Electronic Communications of the EASST
Volume 46 (2011)

Proceedings of the
11th International Workshop on
Automated Verification of Critical Systems
(AVoCS 2011)

Symbolic Model Checking and Safety Assessment of Altarica models

Marco Bozzano [1], Alessandro Cimatti [1], Oleg Lisagor [2],
Cristian Mattarei [1], Sergio Mover [1], Marco Roveri [1] and Stefano Tonetta [1]

15 pages

Guest Editors: Jens Bendisposto, Cliff Jones, Michael Leuschel, Alexander Romanovsky
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Symbolic Model Checking and Safety Assessment of Altarica models

Marco Bozzano [1], **Alessandro Cimatti** [1], **Oleg Lisagor** [2], **Cristian Mattarei** [1], **Sergio Mover** [1], **Marco Roveri** [1] **and Stefano Tonetta** [1]

[1]Fondazione Bruno Kessler, Trento, Italy
[2]The University of York, York, United Kingdom
{bozzano,cimatti,mattarei,mover,roveri,tonettas}@fbk.eu
oleg.lisagor@cs.york.ac.uk

Abstract: Altarica is a language used to describe critical systems. In this paper we present a novel approach to the analysis of Altarica models, based on a translation into an extended version of NuSMV. This approach opens up the possibility to carry out functional verification and safety assessment with symbolic techniques. An experimental evaluation on a set of industrial case studies demonstrates the advantages of the approach over currently available tools.

Keywords: Model Checking, Safety Assessment, Fault Tree Analysis, Altarica

1 Introduction

The dramatic increase in complexity of safety-critical systems in recent years has motivated a growing interest in model-based techniques for system verification. Such techniques must be able to verify functional correctness, but also to carry out safety assessment, that is, assess system behavior in the presence of faults [Ba03, ÅBB06, BV10]. In particular, there has been a growing interest in formal verification tools that can automate the generation of artefacts such as Fault Trees and Failure Mode and Effects Analysis (FMEA) tables [FSA, BV07, BCK$^+$10].

One of such tools is Cecilia OCAS [BBC$^+$04] – a model-based safety assessment platform developed by Dassault Aviation, based on the Altarica [Alt, AGPR00] language. Altarica has been used in the past for safety assessment of industrial systems, see, e.g., [BCS02, BBC$^+$04]. Moreover, OCAS is being used at an industrial level for architectural safety assessment of avionics systems. For example, the Flight Control System of Falcon 7x aircraft has been certified on the basis of the OCAS analysis. OCAS is equipped with different model analysis tools, the main ones are a trace simulator, and a sequence generator to generate minimal cut sets. However, these tools are neither able to perform an *exhaustive* space examination, nor they are able to model check *temporal* properties; even reachability analysis is bounded in depth. Furthermore, developed as an in-house tool, the OCAS sequence generator does not correctly implement language features that are not used within Dassault Aviation. In particular it is unable to adequately explore non-deterministic instantaneous transitions, potentially leading to incomplete analysis results (although the tool can be configured to provide a warning). Finally, the OCAS sequence generator is based on explicit state techniques, hence it suffers from the state-explosion problem.

In this paper we propose a fully symbolic approach that overcomes these limitations, and allows for the industrial usage of advanced symbolic verification and safety assessment techniques. Our approach is based on the translation to an extended version of NuSMV [NuS], and is tightly integrated with the OCAS environment. NuSMV is a state-of-the art symbolic model checker providing cutting-edge model checking technologies such as BDD-based [Bry92] and SAT-based Bounded Model Checking (BMC) [BCCZ99] techniques. It supports both temporal model checking (CTL and LTL temporal logics) and safety assessment, e.g., Fault Tree Analysis (FTA) and FMEA, through its add-on NuSMV-SA. NuSMV has been used in several industrial contexts, for instance for verification and validation of aerospace systems [BCK+10].

More specifically, our contribution is as follows. First, we have isolated a fragment of Altarica in the Dataflow formulation, the Altarica dialect implemented in Cecilia OCAS. As the semantics for this fragment is not fully documented, an additional effort has been required to provide a formal definition for its semantics, by adaptation from the general definition of [AGPR00], and to validate its correctness with respect to the behavior shown by OCAS and user expectations. In the course of our work, we have identified model features that are not correctly managed in OCAS, clarified their intended semantics, and reflected it in our tool. Based on the semantics, we have implemented a translator to convert Altarica models into NuSMV. The translation uses *HyDI* [CMT11] as an intermediate language. The use of *HyDI* proved to be convenient as it provides primitives to deal with networks of automata, and different mechanisms for synchronizing them. The translator has been incorporated as a plugin, named the NuSMV/OCAS plugin, into the OCAS environment, and it provides the following functionalities: invariant checking, temporal model checking, and fault tree generation.

The NuSMV/OCAS plugin has been developed within the MISSA project [MIS] (More Integrated Systems Safety Assessment), an EC-sponsored project involving various research centers and industries from the avionics sector. We evaluated the plugin on a set of industrial-size case studies developed in MISSA, and compared it with existing tools available in OCAS. The results of the evaluation clearly show a significant advantage of symbolic techniques over explicit-state techniques currently provided by OCAS, in terms of performance.

The paper is organized as follows. In Section 2 we give a short overview of the Altarica syntax and semantics. In Section 3 we present the design of the translation. In Section 4 we describe the integration into OCAS. In Section 5 we discuss the experimental evaluation. Finally, in Section 6 we present some related work, and in Section 7 we conclude and discuss future work.

2 Overview of Altarica

In this section we briefly describe the syntax of the Altarica language (Dataflow dialect implemented in Cecilia OCAS) and its semantics - we refer the reader to [Alt, AGPR00] for additional details. A simple example of Altarica model is presented in Figure 1. It consists of two counters modulo 4 and an adder. The base component of an Altarica model is called *node*. Its structure may comprise the following sections:

- *sub*: used to describe the hierarchy of the Altarica nodes; in this section, it is possible to instantiate the *subnodes* which are the children of the current node;

- *state*: this section is used to declare the state variables of the (basic) node; the value of these variables may change only upon firing of an event; this implies that their value does not change in between two consecutive event firings (while other components are executing);

```
node adder
  flow
    input1:[0,3]:in;
    input2:[0,3]:in;
    value_out:[0,7]:out;
  state
    value:[0,7];
  event
    add,
    fault_add;
  trans
    value < 7 |- add -> value := input1 + input2;
    true |- fault_add -> value := 7;
  init
    value := 0;
  assert
    value_out = value;
  extern
    law <event fault_add> = Exponential(0.1);
edon

node observer
  flow
    out_ok:bool:out;
    input1:[0,3]:in;
    input2:[0,3]:in;
    inputS:[-1,6]:in;
  assert
    out_ok = (inputS = (input1 + input2));
edon
```

```
node counter
  flow
    value_out:[0,3]:out;
  state
    value:[0,3];
  event
    inc, reset;
  trans
    value < 3 |- inc -> value := value + 1;
    value = 3 |- reset -> value := 0;
  init
    value := 0;
  assert
    value_out = value;
edon

node main
  event
    total_reset;
  sub
    c1: counter;
    c2: counter;
    add: adder;
    obs: observer;
  sync
   <total_reset, c1.reset, c2.reset>;
  assert
    c1.value_out = add.input1,
    c2.value_out = add.input2,
    c1.value_out = obs.input1,
    c2.value_out = obs.input2,
    add.value_out = obs.inputS;
edon
```

Figure 1: An example Altarica model

- *init*: this section is used to specify the initial value of state variables;
- *event*: used for defining the events that can be fired and, thus, trigger a state transition;
- *flow*: this section declares flow variables, used to describe the connections with the other components; flow variables are linked to state variables by means of assertions; there are two types of flow variables, namely *input* and *output* flow variables;
- *trans*: this section is used to describe the transitions of the system; each transition consists of a guard, the firing event, and a list of assignments; the assignments specify how the system state changes when the corresponding event is fired; the guard is a precondition that has to be satisfied for the transition to be taken;
- *assert*: used to establish links from a flow variable to a state variable or another flow variable; more specifically, it declares a set of equalities either between an output flow variable and an expression over input flow and state variables (*internal assert*), or between an input flow of a subnode and the output flow of another subnode (*in-out assert*), or between an input flow of the node and an input flow of a subnode (*in-in assert*), or between an output flow of the node and an output flow of a subnode (*out-out assert*);
- *sync*: used to define the synchronizations; a synchronization associates an event of the node to the events of the subnodes; there are three types of synchronizations, namely *strong sync*, *weak sync*, and *Common Cause Failure* (CCF) (cf. end of this section);
- *extern*: used to associate events with *priorities* and optional *laws*; priorities and some of the laws constrain permissible order of event firing.

An Altarica model is a hierarchical graph composed of nodes. At the same level of the hierarchy, nodes communicate through flows and synchronizations. The hierarchy yields a tree structure, where two types of nodes are possible:

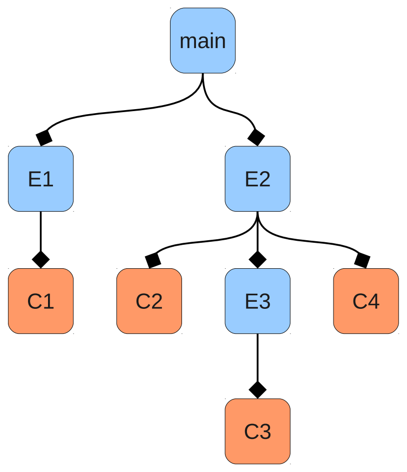


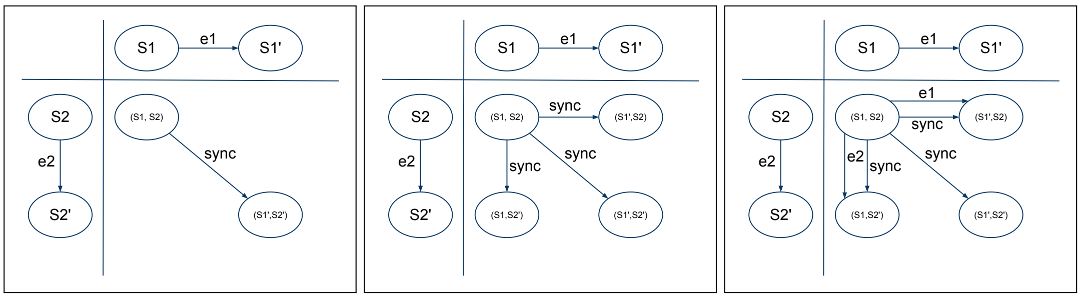
Figure 2: Altarica hierarchy

- *component* (blue in Figure 2): represents a single process of the system, it cannot contain definition of subnodes or synchronizations;
- *equipment* (orange in Figure 2): represents a container for nodes; it may contain declarations of subnodes and synchronizations, but it cannot have state variables.

As shown in Figure 2, this structure imposes that the *component* nodes represent the leafs, whereas the *equipment* nodes are containers for the components. Moreover, there is a special equipment node called *main*, which represents the root of the full Altarica model.

The semantics of the Altarica model is defined in terms of Interfaced Transition Systems (ITSs) (cf. [AGPR00, Mat11]). Intuitively, the ITS associated with a component is given straightforwardly by the state variables (which define the states), the initial condition, the transitions, the events and flow variables (which define the observations) of the node. The ITS associated to an equipment node is given by the composition of the ITSs associated with the subnodes taking into account synchronizations. The mechanisms for the different synchronizations are illustrated in Figs. 3a, 3b and 3c, and explained in more detail in the following:

- *strong sync* (see example in Figure 3a): if we have a strong sync between the events e_1 and e_2, the corresponding processes (components) p_1 and p_2 must move synchronously on such events. This means that the transitions of p_1 fired by e_1 and the transitions of p_2 fired by the event e_2 happen at the same time, and that e_1 is fired if and only if e_2 is fired; as an example, the system in Figure 1 declares a strong synchronization, called $total\_reset$, synchronizing the reset on the two counters;
- *weak sync* (see Figure 3b): this type of synchronization represents a broadcast; participating events happen synchronously as in the strong sync, but only if the corresponding transitions are enabled; this means that if the event e_1 of p_1 is fired and there exists a transition t_2 of p_2 on the event e_2 whose guard is true, then e_2 is fired at the same time as e_1; otherwise (if the guard is false) e_1 is fired and p_2 does not change state; similarly, if e_2 is fired and the guard on e_1 is false, p_1 does not change state;



(a) Strong synchronization (b) Weak synchronization (c) CCF synchronization

Figure 3: Synchronization examples

- *CCF sync* (see Figure 3c): short for Common Cause Failure, this kind of synchronization is similar to a weak synchronization, with the difference that individual processes are also allowed to move on the events independently; this means that either we have a CCF sync involving e_1 and e_2 (with the same rules of the weak sync) or e_1 is fired or e_2 is fired.

The evolution of an Altarica system can be further constrained by associating events with special *laws* and *priorities*. By default, events are considered *stochastic*. These events are typically used to model component failures and can be optionally associated with a probability distribution law (e.g., $Exponential(\lambda)$ law). These laws are used to establish interoperability with commercial RAMS (Reliability, Availability, Maintainability and Safety) analysis tools and do not affect qualitative behaviour of the system. However, a special law – $Dirac(x)$ – is used to mark *instantaneous* and *temporal* events (with $x = 0$ and $x > 0$ respectively). These events fire deterministically x time steps after the guard of the corresponding transition becomes $true$. The definition of the order on events in this case is very involved and we simplify it considering only that, whenever more than one transition is possible at the same time, instantaneous events take precedence. The precedence of transitions can be further constrained by event priorities (events with higher priority are fired first). For the sake of brevity, we do not describe the semantics of priorities in detail – we refer to Section 3 for their encoding.

3 Translation

In this section we describe the encoding of the Altarica language into NuSMV. The formal translation [Mat11] has been designed using *HyDI* [CMT11] as an intermediate language. In the following, we first introduce the *HyDI* language and then we focus on the translation of the main characteristics of Altarica into *HyDI*- we refer to [CMT11] for a discussion of the translation from *HyDI* to NuSMV. In particular, we discuss the management of:

- *hierarchy*: unlike Altarica, *HyDI* does not support hierarchical process definitions;
- *flow variables and assertions*: these definitions cannot be directly mapped into *HyDI*;
- *event priorities*: *HyDI* does not support the definition of event priorities;

- *synchronizations*: Altarica supports three kinds of synchronizations: *strong*, *weak* and *CCF*, whereas *HyDI* supports only the first two.

Finally, we briefly discuss how to model the leaf nodes.

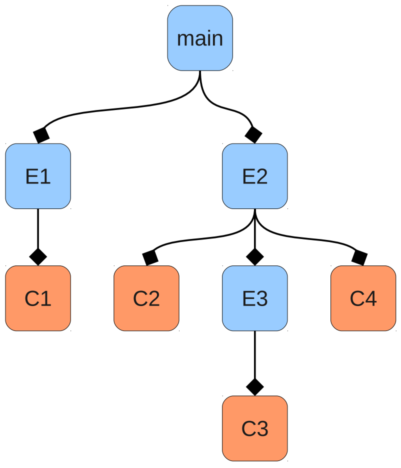
3.1 The *HyDI* language

HyDI is an extension of *SMV* [McM93] that supports the definition of networks of hybrid automata with different kinds of synchronizations. We restrict our presentation to the finite state case, thus ignoring continuous variables and their evolution – see [CMT11] for a complete description. A *HyDI* program is given by a set of *modules*, a set of *processes* and a set of *synchronization* constraints. A *HyDI* module extends *SMV* modules allowing one to specify synchronization constraints. A module contains a set of declarations which define: a set of variables (*VAR*); a set of input variables (*IVAR*); a set of initial constraints (*INIT*) defining the initial states; a set of invariant conditions (*INVAR*) which restricts the valid assignments to the variables; a set of transition constraints (*TRANS*), defining the state transitions. A module can be instantiated in the *VAR* section of another module. The main module is the top-level module of a program and cannot be instantiated. The *HyDI* language allows one to define a network of processes which run asynchronously on private events while they synchronize on shared events. The processes are instantiated in the main module. The network is not hierarchical, since the synchronizations are declared between processes. However, the definition of a single process may be hierarchical, since it can contain the instantiation of sub-modules. The module used to instantiate a process contains the definition of the set of discrete events (*EVENT* section) used to define its synchronization with other processes. In the *HyDI* language a synchronization declares that two events of two processes must be fired at the same time. A variant of this type of synchronization, called "weak" synchronization, allows one to specify a guard which forces the synchronization only if the guard evaluates to true. Finally, the order of occurrence of events can be further constrained with a scheduler, modeled in *HyDI* by variables and constraints in the main module.

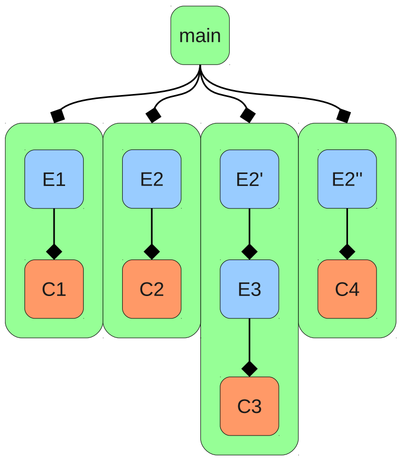
3.2 Hierarchy translation

The network of processes defined by Altarica is hierarchical in that the synchronizations may be specified at the different levels of the Altarica tree structure. Thus, in order to encode the Altarica specification into *HyDI* we perform a flattening of the Altarica hierarchy as depicted in Figure 4b. Each Altarica equipment node is split into several new instances in order to create a hierarchy corresponding to the paths from the root to each leaf. This flattening is possible since the instances of the equipment nodes cannot have definition of state variables.

For the flattening it is necessary to perform some additional transformations on the resulting structure because of the constraints imposed by the *HyDI* language. In Altarica synchronization definitions can be specified at all levels of the hierarchy (i.e., in the equipment nodes). In *HyDI* they must be in the main module. Thus, we need to move all the synchronization definitions in the top level *HyDI* main module. Another difference between *HyDI* and Altarica concerns the definition of discrete events used in the synchronizations. In *HyDI* the declaration of discrete events is done in the module definition of each instance and, thus, new events cannot be declared



(a) Altarica structure



(b) *HyDI* structure

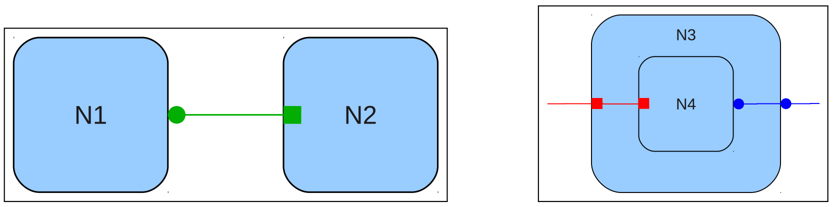
Figure 4: Hierarchy translation

in a submodule. Altarica, on the other hand, requires them to be specified within the leafs (i.e., in the component nodes). Our solution restructures the Altarica hierarchy in such a way all the events present in the original Altarica structure are declared in the definition of an instance in *HyDI*, and passed as parameters to the submodules. The drawback of this encoding consists in the possible growth in terms of resulting model size. However, this solution does not increase the complexity and also it permits to greatly simplify the translation from Altarica to *HyDI*.

3.3 Variables and assertions translation

Altarica allows one to define two types of variables: state variables (which represent the internal state of the system) and flow variables (used to expose the internal state and to link the different components). The translation of the state variables is straightforward, as they also become state variables in *HyDI*. The translation of the flow variables is carried out as follows:

- *Internal assert*: the link between output flow and state variables is expressed by an assertion. In this case the flow variable is represented as a NuSMV *define* on the state variable;
- *In-Out* (Figure 5a): in this case we have a link connecting an input flow of one component with an output flow of another component. In this case the direction is explicitly expressed by the flow labels. This is translated by passing the state variable referred to by the output flow as a parameter to the module translating the component with the input flow;
- *In-In* (Figure 5b red links): this situation is represented by the direct forwarding of an input flow to a subcomponent. In this case the solution is analogous to the previous case, with the difference that the external component plays the writer role;
- *Out-Out* (Figure 5b blue links): this case is similar to the previous one with the difference that the subcomponent plays the role of writer.



(a) Case *In-Out* (b) Case *In-In* and *Out-Out*

Figure 5: Flow translation cases

3.4 Priority, synchronization and leaf node translation

Event priorities and *Dirac(x)* laws in Altarica impose a partial order on the firing of the events. We distinguish between events with *Dirac(0)* law (which have higher priority) and events with *Dirac(x)* with $x > 0$[1]. Within each of these two classes, events are ordered by the explicit definition of the priority (an integer number). The induced partial order among the events is encoded as a scheduler in the main module of the *HyDI* translation.

The Altarica language permits the definition of three possible kinds of synchronizations between events: strong, weak, and CCF (see Figure 3 and Section 2). *HyDI* has native support for the weak and strong synchronizations, while there is no support for the CCF synchronization. We encode the CCF synchronization taking into account its semantics: a CCF involving two events e_1 and e_2 is either a weak synchronization among e_1 and e_2, or simply event e_1 or event e_2 in isolation. Thus, we duplicate events e_1 and e_2 in e'_1 and e'_2, respectively, to enable for the two events to occur in isolation, and we add a new weak synchronization between e_1 and e_2.

The translation of the leaf nodes is straightforward. Each leaf node maps to an SMV module. Each state variable is encoded into an SMV state variable of the same type. The Altarica init and trans sections directly translate into SMV INIT and TRANS formulas, respectively.

4 Tool Integration and Functionalities

In the following we describe the architecture of the NuSMV/OCAS plugin and its functionalities.

4.1 The NuSMV/OCAS plugin

The NuSMV/OCAS plugin has been developed in Python. It is composed of four main components, as illustrated in Figure 6:

- *Property*: this block provides a GUI to specify the (temporal) properties to be verified and the analysis parameters, and to invoke the verification and safety assessment routines; it extends the 'Altarica property' block, which allows only to compare a variable with a value. In the example of Altarica model presented in Figure 1 OCAS needs an observer

[1] Temporal events, i.e. those with *Dirac(x)* law for $x > 0$, in OCAS are given an operational semantics based on event queues and recursive evaluation; in this work, we have used a simplified semantics, that was sufficient for our purposes, and reduces to the original semantics under suitable hypotheses.

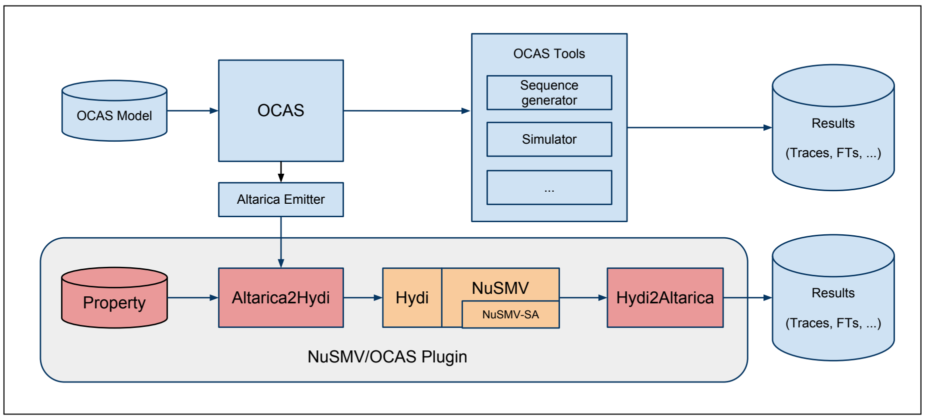


Figure 6: The NuSMV/OCAS plugin and its integration into OCAS

that internally evaluates if the output of the adder is the sum of the two counters (see *out_ok*). With our plugin this check is possible directly from the GUI;

- *Altarica2HyDI*: this module is responsible for the translation of the Altarica model into the equivalent *HyDI* specification to be given as input to the extended version of NuSMV (the NuSMV model checker extended with the NuSMV-SA and *HyDI* plugins);

- *HyDI/ NuSMV* : the verification engine;

- *HyDI2Altarica*: this module is responsible for the back conversion of the results generated by NuSMV to a format that can be visualized or executed within OCAS. In particular, it is responsible for the conversion of the traces generated by NuSMV (corresponding to a simulation or to a counterexample to a property) into the *XML* format accepted by OCAS.

The translation from Altarica to *HyDI*, provided by the *Altarica2HyDI* component, is performed in three main steps (see Figure 7):

1. *Parsing*: this module generates an abstract syntax tree (*AST*) of the Altarica design. This module relies on the ANTLR[2] parser generator;

2. *Preprocessing*: this module analyzes the *AST* generated at parsing time to build a new *AST* corresponding to the flattened Altarica model. Moreover, it collects *common information* about the structure of the design, that is re-used in the following steps of the translation;

3. *Translation*: this module, based on the new *AST* and on the structural information previously gathered, generates an in-memory Python structure corresponding to the *HyDI* model. This structure is then dumped into a textual file to be given as input to NuSMV.

The plugin calls NuSMV, waits for the results, and then converts them back into a format that can be imported into OCAS (e.g., simulation traces to be given as input to the sequence generator).

[2] ANother Tool for Language Recognition (ANTLR), http://www.antlr.org.

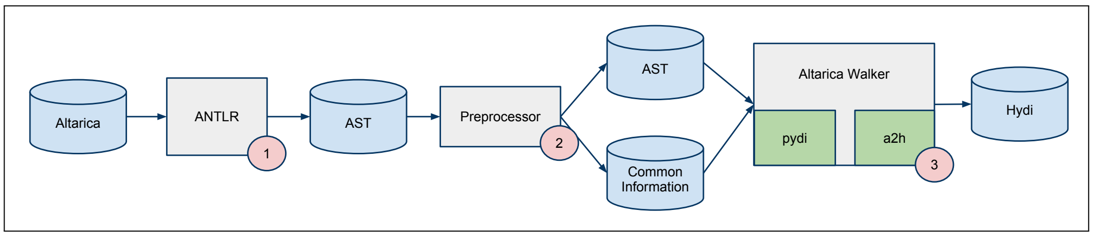


Figure 7: The *Altarica2HyDI* component

4.2 Functionalities

The NuSMV/OCAS plugin relies on NuSMV, that provides standard BDD-based (CTL and LTL) model checking techniques [McM93], and SAT-based LTL Bounded Model Checking (BMC) techniques [BCCZ99]. It allows one to perform guided and random simulation, and to re-execute partial traces. Moreover, it provides optimized model checking algorithms, developed in the MISSA project, which aim at reducing the state explosion problem with techniques that combine BDD and SAT for the verification of invariants. For formal safety assessment the NuSMV/OCAS plugin relies on an extended version of the NuSMV model checker, comprising NuSMV-SA [BV07]. NuSMV-SA allows one to investigate the behavior of a system in degraded conditions (that is, when some parts of the system are not working properly, due to malfunctioning). Key techniques in this area are (dynamic) FTA (Fault Tree Analysis), (dynamic) FMEA (Failure Modes and Effects Analysis), fault tolerance evaluation, and criticality analysis. NuSMV-SA provides advanced and very optimized techniques for the generation of (dynamic) FT and of (dynamic) FMEA tables. NuSMV-SA provides three main engines for safety assessment. The first two are based on classical BDD-based or on SAT-based techniques. The BDD-based engine is complete, but if the model is huge may not scale well. The SAT-based approach is incomplete but allows one to handle very large domains. These two basic approaches are complemented with a third complete approach, developed in the MISSA project, that combines BDD and SAT. It first uses BMC techniques, up to a given depth, to prune the search space, and then it performs an exhaustive analysis on the reduced model using BDD-based model checking algorithms.

5 Experimental Evaluation

5.1 Validation of the translation

As the formal semantics of the Altarica dialect used in OCAS is not fully documented, before starting an experimental evaluation on realistic case studies, we were confronted with the issue of validating the semantics we implemented with respect to the one implemented in OCAS. For the validation we focused on trace simulation generation and trace execution functionalities that are common to both tools. We used several small handcrafted models developed for checking some specific conditions. Then, we used some realistic case studies developed within MISSA.

The validation of the tool was done using the possibility offered by OCAS to re-execute a simulation trace on the Altarica model, using the internal trace simulator. We generated a simulation trace with the NuSMV/OCAS plugin, and then we re-executed it in the OCAS environment. The validation flow we used can be summarized as follows (compare Figure 8):

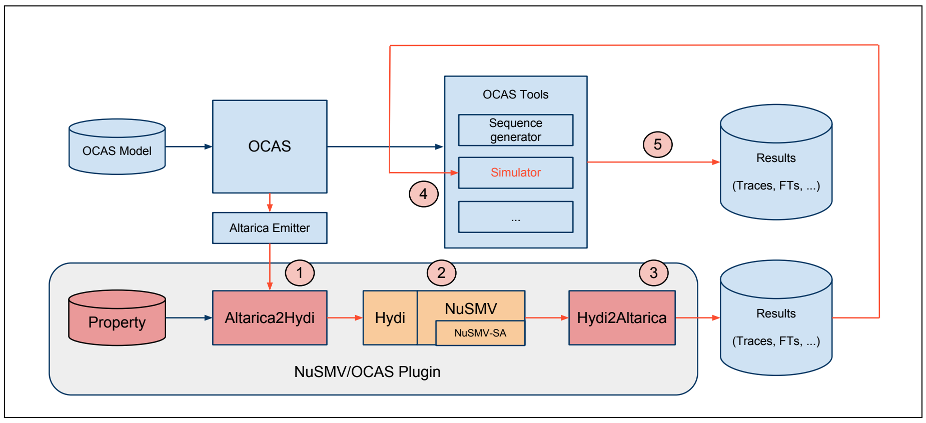


Figure 8: Trace-based validation

1. we translate the Altarica model provided by OCAS into *HyDI*, and then into *SMV*;
2. we either verify properties known to be not satisfied, or we generate random simulation traces in order to obtain an execution trace, that we save in the NuSMV *XML* format;
3. we translate the trace provided by NuSMV into the OCAS *XML* format;
4. we load the trace generated in the previous step into the trace simulator of OCAS;
5. we verify that the state reached at the end of the trace execution is compatible with the property, and with the state reported as final in the simulation trace.

Whenever a discrepancy was detected, a thorough analysis of the simulation execution in OCAS was carried out to identify the cause of the discrepancy and - if needed - come up with a fix in the translation to capture OCAS semantics. In a few cases, the behavior shown by OCAS was found to be incorrect by the users, hence not reflected in the translator (cf. Section 1).

5.2 Verification and safety assessment on industrial case studies

In this section we discuss the comparison between the common functionalities provided by the OCAS sequence generator and the NuSMV/OCAS plugin. The sequence generator of OCAS is able to perform Fault Tree Analysis (generation of minimal sequences) up to a bounded depth (at most, 9). For a fair comparison, we then compared this feature with the Fault Tree Analysis provided by NuSMV-SA that relies on the mixed BDD+SAT approaches[3].

For the experimental evaluation we used four industrial models developed in MISSA. The ELEC_1, ELEC_2, and ELEC_3 models describe a simplified electrical power distribution system (that resembles that of the A320 aircraft), at different levels of detail. The BRSYS model is a realistic model of the braking system of an aircraft. The properties to be analyzed formalize different failure conditions (e.g., “Loss of deceleration capability during landing” for the BRSYS model). The characteristics of the models are reported in Table 1. This table also shows the time

[3] We also used the NuSMV/OCAS plugin to verify temporal properties of the Altarica design; as this functionality is not available in OCAS, we do not report the results here.

	# States	# Nodes	Translation time	Translation memory
ELEC_1	$1.49x10^5$	41	1.127s	27MB
ELEC_2	$2.64x10^5$	44	2.782s	38MB
ELEC_3	$2.0x10^7$	51	2.811s	37MB
BRSYS	$3.8x10^{25}$	135	9.820s	69MB

Table 1: Characteristics of the industrial case studies and translation requirements

	SG	Depth search
ELEC_1	2.4s	9
ELEC_2	653.7s	7
ELEC_3	97.7s	6
BRSYS	16.4s	3

	BDD+SAT	BDD
ELEC_1	1.5s	0.856s
ELEC_2	94.7s	189.2s
ELEC_3	95.6s	120.2s
BRSYS	22.36s	771.7s

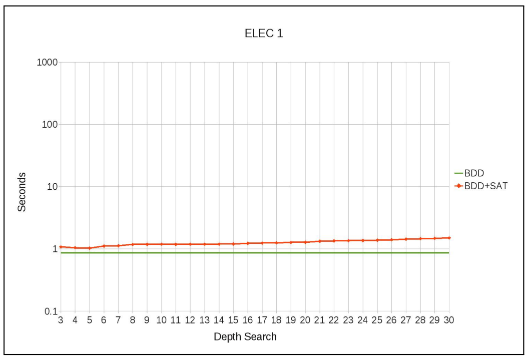
Table 2: Sequence Generator and NuSMV performance

and memory requirements needed to translate the model into an equivalent *HyDI* specification. Note that time and memory increase with the model complexity (however, the translation is performed only once for each given model, whenever several properties have to be verified).

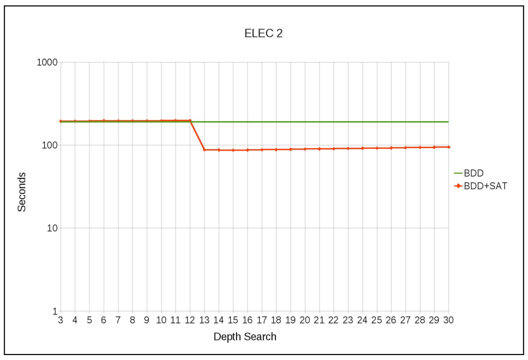
We executed the tests on a laptop equipped with an Intel 3GHz CPU, and with 4GB of RAM running Windows 7. We used a memory limit of 1GB and a timeout of 1000 seconds. We first compared the performances of the OCAS Sequence Generator and NuSMV. The results are presented in Table 2. The Sequence Generator reached the timeout for all models except ELEC_1; in case of timeout, Table 2 reports the maximum search depth explored, and the corresponding time. For ELEC_1, the Sequence Generator reached the maximum allowed depth (that is, 9). On the right hand side, we report the performance of NuSMV using the BDD and BDD+SAT algorithms. We notice that NuSMV never timeouts, and BDD+SAT performs consistently better than pure BDD, except on the simplest model. Moreover, it is always faster than the Sequence Generator, except on the BRSYS model (where OCAS timeouts at depth 4). Furthermore, it should be noted that NuSMV performs an exhaustive search, whereas the Sequence Generator is incomplete, that is, it is not guaranteed to find all the cutsets. Concerning memory, NuSMV used up to 36 MB, whereas OCAS allocated up to 100MB. A detailed comparison is difficult, as it is not possible to trace precisely how OCAS uses the allocated memory. We also evaluated the scalability of the NuSMV/OCAS plugin, as shown in picture 9. We notice that NuSMV has a behavior which is nearly independent of the depth of the verification. We remark that the 'step' behavior which is visible in some BDD+SAT plots is due to the fact that for higher depths, SAT may be able to find additional results, that are used to prune the search space before BDD is run.

6 Related Work

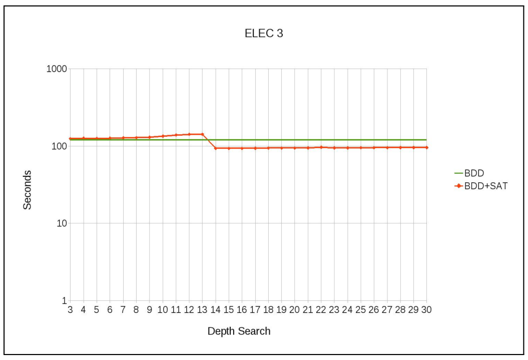
The original language of Altarica, developed by LaBRI, is based on the notion of interfaced constraint automata. A restricted dialect - Altarica Dataflow - was later developed to restrict the complexity of the models and, under certain constraints, permit synthesis of the fault trees [BDRS06, Rau02]. Dialects of Altarica are supported by a number of tools ranging from the academic



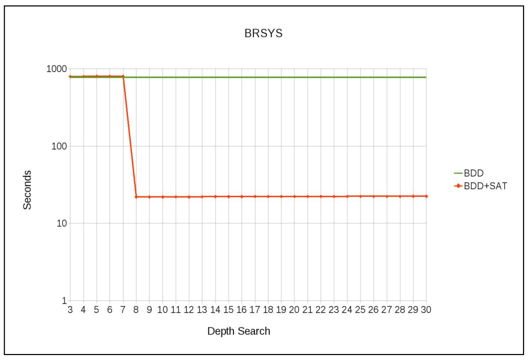
(a) ELEC_1



(b) ELEC_2



(c) ELEC_3



(d) BRSYS

Figure 9: Scalability of BDD and BDD+SAT technology

toolset developed and maintained at the University of Bordeaux [Alt] to SIMFIA [SIM], a modelling, simulation and RAMS analysis environment developed by EADS APSYS - that supports a Dataflow dialect similar to that implemented by OCAS. Another workbench, COMBAVA, has been previously developed by ARBoost Technologies but is now obsolete. To our knowledge, OCAS is the most industrially mature of existing toolsets. OCAS is tightly integrated with Cecilia ARBOR - a Fault Tree Analysis software. Quantitative and Qualitative analysis of fault trees performed in both Cecilia ARBOR and SIMFIA Safety modules are based on Aralia [Rau01]. Whilst there also exists a plugin for synthesis of fault trees (implementing the algorithm of [Rau02]), such functionality is only available for a very restricted subset of Altarica Dataflow.

There are other model checkers that support Altarica, in particular MEC 5 [MEC] and Arc [Arc]. MEC 5 is a somewhat outdated model checker that is now superseded by Arc. Arc is a more recent, BDD-based model checker based on the Altarica language, which supports CTL* temporal logics and μ-calculus. Arc is not currently linked to OCAS and the interoperability with a MEC 5 plugin has not been supported in newer versions of OCAS. Moreover, neither Arc nor its predecessor MEC support safety assessment functionalities. Altarica studio [GPV11] is a prototypical toolset, based on Arc, for model-based formal analyses. To our knowledge, safety assessment functionalities are not available in Altarica studio, yet. A thorough comparison of the model checking engines is hard because of differences in the dialects (and flavours thereof) of Altarica supported by the different tools. This work has been focused on (a variant of) Altarica Dataflow - a more extended comparison will be targeted for future work.

7 Conclusions and Future Work

In this work we have presented a novel encoding of Altarica models into NuSMV, which enables verification and safety assessment of Altarica models using state-of-art symbolic model checking and formal safety assessment techniques. We have integrated the encoder as a plugin into the OCAS environment, and we have experimentally demonstrated the feasibility of the approach by evaluating the plugin on a set of industrial case studies. As part of our future work, we plan to address the semantics of Altarica temporal events, which was simplified in the current implementation. Finally, we plan to investigate a timed extension of Altarica, along the lines of [CPR04]. This extension fits very naturally in our framework, given that the *HyDI* language provides a native support for encoding networks of timed (more in general, hybrid) systems.

Acknowledgements: This work has been supported by the E.C. project MISSA, contract no. ACP7-GA-2008-212088. We would like to thank Chris Papadopoulos (Airbus UK), Pierre Bieber and Christel Seguin (Onera), Xavier Leduc and Valerie Sartor (Dassault Aviation), Laurent Sagaspe (EADS APSYS) and Antonella Cavallo (Alenia Aeronautica) for their precious support and advice for the development, integration and evaluation of the plugin. The ELEC models used for evaluation have been originally developed by ONERA and, in some cases, expanded by Alenia Aeronautica. The BRSYS model has been developed by EADS APSYS.

Bibliography

[ÅBB06] O. Åkerlund, P. Bieber, E. Böede et al. ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. In *Proc. ERTS*. 2006.

[AGPR00] A. Arnold, A. Griffault, G. Point, A. Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae* 40:109–124, 2000.

[Alt] The Altarica language. `http://altarica.labri.fr/forge`.

[Arc] The Arc model checker. `http://altarica.labri.fr/forge/projects/arc/wiki`.

[Ba03] M. Bozzano, et al. ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In *Proc. ESREL*. Pp. 237–245. Balkema Publisher, 2003.

[BBC$^+$04] P. Bieber, C. Bougnol, C. Castel, J.-P. Christophe Kehren, S. Metge, C. Seguin. Safety Assessment with Altarica. In *Building the Information Society*. IFIP International Federation for Information Processing 156, pp. 505–510. 2004.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS*. LNCS 1579, pp. 193–207. Springer, 1999.

[BCK+10] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri. Safety, Dependability, and Performance Analysis of Extended AADL Models. *The Computer Journal* doi: 10.1093/com, March 2010.

[BCS02] P. Bieber, C. Castel, C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In *Proc. EDCC-4*. LNCS 2485, pp. 19–31. Springer, 2002.

[BDRS06] M. Boiteau, Y. Dutuit, A. Rauzy, J.-P. Signoret. The AltaRica Data-Flow Language in Use: Modelling of Production Availability of a Multi-State System. *Reliability Engineering and System Safety* 91(7):747–755, 2006.

[Bry92] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys* 24(3):293–318, 1992.

[BV07] M. Bozzano, A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *Software Tools for Technology Transfer* 9(1):5–24, 2007.

[BV10] M. Bozzano, A. Villafiorita. *Design and Safety Assessment of Critical Systems*. CRC Press (Taylor and Francis), an Auerbach Book, 2010.

[CMT11] A. Cimatti, S. Mover, S. Tonetta. HYDI: a language for symbolic hybrid systems with discrete interaction. Technical report, Fondazione Bruno Kessler, 2011. https://es.fbk.eu/people/mover/hydi

[CPR04] F. Cassez, C. Pagetti, O. Roux. A timed extension for AltaRica. *Fundamenta Informaticæ* 62(3–4):291–332, 2004.

[FSA] The FSAP/NuSMV-SA platform. `http://es.fbk.eu/tools/FSAP`.

[GPV11] A. Griffault, G. Point, A. Vincent. Altarica-studio : the easier way to do model checking. In *Proc. MBSAW 2011*. 2011.

[Mat11] C. Mattarei. Definizione e sviluppo di una traduzione formale da Altarica ad Hydi per la verifica di sistemi avionici. Master's thesis, Università degli studi di Trento, 2011.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MEC] The MEC model checker. `http://altarica.labri.fr/forge/projects/mec/wiki`.

[MIS] The MISSA Project. http://www.missa-fp7.eu.

[NuS] The NuSMV model checker. `http://nusmv.fbk.eu`.

[Rau01] A. Rauzy. Mathematical Foundations of Minimal Cutsets. *IEEE Transactions on Reliability* 50(4):389–396, 2001.

[Rau02] A. Rauzy. Mode Automata and Their Compilation into Fault Trees. *Reliability Engineering and System Safety* 78(1):1–12, 2002.

[SIM] SIMFIA. `http://www.apsys.eads.net/en/17/Software`.