, all these requirements were described using GOAL observers [2]. Then, in order to be handled with IF tools, they were translated manually into temporal logic formulae or finite-state automata.

4 Verification

Formal verification is certainly the most challenging phase in a formal development process. It is the only way to provide an earlier and effective feedback about the behavior of the system, regarding its environment and its requirements. In order to master the complexity of the verification we propose to split it into five independent steps, ranging from the simplest static analysis to the most powerful model-checking techniques (see figure 7). This section presents the concrete verification results obtained on the Ariane-5 specification, following this methodological guidelines.

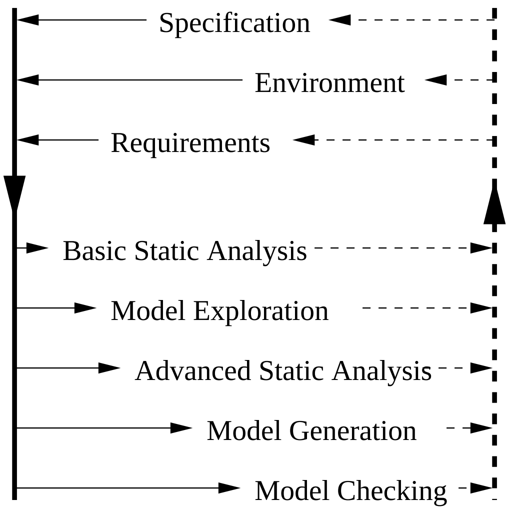

Figure 7: Verification methodology.

4.1 Basic static analysis

During this first step, simple analysis are applied on the specification. They include both sanity tests and some simple static analysis.

In the first category, we mention basic tests on variables and signals. For instance, the user detects variables or timers never assigned nor used. Furthermore, variables which might be used without being initialised are computed. Moreover, signals which are never sent nor received are also indicated to the user. In the second category, we mention techniques such as live variables analysis or constant propagation. They give us much more accurate information about the use of the variables in the program while detecting various kinds of redundancy, such as unused variable definitions or any other form of dead-code.

4.2 Model exploration

The validation process continues with a debugging stage. With no sake for exhaustivity, the user begins to explore the model of the specification, in a guided or random manner. Simulation states do not need to be stored as the complete model might not be explicitly constructed at this moment.

The aims at this stage are multiples. Firstly, the user could inspect and validate known scenarios about the functioning of the specification. Secondly, the user can *test* simple safety properties, which might hold on all execution paths. Such properties might range from generic ones, such as deadlocks, signal loss or wrong timer setting detection, to more specific ones, application dependent. In general, they are tested either using specific code instrumentation, or using external observers. When an error is found, a diagnostic scenario can be produced at this step by the ObjectGeode simulator.

Example 4.1 *By inspecting a diagnostic scenario leading to a timed exception (e.g, unexpected timeout signal) we found an inconsistence between several constants used to control the firing of the* EPC*. A simplified* Msc *corresponding to this scenario is presented in figure 4.2. On one hand, the sending of the* desactivation *signal is conditioned by the reception of the* status *signal. On the other hand,* status *is sent at time* H0+t1 *while* desactivation *must be sent at time* H0+t2*. The error occurred here because* t1 *and* t2 *were defined such that* t1 *was greater than* t2*.*

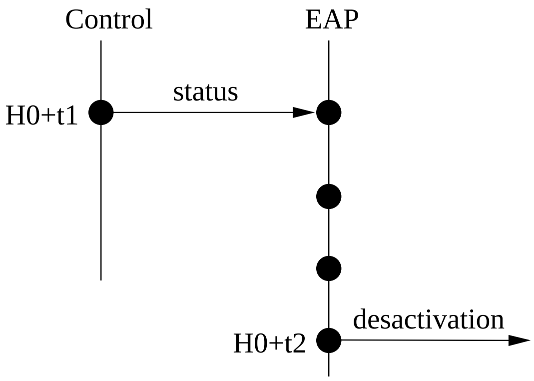

Figure 8: Diagnostic trace.

4.3 Advanced static analysis

The aim at this step is to prepare the specification to an exhaustive simulation. Optimisation based on static analysis results are intensively applied in order to reduce both the state vector and the state space, while completely preserving its behavior.

Different kinds of optimisations are currently available. The first one is variable and timers recovery, which consists in diminishing the number of variables and timers respectively used inside the specification. This optimisation exploits results obtained by *live* and *dependency* analysis:

- a variable (respectively clock) is *live* in a control state if it will be used before being assigned on some path starting on that state;
- two variables (respectively clocks) are *dependent* in a control state if their difference is constant and can be statically computed at that state;

The theoretical result used here is that any program can be rewritten using at most n variables, where n is the maximal number of live and functionally independent variables at some point in the program.

Example 4.2 *The initial* Sdl *version of the flight program used no less than 130 timers. Using our static analysis tool we were able to reduce them to only 55 timers, functionally* *independent ones. Afterward, the whole specification was rewritten taking into account the redundancies discovered by the analyzer.*

A second optimisation attempts to identify live equivalent states by introducing systematic resets for dead variables in the specification. In this way, it prevents to distinguish between simulation states which differ only by values of dead variables. This technique is very effective given that it can be applied locally at control-state level (contrarily to variable recovery which applies only if some condition holds on all control states of the program).

Example 4.3 *For this case study, the live reduction was not so impressive due to the reduced number of variables (others than clocks) used in the specification. Anyway, our initial attempts to generate the model without live reduction failed. Finally, using live reduction we were able to build the model but still, it was of unmanageable size, about* $2 \cdot 10^6$ *states and* $18 \cdot 10^6$ *transitions.*

Finally, the last optimisation we mention here is slicing [21]. This technique consists in statically extracting the part of the specification which is relevant to a slicing criterion i.e, here derived from a fixed property to be verified. The sliced part might be significantly smaller than the entire specification since, contrarily to previous optimisation techniques, slicing does not aim to preserve *all* the behaviors but only those which might influence the validity of the chosen property. In particular, we used this slicing technique to automatically eliminate some silent SDL processes, which do not perform any "relevant" action.

4.4 Model generation

The model generation step aims to explore completely the model of the specification by exhaustive simulation. By specification we mean here either the complete one, or a sliced version with respect to some fixed property.

This step might be extremely difficult given the apriori exponential size of the model. In order to deal with, the user controls both the representation scheme for states and sets of states and the exploration strategy. For example, the use in IF of a symbolic representation for timers i.e, using difference-bound matrixes to represent zones and regions [3], is particularly useful when dealing with a large time horizon and irregular timing constraints. Instead of representing each single particular point in time, this kind of representation allow us to handle set of equivalent points with respect to their future behavior. In particular, such symbolic representation is of real interest here because of the wide spectrum of timers values: for example, very short ones for regulation timers (measured in milliseconds, see figure 4) and longer ones for control timers (measured in minutes, see figure 6).

Concerning the exploration strategy, the use of partial order techniques [12] is clearly of value in the exploration of asynchronous communicating systems. Thus, spurious interleavings initiated either by internal actions or by the consumption of messages from communication buffers could be eliminated still preserving all the observable behavior of the specification. Nevertheless, special care must be taken with respect to time: since time is global and clocks are synchronised there exists implicit dependencies induced by time progress (e.g, time progress may disable some observable, relevant actions). In order to avoid this problem, we implemented in IF a restricted variant of partial order reduction in which outputs and time progress transitions are always considered as observable.

For example, let us consider a generic situation which occurs frequently in the flight program: a multicast communication which involves one sender and n receivers. As the communication in SDL is asynchronous buffered, even if all the buffers are empty we obtain 2^n intermediate states, due to all possible interleavings of receiver inputs. Fortunately, using partial order reduction, the combinatorial explosion disappears: inputs are executed in some order, and only n intermediate states are explored.

Example 4.4 *The use of partial order reduction was mandatory in order to construct models of reasonable size. Here, we reduce the size of the model with 3 orders of magnitude i.e, from* $2 \cdot 10^6$ *states and* $18 \cdot 10^6$ *transitions to* $1.6 \cdot 10^3$ *states and* $1.65 \cdot 10^3$ *transitions, which could be easily handled by* CADP *model-checkers.*

In practice, we consider two different situations regarding the environment. The first one is *time-deterministic*, which means that all environment actions (in particular the control part) take place at precise moments in time. The second one is *time-nondeterministic* which means that environment actions take place with some degree of time uncertainty (within a predefined time interval). From the environment point of view, the later situation corresponds to a whole set of scenarios, whereas the former situation focus only on a single one. Table 1 presents in each case the sizes of the models obtained depending on the generation strategy used.

		time deterministic	time non-deterministic
model generation	− live reduction − partial order	state explosion	state explosion
	+ live reduction − partial order	2201760 st. 18706871 tr.	state explosion
	+ live reduction + partial order	1604 st. 1642 tr.	195718 st. 278263 tr.
model verification	model minimisation	$\sim$ 1 sec.	$\sim$ 20 sec.
	model checking	$\sim$ 15 sec.	$\sim$ 120 sec.

Table 1: Verification Results.

4.5 Model-checking

Once the model being generated, several model-checking techniques can be applied to verify expected properties on the specification. Nevertheless, on-the-fly verification methods i.e, which combines the model-generation and model-checking steps, might be used.

Using the IF validation toolbox, two approaches can be followed to express these properties. First, temporal logic formula could be verified using EVALUATOR, the CADP μ-calculus evaluation tool.

Example 4.5 *The requirement expressing that* the stop sequence no. 3 occurs only during the flight phase, and never on the ground phase *can be expressed by the following temporal logic formula, verified with* EVALUATOR*:*

$$\neg\ \mu X. < EPC!Stop\_3 > tt\ \wedge\ < \overline{EAP!Fire} > X$$

Intuitively, it express that it is not possible to reach a state where is possible to perform the stop sequence no. 3 *without executing in the past the firing of the* EAP *(which denotes the beginning of the flight phase).*

A second approach, usually much more intuitive for a non expert end-user, consists in computing an abstract model (with respect to a given observation criteria) of the overall behavior of the specification. Such a model can be then visualised and possible incorrect behaviors can be detected. These abstract models are computed by ALDEBARAN and, depending on the (bi)-simulation relation used, they preserve different classes of properties.

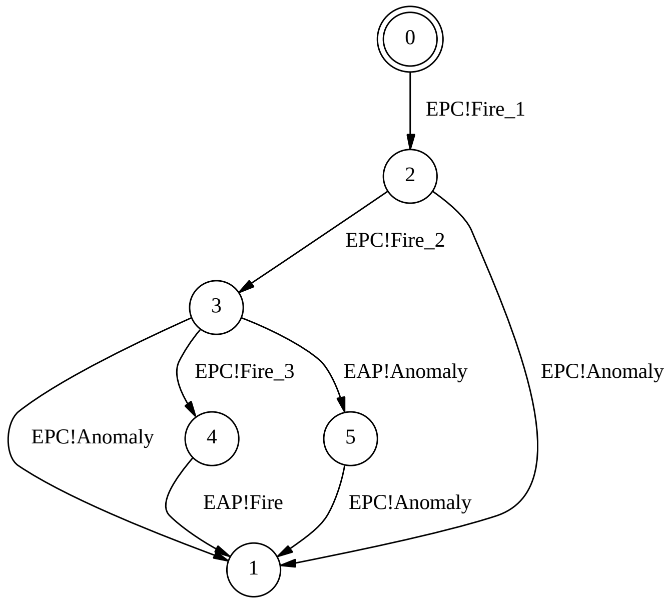

Figure 9: Minimal model.

Example 4.6 *All safety properties involving the firing actions of the two principal stages,* EAP *and* EPC*, and the detection of anomalies are preserved on the graph from figure 9 generated by* ALDEBARAN*. It is the quotient model with respect to safety equivalence [5] while keeping observable only the actions above. For instance it is easy to check on this abstract model that, whenever an anomaly occurs* before *action* `EPC!Fire_3` *(ignition of the Vulcain engine), then nor this action nor* `EAP!Fire` *action are executed and therefore the entire launch procedure is aborted.*

Table 1 gives the average time required for verifying each kind of property (by temporal logic model checking and model minimisation respectively).

5 Conclusions

In this paper we described a practical experiment on the validation of a real-time embedded software specification, the configuration and regulation part of the Ariane-5 Flight Program.

First, this software has been formally specified in SDL by reverse engineering. Then, following a set of general methodological guidelines, the specification has been continuously improved and the twenty expected requirements were all verified on the final version. In particular, the combination of different optimisation techniques, operating either at the *source level* (like static analysis or slicing) or at the *semantic level* (like partial-order reductions) happened to be particularly useful in order to deal with large size state spaces. Nevertheless, this work covers only a limited part of the development process of real-time embedded systems: the specification that has been developed and validated is abstract and rather "far" from the existing executable code. This approach is therefore well-adapted in the earlier phases of development but applying it to more concrete designs could become problematic in practice.

The main difficulty of this case-study comes from the combination of various kind of time constraints. On one hand, the functionality of the flight program strongly depends on an absolute time: coordination dates are frequently exchanged between components in order to synchronise their behaviour during the whole flight. On the other hand, this system has to be verified within a partially constrained environment, reacting with some degree of temporal uncertainty. In this experiment, this expressivity problem was solved at the IF level thanks to explicit urgency attributes. Clearly, such features should be made available at specification level. In particular, ongoing work address the introduction of high-level time and performance annotations in SDL [9].

Another future direction of investigation is the *synchronous/asynchronous* interaction. Currently, with SDL we were able to build an abstraction of the flight program, as an *asynchronous* interaction of several processes, which express the overall sequential behavior and the most important timing constraints on it. However, this specification is not complete, at least because very important program parts, such as navigation and control, must be almost completely abstracted away, because they cannot be described in SDL. Such parts describing intensive data-flow transformations have to be executed in a *synchronous* manner, and are usually described using synchronous languages such as LUSTRE [13]. Unfortunately, from the synchronous side, inherently sequential parts with asynchronous interaction also could not be properly expressed. Nevertheless, this kind of dual design where coexist both asynchronous components and synchronous ones is not an exception and occurs very often in real-time applications design practice. We plan for the future to investigate how to combine, in a sound manner, synchronous and asynchronous descriptions, and the possible tool support for doing it, in order to exploit at best the advantages conferred by both programming paradigms.

References

[1] Telelogic AB. *SDT Reference Manual*. `http://www.telelogic.se`.

[2] B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *Proceedings of SDL FORUM'95*. Elsevier, 1995.

[3] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference (Holstein, Germany)*, volume 1536 of *LNCS*. Springer, September 1997.

[5] A. Bouajjani, J.Cl. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for Branching Time Semantics. In *Proceedings of ICALP'91*, volume 510 of *LNCS*. Springer, July 1991.

[6] M. Bozga, J.Cl. Fernandez, and L. Ghirvu. State Space Reduction based on Live Variables Analysis. In A. Cortesi and G. Filé, editors, *Proceedings of SAS'99 (Venice, Italy)*, volume 1694 of *LNCS*, pages 164–178. Springer, September 1999.

[7] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: A Validation Environment for Timed Asynchronous Systems. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of CAV'00 (Chicago, USA)*, volume 1855 of *LNCS*. Springer, July 2000.

[8] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF: An Intermediate Representation for SDL and its Applications. In R. Dssouli, G. Bochmann, and Y. Lahav, editors, *Proceedings of SDL FORUM'99 (Montreal, Canada)*, pages 423–440. Elsevier, June 1999.

[9] M. Bozga, S. Graf, L. Mounier, I. Ober, J.L. Roux, and D. Vincent. Timed Extensions for SDL. In *Proceedings of SDL FORUM'01*, LNCS, 2001. to appear.

[10] J.Cl. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings of CAV'96 (New Brunswick, USA)*, volume 1102 of *LNCS*, pages 437–440. Springer, August 1996.

[11] J.Cl. Fernandez, C. Jard, T. Jéron, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29, 1997.

[12] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State Explosion Problem*, volume 1032 of *LNCS*. Springer, January 1996. ISBN 3-540-60761-7.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *IEEE*, 79(9), September 1991.

[14] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.

[15] I-Logix. *StateMate*. `http://www.ilogix.com/`.

[16] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1988.

[17] ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, November 1999.

[18] K.L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem.* Kluwer Academic Publisher, 1993.

[19] OMG. Unified Modeling Language Specification. Technical Report OMG UML v1.3 – ad/99-06-09, Object Management Group, June 1999.

[20] Verilog. *ObjectGEODE Reference Manual.* `http://www.verilogusa.com/`.

[21] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984.

[22] S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.