System to Software Integrity: A Case Study[1]

Matteo Bordin, Cyrille Comar, Ed Falis, Franco Gasperoni, Yannick Moy, Elie Richa, Jérôme Hugues
{bordin, comar, falis, gasperoni, moy, richa}@adacore.com, jerome.hugues@isae.fr

1. Introduction

It is widely acknowledged that the main source of cost for developing high-integrity software systems is their verification. A significant portion of this verification cost is spent assessing that software complies with its requirements.

Over the years several different methods have been developed to address this issue, in particular: testing, peer reviews, formal verification and automatic code generation. It is more and more frequent that these verification strategies are mixed within the same system, so as to adopt the most appropriate one for each component. This increases the complexity of the integration phase because it has to cope with multiple formalisms, development and verification methods.

Our goal is to propose a pragmatic process to integrate components developed using different methods into a single system and demonstrate that properties already verified for each component in isolation are preserved in their composition. This process leverages AADL as a pivotal modeling language for system specification and relies on specific verifications between the latter and the components developed using heterogeneous modeling and programming languages, namely Simulink for computation intensive parts and Ada/SPARK 2014 for other components.

Our paper proceeds as follows. First we provide a high-level overview of our approach and enumerate the current methods for addressing the property preservation problem. Then we illustrate practically our approach using the Nose Gear Challenge problem, a simplified yet complete example of a high-integrity real-time system. We then conclude by comparing our approach to the state of the art.

2. Overview of our approach

The core problem we address in this paper is the assessment of property preservation when integrating heterogeneous components. Regardless of the safety standard of reference, the term “property” indicates a specific refinement level of requirements: properties may be system requirements, software requirements, high-level requirements (HLR), low-level requirements (LLR) and so forth.

Complex systems are built using a mixture of a top-down and bottom-up approach. In a top-down approach, the system is decomposed into atomic components which are then developed in isolation and integrated. In a bottom-up approach, existing components are adapted to be reused and integrated in a complete system.

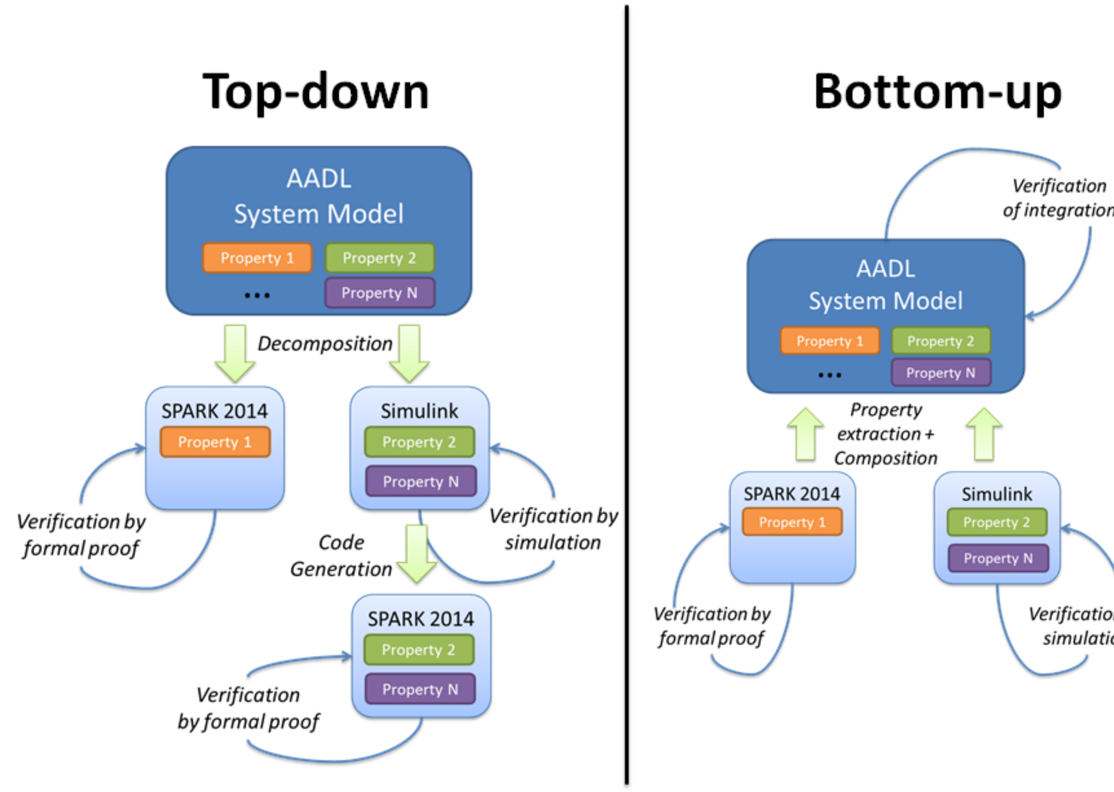
When adopting a top-down approach, properties can be violated when the implementation does not correctly implement its specification. For example, a Simulink model may not guarantee the computational precision required by system specifications. To avoid this problem, we propose to represent the system architecture and all its constraints in the form of AADL models entities, completed with annex language such as REAL or BLESS[2]. The system is decomposed into atomic components and each component is developed using the most appropriate technology, for example Simulink or SPARK 2014. At this point it is fundamental to translate downwards the system constraints in the implementation language. For example, REAL

[1] The work described in this paper has been funded via the FUI 11 project “Project P”, FUI 10 project “Hi-Lite” and Artemis project “Safecer”.
[2] REAL and BLESS are two AADL annexes for expressing structural constraints on architectural models, these will be merged in one annex language called ACL: AADL Constraint Language.

constraints can be translated into Simulink assertion blocks or into SPARK 2014 contracts. This property translation process guarantees that all system properties are allocated and verified on the atomic components. The property translation continues until a property is verified: for example, a constraint can be verified by simulation in Simulink, and if such verification is not deemed sufficient, the property can be translated to SPARK 2014 and formally verified on the SPARK code.

When adopting a bottom-up approach, properties may be violated when sibling components do not respect their mutual interface contracts. For example, a control algorithm requiring a 20Hz execution rate may be integrated in a real-time architecture unable to guarantee such a property. To avoid this problem, we propose to formalize in AADL all the relevant properties of the components being integrated. We are interested in properties that are relevant from a system perspective, both implicit and explicit. For example, a Simulink algorithm implicitly assumes that its persistent data is not changed between two consecutive computations. Likewise, a SPARK program explicitly states its system-relevant contracts in the top-level module. Once the properties have been extracted, the pivotal AADL model can be used to verify the correctness of the integration of atomic components.

The two approaches are summarized in the following picture:



Before proceeding with our paper, we would like to highlight that our approach works only when using languages providing design-by-contract facilities. This is of course the case of Simulink (assertion blocks) and SPARK 2014. However, other languages present similar features: UML can use OCL constraints and C can be decorated with ACSL[3] contracts. We chose to consider Simulink and SPARK because Simulink is the most widely used modeling language in high-integrity real-time systems and SPARK is the only commercially-supported programming language amenable for our domain of interest and offering design-by-contract.

[3] http://frama-c.com/acsl.html

From now on, we will concretely show how properties can be translated between different languages and abstraction levels, and we will discuss the level of automation attainable with currently available tools.

3. Related work

Regardless of whether the development follows a bottom-up approach, a top-down approach or a mix of both, one core activity is the low-level verification of components in isolation, that is the verification that source code correctly implements its specifications. In this section, we list six different ways to achieve this goal. We provide this list upfront to better position our approach with respect to the current state of the art.

3.1 Peer review

This approach is the classical one and consists of having an experienced person review the specification and implementation looking for pitfalls and errors. This static verification is typically manual but can be supported by appropriate verification tools such as CodePeer[4]. This approach is easy to deploy, but quite costly and error-prone, as it relies on manual labor.

3.2 Extensive testing

Testing is a dynamic form of verification that infers correctness of the implementation by checking that the compiled executable behaves as expected with respect to an oracle. The oracle may be manually produced. However, if the specification is executable, the oracle may be produced by executing the specification with a set of test data and checking that the implementation produces the same output. A typical example of the latter is comparing Simulink simulation with the behavior of its source code translation. Another example is checking compiled contracts dynamically during the testing of the executable. Testing is relatively easy to deploy, but it cannot of course be extensive.

3.3 Qualified automatic code generation

This approach consists of the automated production of the implementation of a component from its specification, such as the automatic generation of a C language implementation from a Simulink control algorithm. In this case, the correctness of the implementation is inferred from the correctness of the code generation itself. The latter is verified once and for all as a COTS tool. Depending on the safety standard of reference, this verification may be lightweight or extremely costly. Examples of such qualified code generators are SCADE KCG[5] and the Simulink code generator developed within the FUI project Project P. The main advantage of this approach is the low-cost of the verification activities across various projects once the code generator is qualified.

3.4 Formalize requirements as source code contracts

This approach consists of formalizing the requirements using elements of the lower level abstraction (source code) and verifying the consistency between the formalized requirements and the source code, either statically or dynamically [ARTIFACT]. It is the approach used in the Unit Proof methodology at Airbus [AIRBUS] and the SPARK contract-based verification technology [SPARK_CASE_STUDY]. Low-level requirements are expressed as subprogram contracts (typically preconditions and postconditions), and the source code is verified against these contracts, either through testing or through formal verification, or a mix of both [TEST_AND_PROOF].

4.5 Translation of contracts across different abstraction levels

This approach consists of expressing the same property at different levels of abstraction and verifying it at every abstraction level. It is the approach used in this paper when translating Simulink assertion blocks (observers) into SPARK contracts. The benefit of performing the verification at each level is that errors in the (automatic or manual) translation of the software can be detected at the lower level. The benefit of translating

[4] http://www.adacore.com/codepeer
[5] http://www.esterel-technologies.com/products/scade-suite/generate/qualified-code-generation

the property automatically is that it can be expressed at the most natural level, and still be used at the lower level.

3.6 Property extraction

The last verification strategy consists of extracting implicit properties out of an implementation and comparing them against a specification. For example, source code can be reverse-engineered to a formal model, and the formal model can be checked (statically or dynamically) against a set of properties. ADI Beacon[6] uses this approach to reverse-engineer SPARK code generated from Simulink and compare it against the source Simulink model by using traceability information encoded in the source code. The main advantage of this approach is that, in theory, it works a posteriori on any implementation. However, it usually requires dedicated tools which need extensive verification and validation and is limited by the capability of programming languages to express complex specifications.

In the industrial state-of-the-art, the most used approaches are peer review and testing. All other approaches (and testing as well) greatly benefit from a specification written in a computable specification language like AADL, UML, Simulink or SPARK.

4. Our goal: to improve the integration process

In the previous section, we listed the current state-of-the art for property preservation. Due to the increasing complexity of modern systems and the extensive use of longer supply chains for software components, it is unlikely that the same method is used for all components of a system. The deployment of multiple languages, tools and methods is very attractive because it decreases the development costs of each single component: a control algorithm is better developed in Simulink, while a high-security system is better formally specified and developed in SPARK.

Unfortunately, the use of heterogeneous languages and methods increases the cost of verification. This is the specific aspect we want to tackle in this paper. Our main goal is to demonstrate how we can abstract integration-relevant properties out of each component and use them to verify their integration within a complete system. More concretely, we aim to:

A. Use an AADL architectural model as a pivot element to link components developed using heterogeneous methods and tools;

B. Formalize in the above AADL model the assumptions on which each component relies when it is verified in isolation. These assumptions can be extracted from the atomic components when following a bottom-up approach; and can be translated to the implementation languages when following a top-down approach.

C. Verify that the above assumptions still hold in the composed system.

As explained, this approach is intended to fit both the top-down and bottom-up approaches. The main challenges are:

1. To formalize how to translate integration-relevant properties across languages; and
2. To develop automated tools performing most verification tasks.

To give a practical evidence of our idea, we decided to develop a system whose components are developed using the methods identified in section 3, and in particular:

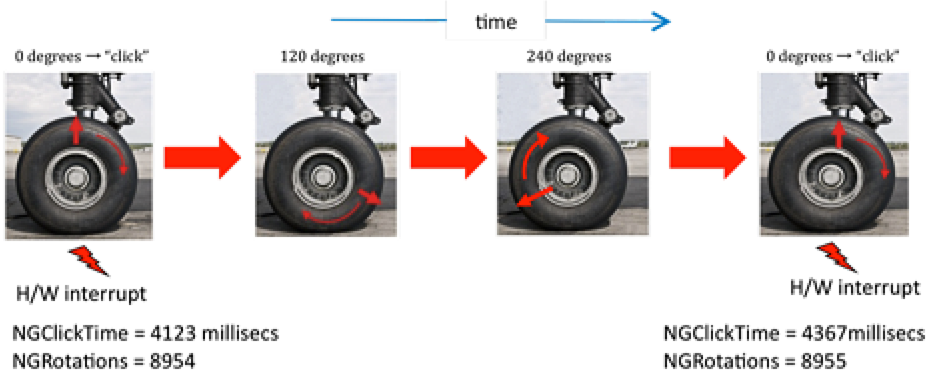
- Use of qualifiable automatic code generators from Simulink
- Use of contract-based programming with formal verification
- Use of peer reviews and extensive testing.

[6] http://www.adi.com/products/b4s

The system we decided to develop is described in the next section.

5. The Nose Gear Challenge

The Nose Gear challenge[7] was proposed in 2011 by the organizers of the second Workshop on Theorem Proving in Certification as a small, yet realistic case of a critical system, to demonstrate and compare benefits and limitations of formal methods. The problem is that of computing a simple function that estimates the velocity of an aircraft while moving on the ground. The only information available comes from an interrupt routine which is called each time the wheel completes a full rotation (the “click”). It updates the value of two 16-bit counters called NGRotations and NGClickTime which denote respectively the count of full rotations of the wheel, and the time of the last click. A third 16-bit counter Millisecs updated every millisecond denotes the current time. The following diagram illustrates this behavior:



Two requirements were specified on the system:

1. A flag **estimatedGroundVelocityIsAvailable** shall be set to **true** if the velocity has been correctly estimated and to **false** otherwise. When available, the estimated value of the velocity shall be within 3km/h of the true velocity of the aircraft at some moment within the past 3 seconds.
2. Estimated velocity should be available during taxi, takeoff and landing. We assume the maximum velocity for a typical airliner like the A320 to be at most 200 km/h.

The challenge is to develop a solution to this problem which provides the required assurance that the software developed complies with requirements 1 and 2 above, while at the same time following a strawman certification standard[8].

The original version of the challenge presented above could be developed using just Simulink. In fact, QinetiQ developed in 2011 a solution entirely in Simulink, using a non-qualifiable code generator and a verification tool able to compare the generated code with the Simulink model to formally prove their consistency. To better fit it to our needs, we decided to extend the original scope of the challenge by adding two features to the system :

1. a logging mechanism should store all events and computations of the last 5 minutes.
2. a simple graphical user interface should display the current estimated velocity when it is available and the log of events.

Our solution to the extended Nose Gear challenge consists of:

1. a model of the architecture in AADL;
2. a model of the velocity computation in Simulink;

[7] http://www.cl.cam.ac.uk/~mjcg/FMStandardsWorkshop/NoseGear.html
[8] http://www.cl.cam.ac.uk/%7Emjcg/FMStandardsWorkshop/standard.pdf

3. an implementation of the logger in SPARK;
4. an implementation of the GUI in Ada.

5.1 Modelling the architecture in AADL

For the ease of discussion, we start by presenting a top-down approach.

We begin by defining the global system architecture using AADL. This language offers inheritance and refinement features that allow us to start with a very abstract representation of the system and refine it step by step as design decisions are made, while preserving information along the process. In this manner it is possible to express high-level properties of abstract components of the architecture early in the design, and keep that information attached to the components throughout the refinement process.

Two modeling stages have been defined.

- Stage 0 represents the interfaces of each block of the system. It is complemented with sequences of subprogram calls modeled using the AADL Behavior annex and BLESS annotations [BLESS] representing invariants deduced from LLR.
- Stage 1 refines the previous model, using AADL inheritance mechanism, turning abstract components into concrete ones (threads, devices) and associating components to execution resources (CPUs, communication buses, etc.).

The source code in Ada is generated from the Ocarina [OCARINA] code generator for AADL. Ocarina targets a reduced middleware that is compatible with the Ravenscar profile for high-integrity concurrent systems. The static analyzer CodePeer is used to detect all potential run-time errors in the generated code (not the run time library, independently certified), which were manually reviewed. CodePeer generates error messages for all potential run-time errors (including reads of uninitialized data), and warnings for some cases of suspicious constructs (for example dead code). Analysis of the generated code by Ocarina resulted in 30 messages (18 errors and 12 warnings) which were all manually inspected. Half of them were intentional, typically tests always true which is common in generated code, and functions always failing by returning an exception because not yet implemented, but never called. The other half were false positives, where the static analyzer misidentifies a potential problem though lack of knowledge of the execution context, which were easily diagnosed as such.

AADL can be used to store various system metrics, including performance information. We use this mechanism to store relevant information for all LLRs.

Let us recall an interrupt is triggered at every rotation of the wheel. HLR#2 provides a maximum speed for the plane, that can be linked to the maximum number of interrupts to be handled by the CPU hosting the Nose Gear. Using the AADL Constraint Annex [ACL], we can model compatibility relationships between properties of the model, and thus confirm the target CPU is a valid choice for deploying the system. This is depicted in the example below.

```
system Airliner
properties
  Plane_Properties::Takeoff_Speed => 200 kph;
  Plane_Properties::Wheel_Perimeter => 5 m;
  -- ...
end Airliner;

system implementation Airliner.i
subcomponents
   CPU : processor Nosegear_CPU
     { Plane_Properties::IT_Processing_Time => 1 us; };
  -- ...
```

```
annex ACL {**
theorem check_IT_Rate -- Evaluation HLR#2 is met
 foreach p in processor_set do
  var ISR_WCET :=
     Get_Property_Value (p, "plane_properties::it_processing_time");
  sys := {s in System_Set | Is_Subcomponent_Of (p, s)};
  var Speed :=
    Get_Property_Value (sys, "plane_properties::takeoff_speed");
  var Wheel :=
    Get_Property_Value (sys, "plane_properties::wheel_perimeter");
  var Max_ISR_Freq := (head (Speed) / head (Wheel));
   -- Maximum number of ISRs per sec.
  check (Max_ISR_Freq * ISR_WCET < 1);
 end check_IT_Rate; **};
end Airliner.i;
```

HLR#1 is concerned with the availability of a result based on interrupts received. Addressing it completely would require physics modeling of the aircraft which we consider to be out of the scope of the tools presented in this paper. To simplify, we state that the velocity estimation shall be available when all the data used for the computation is less than 3 seconds old. This requirement is formalized as a BLESS assertion attached to the calculation component of the architecture. In that assertion `NGClickTime^(-1)` refers to the value of `NGClickTime` at the previous activation of the periodic thread.

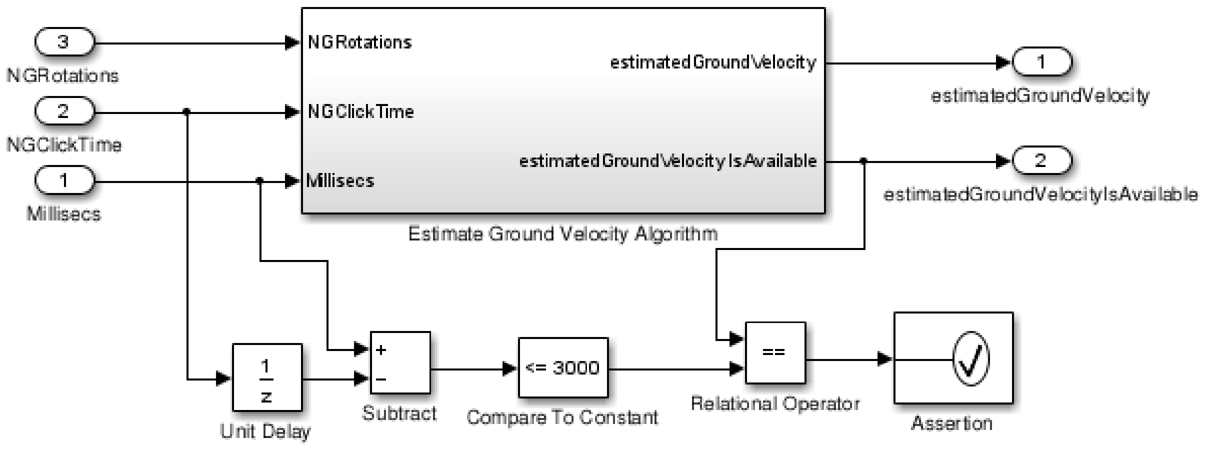
```
thread Velocity_Calculation
features
   --   ...
annex BLESS {**
  assert --  estimatedGroundVelocityIsAvailable is set to true if and only if
         --  the sensor data used for the computation is no older than than 3000ms.
     << hlr_availability: :
        (estimatedGroundVelocityIsAvailable) iff
                 ( (Millisecs - NGClickTime^(-1)) <= 3000 ) >>
```

Such a contract requires the full implementation to be verified. Here, we use AADL only to store the contract, and trace it to the corresponding support entity: a Simulink model. We will discharge this contract later.

5.2 Modelling the velocity computation in Simulink

The computation algorithm is designed using Simulink. To ensure property preservation, we translate the BLESS formalization into the Simulink model with the help of an *Assertion* block as depicted in the following diagram. Our goal is to further translate it down to SPARK. The property is verified by simulation in Simulink, by formal proof in SPARK and can be dynamically checked during integration phase because SPARK contracts can be executed as run-time assertions. This approach honours our original idea of translating properties until we are sufficiently sure they truly hold.

The source code in SPARK is generated from the Simulink model by using the code generator developed with Project P. A contract for the main procedure Nose_Gear_Comp is generated from the Simulink model as well. The `Old_NGClickTime_memory` variable used in the contract is a persistent variable storing the value of `NGClickTime` at the previous invocation (*i.e.* the output of the *Unit Delay* block).



```
procedure Nose_Gear_Comp
  (NGRotations                         : in  Unsigned_16;
   NGClickTime                         : in  Unsigned_16;
   Millisecs                           : in  Unsigned_16;
   estimatedGroundVelocity             : out Long_Float;
   estimatedGroundVelocityIsAvailable  : out Boolean)
 with Post =>
      --  @llr llr_availability
      --  The ground velocity shall be available if and only if the data used for the
      --  computation is no older than 3000ms.
      (estimatedGroundVelocityIsAvailable =
           (Millisecs - Old_NGClickTime_memory <= 3000));
```

In this component, property preservation is demonstrated by translating properties across abstraction levels, by automatic code generation and by formal proof. Note that the SPARK toolset currently proves the absence of run-time errors in the implementation of `Nose_Gear_Comp`, but not yet that the postcondition above is respected, due to a current limitation related to the use of modular types in Ada. This limitation is planned to be lifted in the future.

5.3 Implementing the logger in SPARK

The logger API consists of two subprograms: a function Log_Content to retrieve the content of the log (say, to display it on a control panel) and a procedure Write_To_Log to add an entry to the log. Only the most recent 600 entries are saved, which corresponds to 5 minutes of logging at a rate of one event every 500 ms.

A detailed functional specification of Write_To_Log was written as a SPARK contract:

```
procedure Write_To_Log (E : Log_Entry)
--  @llr Write_To_Log
with Contract_Cases =>
     --  The logger component shall be able to accept a new logging message.
     --  For an old empty log, the new content is the new entry alone.
     (Is_Empty =>
            Log_Content = Singleton_Log (E),

     --  For an old full log, the new content is the old one, with the
     --  oldest entry removed, plus the new entry.
     Is_Full =>
            Log_Content =
            Log_Content'Old (Log_Content'Old'First + 1 .. Log_Content'Old'Last)
            & E,
     --  For an old log neither empty not full, the old content is
     --  preserved, and the new entry added.
     others =>
            Log_Content = Log_Content'Old & E);
```

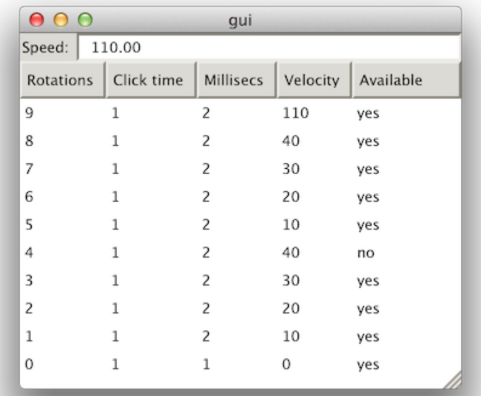
This contract is expressed as a set of disjoint and complete cases, using the Contract_Cases notation. It was checked automatically against the implementation of Write_To_Log using the formal verification tool GNATprove in the SPARK toolset. By using SPARK for specifying the logger and verifying its implementation, the following formal guarantees are obtained:

1. There will not be any reads of uninitialized data at run time.
2. There will not be any run-time errors.
3. The contract of Write_To_Log is respected by its implementation.

Currently, the SPARK toolset manages to prove automatically points 1 and 2 above, but not point 3. This is a temporary limitation of the toolset that is being worked on.

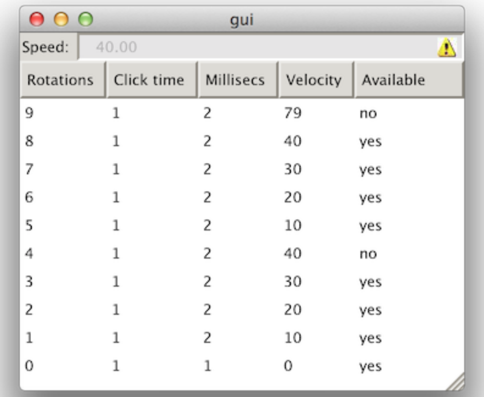
5.4 Implementing the GUI in Ada

The GUI was developed in Ada, based on the GtkAda library providing a binding to GTK graphical library. The requirements for the GUI were expressed in natural language, and checked manually. The requirements were that the GUI would display the most recent entries in the log with the most recent at the top and the least recent at the bottom, and the current computed velocity if available. The left panel below shows the appearance of the GUI when the velocity is available, and the right panel the appearance of the GUI when the velocity is not available:



Speed: 110.00

Rotations	Click time	Millisecs	Velocity	Available
9	1	2	110	yes
8	1	2	40	yes
7	1	2	30	yes
6	1	2	20	yes
5	1	2	10	yes
4	1	2	40	no
3	1	2	30	yes
2	1	2	20	yes
1	1	2	10	yes
0	1	1	0	yes



Speed: 40.00

Rotations	Click time	Millisecs	Velocity	Available
9	1	2	79	no
8	1	2	40	yes
7	1	2	30	yes
6	1	2	20	yes
5	1	2	10	yes
4	1	2	40	no
3	1	2	30	yes
2	1	2	20	yes
1	1	2	10	yes
0	1	1	0	yes

The CodePeer static analysis tool was applied on this code to check for possible run-time errors, and all alarms were manually inspected.

6. Recap system verification

The following table recaps how we decomposed the complete system and which techniques we used to verify property preservation.

Component	Specification written in	Implemented in	Property preservation verified by
Real-time architecture	AADL	Ada Ravenscar via automated code generation	Semi-automated peer review using CodePeer
Control algorithm	Simulink (with assertion blocks translated from AADL)	SPARK via automated code generation	● Translation of properties from AADL to Simulink and from Simulink to SPARK ● Qualified code generator ● Formal proof on SPARK code
Logger	SPARK contracts manually propagated from BLESS assertions	SPARK	Formal proof on SPARK code
GUI	Natural language	Ada	● Semi-automated peer review with CodePeer ● Testing

The table above demonstrates how different techniques for property verification can be deployed and integrated in the same system while ensuring traceability of property preservation and providing enough confidence that the property truly holds.

We now miss the bottom-up part of our process: showing that the properties verified in isolation on each component still hold in the integrated system. To do this we need to show that assumptions made for the isolated verifications are not violated by the integration, relying on the main integration artifact: the AADL model. This step requires the reverse translation of contracts from SPARK and Simulink to AADL where they are represented as AADL properties and BLESS assertions attached to component interfaces. For example assertions attached to output ports express guarantees ensured for the destination component, and those attached to input ports express assumptions made by the receiver component. Therefore we believe that a model-level analysis [BLESS] can show the compatibility between guarantees and assumptions of sibling components, however this remains to be explored in upcoming experiments.

6.1 Tool automation

In the table process above, there are three main steps that require specific support to be automated: translation between AADL and Simulink, between Simulink and SPARK and between AADL and SPARK. All verification within the same language (BLESS assertions, SPARK contracts, Simulink assertion blocks) are already available within their respective development environments. Automating the translation of properties across languages is then a major step toward a fully automated property-preserving process where properties are proved and translated at multiple abstraction layers.

We currently can translate Simulink assertion blocks to SPARK code. Translating Simulink code to SPARK contracts is more complicated, but feasible: we currently support a limited set of blocks and and only scalar datatypes in this context.

As for AADL contracts, we know of no existing tools that translate them automatically to Simulink or SPARK mainly because the BLESS language is relatively recent. Moreover the semantics of BLESS contain notions of time that make the translation potentially difficult - this is part of future work.Yet, current experiments demonstrate code generated from Simulink can already be integrated in generated code from AADL. Similarly, the extended coverage of Ada by SPARK 2014 allows for a significant coverage of the generated code for ensuring glue code is matching SPARK code expectations.

7. Conclusion

In this paper, we proposed a pragmatic view to assess property preservation when developing complex systems using heterogeneous languages and integrating existing components.

The approach relies on AADL as a pivot language to express system-level properties and to verify correctness of integration.

In a top-down approach, properties are translated from AADL system models down to software modeling languages (such as Simulink) or programming languages (such as SPARK). Properties are translated until they can be discharged by relying on tools such as provers or qualified code generators.

In a bottom-up approach, integration-relevant properties are extracted from existing components and represented in AADL, where they can be discharged by using BLESS, thus showing that the verification of components in isolation still holds when they are integrated.

In both cases, we need to rely on languages providing design-by-contract facilities: REAL and BLESS for AADL, assertion blocks for Simulink, contracts for SPARK.

The main open point is related to the degree of automation to move from one abstraction level to the other. In a top-down approach, code generators can easily take care of producing an implementation, however, translation of properties is a more complex task. We currently support this translation in a limited form from Simulink to SPARK. No such automation exists yet for BLESS and REAL.

In a bottom up approach the current tool support is lacking many elements: ensuring that AADL generated code can support verification activities for SPARK code (e.g. bounding the number of events, respect of timing invariant, etc). This specific challenge is conceptually similar to an advanced reverse engineering tool and will require more study from our side.

Our future work will then focus on improving the automation of the property translation process, in particular increasing the support for the generation and extraction of SPARK contracts from and to AADL.

References

[ACL] J. Hugues and S. Gheorghe, *The AADL Constraint Annex*, in SAE 2013 AeroTech Congress & Exhibition, 2013

[AIRBUS] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels and Benjamin Monate, *Testing or Formal Verification: DO-178C Alternatives and Industrial Experience*, IEEE Software Special Issue on Safety-Critical Software, 2013

[BLESS] Larson, B., Chalin, P., and Hatcliff, J. *BLESS: Formal specification and verification of behaviors for embedded systems with software*. In NASA Formal Methods, G. Brat, N. Rungta, and A. Venet, Eds., vol. 7871 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 276–290.[ARTIFACT] José Ruiz, Cyrille Comar and Yannick Moy, *Source Code as the Key Artifact in Requirement-Based Development: The Case of Ada 2012*, Proceedings of the conference on Reliable Software Technologies – Ada-Europe 2012

[OCARINA] Ocarina and AADL resources, http://www.openaadl.org, 2013

[SPARK_CASE_STUDY] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré and Yannick Moy, *Rail, Space, Security: Three Case Studies for SPARK 2014*, Embedded Real Time Software and Systems 2014

[TEST_AND_PROOF] Cyrille Comar, Johannes Kanig and Yannick Moy, *Integrating Formal Program Verification with Testing*, Embedded Real Time Software and Systems 2012