ASTERIOS Checker: A Verification Tool for Certifying Airborne Software

Amira Methni[1], Emmanuel Ohayon[1], and François Thurieau[2]

[1] Krono-Safe – Massy, France
{amira.methni, emmanuel.ohayon}@krono-safe.com
[2] Safran Electronics & Defense – Massy, France
francois.thurieau@safrangroup.com

Abstract. As the number of embedded systems has grown regularly over the past decades, the development and *certification* costs of safety-critical software has increased accordingly. For the aeronautics industry, certification activities are covered by DO-178C, which provides guidance for developing airborne software; and its companion document DO-330 covers the qualification of tools used for the development of such software.
In this paper, we present *ASTERIOS*, a solution for the design, generation and execution of safety critical real-time applications; then we present the certification strategy we advocate for systems developed using our technology. This strategy relies on the use of an automated verification tool called *ASTERIOS Checker*, qualified in accordance with DO-330. This paper presents the technology behind the code generation engine of *ASTERIOS* and the verification activities automated by *ASTERIOS Checker*. It shows how the use of such an automated, qualified tool enables to benefit from design abstractions and relatively complex code generation engines while developing certified systems at the highest level of certification.

Keywords: DO-178C, DO-330, *ASTERIOS*, Psy model, real-time, certification

1 Introduction

Embedded systems have spread across all major industrial fields: automotive, industry, medical devices, and of course aerospace. As they become increasingly complex, their design and reliability become challenging. For safety-critical software, the stakes can be huge as their failure may lead to the loss of entire, potentially costly systems, or even put human lives at risk. Obviously, such systems require a high degree of quality in order to ensure an adequate level of confidence to them. For airborne systems, DO-178C [14], *Software Considerations in Airborne Systems and Equipment Certification*, provides guidance for developing software. DO-178C aims to apply a rigorous development process in order to prevent the occurrence of errors and guarantee the safety and reliability of airborne software. It defines five assurance levels, depending on the risk and the effects of a failure of the system, commonly referred to as DAL (Development Assurance level). Level A (DAL-A) is the highest one and is applied to software for which failure could cause a catastrophic event like death.

Designing safety-critical real-time applications meeting DO-178C is not an easy task. There are a few certified hard Real Time Operating Systems (RTOS) supporting such applications, such as VxWorks [4] or PikeOS [6]. For those solutions, it is still up to the user to certify the application to be run by the RTOS. To that end, ANSYS [7] provides the ANSYS SCADE solution, a model-based embedded software development environment with a code generator qualified at TQL-1, to design and generate software systems certified under DO-178C level A, among other norms. MathWorks [3] provides Code Inspector, a verification tool, qualified at TQL-4 and used to check the equivalence between the software model designed using Simulink and the C code generated by their tool RTW. However, the hard real-time constraints expressed locally or throughout the systems are out of the scope of these tools, and remain to be written as configuration tables for the underlying RTOS. This comes by with a hard and costly integration process.

Krono-Safe [1] proposes the *ASTERIOS* technology, which was originally developed with the French Alternative Energies and Atomic Energy Commission (CEA) [2]. It provides a set of tools to design safety-critical real-time applications, supporting single and multi-core architectures, along with a small foot-print real-time kernel (RTK) in charge of running the application on the embedded platform. *ASTERIOS* defines a real time programming model called *Psy* (for *Parallel SYnchronous*), based on the time-triggered approach [12,8] (more recently referred to as *Logical Execution Time* [10,11]) to express the real-time architecture with timing, communication and spatial partitioning constraints.

The *Psy* model was originally created for the OASIS technology [9], used in the field of nuclear energy. A set of compilation tools produces the executable object code of the application along with runtime tables that implement the temporal behavior, and configure the RTK according to the underlying hardware specificities. In order to certify applications written with *ASTERIOS*, Krono-Safe proposes *ASTERIOS Checker*, a qualified automated verification tool. It allows to claim certification credits for automatically checking the conformity of the real-time application configuration generated with *ASTERIOS*.

After a brief overview of the *ASTERIOS* technology in section 2, this paper sketches the certification strategy adopted to qualify the *ASTERIOS* tool suite for aerospace applications (section 3.1). We provide our experience report on developing and qualifying a verification tool in compliance with DO-330 [5], meant to be used in the process of certifying a system produced by a complex, non-qualified code generation toolchain. We also provide a description of one of the most interesting functions implemented by our tool: the verification of the static scheduling plan. At last, we report about the first-hand foreseen experiences of using *ASTERIOS* Checker for aerospace industrial application at Safran Electronics & Defense.

2 *ASTERIOS* Technology Overview

2.1 Psy Concepts

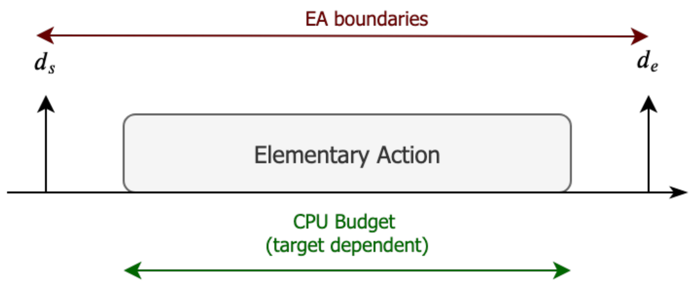

Fig. 1: An Elementary Action

The *Psy* model is a parallel and multi-tasking model where an application is composed of *Agents* (much like Unix processes). An agent is the execution unit in *Psy*: it is a sequence of execution windows, called *Elementary Actions* (EAs) that share the same execution context. As illustrated by Fig. 1, an EA is a sequence of instructions whose execution is constrained by an earliest start date d_s and a deadline d_e. For simplicity, the EA represented above is not preempted, but we note that depending on the scheduling policy, the EA can be executed in several timeslots, as long as its total execution span remains between the EA boundaries. An EA has a CPU budget b that defines the CPU time granted to complete its execution. This time is target-dependent and represents the processing of the task, including the CPU time required to run system services such as inter-task communication.

Since the *Psy* model is based on a Time Triggered approach [12,9], evolutions of the system are virtually made visible only at fixed instants in time, known at compile-time. The boundaries (d_s, d_e) of all EAs correspond to synchronization points of the real-time application, called *Temporal Synchronization Points* (TSP). TSPs define the cadence of tasks, and are also the formal instants in time where the global state of the application can change. To express the timing behavior on EAs, *Psy* introduces the concept of *sources.* A *source* is basically a periodic hardware interrupt (e.g. delivered by a hardware timer) also called a *tick*: the *source* sets the smallest time unit for defining EAs boundaries. Thus, a TSP in the system always corresponds to a source tick. All those concepts are implemented by the *PsyC* programming language.

2.2 The PsyC language

```
/* Declare a clock of period 2 (i.e. tick every 2 ms) and a clock of period 3
 * (i.e. tick every 3 ms).
 *
 *             0   1   2   3   4   5   6
 *  realtime  |---|---|---|---|---|---|->
 *
 *  c2        |-------|-------|-------|->
 *
 *  c3        |-----------|-----------|->
 */
source realtime;
clock c2 = 2 * realtime;
clock c3 = 3 * realtime;

/* Below, an agent declaration. ‘‘uses’’ defines the source on which the
 * agent is synchronized. ‘‘starttime’’ defines the earliest start date of
 * the first EA. The latter is considered as an "advance" from 0. The agent
 * starts at the third tick of c2, so at 6 ms. */
agent Ag0(uses realtime, starttime 3 with c2)/* 0:AG */
{
  /* "start" is the entry point of the agent. It is executed  infinitely
   * i.e. equivalent to a loop in C. The agent can define multiple bodies
   * but we do not introduce them here. */
  body start
  {
    /* ‘‘f1()’’ must be executed before the next tick of  clock c3 */
    f1();  /* User C code */
    timebudget B1, advance 1 with c3;  /* 1:ADV */

    /* The expression ‘‘something’’ cannot be evaluated  offline by the
     * compiler: it may be for instance the result of an I/O operation */
    if (something)
    {
      /* ‘‘f2()’’ must be executed before the next tick of c2 */
      f2();    /* User C code */
      timebudget B2, advance 1 with c2; /* 2:ADV */
    }
  }
}
```

Fig. 2: PsyC Application Example

The *PsyC* programming language is an extension of the C syntax: the additional keywords and grammar rules defined by *PsyC* implement the *Psy* concepts. Fig. 2 illustrates a simple example of an agent called `Ag0`. To define when an EA can start and stop, the *PsyC* introduces the keyword `advance`. The latter defines the deadline of the current EA and the start date of the next EA. Its arguments are a literal integer expression and an optional `with` qualifier to use a different clock than the base clock of the agent. The semantic of an `advance` n `with` c statement is as follows: from the current date, advance to the n^{th} tick of the clock c. For example, at date 5, `advance 3 with c2`, advances the time up to the date 10. The `timebudget` qualifier enables to define the CPU time allocated to the EA completing its execution on this `advance` statement, or a portion of it. For the example of Fig. 2, the EA covering `f1()` has a budget of `B1`. The values of those budgets are set in a separate file with a straightforward syntax (*.bgt* file). Note that the end-user is expected to provide the CPU budgets that is consistent with his estimations of the corresponding WCET (Worst Case Execution Times). *ASTERIOS* comes with a set of profiling features to help the end-user in this process.

The code of an agent is defined inside a `body`, which by default is an infinite loop. Several `body`s can be defined, and dynamic transitions between the bodies can be specified to easily implement a temporal finite-state machine.

Communications in *Psy* do not use shared variables or memory locks as it would be a source of non-determinism. Instead, communications are serviced through two deterministic channel types: sampled data flows, called *temporal variables*, and FIFO messages, called *streams*. For the sake of concision, we won't go into more details here; interested readers can refer to [13] for more information about the deterministic communication paradigm, and other advanced *Psy* concepts.

2.3 The *ASTERIOS* suite

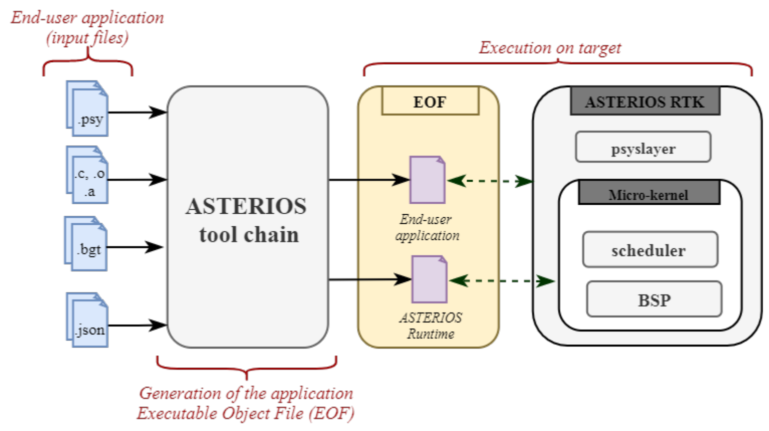

Fig. 3: *ASTERIOS* Suite Overview

The *ASTERIOS* suite consists in the following elements (from right to left on figure 3):

– The real-time kernel, called *ASTERIOS RTK* (or *RTK* for short). It is responsible for tasks scheduling, enforcing spatial partitioning between tasks, and managing errors. It is made of two software components: one generic, and one hardware specific, called the Board Support Package (BSP). It is written mainly in C, with some low-level functions written in assembly language in the BSP. The RTK also provides a micro-kernel service layer called the *psyslayer*, mostly in charge of implementing the communication services. It is designed to execute functions with different DALs by an end user using *ASTERIOS* solution.

- The end-user application, including the real time architecture designed in *Psy* and the functional code developed by the user. These elements are defined through:
  1. *.psy* file(s) describing tasks real-time behavior, timing and communication constraints in PsyC,
  2. *.c*, *.o* and *.a* files implementing the functional code of the tasks,
  3. *.bgt* file(s) defining CPU budget times for the user code,
  4. *.json* file(s) specifying the configuration of the target i.e., stack sizes, allocation of tasks to cores, etc.
- The configuration of the RTK for the end-user application, called the *ASTERIOS Runtime*. It contains data used by the RTK to manage the scheduling of the tasks, the memory sizing and partitioning, and core allocation. These data depend on the real time architecture *Psy* defined by the user, and are automatically generated by our compiler.

From the task cadence and the list of required CPU budget times defined by *.psy* and *.bgt* files, the *ASTERIOS* tool chain computes offline a preemptive static scheduling table, called *Repetitive Sequence of Frames* (RSF). It is composed of intervals of fixed duration, where each interval contains a sequence of frames. A frame is a fixed CPU time allocated to a single agent, or an *idle* frame.

ASTERIOS RTK's scheduler main functionality is to follow the RSF exactly as provided by the toolchain. Tasks notify the scheduler when an EA has completed, and the latter switches to the next frame. When an EA does not complete within the requested budget time, the scheduler detects the error and applies a sanction configured by the user.

3 *ASTERIOS Checker*

3.1 Certification Strategy using Qualified Automated Tools

Since embedded end-user applications may need to be certified up to DAL-A level, both the generated Runtime and the RTK need to be certified at DAL-A as well. The RTK is bundled as a standalone binary file with a complete DAL-A certification kit. This approach cannot be applied to the Runtime though, as it is re-generated for each application. Besides, some additional executable code is also generated by the *ASTERIOS* toolchain into the application binary file (related to task definition, communication channels instantiation, real-time constraints implementation, ...). Obviously, this generated code needs to be certified at DAL-A as well.

For that purpose, *ASTERIOS* includes qualified tools as defined by DO-330 [5] to automate some certification activities, and thus gain certification credits. According to the DO-178C (§12.2.1), a tool needs to be qualified when it is used to "*eliminate, reduce or automate software life cycle processes*", without its output being itself verified. Thus, the purpose of Tool Qualification is to get confidence on the tool's functionalities and on its output. Section 12.2 of DO-178C defines three criteria to classify tools used in a certification process, and subsequently determines the required *Tool Qualification Level* (TQL).

Criteria 1 are basically development tools, whose "*output is part of the airborne software and thus could insert an error*". In other words, an error in such a tool may impact the safety of airborne system by inserting erroneous code or data. Examples are code generators, compilers, linkers, etc.

Criteria 2 are tools "*that automate verification process(es) and thus could fail to detect an error*", and whose output is used to reduce or eliminate both development *and* verification activities other than those automated by that tool. Examples include formal method tools like static code analyzers.

Criteria 3 are verification tools that, "*within the scope of [their] intended use, could fail to detect an error*". Examples are test case generators, coverage tools, etc.

Based on the tool criteria and the Design Assurance Level of the software for which the tool is used, a TQL is assigned. DO-330 [5] provides guidance and objectives for qualifying the tool depending on its TQL, where TQL-1 requires the most rigorous qualification process, and TQL-5 the least. The TQL draws a set of qualification objectives, activities, guidance and life cycle data to be produced. DO-330 defines 76 objectives summarized in 11 tables (Annex A). The process of qualification is then to demonstrate that each objective is satisfied by the tool.

For DAL-A software, when tools eliminate, automate or reduce processes required by the DO-178C they should be TQL-1. Thus, if the *ASTERIOS* suite were to be certified, the code generation tool chain qualification level would be TQL-1, for which objectives are roughly equivalent to DAL-A certification for airborne software. But, qualifying the whole tool chain, along with all its third-party libraries and runtime environment at this level would be tedious, time-consuming and costly: consider for instance that, the code generator being written in C++, the whole standard C++ library needs to be qualified as well.

ASTERIOS Checker is a verification tool whose purpose is to automatically verify that the outputs of the non-qualified code generator — namely the Runtime and the user application, are compliant with its inputs — namely the source files *.psy*, *.c/.o/.a*, *.bgt* and *.json* files. The scope of these verifications is both on the generated C source files, the object files, and the final binary file meant to be loaded on the embedded hardware. These verifications partially cover the objectives listed in tables A-4 and A-5 of DO-178C [14]. As such, *ASTERIOS Checker* is a Criteria 3 tool, meant to automate some verification activities for a DAL-A system, and thus needs to be qualified at TQL-5.

The root document of a certification process is the *Plan for Software Aspects of Certification* (PSAC): it is the "entry point" for a certification authority, providing both a complete system overview and a description of the certification plan. This document is used to justify that the software life cycle complies with the objectives of DO-178C. When qualified tools are used for some of the certification processes, their intended use and qualification level justification are provided in the PSAC.

As a third-party technology provider, Krono-Safe can only contribute to the certification data through a "certification kit". This kit includes for instance a user manual for the code generator that defines a usage domain for certification purposes, PSAC elements for the *ASTERIOS* RTK, and certification considerations for using *ASTERIOS Checker* which should also be included in the final PSAC of the end-user.

DO-178C introduces in section 2.5.1 the concept of *Parameter Data Item* and defines it as: "*a data set that influences the behavior of the software without modifying the Executable Object Code and is managed as a separate configuration item is called a parameter data item*". The PDI should be assigned the same software level as the software component using it. The *ASTERIOS* Runtime is considered as a PDI: it is a set of binary data generated by the code generator, whose purpose is to configure the behavior of the RTK e.g. by defining scheduling tables, configuring communication buffers, or declaring task descriptors. This approach facilitates the automated verification activities, as the C data structures generated for the Runtime are no longer considered as generated source *code* — which they are not in practice: they are only configuration tables, not executable instructions.

The list of certification credits claimed by the use of *ASTERIOS Checker* are not listed here for the sake of brevity. We'll just state that most of the objectives of Table A-4 (*Verification of Outputs of Software Design Process*) and A-5 (*Verification of Outputs of Software Coding and Integration Processes*) of DO-178C, Annex A are covered by our verification tool for the generated code and data.

3.2 ***ASTERIOS Checker* Overview**

Activities defined in the previous section are covered by a set a verifications conducted by *ASTERIOS Checker*, which ensures that the code generation toolchain has produced the expected output. *ASTERIOS Checker* is currently under development, by a team independent from the one developing the code generation toolchain. In order to avoid common mode failures, the checker tool is developed with a different programming language (Python), and uses different algorithms than those implemented by the PsyC compiler. The typical approach for most of the verification functions is to avoid comparing the outputs of the PsyC compiler against an expected output that would be similarly produced by the checker tool. Instead, we always try to validate that the code or data generated by the PsyC compiler remain within an acceptable range of possible outputs. This approach is illustrated in the next subsections with the example of the verification of the scheduling plan.

The verification functions of *ASTERIOS Checker* are covered by several verification components. Each of them is dedicated to detect a specific error that could be introduced by the code generation toolchain. Three main components of the checker tools are:

– *RSF Checker*: verifies the scheduling tables, CPU budgets, TSPs projection, execution order constraints.
– *BIN Checker*: verifies that the final *Executable Object Code* (EOC) is consistent with the memory layout defined by the input configuration files.
– *Sizer Checker*: verifies the size of the generated communication buffers.

Each component performs a verification function to achieve and to automate some of the verification activities required to reach DO-178C objectives. For instance, *BIN checker* claims certification credits for the objective of Table A-5.7 (*Output of software integration process is complete and correct.*). Besides, as the RSF is a Parameter Data Item, *RSF Checker* satisfies a part of the objectives of Tables A-4.9 and A-5.8 on software architecture consistency, and PDI files verification and correctness.

As stated in DO-330, the development of *ASTERIOS Checker* goes through a complete software life cycle including planning, requirement, design, verification, validation, quality assurance, etc. DO-330 states that for TQL-5 qualified tools, 15 objectives need to be satisfied, two of which require independence. Compliance to those objectives is proved by providing reports and qualification artifacts to be evaluated by certification authority.

As stated before, the novelty brought by *ASTERIOS Checker* as a qualified verification tool comes from the variety of verification operations that are automated, covering a complete code generation toolchain that implements an entire programming abstraction. In the next subsection we focus on the *RSF Checker* component to illustrate one of the most advanced verification functions implemented by our tool.

3.3 ***RSF Checker***

RSF Checker ensures that the scheduling plan produced by *ASTERIOS* is correct with regard to the *Psy* architecture and the CPU budgets defined by the user. To do that, it verifies that for each agent, the RSF provides in order the CPU time for all EAs declared in the *.psy* file and that the total CPU times of the frames composing each EA is greater than or equal to the CPU time required by the user (defined into *.bgt* files).

To that end, for each agent ag in the application, we build a directed cyclic graph called *Temporal Control Flow Graph* ($TCFG$). The latter captures the temporal behaviors and the control flow of ag, and is defined by $TCFG = (S_{ag}, T_{ag})$ where:

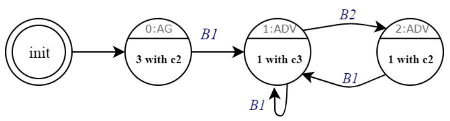

Fig. 4: $TCFG$ of $Ag0$. The *starttime* is viewed as an advance statement. Transitions are labeled with symbols where $B1 = 100\mu s$ and $B2 = 150\mu s$

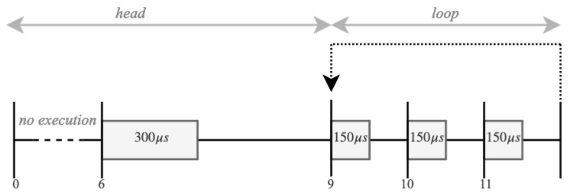

Fig. 5: RSF. The first interval has a length of 6 source ticks which is the start time of the application. The loop length is 3 source ticks

– S_{ag} is the set of states. Each state $s \in S_{ag}$ corresponds to a temporal constraint, i.e., an `advance` statement in the Psy agent. The set S_{ag} includes one initial state of the graph called *init*,
– $T_{ag} \subseteq S_{ag} \times B \times S_{ag}$ is the set of transitions, where $B \in \mathbb{N}^*$ is the set of budgets declared in *.bgt* file. Each $tr \in T_{ag}$ corresponds to an EA identified by (s_i, b, s_j), which contains the functional C code between two advance statements s_i and s_j and holds the maximum CPU time b required for that EA.

Fig. 4 shows the $TCFG$ of $Ag0$. Each state holds the value and the clock of its corresponding advance statement. Exploring the $TCFG$ consists in computing all execution paths of an agent, starting from *init* state at date 0. An execution path is a sequence of (s_i, d_i) where $s_i \in S_{ag}$ and d_i is its exploration date (in source ticks). For $Ag0$, the execution path starting from *init* to $0 : AG$ is: $(\text{``}init'', 0), (\text{``}0 : AG'', 6), (\text{``}1 : ADV'', 9), \ldots$.

The RSF given by Fig. 5 can be viewed as a finite sequence of intervals $RSF = \{l_1, \ldots l_n\}$. Each interval l_i is a list of agent's frames $l_i = \{f_1, f_2, \ldots, f_m\}$, where a frame f_i is identified by its given CPU time. Each interval start corresponds to a TSP identified by an absolute date. The RSF holds a special interval that marks the looping interval (e.g. for $Ag0$, interval start of the RSF loop is 9).

The verification of CPU budgets consists in exploring simultaneously the $TCFG$ and the RSF. More formally, for each transition (s_i, b, s_j) of the $TCFG$, namely a path $(s_i, d_i), (s_j, d_j)$, check that:

– d_i and d_j are TSPs in the RSF, and
– $b \leq \sum f_1, \ldots, f_n$ where $f_1, \ldots, f_n$ are CPU of frames of the interval delimited by $[d_i, d_j]$.

The exploration stops if one of the followings conditions is true:

– a state of $TCFG$ has no corresponding tick in the RSF ;
– the transition (s_1, b, s_2) of the $TCFG$ has not the sufficient budget in the RSF ;
– the execution path has already been explored.

The verification shall explore all execution paths of each agent in the application. The execution paths of one agent can be viewed as a tree as illustrated by Fig. 6. The exploration may stop every time it reaches an "equivalent temporal state". Two exploration states are said to be equivalent when they have the same possible futures, thus ensuring that the execution paths beyond have already been explored.

We notice that the verification scalability is not yet an issue. The tool can explore roughly 250 states per second, when executed on an average workstation from 2016 with the Python 3.6 (CPython) interpreter, enabling to verify the most complex applications of the tool test suite in a matter of seconds.

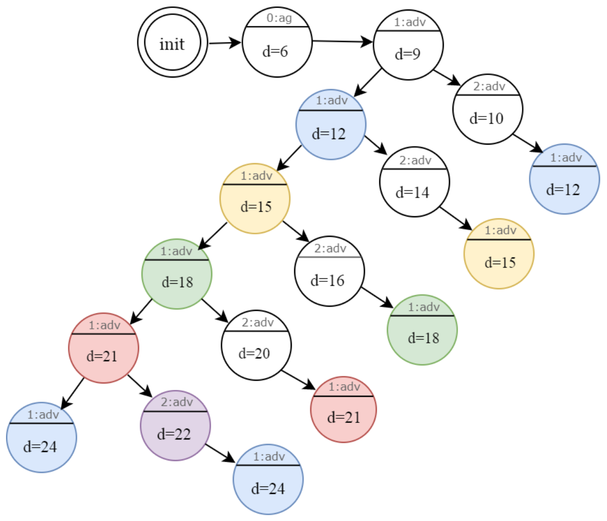

Fig. 6: Tree of all execution paths of $Ag0$. Equivalent temporal states have the same color.

4 Industrial Application at Safran Electronics & Defense

4.1 Use of Qualified Verification Tools

A major contribution of DO-178C over DO-178B was the refinement of guidelines regarding the use of tools to automate or avoid altogether some of the certification activities. It has reinforced the use of Criteria 3 tools in particular, as they “only” require TQL-5 qualification, while increasing safety by automating otherwise tedious and error-prone verification tasks.

Safran Electronics & Defense uses several such verification tools, including for automating verification activities on source code produced by non-qualified generators. Experience shows however that in-house development of verification tools for third-party non-qualified code generators is difficult and costly, by lack of specification data. Therefore, it makes sense for third-party technology providers to also offer such qualified verification tools, provided that they can show evidence of protection against common mode failures; i.e. an error that would cause a failure of both the generation tool, and of the verification tool supposed to detect that error.

4.2 Supporting a Paradigm Shift with a Verification Tool

The use of the *ASTERIOS* toolsuite brings novelty (and thus inevitably some part of challenge) to the certification process of airborne systems. The *Psy* programming model defines a design abstraction, which offers by construction key safety properties such as communication determinism and reproducibility. A byproduct of this abstraction is that the toolsuite can automate the generation of configuration elements that once were the task of system integrators, such as the scheduling table. Thus, the use of this technology in a certified airborne system depends on qualified verification tools to make sure that these safety properties are enforced, and that the configuration elements were generated accordingly. As such, the transformation rules verified by *ASTERIOS Checker* are more complex for instance than those implemented by a verification tool that enforces a set of coding rules, or one that verifies straightforward syntactic transformations.

The foreseen applications of the *ASTERIOS* toolchain and its verification tool by Safran Electronics & Defense are multiple and include landing systems, engine regulation or inertial navigation. The complexity of these systems varies from 100,000 to 700,000 source lines of codes: they are made of typically a dozen of periodic tasks of different frequencies, with ratios going up to 400. Current benchmarks show that this complexity can be easily handled by the checker tool, in a matter of minutes in worst case scenarios.

The final certification of these systems is planed in three to four years from now.

Looking even further, evolutions of *ASTERIOS Checker* should support incremental certification of systems built with *ASTERIOS*. The *Psy* programming model encourages a modular design, by de-coupling the expression of hard real-time constraints from integration considerations such as scheduling tables, or communication buffer sizing (as these elements are generated). *ASTERIOS Checker* should therefore facilitate the addition of new features to an already certified system, by enabling automatic verification of the newly added components only – while relying on previous verifications performed on the initial version of the application.

5 Conclusion

In this paper, we have presented the *ASTERIOS* tool-suite with a strong focus on *ASTERIOS Checker*: a TQL-5 qualified tool that automates verification activities for certified airborne applications developed with *ASTERIOS*. The core of the tool-suite being a non-qualified source-to-source compiler and configuration generator, *ASTERIOS Checker* is a key element in the certification strategy of such airborne applications.

Past and current experiences both at Krono-Safe and at Safran Electronics & Defense show that, when possible, qualifying an automated verification tool for generated applications is a more tractable and less costly approach than qualifying code generators themselves. Besides, in the present case, the verification functions can show a relative algorithmic complexity for a qualified tool, enabled by the fact that "only" TQL-5 is required.

References

1. `http://www.krono-safe.com/`
2. `http://www.cea.fr/`
3. Simulink Code Inspector. `www.mathworks.com`
4. VxWorks. `https://www.windriver.com/products/vxworks/`
5. DO-330: Software Tool Qualification Considerations (Dec 2011)
6. PikeOS Safe Real-Time Scheduling. Tech. rep., SYSGO (2016)
7. ANSYS: ANSYS SCADE Suite. `https://www.ansys.com` (2019)
8. Chabrol, D., Roux, D., David, V., Jan, M., Ait Hmid, M., Oudin, P., Zeppa, G.: Time- and Angle-Triggered Real-Time Kernel. In: 2013 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1060–1062 (2013)
9. Chabrol, D., Vidal-Naquet, G., David, V., Aussagues, C., Louise, S.: OASIS: A chain of development for safety-critical embedded real-time systems. In: 2nd European Congress Embedded Real Time Software. (ERTS 2004). pp. CD–ROM Proceedings – 10 pages. Toulouse, France (Jan 2004)
10. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: International Workshop on Embedded Software. pp. 166–184. Springer (2001)
11. Kirsch, C.M., Sokolova, A.: The logical execution time paradigm. In: Advances in Real-Time Systems, pp. 103–120. Springer (2012)
12. Lemerre, M., Ohayon, E., Chabrol, D., Jan, M., Jacques, M.: Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops. pp. 41–48 (March 2011)
13. Ohayon, E.: Deterministic Real-Time Communication Paradigm. `http://www.krono-safe.com/deterministic-real-time-communication-paradigm`
14. RTCA DO-178, R., EUROCAE: Software Considerations in Airborne Systems and Equipment Certification (2011)