Safety Assessment of AltaRica models via Symbolic Model Checking

Marco Bozzano[a], Alessandro Cimatti[a], Oleg Lisagor[b], Cristian Mattarei[a], Sergio Mover[a], Marco Roveri[a], Stefano Tonetta[a]

[a]*Fondazione Bruno Kessler, Trento, Italy*
[b]*The University of York, York, United Kingdom*

Abstract

AltaRica is a language used to describe safety critical systems that has become a de-facto European industrial standard for Model-Based Safety Assessment (MBSA). However, even the most mature tool for the support for MBSA of AltaRica models, i.e. Dassault's OCAS, has several limitations. The most important ones are its inability to perform many analyses exhaustively, severe scalability issues, and the lack of model checking techniques for temporal properties.

In this paper we present a novel approach for the analysis of AltaRica models, based on a translation into an extended version of the model checker NuSMV. The translation relies on a novel formal characterization of the Dataflow dialect of AltaRica used in OCAS. The translation is formally defined, and its correctness is proved. Based on this formal characterization, a toolset has been developed and integrated within OCAS, thus enabling functional verification and safety assessment with the state of the art techniques of NuSMV. The whole approach is validated by an experimental evaluation on a set of industrial case studies, which demonstrates the advantages of the proposed technique over the currently available tools.

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

1. Introduction

Safety-critical systems are traditionally subject to a high degree of authority, which mandates the application of suitable techniques for safety assessment (e.g., Fault Tree Analysis (FTA) [1, 2] or Failure Modes and Effects Analysis (FMEA)), to analyze the behavior of the system in presence of faults.

The dramatic increase in complexity of computer-based systems has motivated, in recent years, a growing industrial interest in *model-based* safety assessment (MBSA) methods. In contrast to traditional safety assessment, where each safety-significant event (e.g., a system hazard, or component failure mode) is subject to separate analysis, MBSA methods are based on a single "safety model" of a system, typically written in a suitable modeling language. Analyses with respect to particular safety-significant events are conducted on the basis of such "safety models" with a high degree of automation [3]. Many of the MBSA techniques allow the verification of the functional correctness of the system (e.g., with respect to functional safety requirements) as well as to assess system behavior in the presence of faults [4, 5, 6]. In particular, formal verification tools based on model checking have been extended to automate the generation of artifacts such as Fault Trees (FTs) and FMEA tables [7, 8, 9].

AltaRica [10, 11] is a modelling language for MBSA, specifically developed by a consortium of French industrial and academic partners for safety assessment of industrial systems (see, e.g., [12, 13]). AltaRica has been used in several projects by companies and consortia like Airbus, Alenia Aeronautica, Alstom Rail, Dassault Aviation, EADS, France Telecom, Schneider Electric, Thales, Total and Turbomeca, and has become a de-facto European industrial standard over the course of the last decade or so. In fact, to authors' knowledge, it is the only MBSA language that has been successfully used for certification of industrial safety-critical systems.

AltaRica is the modeling language used in Cecilia OCAS [13] (OCAS for short), a model-based safety assessment platform developed by Dassault Aviation. OCAS provides a graphical modelling environment, which allows us to describe dynamic transition systems and it is well understood and accepted by the engineers. OCAS is also equipped with different model analysis tools, including a trace simulator (able to generate possible evolutions of the system), and a sequence generator that is used to generate minimal cut sets [1, 2]. Interestingly, OCAS is the only MBSA tool that was accepted by regulatory agencies as a basis for certification of safety-critical systems. The tool was qualified to the requirements of DO-178b [14], the industrial standard for software development and assurance in the aviation sector. OCAS demonstrated its usefulness for the architectural safety assessment of avionics systems, for example the OCAS analysis was the basis for the certification of the Flight Control System of Falcon 7x aircraft.

However, the size reached by industrial systems constantly increases the labor and costs in certification, and calls for increasingly automated and highly scalable

techniques and tools. In particular, the analysis tools in OCAS are subject to some important limitations. First, some analyses are not able to perform an *exhaustive* space examination, and are not complete. For example, reachability analysis is bounded in depth; similarly, the OCAS sequence generator cannot explore non-deterministic instantaneous transitions, potentially leading to incomplete analysis results. Second, the OCAS sequence generator demonstrates limited scalability in presence of industrial-sized systems, hindering the generation of important artifacts such as Fault Trees. Finally, OCAS does not support the representation and analysis of temporal properties of AltaRica models, which makes model validation an extremely hard task.

In this paper we propose an approach that overcomes these limitations. The idea is to translate AltaRica models into HyDI [15], one of the input languages of NuSMV [16]. NuSMV is a state-of-the-art symbolic model checker, providing cutting-edge model checking technologies such as Binary Decision Diagrams (BDD)-based [17] and SAT-based Bounded Model Checking (BMC) [18] techniques. NuSMV 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 [9], and is one of the most mature formal verification tools available to date. The translation enables the use of the NuSMV functionalities, thus extending the existing analysis techniques for AltaRica models.

Although in principle simple, realizing this flow posed a key challenge: the formal characterization of the AltaRica language defined in [11] lacks a clear link between the concrete syntax of the language and the abstract syntax (and thus its semantics). Our first contribution is to provide a formal characterization of the AltaRica language in the Dataflow formulation used in OCAS. The formalization includes a rich abstract syntax, and the corresponding semantics, taking into account the recursive definition of nodes, input/output flows, and their connections through assertions.

Second, we formally define a translation from AltaRica to HyDI, and prove its correctness, showing a one-to-one correspondence in the behaviors of the models. This result is the foundation for the correctness of the overall approach, and it allows us to map artifacts from HyDI (e.g., counterexample traces produced by NuSMV) back to AltaRica, and constitutes a key enabler for a future qualification of the certification process.

Based on this result, we developed the NuSMV/OCAS plugin, which extends the OCAS environment with the following functionalities: invariant checking,

temporal model checking, and fault tree generation. The NuSMV/OCAS plugin was developed within MISSA (More Integrated Systems Safety Assessment), a European Community-sponsored project involving various research centers and industries from the avionics sector [19]. The industrial partners conveyed important insights on the functionalities of interest, and provided some case studies.

The NuSMV/OCAS plugin was validated with respect to the behavior shown by the OCAS simulator. We evaluated both behaviors on a set of industrial-size case studies developed in MISSA, and compared it, when possible, with the existing analysis tools of OCAS. The results of the evaluation clearly show that this activity resulted in a significant improvement of OCAS, both in terms of available functionalities, and in terms of performance improvement.

The paper is organized as follows. First, in Section 2 we discuss our approach with the existing related works, then in Section 3 we give an overview of the AltaRica Dataflow formulation while in Section 4 we introduce the HyDI language. We present an informal description of our translation in Section 5. In Section 6 we present the newly defined abstract syntax and semantics of AltaRica and its translation to HyDI. In Section 7 we present the integration into OCAS. Finally, in Section 8 we discuss the experimental evaluation and in Section 9 we conclude and discuss future work.

2. Related Work

Since early 1990s a large number of MBSA languages and notations became available. The most prominent languages include Failure Propagation and Transformation Notation and Calculus (FPTN and FPTC) [20, 21], Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [22], Error Model Annex of Architecture Analysis and Design Language (AADL) [23], and AltaRica language. In the authors' experience, only latter three languages have matured beyond the level of research prototypes.

The original language of AltaRica, developed by LaBRI, is based on the notion of interfaced constraint automata, and is highly expressive. In this paper, we adopt the restricted dialect of AltaRica, called AltaRica Dataflow, that was later developed to restrict the complexity of the models and, under certain constraints, to permit the synthesis of the fault trees [24, 25].

The reference formal characterization of AltaRica is defined in [11]. Here we provide a richer formal characterization, driven by the need to recursively define a translation towards the HyDI language. In particular, we defined an abstract syntax of AltaRica, and a recursively defined semantics. In particular, our abstract

syntax differs from the one presented in [11] in that it directly defines concepts like the input and output flows, which are widely used in the concrete language, while it simplifies less useful notions, like the global coordination expressed by observer nodes.

There exist several tools for modeling in (various dialects of) AltaRica. Among these, we mention the academic toolset developed and maintained at the University of Bordeaux [10]; SIMFIA [26], a modelling, simulation and RAMS analysis environment developed by EADS APSYS that supports a Dataflow dialect similar to that implemented by OCAS; COMBAVA, developed by ARBoost Technologies, and now obsolete.

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 [27]. Whilst there also exists a plugin for the synthesis of fault trees (implementing the algorithm of [25]), such functionality is only available for a very restricted subset of AltaRica Dataflow.

In practice, AltaRica models in OCAS are analysed by the sequence generator, which analyses the model and, depending on configuration, produces a set of either sequences or sets of events that are sufficient for a particular condition in the model to hold. Informally, if model events represent failures of components and selected condition of interest represents a hazard, the sequence generator produces a close equivalent of FTA's minimal cut sets.

There are other model checkers that support AltaRica, in particular MEC 5 [28] and Arc [29]. 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 [30] 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 AltaRica Dataflow - a more extended comparison will be targeted for future work.

3. The AltaRica language

In this section we briefly describe the syntax of the AltaRica language, and in particular, the Dataflow dialect and its semantics - we refer the reader to [10, 11] for additional details. A simple example of an AltaRica model is presented in

```
node main
  event
    total_reset;
  sub
    cs: counters;
    add: adder;
    obs: observer;
  sync
    <total_reset, cs.c1.reset, cs.c2.reset>;
  assert
    cs.output1 = add.input1,
    cs.output2 = add.input2,
    cs.output1 = obs.input1,
    cs.output2 = obs.input2,
    add.value_out = obs.inputS;
edon

node counters
  flow
    output1:[0,3]:out;
    output2:[0,3]:out;
  sub
    c1: counter;
    c2: counter;
  assert
    c1.value_out = output1,
    c2.value_out = output2;
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 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;
edon
```

Figure 1: Adder example in AltaRica

Figure 1. The system consists of two modulo 4 counters and an adder. The base component of an AltaRica model is called a *node*. A node is composed by the following sections:

- *event*: used for defining the events that can be fired and, thus, trigger a state transition;
- *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);

- *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;
- *init*: this section is used to specify the initial value of state 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*);
- *sub*: used to describe the hierarchy of the AltaRica nodes; in this section, it is possible to instantiate the *subnodes* that are the children of the current node;
- *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) (c.f. end of this section).

An AltaRica model is a hierarchical tree composed of nodes. The nodes can communicate through links defined in their common parent nodes. These links can be expressed as equalities over the flow variables (i.e. in the assert section), or via synchronizations. The AltaRica structure is composed of two types of nodes:

- *equipment* (*main* and *cs* white squares in Figure 2): represents a container for nodes; it may contain declarations of subnodes, synchronizations and flow links between subnodes, but it cannot have state variables;
- *component* (gray in Figure 2): represents a single process of the system, it cannot contain definition of subnodes or synchronizations.

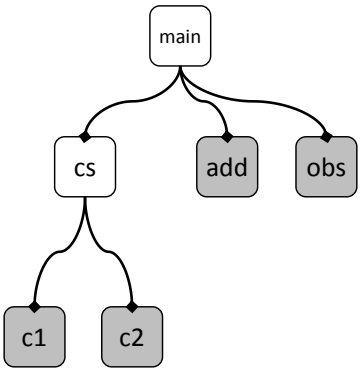


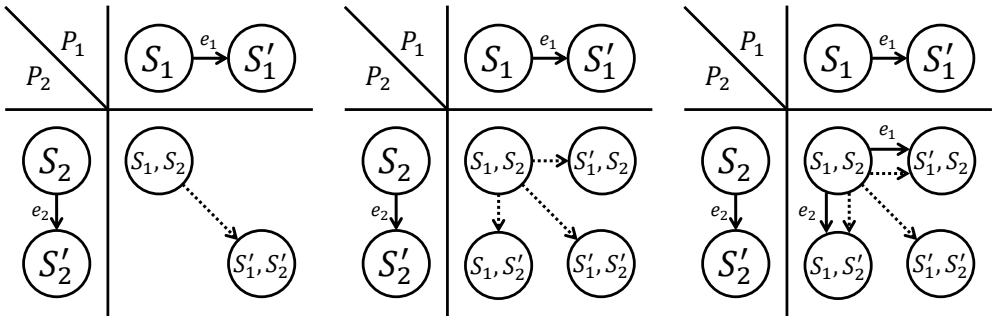
Figure 2: Hierarchical view of adder example

as shown in figure 2, the *component* nodes represent the leaves, 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) (c.f. [11, 31]). 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.

Strong Sync. As in the example in Figure 3a, if we have a strong sync between the events e_1 and e_2 (in AltaRica expressed as $\langle e_1, e_2 \rangle$), 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 atomically, 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. Figure 3b expresses the behavior of weak sync. 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



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

Figure 3: AltaRica synchronizations: in each figure the bottom right corner represents the allowed transitions in the composition of the two systems. Dashed arrows represent a transition where both P_1 and P_2 move (i.e. the fired event in the composition is e_1/e_2), while solid arrows represent a local transition.

the guard on e_1 is false, p_1 does not change state. Such kind of synchronization is expressed in the AltaRica language using as $\langle e_1|e_2|\ldots|e_n\rangle$.

CCF Sync. The Common Cause Failure (see Figure 3c) synchronization is similar to a weak synchronization, with the difference that individual processes can move on the events independently. This means that either we have a 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 motivation for CCF sync is to describe the condition when two components can fail due to an internal error, represented by the e_1 and e_2 events, or by means of a common cause, for instance the engine burst. In this case we can express such condition using CCF (with concrete syntax as $\langle e_1?e_2\rangle$), where each failure can be caused by the engine explosion (i.e. the weak sync) or by internal errors.

The evolution of an AltaRica system can be further constrained by associating events with a specific priority. This definition is possible by providing a partial order in the *event* section e.g. the highest priority of event e_1 with respect to e_2 it is expressed with $e_1 < e_2$. Differently, some AltaRica based tools, support the event priority feature by extending the language with a specific section called *extern*. In the case of OCAS, the partial ordering of events is provided by labeling them with probability distribution laws e.g. the definition of an event of type $Dirac(2)$ has priority respect to a $Dirac(3)$, but not to a $Dirac(0)$. These laws, considering $Dirac(x)$ as well as $Exponential(x)$, are used to establish interoperability with commercial RAMS (Reliability, Availability, Maintainability and Safety) analysis tools and do not affect the qualitative behaviour of the system.

4. The HyDI language

The HyDI [15] language extends the standard symbolic language SMV, the main input language of the NuSMV model checker, with continuous variables and synchronization aspects. In the following, we disregard the continuous aspects of HyDI, since they are not used in the translation from AltaRica. Compared to SMV, HyDI models asynchronous components, instead of synchronous systems, which communicate through message passing and shared variables. Since NuSMV provides an automatic mapping from HyDI to SMV, HyDI models can be analyzed by the safety-assessment techniques implemented in NuSMV. A direct mapping from an asynchronous language, like AltaRica, to SMV is non-trivial, time-consuming and error prone, since the translation requires to manually encode all the asynchronous aspects of the system. In practice, this amounts to modify the behavior of all the asynchronous processes adding idle transitions, and to enforce the complex synchronization constraints. Thus, the main advantage of HyDI over SMV is to directly model asynchronous systems.

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. Moreover, a module defines a set of input parameters. A module can be instantiated in the *VAR* section of another module. The module instantiation defines the actual parameters passed to the module, thus enabling the sharing of variables among different instances. 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.

5. Translation from AltaRica to HyDI

5.1. Overview

In this section we describe the encoding of the AltaRica language into NuSMV. The formal translation [31] has been designed using HyDI [15] as an intermediate language. We refer to [15] for a discussion of the translation from HyDI to NuSMV.

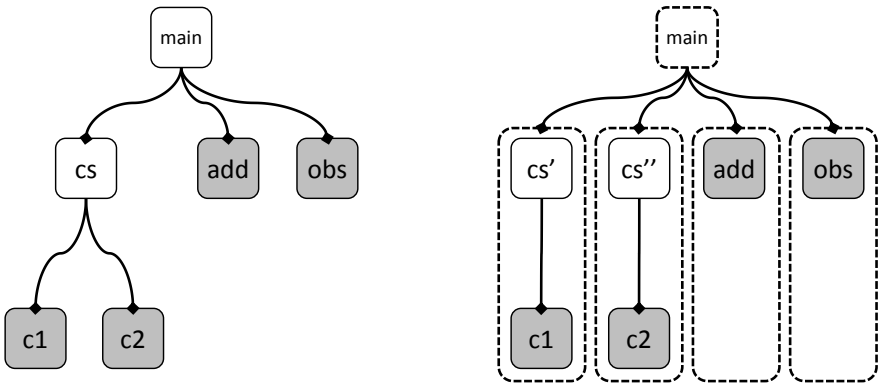
In the following we detail the main steps of the translation:

- *Flattening of the hierarchy*: unlike AltaRica, HyDI does not support hierarchical process definitions. Hence, we describe a preliminary step of the translation which flattens the structure of the AltaRica program.
- *Translation of flow variables and assertions*: these definitions cannot be directly mapped into HyDI;
- *Translation of 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.

5.2. Flattening of the hierarchy

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 to the top level



(a) AltaRica structure (b) HyDI structure

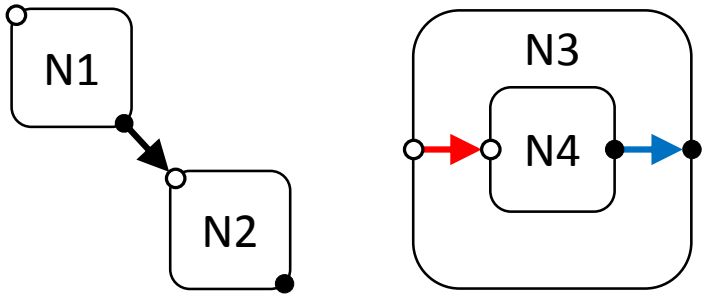
Figure 4: Hierarchy translation

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 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 that 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 greatly simplifies the translation from AltaRica to HyDI. Finally, 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.

5.3. Translation of variables and assertions constraints

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



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

Figure 5: Flow translation cases

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.

5.4. Translation of priorities and synchronization constraints

In AltaRica it is possible to define priorities that induce a partial order among events. This feature is encoded into HyDI by defining an explicit scheduler that imposes such priority constraints over the events.

The AltaRica language permits the definition of three possible kinds of synchronizations between events: strong, weak, and CCF (see Figure 3 and Section 6.2). 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.

6. Formal Properties of the Translation

This section provides a formal definition of the translation from AltaRica to HyDI. We first provide the formal semantics of the AltaRica and the HyDI languages, and then we formally define the translation from AltaRica to HyDI, proving the correctness of the overall approach. The semantics of AltaRica and HyDI has been formalized using a standard model-theoretic approach to the definition of the semantics of formal languages. In particular, AltaRica and HyDI programs are interpreted with a variant of transition systems, as usual in model checking. This was preferred to other approaches to formal semantics, such as for example Structural Operational Semantics [32], that are well established for programming languages.

6.1. Background

In this section we define some background notions used to define the semantics of the AltaRica language. We will define the semantics of AltaRica using an extension of a labeled transition system, called *Interface transition system* [11] (ITS).

Definition 6.1 (Interfaced Transition System [11]). *An* Interfaced Transition System *(ITS) is a tuple* $A = \langle E, O, C, I, \pi, T \rangle$ *where:*

- E *is set of events,* $\varepsilon \in E$*;*
- O *is a set of observations;*
- C *is a set of configurations (i.e. states of the systems);*
- $I \subseteq C$ *is the set of initial configurations;*
- $\pi : C \to O$ *is a function mapping each configuration* $c \in C$ *to an observation* $\pi(c) \in O$*;*
- $T \subseteq C \times E \times C$ *is a set of transitions, such that* $\langle c, \varepsilon, c \rangle$ *for each* $c \in C$*.*

An ITS extends a labeled transition system with a set of observations O and the mapping function π. Intuitively, π defines a mapping from each state of the system, also called configuration, to an observation in O. O is defined generically, and represents a set of states observable by another ITS. The observation are used to expose the internal state of a component to its parent, in a hierarchical composition of ITSs. Note that AltaRica features this hierarchical structure. Also, the label ε is used to denote the *stuttering* event. Intuitively, ε labels the transition $\langle c, \varepsilon, c\rangle$ that does not change the configuration of the ITS. This transition is used to synchronize with other ITS, with the implicit meaning the ITS is not changing its configuration, thus modeling asynchronous behaviors in a synchronous setting.

Given an ITS A and an ordering relation $<_E$ over the events of A, we define the *priority resolution* operation. The operation defines a new ITS $A_{<_E}$, which differ from A only for the transition relation. The transition relation of $A_{<_E}$ contains only the transitions with higher priority (according to $<_E$). Formally:

Definition 6.2 (Priority resolution). *Given an Interfaced Transition System $A = \langle E, O, C, I, \pi, T\rangle$ and an ordering $<_E$ over E, we define the priority resolved ITS $A_{<_E} = \langle E, O, C, I, \pi, T \restriction <_E\rangle$, where $T \restriction <_E = \{\langle c, e, c'\rangle \in T \mid \nexists \langle c, e', c''\rangle \in T, e' <_E e\}$.*

Both AltaRica and HyDI feature a symbolic representation of the system, where set of states and relations are represented by formulas. Given a set of variables V, we use V' to denote the set of next state variables $\{v'\}_{v\in V}$, where v' represents the next value of v. Given a set of variables $V = \{v_1, \ldots, v_n\}$ and a prefix p, we will denote with $p.V$ the set of variables $p.V = \{p.v_1, \ldots, p.v_n\}$, obtained by renaming all the variables by adding the prefix "$p.$". In the following, we use the standard notation for propositional logic $s \models \phi(V)$, to express that s is a model of the formula $\phi(V)$. A state of the system becomes an assignment to the variables V. A set of states is represented by a formula $\alpha(V)$ over the variables V: a state s belongs to the set if s, as an assignment, makes the formula true (namely, $s \models \alpha(V)$)

We will often use an *evaluation function* σ, which returns the set of all the possible assignments to the variables in the set V. If V is the set of variables of a system, then $\sigma(V)$ returns the set of all the states of the system.

Definition 6.3 (Evaluation function). *Given a set of variables $S = \{s_1, \ldots, s_n\}$ the* evaluation function *σ is defined as: $\sigma(S) = \{(s_1 = v_1, \ldots, s_n = v_n) \mid v_i \in Dom(s_i)\}$.*

6.2. The semantics of AltaRica

6.2.1. Syntax

In Definition 6.4 we provide a variant of the abstract syntax of the AltaRica language described in [11]. The motivation for introducing a different definition of the AltaRica syntax is to have an abstract syntax that is closer to the concrete one and that clarifies the definition given in the original presentation. In particular, we define a relational function between flow variables that differs from the original one that is expressed by means of a controller node. Then, for clarity we explicitly split the mapping between state and flow variables and the invariants over flow variables, which are merged in the concrete syntax (section "assert"). Moreover, we need to avoid the possibility of constraining input flow variables due to the fact they are read-only ports. This limitation is guaranteed by the fact that output flows are defined over input flows and state variables. However, we need also to impose a single assertion for each output flow in order to avoid indirect and transitive constraints e.g., if we have $output = input$ and $output = 1$, it implicitly defines that $input = 1$. Moreover, we allow for a hierarchy definition by permitting composition of AltaRica nodes, and this permits us to provide a clear separation between abstract syntax and semantics, which is expressed via the Interfaced Transition System. Finally, our definition of the AltaRica structure is based on a symbolic representation.

Definition 6.4 (AltaRica Node). *An AltaRica node is a tuple* $N = \langle E, <_E, S, F, I, T, P, N_1, \ldots, N_n, A, V \rangle$*, with* $n \geq 0$*, where:*

- E *is a set of events partially ordered by* $<_E$*.*
- S *is the set of state variables;*
- $F = F_I \cup F_O$ *is the set of flow variables, split into input and output flows, with* $F_I \cap F_O = \emptyset$
- $I(S)$ *is the initial formula;*
- $T(S, F_I, \Gamma_E, S')$ *is the transition formula;*
- $P : F_O \rightarrow Expr(S)$ *is a function mapping output flow variables to expressions over state variables;*
- $\forall i = 1, \ldots, n \; N_i = \langle E_i, <_{E_i}, S_i, F_i, I_i, T_i, P_i, N_{i1}, \ldots, N_{im}, A_i, V_i \rangle$*, with* $m \geq 0$*, is an AltaRica node, called sub-node;*

- $A(F, N_1.F_1, \ldots, N_n.F_n)$ *is an invariant formula over the flow variables representing the assert section ($N_i.F_i$ is the set of flow variables of the sub-node N_i); thus, A is in the form of a conjunction of equalities $v = e$, one for every variable v in $F_O \cup \bigcup_{1 \leq i \leq n} N_i.F_{iI}$, where e is an expression over $S \cup F_I \cup \bigcup_{1 \leq i \leq n} N_i.F_{iO}$;*

- $V \subseteq E^? \times E_1^? \times ... \times E_n^?$ *is the set of synchronization vectors defined over the extended sets of events $E_i^? = E_i \cup \{e? \mid e \in E_i - \{\varepsilon\}\}$ (i.e., $E_i?$ contains also a copy of the events marked with the $?$ symbols). If $n > 0$, V always contains the vector $\vec{\varepsilon} = \langle \varepsilon, ..., \varepsilon \rangle$.*

With the symbol Γ_E we denote a variable with $Dom(\Gamma_E) = E$ that is used to represent the current event of the system. For example, the transition constraint $(\Gamma_E = e_1) \rightarrow counter' = counter + 1$ describe a transition where the counter variable is incremented by 1 if the event is e_1 (i.e., the label of the transition is e_1).

An AltaRica program is hierarchically defined as a tree of AltaRica nodes, where only the leaves contain state variables and define a transition relation, while the other nodes define the interaction among their children. We indenty the AltaRica program with the root node of the tree. We define two kinds of nodes, the internal nodes, of type *equipment*, and the leaf node, of type *component*.

Definition 6.5 (Types of AltaRica node). *An AltaRica node $N = \langle E, <_E, S, F, I, T, P, N_1, \ldots, N_n, A, V \rangle$ can be of two different types:*

- Equipment type*: if $S = \emptyset$, $I = True$, $T = True$, and P is such that $P(f) = True$ for each $f \in F_O$.*

- Component type*: if $n = 0$, $A = True$, and $V = \emptyset$.*

The interaction between an AltaRica node and its sub-nodes is defined by the relation over flow variables and the synchronization vectors.

The flow variables represent the configuration a component that are visible by the other components. In particular, the function P associates an expression over the state variables S to a specific flow variable $f \in F_0$. They directly map to an *internal assert* in the concrete syntax. Moreover, the interaction between the flow variables of a component and of its subnodes can be constrained by the invariant section A. The invariant section A represents the *in-out assert*, the *in-in assert* and the *out-out assert* of the concrete AltaRica syntax.

The synchronization vectors model a message-passing style of communication, allowing complex patterns like broadcasting messages. For instance, it is

possible to model the response to a failure in the system. Suppose that, in response to a failure, a controller broadcasts to all the engines the message "*turn_off*". The communication must be non-blocking to model also the failure of the engines (i.e. when the engine cannot be turned off). Suppose the system is composed by 4 components, a controller, an observer and two engines. Then, the synchronization vector $\langle$ *failure_sync*, *failure*, ε, *turn_off?*, *turn_off?* $\rangle$ represents a set of several synchronization instances: {$\langle$ *failure_sync*, *failure*, ε, *turn_off*, *turn_off* $\rangle$, $\langle$ *failure_sync*, *failure*, ε, ε, *turn_off* $\rangle$, $\langle$ *failure_sync*, *failure*, ε, *turn_off*, ε $\rangle$, $\langle$ *failure_sync*, *failure*, ε, ε, ε $\rangle$} (recall that the first event in the tuple is the event of the parent ITS node). The set represents *all* the possible actions performed by the system. The question mark appended to an event in the synchronization vector specifies that the event is non-blocking, thus allowing different instances of synchronization (e.g., the question mark models the fact that an event "can happen", but this is not mandatory). In the following, we use the notation $Inst(v)$ to denote all the instances of a synchronization vector v.

The synchronization vector can represent the three types of synchronization of the concrete syntax of AltaRica. For example, the synchronization vector $\{\langle failure\_sync, failure, \varepsilon, turn\_off, turn\_off\rangle$ represent a strong synchronization on the event $failure\_sync$ of the parent ITS (in the synchronization the first component moves on $failure$, the second moves on stutter, the third and the fourth move on $turn\_off$). A weak synchronization is expressed appending the question mark ? to an event. For example $\langle failure\_sync, failure, \varepsilon, turn\_off?, turn\_off?\rangle$ is a weak synchronization. Finally, a CCF synchronization can be represented with several synchronization vectors. One vector defines a weak synchronization (i.e. if one of its instances fire, it means that a synchronization happened), while the other vectors describe the movement of single components with their own event (i.e. it means that no synchronization happened). For example, the following vectors represent a CCF synchronization: $\{\langle failure\_sync, failure?, \varepsilon, turn\_off?, turn\_off?\rangle$ and $\{\langle failure, failure, \varepsilon, \varepsilon, \varepsilon\rangle, \langle turn\_off, \varepsilon, \varepsilon, turn\_off, \varepsilon\rangle, \langle turn\_off, \varepsilon, \varepsilon, \varepsilon, turn\_off\rangle\}$.

Definition 6.6 (Instances of a synchronization vector). *Given a synchronization vector $v = \langle e_0, \ldots, e_n\rangle$ of an AltaRica node A, $u = \langle u_0, \ldots, u_n\rangle \in Inst(V)$ if for all $1 \leq i \leq n$ it holds that: (i) if $e_i \in E_i$, then $u_i = e_i$; (ii) if $e_i = b?, b \in E_i \setminus \{\varepsilon\}$, then $u_i \in \{b, \varepsilon\}$.*

6.2.2. Semantics

The semantics of an AltaRica program is obtained by applying recursively the definition of the semantics for each subnode of the tree, starting from the root.

Thus, we provide the semantics for the two types of AltaRica nodes.

Given an AltaRica node $N = \langle E, <_E, S, F, I, T, P, N_1, \ldots, N_n, A, V \rangle$, let $ITS_N = \langle E', O', C', I', \pi', T' \rangle$ be the ITS that defines the semantics of N. We define the ITS_N of the root node A by recursively defining the ITS of all its subcomponents in the tree (e.g., similarly to a post-order visit).

ITS of a component node. If N is of type component then ITS_N is defined as:

- $E' = E$;
- $O' = \sigma(F)$;
- $C' = \{c = \langle s, f \rangle \mid s \in \sigma(S), f \in \sigma(F_O), s \models P(f)\}$;
- $I' = \{c \in C' \mid c \models I(S)\}$;
- $\pi' : C' \to O'$ is such that $\forall c \in C', c = \langle s, f \rangle, \pi'(s) = f$;
- $T' = T'' \restriction <_E$, where $T'' = \{\langle c, e, c' \rangle \in C \times E \times C' \mid \langle c, e, c' \rangle \models T\}$;

The ITS_N has the same events of the AltaRica node. Its configurations are all the possible assignments to the state and output flow variables of N that satisfy the function P (i.e. which intuitively define the value of each flow variable as a set of values of S). The observations are defined by all the possible assignments to the flow variables F, and the initial states by all the configurations that satisfy I. Then, the mapping from a configuration c to an observable is the projection of each configuration $c = \langle s, f \rangle$ on S. Finally, the transition relation T' is represented by the set of tuples composed of a source configuration, an event, and a destination configuration that satisfy T, and later restricted according to the priorities $<_e$.

ITS of an equipment node. If N is of type equipment, we define the ITS_N as follows. First, $\forall 1 \leq i \leq n$, N_i is an AltaRica node and $ITS_i = \langle E'_i, O'_i, C'_i, I'_i, \pi'_i, T'_i \rangle$ is its associated ITS.

We define ITS_N as follows:

- $E' = E$;
- $O' = \sigma(F)$;
- $C' = \{(o, c_1, \ldots, c_n) \in O' \times C'_1 \times \ldots \times C'_n \mid (o, c_1, \ldots, c_n) \models A\}$;
- $I' = \{(o, i_1, \ldots, i_n) \in O' \times I'_1 \times \ldots \times I'_n\}$;

- $\pi' : C' \to O'$ is such that $\pi'(o, c_1, \ldots, c_n) = o$;
- $T' = T''$;

In the case of an equipment node, the set of configurations C' is constructed by the product of its observables and the configurations of its subcomponents. Similarly, this happens for the set of initial states. The mapping π' for a configuration $c = \langle o, c_1, \ldots, c_n \rangle$ is simply o.

Instead, the definition of the transition relation T'' is more convoluted and it is defined in a constructive fashion as follows.

1. This step defines the set T_C, which contains all the transitions obtained considering all the instances of synchronization vectors V.
   Let $V_{\text{local}} = \{v \mid |v| = n, \exists i \in [1, n], e_i \in E_i, \forall j \in [1, n], j \neq i, e_j = \varepsilon\}$ be the set of the local events of a component.
   Let $W = \bigcup_{v \in V \cup V_{local}} Inst(v) \times \{v\}$ be the set of pairs composed by an instance of a synchronization vector v (either from V or V_{local}) and the synchronization vector v.
   Then $T_c \subseteq C' \times W \times C'$ contains all the elements $\langle (c_1, \ldots, c_n), \langle e, e_1, \ldots, e_n, v \rangle, (c'_1, \ldots, c'_n) \rangle$ such that:
   - for all $i = 1, \ldots, n$, $\langle c_i, e_i, c'_i \rangle \in T_i$.
   - for all $c \in C$, $\langle c, (\varepsilon, \ldots, \varepsilon, \vec{\varepsilon}), c \rangle \in T_c$.
2. This step defines the set T_B, that restricts T_C considering only the transitions that maximize the number of components that synchronize.
   Let $u = \langle e, e_0, \ldots, e_n \rangle, u' = u = \langle e', e'_0, \ldots, e'_n \rangle$. We say that $u \sqsubset u'$ iff the number of events different from ε in u is greater than u' (i.e. in u more components move).
   We define the partial order $<_W$ on transition instances $\langle u, v \rangle, \langle u', v' \rangle \in W$: $(u, v) <_W (u', v')$ if and only if $v = v'$ and $u \sqsubset u'$.
   Then, we restrict T_C considering only the transitions that have a maximal number of components that synchronize. Formally, $T_B = T_C \restriction <_W$.
3. We define $T_A \subseteq C' \times E' \times C'$, the transition relation restricted to the events E' of ITS_N. $\langle c, e, c' \rangle \in T_A$ either if:
   - there exists $w = (e, e_1, ..., e_n, v) \in W$ such that $e \in E'$ and $\langle c, w, c' \rangle \in T_B$.
   - $\langle c, e, c' \rangle = \langle c, \varepsilon, c \rangle$, for some $c \in C'$.

4. Finally, we apply the ordering $<'_E$ to T_A, i.e. $T'' = T_A \restriction <_E$.

6.3. The semantics of HyDI

In this section, we introduce the HyDI language, its syntax[1] and semantics.

6.3.1. Syntax

A HyDI network is composed of a set of processes, a set of synchronizations and a set of global constraints over the variables of the processes. Each process M is described with a set VAR of state variables.

Each process contains an enumerated type variable EVENT, which is used as guard for the transitions of the process. A specific value S in the domain of EVENT is used to represent the *stuttering* of the process, while other processes are asynchronously active. When a process performs the action S, its state does not change. The synchronizations and global constraints can range over the variables EVENT of a process, thus forcing stuttering and synchronizations.

Formally, an HyDI process M is a tuple $\langle$PARAM, VAR, IVAR, INIT, TRANS, INVAR$\rangle$ where:

- PARAM is the set of parameters;
- VAR is the set of state variables;
- IVAR is the set of input variables;
- INIT is the initial formula over the variables PARAM $\cup$ VAR;
- TRANS is the transition formula over the variables VAR $\cup$ PARAM $\cup$ IVAR $\cup$ VAR$'$ (note that the parameters can only be read by the process);
- INVAR is the invariant formula over the variables PARAM $\cup$ VAR.

An HyDI network H is a tuple $\langle \mathcal{M}$, VAR, IVAR, INIT, INVAR, TRANS, SYNC$_H$, WEAKSYNC$_H$, MAP$\rangle$ where:

[1] In order to ease the presentation and the description of the translation, we give an abstract version of HyDI without considering modules and their instantiation. The reader may refer to [15] for a complete description of the language.

- $\mathcal{M}$ is a set of processes where, for all $M \in \mathcal{M}$, $M = \langle \text{PARAM}_M, \text{VAR}_M, \text{IVAR}_M, \text{INIT}_M, \text{TRANS}_M, \text{INVAR}_M \rangle$ is a process where $\text{EVENT}_M \in \text{IVAR}_M$ (we denote with E_M the domain of EVENT_M),
- VAR is a set of variables (we identify the set of all the variables of a network with $\text{VAR}_H = \text{VAR} \cup \bigcup_{M \in \mathcal{M}} M.\text{VAR}_M$);
- IVAR is a set of input variables (we identify the set of all the input variables of a network with $\text{IVAR}_H = \text{IVAR} \cup \bigcup_{M \in \mathcal{M}} M.\text{IVAR}_M$);
- INIT is an initial condition over VAR_H; INVAR is the invariant formula over VAR_H.
- TRANS is a transition condition over $\text{VAR}_H \cup \text{IVAR}_H \cup \text{VAR}'_H$;
- SYNC_H is a set of *strong synchronizations*, which contains tuples of the form $\{\langle i, j, a_i, a_j \rangle$, where $i, j \in \mathcal{M}$, $a_i \in E_i$ and $a_j \in E_j$;
- WEAKSYNC_H is a set of *weak synchronizations*, which contains tuples of the form $\langle i, j, a_i, a_j, \phi_i, \phi_j \rangle$, where $i, j \in \mathcal{M}$, $a_i \in E_i$, $a_j \in E_j$ and ϕ_i, ϕ_j are formulas over $\text{VAR}_i \cup \text{IVAR}_i$ and $\text{VAR}_j \cup \text{IVAR}_j$ respectively. ϕ_i and ϕ_j are called *guards* of the synchronization;
- MAP is a binding function that maps every $p \in \text{PARAM}_M$ of every $M \in \mathcal{M}$ into an expression $\text{MAP}(p)$ over VAR_H. Given a formula ϕ, $\text{MAPSUBS}(\phi)$ denotes the formula obtained by replacing every occurrence of p in ϕ with $\text{MAP}(p)$, for all $p \in \text{PARAM}_M$ of every $M \in \mathcal{M}$.

HyDI features two different types of synchronizations. A *strong synchronization* $\{\langle i, j, a_i, a_j \rangle$ enforces two processes i and j to perform a transition labeled with the event a_i and a_j at the same time. A *weak synchronization* $\langle i, j, a_i, a_j, \phi_i, \phi_j \rangle$ forces the processes i and j to perform a transition labeled with a_i and a_j at the same time if both ϕ_i and ϕ_j hold. Otherwise, in the case ϕ_i holds and ϕ_j does not hold, i may move on the event a_i, while j stutters. A similar behavior happens by inverting the roles of i and j. Intuitively, when both the transitions labeled with a_i and a_j are enabled, i and j must synchronize. In the case one of the two processes cannot move on the synchronization event, because its guard is false, the other is not blocked and can move on the synchronization event. More general synchronization policies between the processes can be specified by means of the global constraint TRANS, since it can range over the variables EVENT of the

processes. For example, this feature may be used to encode priorities constraints among the synchronization of the network.

Note that we also allow the modeling of *shared variables* between different instances, with the restriction that only the process which declares a variable can write its value, while the other processes can only read the value of the variable. The restriction is expressed in the fact that the formula TRANS_M of each process M can only change its next state variables VAR′ but not the parameters PARAM.

We remark that in the scope of this paper, the sets VAR and IVAR of the network are always empty and that the set IVAR_M of every process M always contains only EVENT_M.

6.3.2. Semantics

We define the semantics of a HyDI network $H = \langle \mathcal{M}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR}, \text{MAP} \rangle$, where every $M \in \mathcal{M}$ is such that $M = \langle \text{PARAM}_M, \text{VAR}_M, \text{IVAR}_M, \text{INIT}_M, \text{TRANS}_M, \text{INVAR}_M \rangle$.

First, for every $M \in \mathcal{M}$, we define the formula $\text{TRANS}^{\text{S}}_M$ as $\text{TRANS}_M \wedge (\text{EVENT}_M = \text{S} \rightarrow \bigwedge_{v \in \text{VAR}_M} v' = v)$ (i.e., the frame conditions are added for the stuttering event).

Then, we define the semantics of the synchronization constraints SYNC_H and WEAKSYNC_H. Each strong synchronization $\{\langle i, j, a_i, a_j \rangle \in \text{SYNC}_H$ is encoded as follows:

$$\text{SYNC}_{\langle i,j,a_i,a_j \rangle} = ((i.\text{EVENT} = a_i) \leftrightarrow (j.\text{EVENT} = a_j)).$$

Then, each weak synchronization $\langle i, j, a_i, a_j, \phi_i, \phi_j \rangle$ is encoded with the following formula:

$$\begin{aligned} \text{WEAKSYNC}_{\langle i,j,a_i,a_j,\phi_i,\phi_j \rangle} = &((i.\text{EVENT} = a_i \wedge \phi'_j) \rightarrow j.\text{EVENT} = a_j) \wedge \\ &((j.\text{EVENT} = a_j \wedge \phi'_i) \rightarrow i.\text{EVENT} = a_i) \wedge \\ &((i.\text{EVENT} = a_i \wedge \neg\phi'_j) \rightarrow j.\text{EVENT} = \text{S}) \wedge \\ &((j.\text{EVENT} = a_j \wedge \neg\phi'_i) \rightarrow i.\text{EVENT} = \text{S}) \end{aligned}$$

where ϕ'_i (ϕ'_j) are obtained from ϕ_i (ϕ_j) by renaming each variable v with $i.v$ ($j.v$) and substituting each parameter p of M_i (M_j) with $\text{MAP}(p)$.

Since the synchronization constraints are declared between pairs of processes, we define the transitive synchronization relation SYNC^* from SYNC and WEAKSYNC. A tuple $\langle i, j, a_i, a_j \rangle$ is in SYNC^* if $\langle i, j, a_i, a_j \rangle \in \text{SYNC}$ or $\langle i, j, a_i, a_j, \phi_i, \phi_j \rangle \in$

WEAKSYNC, for some ϕ_i, ϕ_j, or there exists a sequence of processes $l_1, l_2, \ldots, l_n$ such that $\langle l_k, l_{k+1}, a_{l_k}, a_{l_{k+1}} \rangle \in \text{SYNC}*$ for $1 \leq k < n$, $i = l_1$ and $j = l_n$.

Example 1. *Consider three processes i, j, k with the synchronization constraints $\langle i, j, a_i, a_j \rangle$ and $\langle j, k, a_j, a_k, \phi_j, \phi_k \rangle$. When the process i synchronizes with the process j on the events a_i and a_j, also the process k may also synchronize with the process j, on the event a_k. Thus, in this example* $\text{SYNC}^* = \{\langle i, j, a_i, a_j \rangle, \langle j, k, a_j, a_k \rangle, \langle i, k, a_i, a_k \rangle\}$. *Note that the encoding of the* WEAKSYNC *synchronization forces stuttering of the process k if j moves on a_j and ϕ_k does not hold.*

The relation of SYNC^* is used to encode the interleaving semantics of the processes:

$$\text{INT} = \bigwedge_{\substack{i,j \in \mathcal{M}, \\ i \neq j}} \Big(\bigwedge_{a_j \in E_j} (j.\text{EVENT} = a_j \rightarrow \bigwedge_{\substack{a_i \in E_i, \\ \langle i,j,a_i,a_j \rangle \notin \text{SYNC}^*}} i.\text{EVENT} = \text{S})\Big)$$

The constraint INT encodes the interleaving of the processes in the network. In practice, INT encodes that when some processes move, all the processes that do not synchronize with them must stutter.

Moreover, we define the constraint NOSTUTTER, which ensures that at least one process does not perform the *stutter* action:

$$\text{NOSTUTTER} = \bigvee_{i \in \mathcal{M}} i.\text{EVENT} \neq \text{S}$$

Now we can define the process associated to the HyDI network H as follows:

$$\begin{aligned} M_H \;=\; & \langle \emptyset, \text{VAR}_H, \text{IVAR}_H, \text{INIT} \wedge \textsc{MapSubs}(\textstyle\bigwedge_{M \in \mathcal{M}} \text{INIT}_M), \\ & \text{TRANS} \wedge \textsc{MapSubs}(\textstyle\bigwedge_{M \in \mathcal{M}} \text{TRANS}^{\text{S}}_M)\rangle \wedge \\ & (\textstyle\bigwedge_{w \in \text{WEAKSYNC}} \text{WEAKSYNC}_w) \wedge (\textstyle\bigwedge_{s \in \text{SYNC}} \text{SYNC}_s) \wedge \text{INT} \wedge \text{NOSTUTTER}, \\ & \text{INVAR} \wedge \textsc{MapSubs}(\textstyle\bigwedge_{M \in \mathcal{M}} \text{INVAR}_M)\rangle. \end{aligned}$$

In the last step we define the semantics of a process M with an ITS. Note that this also defines the semantics of the process M_H associated to a network H. We will call ITS_H the ITS that defines the semantic of M_H, and thus of the HyDI program H. The semantics of a process $M = \langle \text{PARAM}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR} \rangle$ is defined by the ITS $C(M) = \langle E, O, C, I, \pi, T \rangle$ as follows:

- E is the set of assignments to the variable IVAR_M,
- C is the set of assignments to the variables $\text{VAR}_M \cup \text{PARAM}_M$,

- $O = C$ and π is the identity function,
- $I = \{s \in C \mid s \models \text{INIT}_M(\text{VAR}) \wedge \text{INVAR}_M(\text{VAR})\}$,
- $T = \{(s_1, a, s_2) \in C \times E \times C \mid s_1, a, s_2' \models \text{INVAR}_M(\text{VAR}) \wedge \text{TRANS}_M(\text{VAR}, \text{PARAM}, \text{IVAR}, \text{VAR}') \wedge \text{INVAR}_M(\text{VAR}')\}$.

6.4. Translation

6.4.1. Flattening of the AltaRica hierarchy

The first step of the translation from AltaRica to HyDI is the flattening of the AltaRica program. The flattening is necessary since the target language, HyDI, does not allow us to define synchronization constraints across intermediate nodes, since it does not handle hierarchical asynchronous processes. Thus, our translation first performs a flattening of the AltaRica program, moving all the synchronization vectors and the constraints among flow variables in the root node of the program. The flattening procedure takes as input an AltaRica program, which represent a tree of AltaRica nodes, and translates them in a *flattened AltaRica node*.

Definition 6.7 (Flattened AltaRica node). *An AltaRica node* $N = \langle E, <_E, S, F, I, T, P, N_1, \ldots, N_n, A, V\rangle$ *is* flattened *if (i)* N *is of type equipment and (ii) each node* $N_1, \ldots, N_n$ *is of type component.*

We use a "dot notation" to identify the variables of subcomponents during the flattening. For example, when flattening the child node $n1$ of a node n, we will write a variable b of $n1$ as $n1.b$ in the node n. We extend the "dot notation" to sets of variables in the intuitive way (i.e. if V is a set of variables, $b.V = \{b.v \mid v \in V\}$). Let FL define the recursive function that flattens an AltaRica node and FL_E a function that substitute a single equipment node with all its children. We will first define FL and then FL_E. We will use the predicates $\text{IsComp}(N)$ (resp. $\text{IsEquip}(N)$) that are true iff N is a node of component (resp. equipment type). We will write $N[N_i/N_j]$ to denote a node which is the copy of N where the child N_j was substituted with the component N_i. Also, let $\text{Children}(N)$ be the list of children of the node N.

$$\text{FL}(N) := \begin{cases} N & \text{if } \forall N_i \in \text{Children}(N), \text{IsComp}(N_i) \\ \text{FL}(N[\text{FL}(N_j)/N_j]) & \text{if } \exists N_j \in \text{Children}(N), \\ & \quad \exists N_l \in \text{Children}(N_j), \text{IsEquip}(N_l) \\ \text{FL}(\text{FL}_E(N)) & \text{if } \exists N_j \in \text{Children}(N), \\ & \quad \forall N_l \in \text{Children}(N_j), \text{IsComp}(N_l) \end{cases}$$

Intuitively, in the first case of $\text{FL}(N)$ N is a flattened node. In the second case, N has at least one child (N_j) that has as child an equipment node (N_l). Thus, FL must be applied recursively to get rid of the nested equipment node N_l. Finally, in the third case all the children of N_j, a child of N, are components node. Thus, N_j is "merged" with N using the function FL_E, that moves all its children in N.

Now, we define the flattening $\text{FL}_E(N)$.

Let $N = \langle E_N, <_{E_N}, S_N, F_N, I_N, T_N, P_N, N_1, \ldots, N_n, A_N, V_N \rangle$ be an AltaRica node such that:

- $\text{IsEquip}(N)$.
- $\text{IsEquip}(N_1)$ (without loss of generality, to simplify the presentation we assume that the first node is of type equipment).
- $\forall C_i \in \text{Children}(N_1)$, $\text{IsComp}(C_i)$.

We will refer to the children of N_1 as $C_1, \ldots, C_k$.

Then, $N' = \text{FL}_E(N)$, with $N' = \langle E_{N'}, <_{E_{N'}}, S_{N'}, F_{N'}, I_{N'}, T_{N'}, P_{N'}, C_1, \ldots, C_k, N_2, \ldots, N_n, A_{N'}, V_{N'} \rangle$, be defined as:

- $E_{N'} = E_N$;
- $<_{E_{N'}} = <_{E_N}$;
- $S_{N'} = \top$;
- $F_{N'} = F_N \cup N_1.F_{N_1}$;
- $I_{N'} = \top$;
- $T_{N'} = \top$;
- $P_{N'}$ is such that $P(f) = \top$ for each f;
- $A_{N'} = A_N \wedge A_{N_1}[N_1.F_{N_1}/F_{N_1}, C_1.F_{C_1}/F_{C_1}, \ldots, C_k.F_{C_k}/F_{C_k}]$ (where $\phi[N_1.V/V]$ denotes the formula ϕ where all the occurrences of variables $v \in V$ have been substituted with $N_1.v$).
- The children of N' are: $C_1, \ldots, C_k, N_2, \ldots, N_n$.

- $V' := V_{\text{exp}} \cup V_{\text{LOCAL}}$, where

$$V_{\text{exp}} := \bigcup_{v \in V_N} \text{exp}(v)$$

$$\text{exp}(\langle e, e_1, \ldots, e_n \rangle) := \begin{cases} \text{exp}'(\langle e, e_1, \ldots, e_n \rangle) & \text{if } \exists l_1, \ldots, l_k, \langle e_1, l_1, \ldots, l_k \rangle \in V_{N_1} \\ \{\langle e, \varepsilon, \ldots, \varepsilon, e_2, \ldots, e_n \rangle\} & \text{otherwise} \end{cases}$$

$$\text{exp}'(\langle e, e_1, \ldots, e_n \rangle) := \{\langle e, l_1, \ldots, l_k, e_2, \ldots, e_n \rangle \mid \langle e_1, l_1, \ldots, l_k \rangle \in V_{N_1}\}$$

$$V_{\text{LOCAL}} := \{\langle \varepsilon, l_1, \ldots, l_k, \varepsilon, \ldots, \varepsilon \rangle \mid \langle e, l_1, \ldots, l_k \rangle \in V_{N_1}, e \notin V\}$$

  The set of synchronization vectors V_{exp} is obtained from the synchronization vectors in V_N, "expanding" all the synchronization vector of V_N that contains an event e_1 with all the synchronization vectors from V_{N_1} that contain e_1. Then, in V_{LOCAL} there are all the synchronizations between the components $C_1, \ldots, C_n$ defined in N_1 which do not synchronize with an external event of N.

6.4.2. Translation from a flattened AltaRica node to a HyDI network

Let $N = \langle E, <_E, \emptyset, F, \top, \top, P, N_1, \ldots, N_n, A, V \rangle$ be a flattened AltaRica node. We translate N in the HyDI network $\langle \mathcal{M}, \text{VAR}, \text{IVAR}, \text{INIT}, \text{TRANS}, \text{INVAR}, \text{MAP} \rangle$.

Intuitively, we translate each component N_i of the flattened node into a process $M_i \in \mathcal{M}$ of the HyDI network. We define VAR $:= \emptyset$, IVAR $:= \emptyset$, INIT $:= \top$, INVAR $:= A$ and MAP $:= P$. TRANS is used to encode the synchronization vector V. In the following, we give the details on the definition of the processes M_i and the condition TRANS.

Given a synchronization vector $v = \langle e_0, \ldots, e_n \rangle$ we write $v[i+1]$ to refer to the i-th element e_i of v.

Let $N_i = \langle E_i, <_{E_i}, S_i, F_i, I_i, T_i, P_i, \top, \emptyset \rangle$ where $F_i = F_{O_i} \cup F_{I_i}$. We translate N_i into the process $M_i = \langle \text{PARAM}_i, \text{VAR}_i, \text{IVAR}_i, \text{INIT}_i, \text{TRANS}_i, \text{INVAR}_i \rangle$ as follows:

- $\text{PARAM}_i := F_{I_i}$;
- $\text{VAR}_i := S_i \cup F_{O_i}$;
- $\text{IVAR}_i := \{\text{EVENT}_i\}$, where the domain of EVENT_i is $\{v.e \mid v[i+1] = e \text{ or } v[i+1] = e? \text{ for some } v \in V\}$;

- $\text{INIT}_i = I_i$;
- TRANS_i is obtained from T_i by replacing, for all $e \in E$, every occurrence of $\Gamma_i = e$ with $\bigvee_{v \in V, v[i+1]=e \text{ or } v[i+1]=e?} \text{EVENT}_i = v.e$.
  In practice, TRANS_i is obtained renaming the event variables and the names of the events in the transition T_i;
- $\text{INVAR}_i := \bigwedge_{f \in F_{O_i}} f = P(f)$.

We translate each synchronization $v = \langle e'_o, e'_1, \ldots, e'_n \rangle, v \in V$, into the SYNC and WEAKSYNC conditions as follows:

- If $v = \langle \varepsilon, \ldots, \varepsilon \rangle$, then we do not declare any synchronization. The silent synchronization is implicit in the HyDI semantics (i.e. the HyDI network allows us to perform a transition where all the instances stutter).
- If $v = \langle \varepsilon, \ldots, \varepsilon, e_i, \varepsilon, \ldots, \varepsilon \rangle$, then we do not declare any synchronization, since transitions labeled with $v.e_i$ may be performed by the i-th process without synchronizing with the other processes.
- Otherwise, let $\text{Idx}_{\text{Weak}}, \text{Idx}_{\text{Sync}}, \text{Idx}_{\varepsilon}$ be three disjoint sets of indexes such that:
  - $\text{Idx}_{\text{Weak}} \cup \text{Idx}_{\text{Sync}} \cup \text{Idx}_{\varepsilon} = \{1, \ldots, n\}$, $\text{Idx}_{\text{Weak}} \cap \text{Idx}_{\text{Sync}} = \emptyset$, $\text{Idx}_{\text{Weak}} \cap \text{Idx}_{\varepsilon} = \emptyset$ and $\text{Idx}_{\text{Sync}} \cap \text{Idx}_{\varepsilon} = \emptyset$,
  - for each $k \in \text{Idx}_{\text{Weak}}$, $e'_j = e_j?$ and $e_j \in E_j$,
  - for each $j \in \text{Idx}_{\text{Sync}}$, $e'_j = e_j$ and $e_j \in E_j$,
  - for each $l \in \text{Idx}_{\varepsilon}$, $e'_l = \varepsilon$,

  then we have that:
  - for all $j_1, j_2 \in \text{Idx}_{\text{Sync}}, j_1 \neq j_2$, $\langle M_{j_1}, M_{j_2}, v.e_{j_1}, v.e_{j_2} \rangle \in$ SYNC.
  - for all $k_1, k_2 \in \text{Idx}_{\text{Weak}}, k_1 \neq k_2$, $\langle M_{k_1}, M_{k_2}, v.e_{k_1}, v.e_{k_2}, \phi_{k_1}, \phi_{k_2} \rangle \in$ WEAKSYNC, where:

    $$\phi_{k_1} = \exists \Gamma_{k_1}, \exists S'_{k_1}, T_{k_1} \wedge \Gamma_{k_1} = e_{k_1}$$
    $$\phi_{k_2} = \exists \Gamma_{k_2}, \exists S'_{k_2}, T_{k_2} \wedge \Gamma_{k_2} = e_{k_2}$$

    ϕ_{k_1} (resp. ϕ_{k_2}) represents the set of states where the node N_{k_1} (resp. N_{k_2}) performs a transition on the event e_{k_1} (resp. e_{k_2}). Recall that S'_{k_1} denotes the set of next state variables of the AltaRica node N_{k_1} (and similarly for S'_{k_2}).

– for all $k_1 \in \text{Idx}_{\text{Weak}}, j_1 \in \text{Idx}_{\text{Sync}}$, $\langle M_{k_1}, M_{j_1}, v.e_{k_1}, v.e_{j_2}, \phi_{k_1}, \top \rangle \in$ WEAKSYNC, where $\phi_{k_1} = \exists \Gamma_{k_1}, \exists S'_{k_1}, T_{k_1} \wedge \Gamma_{k_1} = e_{k_1}$.

Note that in practice we do not perform a real quantification to compute the guards $\phi_{k_1} = \exists \Gamma_{k_1}, \exists S'_{k_1}, T_{k_1} \wedge \Gamma_{k_1} = e_{k_1}$ (and analogously for ϕ_{k_2}) but we exploit the syntax of the concrete AltaRica language. In fact, in the concrete syntax of AltaRica the transition are already disjunctively partitioned by the events that label them.

6.5. Correctness

The translation of a flattened AltaRica node N into the HyDI network H is correct, since the ITSs associated to N and H are equal (modulo an isomorphism).

Theorem 1. *Let N be an AltaRica node and H be its translation to HyDI. Then, the ITS $ITS_N = \langle E_N, O_N, C_N, I_N, \pi_N, T_N \rangle$ of N and the ITS $ITS_H \langle E_H, O_H, C_H, I_H, \pi_H, T_H \rangle$ of H are equal.*

The informal arguments of the correctness are simple. First, we have a bidirectional mapping between each configuration of ITS_N and ITS_H. Thus, the sets of states that satisfy the initial and invariant configuration of both the ITSs are the same. Then, we have that there is a bijection between the transitions in ITS_N and ITS_H, and thus both the transition systems perform the same transitions. The complete proof can be found in Appendix A.

7. Tool Integration and Functionalities

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

7.1. 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 only allows comparing a variable with a value. For example, in the AltaRica model presented in Figure 1, OCAS needs an observer 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;

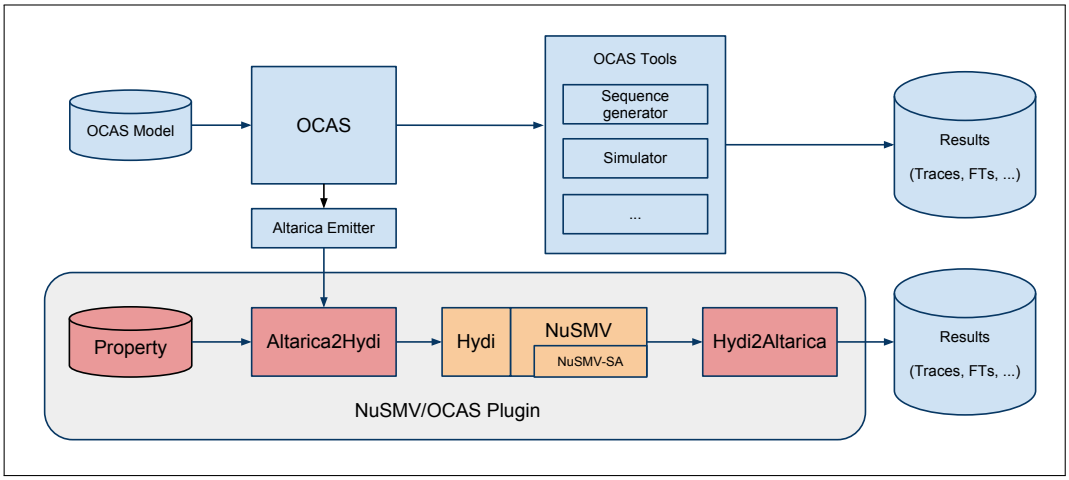


Figure 6: The NuSMV/OCAS plugin and its integration into OCAS

- *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 parser generator [33];

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, which 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

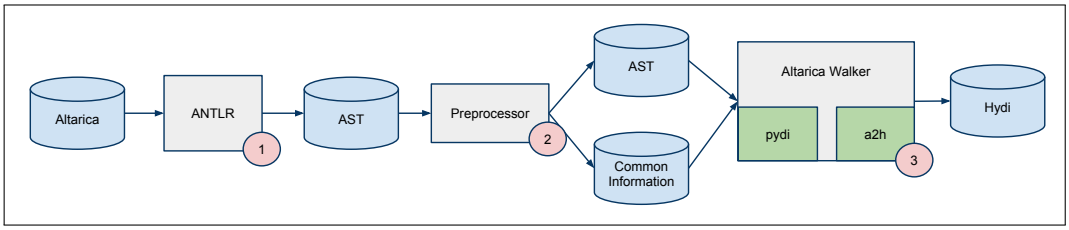


Figure 7: The *AltaRica2HyDI* component

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).

7.2. Functionalities

The NuSMV/OCAS plugin relies on NuSMV, which provides standard BDD-based (for CTL and LTL) model checking techniques [34], and SAT-based BMC (for LTL) techniques [18]. 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 [35, 8]. 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, (dynamic) FMEA, fault tolerance evaluation, and criticality analysis. More specifically, in this context the FTA is strongly related to the computation of cutsets [27], which are an abstraction of the counter examples that prove the reachability of an undesirable behavior. The conditions "the breaking system is not responding" and "the electrical subsystem does not provide a minimum level of power" are some examples of undesirable behaviors, and the model checking based Fault Tree Analysis consists of generating all possible system executions that cause such conditions. Hopefully, the reachability of such behaviors demands the triggering of at least one component's failure, because this is the only reasonable motivation to reach such a hazardous conditions.

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 and called BDD+SAT, that combines BDD and SAT. It first uses BMC techniques, up to a given depth, in order to find as much cutsets as possible (i.e., BMC is not complete). Then a BDD-based model checking algorithm performs an exhaustive analysis, starting from the results obtained with BMC. For instance, the BMC-based analysis (until a specific depth k) can find the cutsets $f_1 \wedge f_2$ and $f_2 \wedge f_3$. Then, in the next step, the algorithm runs the BDD-based analysis to find all the cutsets that are different from $f_1 \wedge f_2$ and $f_2 \wedge f_3$. The BDD+SAT approach performs well than the pure BDD approach since the BMC-based analysis is usually faster than the BDD-based one to find counterexamples, and hence cutsets. Then, these cutsets are used in the BDD-based analysis to prune the search space. The performance of the BDD+SAT approach strongly depends on the total number of cutsets that are discovered in the first phase. In fact, the approach is particularly efficient when the BDD analysis is only used to prove the completeness of the results and does not have to find other cutsets.

8. Experimental Evaluation

8.1. Validation of the translation

The semantics of the language used in OCAS extends the AltaRica Dataflow dialect by extending it with the *extern* section, as described in 3. However, such extended specifications are not fully documented, neither considering the formal semantics. Due to this fact, before starting the 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, in addition to some realistic case studies developed within the MISSA project.

The validation of the tool was done using the possibility offered by OCAS to re-execute a simulation trace on the AltaRica model, using its internal trace simulator. The OCAS tool manual [36] guarantees that the Sequence Generator and the trace simulator use the same semantics. Moreover, our confidence on the correct behavior of both tools is high, since the trace simulator and the Sequence Generator have been extensively tested by Dassault Aviation and, in all our experiments, we did not find any discrepancy in the results provided by the two tools.

In the validation process 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):

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 the OCAS semantics. In a few cases, the behavior shown by OCAS was found to be misleading by the users, but this work allowed us to define a formal interpretation of the OCAS behavior and reach a better understanding of the whole verification platform.

8.2. 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. In particular, both tools are able to peform Fault Tree Analysis (i.e. minimal cut-set generation). The sequence generator of OCAS is able to perform Fault Tree Analysis (generation of minimal cutsets) up to a bounded depth (at most, 9). We compared this feature with the Fault Tree Analysis provided by NuSMV-SA that relies on BDD and the mixed BDD+SAT approaches. Thus, while OCAS can only find the minimal cut-set up to a fixed depth, NuSMV-SA also guarantees the completeness of the results. We also used the NuSMV/OCAS plugin to verify temporal properties of the AltaRica design; as these analysis are not available in OCAS, we can not provide a performance comparison for these functionalities.

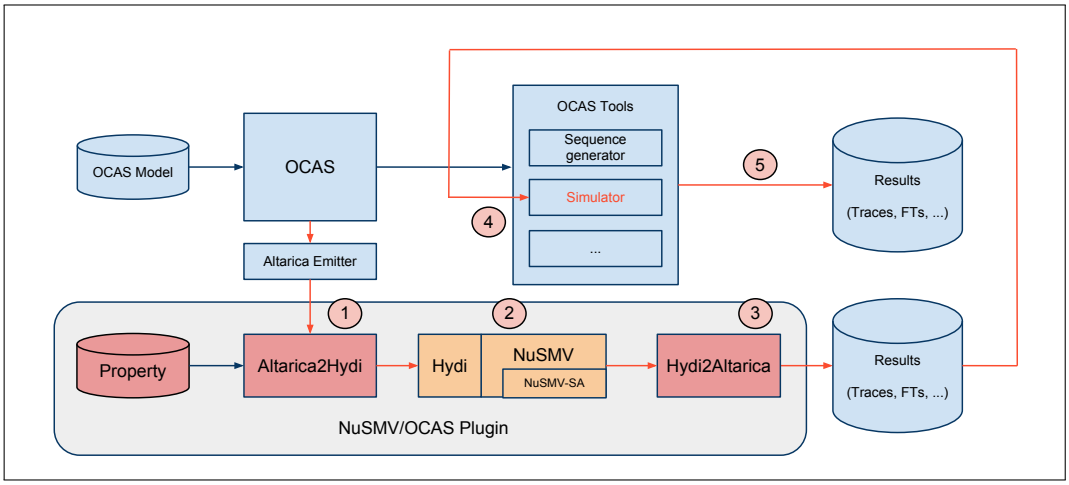


Figure 8: Trace-based validation

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 characteristics of the models are reported in Table 1. The table also shows the time 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).
In the experimental evaluation, both the OCAS Sequence Generator and NuSMV-SA were used to generate the minimal cutset for the same top level event in the same model. In particular, the top level event used for the all the ELEC models was "Loss of power capability", while for the BRSYS model was "Loss of deceleration capability during landing".

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 setup OCAS with default configurations, while NuSMV-SA performs the analysis with FTA algorithms proposed in [35]. We first compared the performances of the OCAS Sequence Generator and NuSMV-SA on the computation of the minimal cutset. The results are presented in Table 2. The Sequence Generator reached the timeout for all models except ELEC_1, and in Table 2 we reported the maximum search depth explored with the corresponding time. For ELEC_1, the Sequence Generator reached the maximum allowed depth

	# States	# Nodes	Translation time	Translation memory
ELEC_1	$1.5 * 10^5$	41	1.127s	27MB
ELEC_2	$2.7 * 10^5$	44	2.782s	38MB
ELEC_3	$2.0 * 10^7$	51	2.811s	37MB
BRSYS	$3.8 * 10^{25}$	135	9.820s	69MB

Table 1: Characteristics of the industrial case studies and translation requirements ("# States" shows the total number of states of the system computed with NuSMV-SA while "# Nodes" reports the number of AltaRica node instances composing the model).

(that is, 9). On the right hand side, we report the performance of NuSMV-SA using BDD and BDD+SAT algorithms. We notice that NuSMV-SA never time-outs, 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-SA performs an exhaustive search, whereas the Sequence Generator is incomplete, that is, it is not guaranteed to find all the cutsets. Concerning memory, NuSMV-SA 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.

The experimental evaluation on the Fault Tree Analysis task is of high interest: it shows that NuSMV-SA performs better than OCAS, even if OCAS is not able to provide a complete answer to the problem.

We also evaluated the scalability of the NuSMV/OCAS plugin, as shown in Figure 9. We notice that NuSMV-SA has a behavior that is nearly independent of the depth of the verification. We remark that the 'step' behavior that is visible in some BDD+SAT plots is due to the fact that for higher depths, SAT may be able to find additional results, which are used to prune the search space before the BDD exploration is run. Differently, when SAT approach is able to find all the minimal cutsets (until the specified depth) the usage of BDD is only an overhead. In effect, in the ELEC_1 test case, pure BDD technique performs better than BDD+SAT.

9. 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 techniques. We have formally proved the correctness of the encoding. Finally, we have integrated the encoder as a

	OCAS		NuSMV-SA	
	SG	**Search Depth**	**BDD+SAT**	**BDD**
ELEC_1	2.4s	9	1.5s	0.9s
ELEC_2	653.7s	7	94.7s	189.2s
ELEC_3	97.7s	6	95.6s	120.2s
BRSYS	16.4s	3	22.36s	771.7s

Table 2: Sequence Generator and NuSMV-SA: FTA performance comparison

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.

The activity gives several insights. First, formal verification technologies can be effectively integrated with a mature and independently-developed modelling language and environment, and can provide significant advances in terms of functionalities and scalability. Second, in order to obtain a well founded solution, a clear understanding and formalization of the source design language must be carried out. A careful choice of the target language is also extremely important: in particular, mapping AltaRica to HyDI, that is inherently able to deal with asynchronous composition, turned out to be a much better choice than mapping AltaRica to the synchronous language of NuSMV.

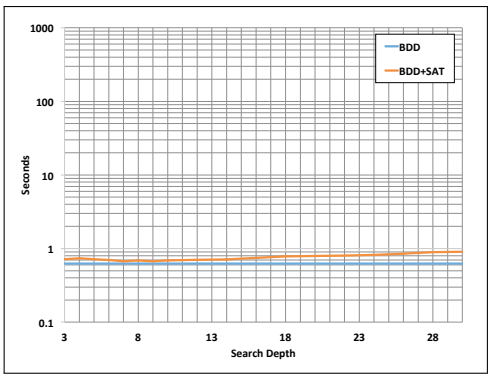
As part of our future work, we plan to analyze the effectiveness of symbolic techniques specialized to asynchronous systems [37, 38]. Finally, we plan to investigate a timed extension of AltaRica, along the lines of [39]. This extension fits very naturally in our framework, given that the HyDI language provides a native support for encoding networks of timed (more in general, hybrid) systems, and can be analyzed by means of symbolic techniques [40].

References

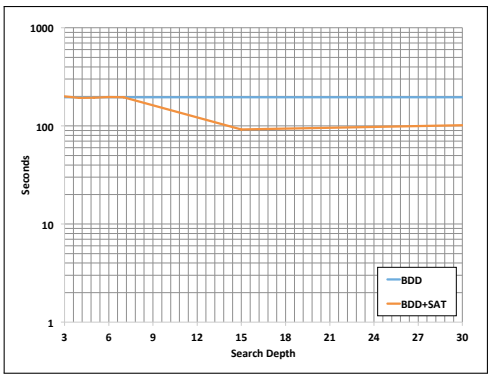
[1] W. Vesely, F. Goldberg, N. Roberts, D. Haasl, Fault tree handbook, Tech. Rep. NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission (1981).

[2] W. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick III, J. Railsback, Fault Tree Handbook with Aerospace Applications, Tech. rep., NASA (2002).

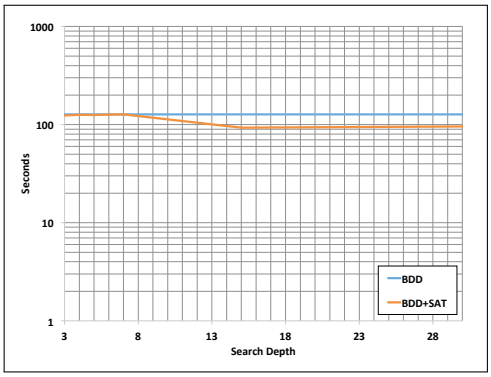
[3] O. Lisagor, T. Kelly, R. Niu, Model-based safety assessment: Review of the discipline and its challenges, in: ICRMS, 2011, pp. 625–632.



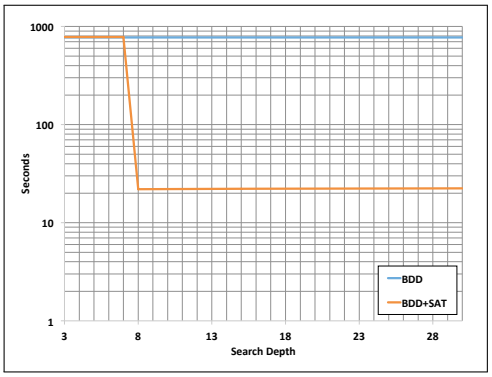
(a) ELEC_1



(b) ELEC_2



(c) ELEC_3



(d) BRSYS

Figure 9: Scalability of BDD and BDD+SAT technology

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

[5] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, A. Griffault, et al., ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects, Proc. ERTS 2006 (2006) 1–11.

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

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

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

[9] 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.

[10] The AltaRica language. `http://altarica.labri.fr/forge`.

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

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

[13] P. Bieber, C. Bougnol, C. Castel, J.-P. Christophe Kehren, S. Metge, C. Seguin, Safety assessment with AltaRica, in: Building the Information Society, Vol. 156 of IFIP International Federation for Information Processing, Springer, 2004, pp. 505–510.

[14] L. A. Johnson, Do-178b, software considerations in airborne systems and equipment certification, Crosstalk, October.

[15] A. Cimatti, S. Mover, S. Tonetta, HyDI: A Language for Symbolic Hybrid Systems with Discrete Interaction, in: EUROMICRO-SEAA, 2011, pp. 275–278.

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

[17] R. E. Bryant, Symbolic boolean manipulation with ordered binary decision diagrams, ACM Computing Surveys 24 (3) (1992) 293–318.

[18] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: Proc. TACAS, Vol. 1579 of LNCS, Springer, 1999, pp. 193–207.

[19] The MISSA Project, `http://www.missa-fp7.eu`.

[20] P. Fenelon, J. A. McDermid, An integrated tool set for software safety analysis, Journal of Systems and Software 21 (3) (1993) 279–290.

[21] M. Wallace, Modular architectural representation and analysis of fault propagation and transformation, Electronic Notes in Theoretical Computer Science 141 (3) (2005) 53–71.

[22] Y. Papadopoulos, J. McDermid, R. Sasse, G. Heiner, Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure, Reliability Engineering & System Safety 71 (3) (2001) 229–247.

[23] P. H. Feiler, B. A. Lewis, S. Vestal, The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems, in: Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE, IEEE, 2006, pp. 1206–1211.

[24] 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) (2006) 747–755.

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

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

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

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

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

[30] A. Griffault, G. Point, A. Vincent, AltaRica-studio : the easier way to do model checking (2011).

[31] 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).

[32] G. D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60-61 (2004) 17–139.

[33] T. J. Parr, R. W. Quong, ANTLR: A predicated-LL (k) parser generator, Software: Practice and Experience 25 (7) (1995) 789–810.

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

[35] M. Bozzano, A. Cimatti, F. Tapparo, Symbolic Fault Tree Analysis for Reactive Systems, in: Proc. Symposium on Automated Technology for Verification and Analysis (ATVA 2007), 2007, pp. 162–176.

[36] OCAS Module System Design and Analysis, User's manual, Dassault Aviation (September 2007).

[37] A. J. Yu, G. Ciardo, G. Lüttgen, Decision-diagram-based techniques for bounded reachability checking of asynchronous systems, STTT 11 (2) (2009) 117–131.

[38] J. Rintanen, K. Heljanko, I. Niemelä, Planning as satisfiability: parallel plans and algorithms for plan search, Artif. Intell. 170 (12-13) (2006) 1031–1080.

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

[40] A. Cimatti, S. Mover, S. Tonetta, Smt-based scenario verification for hybrid systems, Formal Methods in System Design 42 (1) (2013) 46–66.

Appendix A. Proofs

Theorem 1. *Let N be an AltaRica node and H be its translation to HyDI. Then, the ITS $ITS_N = \langle E_N, O_N, C_N, I_N, \pi_N, T_N \rangle$ of N and the ITS $ITS_H \langle E_H, O_H, C_H, I_H, \pi_H, T_H \rangle$ of H are equal.*

Proof. Recall that there exists a bidirectional mapping between each configuration $c \in C_N$ and each tuple $\langle o, c_1, \ldots, c_n \rangle \in O_N \times C_1 \times \ldots \times C_n$. Also, there exists a bidirectional mapping between each state $q \in C_H$ and an assignment μ_q to the variables V_H of the process of the HyDI network.

Thus, there exists a bijection $\xi : C_N \to C_H$ such that, given a state $c \in C_N$, defined from the tuple $\langle o, c_1, \ldots, c_n \rangle \in O \times C_1 \times \ldots \times C_n$, $\xi(c) = q$, where $q = \mu_q(V_H)$ and μ_q is the same assignment determined by $\langle o, c_1, \ldots, c_n \rangle$. Let $\xi^{-1} : Q \to C_N$ be the inverse mapping from Q_H to C_N.

Now we have to prove that the transition relations of ITS_N and ITS_H are equivalent. Formally, it means that $\langle c, e, c' \rangle \in T_N$ iff $\langle \xi(c), e, \xi(c') \rangle \in T_H$.

Let $M_H = \langle \text{PARAM}_H, V_H, W_H, \text{INIT}_H, \text{TRANS}_H, \text{INVAR}_H \rangle$ be the process associated to H. Thus, ITS_H is the ITS associated to M_H.

($\Rightarrow$) We prove that if $\langle s, e_k, s' \rangle \in T_N$, then $\langle q, e, q' \rangle \in T_H$, where $\xi(s) = q$ and $\xi(s') = q'$. Let μ_s be the assignment to the variables of N corresponding to $\langle o, c_1, \ldots, c_n \rangle = \text{S}$. Since $\mu_s \models A_N$ and $\mu_{s'} \models A_N$, also, $\mu_q \models \text{INVAR}_H$ and $\mu_{q'} \models \text{INVAR}_H$. By the definition of N, there exists a tuple of the synchronization vectors $\langle \langle e_0, \ldots, e_n \rangle, v \rangle \in \bigcup_{v \in V} Inst(v) \times \{v\}$ such that $s \overset{\langle e_0, \ldots, e_n \rangle, v}{\rightarrow} s'$. Consider the following cases:

- if $\langle \langle e_0, \ldots, e_n \rangle, v \rangle = \langle \langle \varepsilon, \ldots, \varepsilon \rangle, \varepsilon \rangle$, then there exists a transition $q \overset{a}{\rightarrow} q'$, where $a = \langle M_1.\text{EVENT} = \varepsilon, \ldots, M_n.\text{EVENT} = \varepsilon \rangle$ (Note that the silent event ε cannot be used on arbitrary transitions inside a node, but it labels an implicit self loop).

- Otherwise, let $\text{Idx}_{\text{Weak}}, \text{Idx}_{\text{Sync}}, \text{Idx}_{\varepsilon}$ be three disjoint sets of indexes such that:

  - $\text{Idx}_{\text{Weak}} \cup \text{Idx}_{\text{Sync}} \cup \text{Idx}_{\varepsilon} = \{1, \ldots, n\}$, $\text{Idx}_{\text{Weak}} \cap \text{Idx}_{\text{Sync}} = \emptyset$, $\text{Idx}_{\text{Weak}} \cap \text{Idx}_{\varepsilon} = \emptyset$ and $\text{Idx}_{\text{Sync}} \cap \text{Idx}_{\varepsilon} = \emptyset$,
  - for each $k \in \text{Idx}_{\text{Weak}}$, $e'_j = e_j?$ and $e_j \in E_j$,
  - for each $j \in \text{Idx}_{\text{Sync}}$, $e'_j = e_j$ and $e_j \in E_j$,
  - for each $l \in \text{Idx}_{\varepsilon}$, $e'_l = \varepsilon$,

  then there exists a transition $q \overset{a}{\rightarrow} q'$, where

  $$a = \bigcup_{j \in \text{Idx}_{\text{Sync}}} M_j.\text{EVENT} = e_1 \cup \bigcup_{k \in \text{Idx}_{\text{Weak}}} M_k.\text{EVENT} = e_k \cup \bigcup_{l \in \text{Idx}_{\varepsilon}} M_k.\text{EVENT} = \varepsilon.$$

  Let $\langle e'_0, \ldots, e'_n \rangle$ be the vector of synchronization v. For each $1 \leq i \leq n$:

  - $e_i = \varepsilon$ and $e'_i = \varepsilon$. By the AltaRica semantics, we know that there are no synchronization vectors $v_2 \in V$ such that $\langle e'_0, \ldots, e_i \neq \varepsilon, \ldots, e'_n \rangle$ which can be fired from q. Moreover, the stuttering condition of HyDI forces $i.\text{EVENT} = \varepsilon$, since i does not participate in the synchronization.

– $e_i = e'_i, e_i \neq \varepsilon$. First, $\langle c_i, e_i, c'_i \rangle \models T_H$, thus M_i moves from c_i to c'_i on e_i. Moreover, for each $j \in \text{Idx}_{\text{Sync}}, j \neq i$, M_j moves on $v.e_j$ due to the strong synchronization condition. For each $k \in \text{Idx}_{\text{Weak}}, k \neq i$, by the weak synchronization constraint we have that either $e_k = \varepsilon$, and thus M_k stutters, or M_k synchronizes on $v.e_k$.

($\Leftarrow$) We want to show that, for all $q, q' \in C_H$, for all $e \in E_H$, if $\langle q, e, q' \rangle \in T_H$ then $\langle \xi^{-1}(q), e, \xi^{-1}(q') \rangle \in T_N$. Let $s = \xi^{-1}(q)$ and $s' = \xi^{-1}(q')$. If $\langle q, e, q' \rangle \in T_H$, then, by definition of ITS_H, $\langle q, e, q' \rangle \models \text{TRANS}_H$. In particular, e is an assignment to all event variables EVENT_M of the processes in $\mathcal{M}_H$. We want to show that there exists a corresponding $\langle s, e', s' \rangle \in T_c$. We can have three possible forms of e. 1) $e = \langle \text{S}, \ldots, \text{S} \rangle$; by definition of the AltaRica node, synchronization instance, and the ITS associated to the node, we have that $e \in V_N$, $e \in Inst(e)$, and $e \in T_C$ and trivially $\langle s, e, s' \rangle \in T_B$. 2) $e = \langle \varepsilon, \ldots, e_i, \ldots, \varepsilon \rangle$ and e_i does not occur in V_N; by definition of the ITS associated to an AltaRica node, we have that $\langle s, e, s' \rangle \in T_c$ and trivially $\langle s, e, s' \rangle \in T_B$. 3) $e = v.e_c$ with $e_c \in Inst(v)$ for some $v \in V_N$; by definition of the ITS associated to an AltaRica node, we have that $\langle s, \langle e_c, v \rangle, s' \rangle \in T_c$; if v is a strong sync then trivially $\langle s, e, s' \rangle \in T_B$; if instead v is weak, since $\langle q, e, q' \rangle \models \text{TRANS}_H$, we are guaranteed that $e_i = \text{S}$ only if q does not satisfy the guard of the weak synchronization, and therefore e is the greatest synchronization among the synchronization instances of v; thus, $\langle s, e, s' \rangle \in T_B$. Thus there exists a corresponding transition in T_N.

This concludes our proof.

$\diamond$