View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Open Archive Toulouse Archive Ouverte

Open Archive Toulouse Archive Ouverte (OATAO)

OATAO is an open access repository that collects the work of some Toulouse researchers and makes it freely available over the web where possible.

This is an author's version published in: https://oatao.univ-toulouse.fr/19368

Official URL: http://dx.doi.org/10.1007/978-3-319-64119-5_5

To cite this version :

Brunel, Julien and Feiler, Peter and Hugues, Jérôme and Lewis, Bruce and Prosvirnova, Tatiana and Seguin, Christel and Wrage, Lutz Performing Safety Analyses with AADL and AltaRica. (2017) In: The 5th International Symposium on Model Based Safety Assessment (IMBSA 2017), 11 September 2017 - 13 September 2017 (Trento, Italy).

Any correspondence concerning this service should be sent to the repository administrator:

tech-oatao@listes-diff.inp-toulouse.fr

Performing Safety Analyses with AADL and AltaRica

Julien Brunel[1], Peter Feiler[4], Jérôme Hugues[3], Bruce Lewis[5], Tatiana Prosvirnova[2], Christel Seguin[1], and Lutz Wrage[4]

[1] ONERA, 2 avenue Edouard Belin, 31055 Toulouse
julien.brunel,christel.seguin@onera.fr
[2] IRT Saint-Exupéry, 118 Route de Narbonne, 31432 Toulouse, France
tatiana.prosvirnova@irt-saintexupery.com
[3] ISAE SUPAERO, 10 avenue Edouard Belin, 31055 Toulouse, France
jerome.hugues@isae-supaero.fr
[4] Carnegie Mellon University, Software Engineering Institute, 4500 Fifth Ave, Pittsburgh, PA 15213, USA phf,lwrage@sei.cmu.edu
[5] US Army, AMRDEC, Huntsville, AL 35898, USA bruce.a.lewis.civ@mail.mil

Abstract. AADL and AltaRica languages can be used to support the safety assessments of system architectures. These languages were defined with different concerns and this paper aims at presenting their principles and how they can be related. A translator from AADL to AltaRica is proposed and its prototype is applied to a simplified flight control system of a UAV. The resulting AltaRica model has been analyzed with the AltaRica safety tools and the experimental results are discussed.

Keywords: AADL – AltaRica – MBSA – Safety patterns

1 Introduction

The interest of industrial community in Model-Based System Engineering (MBSE) and Model-Based Safety Assessment (MBSA) is gradually increasing. In this paper we consider two modeling languages: AADL and AltaRica.

AADL (Architecture Analysis and Design Language) is a multi-concerns modeling language dedicated to distributed real-time embedded systems [11]. It proposes several annexes to describe embedded systems behavior. The AADL Error Model V2 (EMV2) [6] is an error annex focused on Safety Analyses. It offers a terminology and an ontology to capture key features of failure/error propagations.

AltaRica [2, 8] is a high level modeling language dedicated to Safety Analyses. Its formal semantics allowed the development of a set of efficient assessment tools, such as compilers to Fault Trees [10, 9] and Markov chains, stochastic and stepwise simulators. It is in the core of several commercially distributed integrated modeling and simulation environments and has been successfully used to perform industrial scale experiments [1].

In this article we study the mapping between AADL EMV2 and AltaRica concepts. To illustrate our purpose, we use as a study case a simplified flight control system of a small UAV quadcopter with a particular focus on safety mitigation architectures, also called safety patterns. The transformation of AADL EMV2 models to AltaRica is interesting because it enables us to enlarge the set of safety assessment tools for AADL and to perform cross check verifications. Indeed, to implement the transformation we use the standardized version of AltaRica Data-Flow, which is a subset of AltaRica 3.0 [8]. AltaRica 3.0 is supported by the OpenAltaRica platform, which is free of use for research and education purposes. This platform already includes a Fault Tree compiler and a stepwise simulator. A stochastic simulator and a sequence generator are currently under development.

The remainder of this article is organized as follows. Section 2 describes a simplified flight control system of a small UAV quadcopter which is used as a running example of this article. Section 3 gives an overview of AADL EMV2 concepts. Section 4 introduces the AltaRica modeling language. Section 5 explains AADL to AltaRica translation principles. Section 6 presents some analysis results of this study. Section 7 summarizes related works. Section 8 concludes this article.

2 Running example

The case study is inspired by the flight control system of a small UAV quadcopter. We address a simplified software architecture that encompasses different significant safety patterns.

High level functional architecture

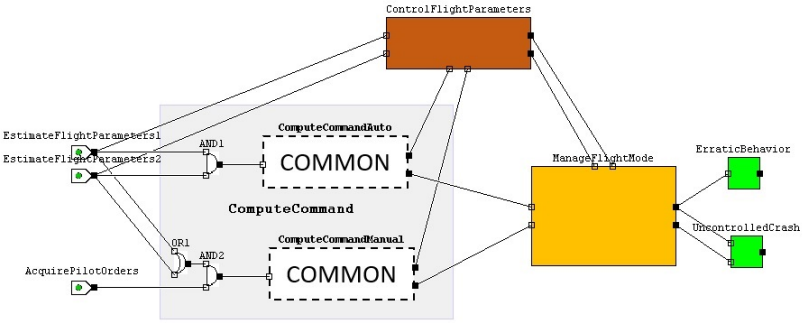

Fig. 1. Overview of the flight control software architecture

The control is achieved by three main functions. The *ComputeCommand* function aims at computing the commands to control the helixes in automated or

manual mode. The function needs two different flight parameters to be performed correctly in the automated mode whereas it needs at least one flight parameter and the pilot order in the manual mode. The quality of the inputs and outputs of this function is checked by the function *ControlFlightParameters*. The function *ManageFlightMode* adapts the piloting mode and the applicable order according to the checked status.

Failure modes and failure conditions

Each function may fail and its computation may be *erroneous* or *lost*. The loss of control is not catastrophic as long as the crash is controlled in an acceptable area. The erroneous control is a worse case because the UAV may fly away and lose the separation with other aircraft or it may crash over populated area. So the system safety assessment is needed to compute the causes and probability of occurrences of such failure conditions.

In this study we consider two failure conditions:

- FC01: "Quadcopter fly away, i.e. erratic behavior, potentially fly and crash in an unauthorized area, leading at worst to fatalities".
- FC02: "Uncontrolled crash, i.e. loss of the quadcopter control".

Safety patterns

Several safety patterns are introduced to reduce the failure occurrences or their effects. A COMMON (COMmand and MONitoring) architecture is proposed to detect and mitigate the effect of potential erroneous computations.

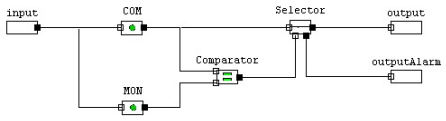

Fig. 2. Command and Monitoring pattern

The pattern, shown Fig. 2, works as follows. We assume that the computation is achieved in parallel by two different channels so that a computation error of one channel may be detected by comparison with the other channel. If no alarm is raised, the order computed by the command lane is applied. Otherwise, the computed order is no more selected and the order is lost.

The mode manager is proposed to adapt as long as possible the control architecture according to the integrity and availability of its basic functions. Initially, the engaged mode is the automated mode. When the check of the flight parameters detects that the automated flight is no longer safe, the manual mode is engaged. A crash mode is engaged when the manual mode is also estimated unsafe.

Modeling and Safety assessment needs
This case study is modeled using the AADL notation and the AltaRica formal language to better understand each notation and the principles of the translation of the AADL model into an AltaRica one. Then the translation is applied to produce an AltaRica model from the AADL one. AltaRica fault tree generator [9] is used for both the hand made AltaRica model and the generated one. The comparison of the Minimal Cut Sets for FC01 and FC02 contributes to the validation of our translation process.

3 AADL EMV2 presentation

The SAE "Architecture Analysis and Design Language" (AADL) [11] is a language for model-based engineering of embedded real-time systems. The AADL allows for the description of both software and hardware parts of a system. It focuses on the definition of clear block interfaces, and separates the implementations from these interfaces. From the separate description of these blocks, one can build an assembly of blocks that represents the full system. To take into account the multiple ways to connect components, the AADL defines different connection patterns: subcomponent, connection, binding.

An AADL description is made of *components*. Each component category describes well-identified elements of the actual architecture, using the same vocabulary of system or software engineering. The AADL standard defines software components (`data`, `thread`, `subprogram`, `process`), execution platform components (`memory`, `bus`, `processor`, `device`, ...) and composite components (`system`, `abstract`). Besides, the language defines precise legality rules for component assemblies, and both its static and execution semantics.

The AADL defines the notion of *properties*. They model non-functional properties that can be attached to model elements (components, connections, features, instances, etc.). Properties are typed attributes that specify constraints or characteristics that apply to the elements of the architecture such as clock frequency of a processor, execution time of a thread, bandwidth of a bus, implementation of the functional part.

From this core of elements, AADL allows the designer to attach annex elements that further refine one dimension of the system. The Error Modeling Annex V2 (EMV2) addresses safety modeling concerns. See [6] for more details.

EMV2 supports architecture fault modeling at three levels of abstraction:

- Focus on error propagation between system components and with the environment: modeling of fault sources, along with their impact through propagation. It allows for safety analysis in the form of hazard identification, fault impact analysis, and stochastic fault analysis.
- Focus on component faults, failure modes, and fault handling: fault occurrences within a component, resulting fault behavior in terms of failure modes, effects on other components, the effect of incoming propagations on the component, and the ability of the component to recover or be repaired.

It allows for modeling of system degradation and fail-stop behavior, specification of redundancy and recovery strategies providing an abstract error behavior specification of a system without requiring the presence of subsystem specifications.

– Focus on compositional abstraction of system error behavior in terms of its subsystems.

In addition, EMV2 introduces the concept of error type to characterize faults, failures and propagations. It includes a set of predefined error types as starting point for systematic identification of different types of error propagations providing an error propagation ontology. Users can adapt and extend this ontology to specific domains.

As an illustration consider an AADL EMV2 model given below. It represents a function with an input and an output. It may be in three states: *s_Ok*, *s_Erroneous* and *s_Lost*, representing respectively the nominal behavior, the erroneous behavior and the loss of the function. If the function is in the state *s_Ok* and it receives an error on its input, it propagates it on the output. If the function is in the state *s_Erroneous* it propagates an error on its output and so on.

```
abstract BasicInOutFunction
  features
    input : in feature;
    output : out feature;
  annex EMV2 {**
    use types FunctionFailureModesLib;
    use behavior FunctionFailureModesLib::ErroneousLostBehavior;
    error propagations
      input  : in  propagation{BasicFunctionFailures};
      output : out propagation{BasicFunctionFailures};
    end propagations;
    component error behavior
      propagations
        s_Ok -[input]-> output;
        s_Erroneous -[]-> output{ERRONEOUS};
        s_Lost -[]-> output{LOST};
    end component;
  **};
end BasicInOutFunction;
```

EMV2 follows regular convention for the description of state transition and error propagation, $\langle initial\_state \rangle - [trigger] \rightarrow \langle error\_event \rangle$, that reads as follows: when the system is in state $\langle initial\_state \rangle$, it propagates the corresponding $\langle error\_event \rangle$. This propagation may be controlled by the *trigger*.

4 AltaRica presentation

AltaRica is a high level formal modeling language dedicated to Safety Analyses [2]. Its Data-Flow version has been created to handle industrial scale models [4]. A number of assessment tools have been developed ([10], [12]). AltaRica Data-Flow is at the core of several Modeling and Simulation tools and has been successfully used for industrial applications [1]. In 2011, an initiative was launched to standardize the syntax of AltaRica Data-Flow.

To implement the transformation of AADL to AltaRica we use the standardized version of AltaRica Data-Flow – a subset of AltaRica 3.0 [8], supported by the OpenAltaRica platform. This platform is developed by IRT SystemX and is free of use for research and education purposes. It already includes a Fault Tree compiler [9] and a stepwise simulator. Other tools are under development.

In this article we only focus on concepts of AltaRica Data-Flow illustrated using the running example. The interested reader can refer to [3] to know more about AltaRica 3.0.

Basic blocks

The following AltaRica code represents the behavior of a basic function.

```
domain BasicFunctionStatus {OK, LOST, ERRONEOUS}
class BasicFunction
  BasicFunctionStatus status (init = OK);
  event fail_loss (delay = exponential(0.001));
  event fail_error (delay = exponential(0.0005));
  transition
    fail_loss: status == OK -> status := LOST;
    fail_error: status == OK -> status := ERRONEOUS;
end
```

States: The internal state of the function is represented by means of the state variable *status*, which takes its value in the domain *BasicFunctionStatus*. So, the function can be in three states: *OK* representing the nominal behavior, *LOST* (loss of the function), and *ERRONEOUS* (erroneous behavior). The initial value of the state variable is specified by the attribute *init*.
Events: The state of the function changes under the occurrence of an event, introduced with the keyword *event*. In our example, the function has two failure events: *fail_error* (erroneous behavior), and *fail_loss* (loss of the function). A delay is associated with each event by means of the attribute *delay*. Delays of the events *fail_loss* and *fail_error* are random exponentially distributed variables.
Transitions: A transition is a triple $e : G \rightarrow P$, where e is an event, G is a Boolean expression, the so-called guard of the transition, P is an instruction, the so-called action of the transition. In the example above if the state of the function is *OK*, then two transitions are fireable: the transition labeled with the event *fail_loss* and the transition labeled with the event *fail_error*. If the delay drawn for the transition *fail_loss* is the shortest, then this transition is fired and the variable *status* is switched to *LOST*.

Flow propagations: In AltaRica the propagation of errors/failures and nominal values is done in the same way: via flow variables and assertions. The value of flow variables are recalculated after each transition firing by means of assertions. Assertions are instructions as are actions of transitions. The difference is that actions of transitions assign only state variables, while assertions assign only flow variables. Consider, for example, the following AltaRica code:

```
class BasicInOutFunction
  extends BasicFunction;
  BasicFunctionStatus input, output (reset = LOST);
  assertion
    output := if (status==OK) then input else status;
end
```

There are two flow variables: *input* and *output*, taking their values in the domain *BasicFunctionStatus*. They represent respectively the quality of the data received and sent by the function. The assertion states that if the state of the function is *OK*, then its output is equal to its input, if its state is *LOST* then its output is also *LOST*, otherwise it is *ERRONEOUS*.

Hierarchical models

In AltaRica Data-Flow components are represented by classes. Classes can be instantiated in other classes in order to create hierarchical models. Their inputs and outputs can be connected via assertions.

The Figure 2 is a graphical view of the class *COMMONPattern*. This class contains two instances of the class *BasicInOutFunction*, named *COM* and *MON*, one instance of the class *Selector* named *selector* and one instance of the class *Comparator* named *comparator*. The *Comparator* (cf code below) and *Selector* are considered as component connectors that are free of failure modes in our case study. Other direct connections between components are represented by plane lines in the figure and by equality assertions linking input and output of connected components in the class *COMMONPattern*.

```
class Comparator
  BasicFunctionStatus input1 (reset=LOST);
  BasicFunctionStatus input2 (reset=LOST);
  Boolean alarm (reset=false);
  assertion
    alarm:= if (input1==input2) then false else true;
end
```

Flight modes and reconfigurations

Flight modes can be represented by a state variable *mode* which takes its value in the domain *FlightModeDomain*. The reconfigurations are represented by immediate transitions, introduced by the attribute delay equal to Dirac(0). They are fireable as soon as their guards are true. As an illustration consider a part of the AltaRica model representing the function *ManageFlightMode* of the running example.

```
domain FlightModeDomain{AUTO, MANUAL, CRASH, DANGER}
class ManageFlightModeFunction
  Boolean inputAlarm, inputCrashAlarm (reset = false);
  FlightModeDomain mode(init = AUTO);
  event GoToManualMode (delay = Dirac(0));
  ...
  transition
    GoToManualMode:
      (mode == AUTO) and inputAlarm and not inputCrashAlarm ->
   mode := MANUAL;
...
end
```

In this model, we define an immediate event *GoToManualMode* and the associated transition, which represents the reconfiguration: while in the automated mode, if the alarm is received (*inputAlarm* is true), then the transition is fired immediately and the mode is switched to manual.

5 AADL EMV2 to AltaRica translation

Models in AADL with EMV2 and AltaRica are structurally similar so that it is possible to translate one notation into the other. On the structural side AADL components correspond to AltaRica classes. AADL subcomponents correspond to AltaRica variables whose type is an AltaRica class. Thus, given an AADL model it is possible to create an AltaRica model that exhibits the same hierarchical containment structure.

For safety analysis we are interested in the occurrence and propagation of faults throughout the analyzed system. The interface of an AADL component with regard to fault propagation is given by its error propagations as defined in the component's EMV2 annex subclause. Each of these EMV2 error propagations corresponds to an AltaRica flow variable whose domain can be derived from the set of faults that are associated with the error propagation.

Several AADL model constructs are involved in the propagation of faults from one AADL component to another. (a) Connections modeling data and control flow can propagate faults related to, for example, data validity and timing. (b) Faults that occur in an execution platform component, e.g. power loss, propagate to software components that are bound to, i.e. propagation via binding properties. (c) Additional propagations without an explicit path in the architectural model are defined in an EMV2 annex subclause, e.g. fault propagation due to heat transfer between hardware components located in close proximity. All these constructs are translated into "external" AltaRica assertions, i.e. assertions connecting flow variables from different classes.

The internal fault behavior of an AADL component is given using the AADL EMV2 *error behavior* and *component error behavior* constructs. The translation of these constructs to AltaRica proceeds as follows:

- An EMV2 *error event* describes the occurrence of an internal fault that happens in the component. Each such event is translated into an AltaRica event. The occurrence probability of an error event is given by the value of property *EMV2::OccurrenceDistribution* which is translated into a delay attribute for the AltaRica event.
- EMV2 *error states* are defined as an identifier, i.e. the state's name. All error states of an AADL component are translated into a single state variable in AltaRica. The domain of this variable is the set of symbols created from the EMV2 error state names.
- EMV2 *state transitions* are translated to AltaRica transitions. If the the transition is caused by an internal fault, i.e., an error event, it is translated into an AltaRica transition that is enabled by the corresponding AltaRica event and uses the source state as the guard. The action assigns the target state to the state variable. If, on the other hand, the transition is caused by incoming fault propagations, we create an AltaRica transition that is always enabled and translate the EMV2 error condition into the guard condition. To enable this kind of transition we add an event with the attribute delay equal to $Dirac(0.0)$ (immediate event) to each AltaRica class.
- EMV2 *out propagation conditions* determine the error type produced at an out propagation based on the state and an error condition involved in propagations. Each out propagation condition is translated to an *internal* AltaRica assertion that sets the value of an outgoing flow variable to the symbol representing the propagated error type.

The following two listings show a simple example in AADL, the comparator component used in the COMMON pattern, and the AltaRica code generated by the translator. It is easy to see the correspondence between the two models.

```
abstract Comparator
  features
    in1: in data port;       -- ports defined in AADL core
    in2: in data port;       -- language
    out0: out data port;
  annex EMV2 {**
    use types FailMode;
    error propagations       -- propagated errors defined in
                             -- EMV2 annex
      in1: in propagation {lost, err};
      in2: in propagation {lost, err};
      out0: out propagation {lost, err};
   end propagations;
   component error behavior  -- no error states needed
     propagations
                             -- propagated error if in1 = in2
       all -[in1{lost} and in2{lost}]-> out0{lost};
       all -[in1{err} and in2{err}]-> out0{err};
```

```
                            -- propagated error if in1 != in2
       all -[in1{noerror} and in2]-> out0{lost};
       all -[in1 and in2{noerror}]-> out0{lost};
       all -[in1{lost} and in2{err}]-> out0{lost};
       all -[in1{err} and in2{lost}]-> out0{lost};
                            -- default: no error is propagated
     end component;
  **};
end Comparator;
```

Note that in EMV2 it is not possible to compare error types with each other. For example, the Comparator class as shown in the code in Section 4 uses the expression *input1 == input2*. Such comparisons must be modeled in EMV2 by using several out propagations that enumerate all possible combinations of error types occurring at the two propagation points.

```
domain domain_4 {      // domain names are generated
  noerror, lost, err   // constants generated from error types
}
domain domain_5 {
  noerror, lost, err
}
class Comparator_6
  event error_propagation (delay = Dirac (0.0)); // not used
  domain_4 out0 (reset = noerror);     // initially no error
  domain_5 in2_3;
  domain_5 in1_4;
  assertion
    out0 := switch {   // all out propagation conditions
                       // aggregated into one switch statement
      case in1_4 == err and in2_3 == lost: lost
      case in1_4 == lost and in2_3 == lost: lost
      case in1_4 == lost and in2_3 == err: lost
      case in1_4 == noerror and in2_3 != noerror: lost
      case in1_4 != noerror and in2_3 == noerror: lost
      case in1_4 == err and in2_3 == err: err
      default: noerror // noerror is the default
    };
end
```

Modeling the comparison of error types as several out propagation conditions results in a somewhat unwieldy AltaRica assertion since the translator does not perform any simplification of the generated code. The assertion is equivalent to

```
    out0 := if (in1_4 == in2_3) then in1_4 else lost;
```

Resolution of mismatches between AADL EMV2 and AltaRica
There are a couple of differences between AADL EMV2 and AltaRica which must be taken into account when translating between the two formalisms.

One difference concerns the way error types are defined in EMV2. In general error types can be thought of as typed tokens that propagate through an architecture. However, EMV2 error types are organized in a generalization hierarchy, and all EMV2 allows use of generalized error types wherever an error type can occur. For example, *AboveRange* is a subtype of *OutOfRange*, which is a subtype of *DetectableValueError*. The most general error in this generalization chain is the *ItemValueError*. AltaRica does not support a notion of generalization in the definition of constants used to create domains. We solve this mismatch by always replacing generalized error types with the most specific error types. For example, a *DetectableValueError* in an error propagation or condition is replaced with the most specific error types of which *DetectableValueError* is a generalization, namely *OutOfBounds*, *BelowRange*, and *AboveRange*.

Another difference is that EMV2 uses sets of error types to define error propagations and conditions, whereas AltaRica does not support a built-in notion of sets. This leads to difficulties when assigning a domain to an incoming flow variable if the flow is generated based on an AADL feature that has multiple incoming connections (fan-in). Such an AADL model is valid if the set of error types F_i at each of the connected out propagations o_i are contained in the type set at the in propagation: $\forall i \in \{1, ..., n\} F_i \subseteq E$. In general all type sets F_i may be different. To resolve this mismatch we generate multiple flow variables from an EMV2 in propagation, one per incoming connection. As the domain we use the domain generated from the error of the corresponding out propagation at the other end of the connection. This way both flow variables have the same domain as their type and can be connected using an AltaRica assertion.

Unfortunately, the 1 to n translation of in propagations to flow variables complicates the translation of error conditions on transitions and out propagation conditions. The error conditions contain atomic terms of the form $ip(C)$, with $C = \{t_1, t_2, ..., t_k\}$ a set of error types. Such a term is true if and only if one of the error types is propagated in via the in propagation ip. In the simplest case this term is translated into the following Boolean expression in AltaRica: $ip = t_1 \vee ip = t_2 \vee ... \vee ip = t_k$. When the in propagation is split into several flow variables, the resulting AltaRica expression is the disjunction of the expressions generated for each of the new flow variables, which can become long and difficult to read.

Another consequence of the 1 to n relationship between in propagations and flow variables is that the number of generated flow variables is not the same for all instances of an AADL classifier. It depends on the context in which the component is used, in particular, how it is connected to other components in the model. Therefore, it is not possible to generate the AltaRica classes based on the declarative AADL model alone. Instead, we translate each AADL component instance into an AltaRica class, potentially resulting in an AltaRica model with several classes that are identical except for identifiers. This increases the size of the AltaRica model but has not much influence on analyses of the models. As an AltaRica model is "flattened" to transform it into a guarded transition system, it is necessary to insert a full copy of a class for each class instance. This

is similar to what happens during the generation of an AADL instance model, thus resulting in essentially the same guarded transition system.

AADL EMV2 also has the concept of a typed error state. This is an error state that has an error type associated with it. The error types allowed for a state are enumerated in a type set given with the error state declaration. Even though we have not included typed error states in the translation, it could be extended to generate multiple AltaRica domain constants for each typed EMV2 error state, one per declared type token.

Implementation of the translator
We have implemented the AADL EMV2 to AltaRica translator as a plugin to OSATE, the open source AADL tool environment. The translation is implemented using the Atlas Transformation Language. The source code for our translator is available on github[6], and the plugin can be installed into OSATE from our p2 repository.[7]

The current version supports a subset of AADL EMV2 sufficient for the analysis of the UAV system used in our case study. We are planning to extend that translation to include more EMV2 concepts as needed, e.g. error detections and stochastic state transitions.

6 Experiments

The automatically generated AltaRica model can be analyzed with the various tools which are provided within the OpenAltaRica platform. In particular, we can simulate the model in order to observe the effect of different failure scenarios on the overall system. We can also analyze the potential causes of the failure conditions by generating Fault Trees and Minimal Cut Sets (MCS).

Let us consider the failure condition FC02 (uncontrolled crash). The analysis returns 12 MCS presented in Fig. 6.

This result fully complies with the result that we obtained with the hand-written AltaRica model of the quadcopter. As a first remark, all the MCS are of order 2. So, we can deduce that no single failure can lead to FC02. As we can see, the pilot order is of prime importance. This is not surprising since one of the main ideas behind this system is to switch from automatic mode to manual mode in case of a problem (loss of a source for the estimation of flight parameters, loss of the automatic command, ...). Once the system switches to manual mode, if the function that acquires the pilot order also fails, then an uncontrolled crash occurs. A similar argument holds for the alarm of the function *ControlFlightParameters*, which is responsible for detecting a problem and switching to manual mode.

The same analysis has been performed for the failure condition FC01. Again, the result is similar to the result obtained from the hand-written model. For this failure condition, there are 2 MCS of order 1, which correspond to the erroneous

[6] at http://github.com/osate/aadl2altarica
[7] at http://aadl.info/aadl/osate/experimental

AcquirePilOrder.loss	CommandAuto.COMMON.COM.error
AcquirePilOrder.loss	CommandAuto.COMMON.COM.loss
AcquirePilOrder.loss	CommandAuto.COMMON.MON.error
AcquirePilOrder.loss	CommandAuto.COMMON.MON.loss
AcquirePilOrder.loss	EstimFlightParam1.loss
AcquirePilOrder.loss	EstimFlightParam2.loss
ContrlFlightParam.Alarm.loss	EstimFlightParam1.loss
ContrlFlightParam.Alarm.loss	EstimFlightParam2.loss
ContrlFlightParam.Alarm.loss	CommandAuto.COMMON.COM.error
ContrlFlightParam.Alarm.loss	CommandAuto.COMMON.COM.loss
ContrlFlightParam.Alarm.loss	CommandAuto.COMMON.MON.error
ContrlFlightParam.Alarm.loss	CommandAuto.COMMON.MON.loss

Fig. 3. Minimal cut sets for FC2: uncontrolled crash

behavior of each of the sources for the estimation of the flight parameters. This was expected since the system does not implement any mitigation for this specific failure (contrary to the loss of one source, or the erroneous behavior of the command computation).

We have obtained the same results for the handwritten and the automatically generated AltaRica models. It is a first validation step for our translator.

7 Related work

Another translation of AADL to AltaRica, based on the definition of ontologies, has been proposed in [7].

Translations of AADL models into classical safety models exist (see e.g. [6]). It is worth noting that each type of translation takes advantage of different features of the AADL model according to the targeted safety models. For instance, the translation into a Fault Tree requires the features *composite error behavior* and *error events*. Such features describe static dependencies between failure modes of components and are well suited for the compilation into Boolean formulas. *Component error behavior* gives a more dynamic view of the failure propagation and is useful for the computation of probability of occurrences of states of Markov chains or stochastic processes. AltaRica has been designed to integrate in one formalism both static and dynamic features of the failure propagation; it supports not only the computation of the probability of occurrences of states but also the computation of sets/sequences of events leading to undesired states. So, the proposed translation should ease the analysis of reconfigurable systems with multiple modes as our flight control system.

A translation of an AADL subset into NuSMV formal language is a pioneer work to achieve more powerful safety analysis in the framework of the COMPASS project [5]. NuSMV language is close to AltaRica Data-Flow: they have similar expressive power. Several safety tools have been developed for NuSMV. However, AltaRica was designed to support safety analysis whereas NuSMV is a more

general language. Specific annotations are introduced in a native NuSMV model to distinguish error concepts that should appear in a safety report. The available tools are effective [1], but the annotation process raises practical questions when mixed with AADL view. First versions of COMPASS built the annotations by asking end users to extend AADL nominal models in the COMPASS framework. It could be clearer for end users to perform the model extension in an AADL framework compliant both with the principle of the AADL Error Annex and the level of rigor requested by NuSMV as experimented in our work.

8 Conclusions

The AADL is a language that is used to describe the software and hardware architectures of system. The AADL EMV2 enables the extension of the architecture models with artefacts that are relevant to generate safety related models. For instance, Fault Trees can be generated from models extended according the principle of the EMV2. AltaRica is a formal language specifically defined to support advanced safety analysis of complex systems that integrate modes and dynamic reconfiguration on specific conditions.

We have presented an example of a simplified version of such a kind of dynamic systems and we have modeled it both in AltaRica and in the AADL as starting point. This first activity has enabled a better understanding of the philosophy of both approaches. Both languages have to deal with failure propagations inside a system architecture. In the AADL, the failure extension of a component is flexible and it enables the connexion of components that are subject to heterogeneous failure modes. As a counterpart, end users need to be quite explicit on the various cases of composition of the failure modes. In AltaRica, the connexion of component shall be compliant with stronger typing rules. So end users can less easily propagate heterogeneous failure modes. As a counterpart, the connexion is straight forward and the propagation between connected components results directly from the language semantics.

Then we have developed a translator from the AADL EMV2 into AltaRica and we have generated an AltaRica model for our case study. The manual AltaRica model and the generated one have been simulated and analyzed. They have produced the same results. This is a first encouraging validation step for our translator. Moreover, the analysis exhibits relevant sequences of causes of the undesired events for a multi-mode system. Thus the AADL model can be analyzed with latest generation of safety assessment tools.

Further works will aim at short term to more widely validate the translator and the benefit of the coupled approach. At longer term, the AADL EMV2 could be updated to ease modeling in the spirit of some interesting AltaRica features. Conversely, the flexibility of the failure type definition in AltaRica could be extended to better account for extension approach like in the AADL EMV2.

Acknowledgements

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0004294

References

1. Akerlund, O., Bieber, P., Boede, E., Bozzano, M., Bretschneider, M., Castel, C., Cavallo, A., Cifaldi, M., Gauthier, J., Griffault, A., Lisagor, O., Luedtke, A., Metge, S., Papadopoulos, C., Peikenkamp, T., Sagaspe, L., Seguin, C., Trivedi, H., Valacca, L.: Isaac, a framework for integrated safety analysis of functional, geometrical and human aspects. In: Proceedings of 3rd European Congress Embedded Real Time Software, ERTS 2006. Toulouse (France) (2006)
2. Arnold, A., Griffault, A., Point, G., Rauzy, A.: The altarica language and its semantics. Fundamenta Informaticae 34, 109–124 (2000)
3. Batteux, M., Prosvirnova, T., Rauzy, A.: AltaRica 3.0 specification. Tech. rep., AltaRica Association (2015), http://openaltarica.fr/docs/AltaRica 3.0 Language Specification.pdf
4. Boiteau, M., Dutuit, Y., Rauzy, A., Signoret, J.P.: The altarica data-flow language in use: Assessment of production availability of a multistates system. Reliability Engineering and System Safety 91, 747–755 (2006)
5. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended aadl models. Comput. J. 54(5), 754–775 (May 2011), http://dx.doi.org/10.1093/comjnl/bxq024
6. Delange, J., Feiler, P.: Architecture fault modeling with the aadl error-model annex. In: 40th Euromicro Conference on Software Engineering and Advanced Applications. IEEE (2014)
7. Mokos, K., Katsaros, P., Bassiliades, N., Vassiliadis, V., Perrotin, M.: Towards compositional safety analysis via semantic representation of component failure behaviour. In: Proceedings of the 2008 Conference on Knowledge-Based Software Engineering. pp. 405–414. Amsterdam, The Netherlands, The Netherlands (2008)
8. Prosvirnova, T., Batteux, M., Brameret, P.A., Cherfi, A., Friedlhuber, T., Roussel, J.M., Rauzy, A.: The altarica 3.0 project for model-based safety assessment. In: Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS 2013. IFAC, York (Great Britain) (September 2013)
9. Prosvirnova, T., Rauzy, A.: Automated generation of minimal cut sets from altarica 3.0 models. IJCCBS 6(1), 50–80 (2015)
10. Rauzy, A.: Mode automata and their compilation into fault trees. Reliability Engineering and System Safety 78, 1–12 (2002)
11. SAE: Architecture Analysis and Design Language (AADL) AS-5506B. Tech. rep., The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 2.1 (September 2012)
12. Teichteil-Knigsbuch, F., Infantes, G., Seguin, C.: Lazy forward-chaining methods for probabilistic model-checking. In: Advances in Safety, Reliability and Risk Management, pp. 318–326. Informa UK Limited (Aug 2011), http://dx.doi.org/10.1201/b11433-47