Towards formalising AADL in Proof Assistants

Jean-Paul Bodeveix [1] David Chemouil Mamoun Filali [2]
Martin Strecker [3]

IRIT
Université Paul Sabatier
118 route de Narbonne
F-31062 Toulouse

Abstract

This paper presents first steps towards a formalisation of the Architecture Analysis and Design Language, mainly concentrating on a representation of its data model. For this, we contrast two approaches: one set-based (using the B modelling framework) and one in a higher-order logic (using the Isabelle proof assistant). We illustrate a transformation on a simplified part of the AADL metamodel concerning flows.

Key words: AADL, Program Transformation, Theorem Proving

1 Introduction

This paper discusses design decisions for a formal meta-model of AADL (Architecture Analysis and Design Language) [FLV03], a language that has become an official standard of the SAE [4] in October 2004. AADL is the synthesis of previous architecture description languages, such as MetaH [Met97], Acme [GMW97] and Cotre [BRV+03]. It reflects the growing industrial need to model and reason about complex software/hardware artifacts, as found in the avionics and automotive sector. Indeed, the authors of this contribution are partners in the federated project TOPCASED which comprises, among others, Airbus and TNI, and aims at creating a workbench whose main modelling language is AADL.

Due to the complexity of its application domain, the definition of AADL itself is voluminous and not easily accessible. Apart from the AADL standard,

[1] Email: bodeveix@irit.fr
[2] Email: filali@irit.fr
[3] Email: strecker@irit.fr
[4] Society of Automotive Engineers

there is a formalised description in Ecore, a variant of XML which will be summarised in Section 2. By its very nature, the latter can only capture syntactic aspects and limited forms of typing, but not certain consistency conditions of a model ("acyclicty of connections between components").

In the context of the TOPCASED project, we are interested in describing AADL model transformations and reason about them, with the purpose of proving, for example,

- that they preserve the semantics of the model, or at least certain semantic aspects, such as the temporal behaviour. This is necessary, among others, when transforming a model so that it can be handled by other verification tools, such as model checkers.
- that they preserve or establish specific properties. Such a well-formedness check is applicable to refactoring steps that may change the semantics of the model.

Defining a formal semantics for programming languages and verifying the correctness of language transformations or compilers in proof assistants is a well-established practice. For several reasons, AADL does not fit neatly into this scheme:

- The Ecore-style presentation gives it an object-oriented flavour. However, most proof assistants are based on a form of set theory or a higher-order logic comprising a kind of functional language. Even though a coding of object-oriented concepts is possible ("deep embedding"), it is more natural to reason about AADL using concepts of the language offered by the underlying proof assistant ("shallow embedding"), such as its type system and function definition facilities. We will discuss several alternatives in Section 3.
- The sheer size of AADL ($\approx$ 200 classes, often with a large number of attributes) makes it difficult to handle manually. Furthermore, due to the recency of the AADL standard, frequent changes of the meta model can be expected. Therefore, at least a partial automation for converting the Ecore presentation into the definition of the proof assistant of choice is reasonable. We will indicate throughout the text how this could be achieved.

For the time being, we can only give a very preliminary account of our formalisation of the AADL language. Even though still incomplete, a small example in Section 4 shows where we are heading. The paper concludes in Section 5 with a discussion of related work and possible extensions.

2 Language Definitions in Ecore

Ecore is the meta-model of EMF, the Eclipse Modelling Framework. In [BSM+03], EMF is characterized as "essentially the Class Diagram subset of UML". Thus, EMF defines concepts such as packages, classes, objects, and attributes. Ecore itself is presented as an XML schema [Con]; accordingly, Ecore definitions, such as those of AADL, are XML documents. These documents lend themselves to automatic processing.

To get an intuition, consider the definitions of the following classes, slightly simplified for the purpose of this presentation:

```
<eClassifiers xsi:type="ecore:EClass" name="AObject" abstract="true">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="comment"
      unique="false" upperBound="-1" eType="ecore:EDataType"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="NamedElement"
      abstract="true" eSuperTypes="#//AObject">
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
      eType="ecore:EDataType"/>
</eClassifiers>

<eClassifiers xsi:type="ecore:EClass" name="Connection" abstract="true"
      eSuperTypes="#//ModeMember property.ecore#//ReferenceElement">
  <eStructuralFeatures xsi:type="ecore:EReference" name="srcContext"
      lowerBound="1"
      eType="ecore:EClass connection.ecore#//FeatureContext"/>
  <eStructuralFeatures xsi:type="ecore:EReference" name="dstContext"
      lowerBound="1"
      eType="ecore:EClass connection.ecore#//FeatureContext"/>
  <eStructuralFeatures xsi:type="ecore:EReference"
      name="inModeTransitions" upperBound="-1"
      eType="#//ModeTransition" eOpposite="#//ModeTransition/members"/>
</eClassifiers>
```

It defines a class `AObject` (the very top of the AADL class hierarchy), one of its immediate subclasses, `NamedElement`, and a `Connection` class. Class `AObject` can contain an arbitrary number of `comment` attributes (as indicated by the negative *upperBound*), `NamedElement` has a `name` attribute, apart from the `comment` attribute inherited from `AObject`. A `Connection`, with supertypes `ModeMember` and `ReferenceElement`, has three (non-inherited) attributes, among them two references to elements of class `FeatureContext` (the source and destination of the connection).

Ecore itself does not have a formal semantics. Of course, defining a semantics for XML [SW03] does not attribute a meaning to languages (such as Ecore) defined with the aid of XML. Most often, Ecore is put into correspondence with UML or Java[EMF,BSM+03], but again not in a systematic way.

Altogether, we are left on our own to give a precise meaning to the constructs of Ecore, based on an informal understanding of the underlying concepts. This translation, which we will explore for different target formalisms in Section 3, is not limited to the definition of AADL, but can be applied to any language defined in Ecore. However, AADL imposes some additional constraints that cannot be expressed in Ecore but are made precise in the accompanying standard[Int04].

So far, we do not cover all of Ecore, but pay particular attention to the following elements:

- Classes and attributes, as expressed by the `eClassifiers` and `eStructural-Features` elements.
- The class hierarchy, as expressed by the `eSuperTypes` attribute of Ecore classes, and abstractness (attribute `abstract`).
- Types of Ecore attributes (`eType`) as well as multiplicities (`lowerBound` and `upperBound`).

Other elements (`eAnnotations`) are currently not dealt with. The package structure is not respected (i.e., we assume a single package) as Ecore permits mutual dependencies between elements across package boundaries, which would not be acceptable for the systems used in Section 3.

3 Model definitions: Alternatives

In this section, we consider two alternatives for encoding models described in the Ecore language. The first one is based on set theory and the second is based on higher order logics.

3.1 Set based encodings

In this section, we illustrate how an `ecore` description can be encoded in set theory. We use the B [Abr96] syntax. The translation to other set based frameworks like TLA [Lam02], or Isabelle-ZF [Pau04] should be similar. In this section, we sketch the principles of data modeling. In section 4, we present a transformation example relying on such a representation.

We consider a simple class hierarchy where `AObject` is the root, and the classes `ComponentImpl` and `ComponentType` are derived from the class `ComponentClassifier`.

Class hierarchy.

The basic idea of the set based encoding is to represent the *is-a* relation of object oriented programming as the subset relation of set theory and to partition the whole set of objects of type `C` (the `Csub` set) into objects which will be instances of `C` (the `C` set) and objects of which type is a subclass of `C`. Then, such partitions will be the containers for the instances that will be actually created: the partition associated to class `C` will contain the instances of class `C`. It follows, that sets of instances of different classes will be disjoint. At last, when a class `C` is abstract, the container `C` is made empty, i.e., we have the additional property: `C = ∅`. In B, we encode this as follows[5]:

```
SETS
  AObject /* universe of AADL objects */
CONSTANTS
  /* containers */
  ComponentClassifier, ComponentImpl, ComponentType
DEFINITIONS
  ComponentClassifiersub ==
    (ComponentClassifier ∪ ComponentImplsub ∪ ComponentTypesub);
PROPERTIES
  ComponentClassifier ⊆ AObject
 /* containers are disjoint */
∧ ComponentClassifier ⊆
    ComponentClassifiersub - (ComponentTypesub ∪ ComponentImplsub)
∧ ComponentImpl ⊆
    ComponentClassifiersub - (ComponentClassifier ∪ ComponentTypesub)
∧ ComponentType ⊆
    ComponentClassifiersub - (ComponentClassifier ∪ ComponentImplsub)
```

Instances and attributes.

Since we plan to study model transformations, instances must be dynamic and are encoded by the variable `instances` which is a subset of `AObject`. Moreover, as we will see in section 4, a new instance of a class `C` is obtained as an element of `C - instances`. With respect to the previous example, we have:

```
VARIABLES
  /* instances actually created */
  instances
INVARIANT
  instances ⊆ AObject
```

Remarks.

(i) In the B context, for theorem proving purposes, we can also state the following assertion, i.e., a predicate that can be derived from the preceding properties and invariants:

```
ASSERTION
  ComponentImpl ∩ ComponentType = ∅
```

Because there are $\frac{n\times(n-1)}{2}$ derived predicates, such assertions should be automatically generated only for classes with a small number of derivatives.

[5] In B, **FIN**(A) is the set of finite subsets of A; a DEFINITIONS introduce (parameterized) macros.

Attributes.

The attributes of a class are represented as functions. In order to support inheritance, the domain of such functions is the set of instances of the class and its derivatives. Then, we have:

VARIABLES

```
classattribute, ...
```

INVARIANT

classattribute $\in$ Classsub *cap* instances $\rightarrow$ typeattribute

$\wedge$ ...

Reading the attribute i of an instance o consists in evaluating classattribute(o). While the update of and attribute of o is easily expressed through the overloading notation of B:

$$\text{classattribute}(o) := \text{new\_attribute\_value}$$

3.2 Encodings based on Higher Order Logics

In the following, we will present first steps towards an encoding of the AADL meta-model in higher order logic. More specifically, we show how we represent the key elements of Ecore, as introduced in Section 2, in Isabelle/HOL[NPW02], the higher order logic extension of the generic proof assistant Isabelle. We will mainly restrict ourselves to the core of Isabelle/HOL (simply typed Lambda calculus with ML-style polymorphism and inductive datatypes). This has a double advantage: firstly, our development can easily be transposed to other proof assistants with more expressive type systems, such as Coq or PVS; secondly, we remain in the realm of the type systems of traditional programming languages such as ML, so that we get verified code executable [BN00] on a standard platform.

Let us review the main choices: A *deep embedding* does not try to directly represent elements of the language as expressions of the target language (in this case: Isabelle/HOL), but rather encodes them. For example, classes could be represented by the class name, the name of the superclass and lists of their attribute types. Such an approach has been followed in the semantics of Java described in [Ohe01]. We would get a definition like

```
attrib_decl = attrib_name * attrib_type
class = cname * cname * attrib_decl list
```

In our setting, this looks attractive at first glance because it allows for a very uniform treatment, and the translation from Ecore to the language of the proof assistant would be almost immediate. However, there are severe drawbacks: Almost no protection from the underlying type system of the proof assistant is provided. For example, an instance of a class would be a list of attribute values:

```
inst = (attrib_name * value) list
```

The fact that class and instance should have the same number of attributes, with attribute values of the correct attribute types, would have to be expressed as predicates and would not be ensured by the type system.

Furthermore, note that the class structure we are dealing with is not open-ended, as in a traditional object-oriented program. Rather, the class structure is fixed. For example, a `FlowSpec` can be either a `FlowSourceSpec`, a `FlowSinkSpec`, or a `FlowPathSpec`. This fact would again have to be coded explicitly, while we would prefer to appeal to an induction principle.

All this leads us to consider a *shallow embedding*, in which AADL classes are coded as types of HOL, whereas instances of classes correspond to terms of the respective types. Let us once again spell out the desiderata and then see to which degree we can fulfill them:

"Subclassing" should correspond to "subtyping". Unfortunately, in an ML-style type system, we do not have a natural notion of "subtype". At best, we could exploit polymorphism to encode subtyping. In fact, this is the idea underlying Isabelle's extensible records [NW98]. For example, a class `A` with field `a:nat` and its subclass `B` with fields `a:nat, b:bool` would give rise to type definitions

```
datatype
  'a t_A = c_A nat 'a
datatype
  'b t_B = c_B bool 'b
types
  'a A = 'a t_A
  'b B = 'b t_B t_A
```

so a term `c_A 3 (c_B True ())` would be of type `unit B`. A function `get_a`, selecting the `a` attribute and defined as

```
consts
  get_a :: 'a A => nat
primrec
  get_a (c_A a x) = a
```

would be applicable to elements of type A and type B.

This approach is elegant, but:

- The type system does not permit to express the fact that `A` only has specific subclasses, say `B` and `C`.
- We cannot form a collection (as list or set) of all elements of a class and its subclasses, because these elements would have different types, even though being instances of the same type scheme. However, for expressing some well-formedness constraints (such as "no dangling object references"), we have to keep track of all objects under consideration – see below.

In recent years, there has been a surge of research on coding object oriented languages in typed lambda calculi [ACV96,HP96,Com95]. Most of them are based on extensions of System F, thus leave the framework of plain ML-style polymorphism and are therefore not of direct use for us.

A second approach consists in factoring out the class hierarchy. For the above classes A and B, we would get

```
types
 A = nat
 B = nat * bool
```

which would make us lose any connection between A and B. In particular, we could not define an accessor function like get_a in a uniform way, but would have to resort to overloading, a special feature of Isabelle that may not be provided in other proof assistants:

```
consts
  get_a :: 'a => nat
defs (overloaded)
  get_a_A:  get_a == id
  get_a_B:  get_a == fst
```

This lack of uniformity makes us refrain from this solution.

This makes us arrive at the solution that we finally adopt. For each class C, we define two types:

- A type s_C which is the sum of the subtypes of C. For non-abstract classes, we add a further component (of type Unit) for an instance which is of class C an none of its subclasses. For a class without subtypes, we do not construct s_C.
- A type t_C which is the product of the types of the attributes of C and of s_C (if it exists).

For example, for the ComponentClassifier class mentioned in Section 3.1, we get the type definitions

```
datatype
  s_ComponentClassifier =
   cs_ComponentType t_ComponentType
 | cs_ComponentImpl t_ComponentImpl

datatype
 t_ComponentClassifier =
   ct_ComponentClassifier oid s_ComponentClassifier
```

where s_ComponentClassifier codes the fact that this class has subclasses ComponentType and ComponentImpl. The class is not abstract, otherwise we would have another constructor cs_Unit unit. We will return to the oid component in a moment.

When pursuing this construction from the leaves of the class hierarchy to the root, we end up with a single type, corresponding to the class of all objects, AObject in our case. Thus, unfortunately, we have to give up having a one-to-one correspondence between classes and types: Since the goals "subclassing corresponding to subclassing" and "uniform operations for subclasses" are irreconcilable, we opt for abandoning the former in favour of the latter.

We can, however, encode class membership by predicates, such as is_ComponentClassifier. These are used for expressing some consistency conditions on models, as described below.

In general, an AADL model will have a graph structure, so components will be related by references (type EReference in Ecore). In order to model object identities, we introduce a type oid of object identifiers, which we implement as natural numbers to ease certain operations, such as creation of a "new" oid:

```
constdefs
  new_oid :: oid list => oid
  new_oid os == (foldl max 0 os) + 1
```

We augment elements of the root class of the hierarchy, AObject, with an oid attribute

```
 t_AObject =
   ct_AObject oid s_AObject
```

and can now define an AADL model to be a collection of objects:

```
types AADL_Model = t_AObject list
```

Creation of a new instance, such as for class Component (see example Section 4), is done relative to an AADL model M, with an expression such as new_Component M nm.

A minimal requirement on models is well-formedness, which comprises:

- uniqueness of object identifiers
- type-correctness of attributes, in particular of references. Note that this property cannot, in general, be enforced statically by a type system, so a modification of a model will generate proof obligations.

For example we say that a component is well-typed if its implementation reference im is a ComponentImpl, all of its ports po are well-typed, and so are its subcomponent and flow components:

```
consts
  wt_Component :: [AADL_Model, t_Component] => bool
primrec
  wt_Component M (ct_Component im po pa fl) =
  ((option_lift (reftype M is_ComponentImpl) im) /\
   (list_all is_Port po) /\
   (option_lift (reftype M is_ComponentImpl) pa) /\
   (list_all is_Flow fl))
```

This concludes our description our AADL formalisation. We are in the process of refining this model to include further consistency conditions, as expressed in the AADL standard.

4 Example: Flows

In this section, we sketch how a transformation can be considered in our framework. We first give a part of the AADL metamodel and then define a simple transformation that adds a filter to a flow. The example is only developed using the B language.

4.1 Part of the AADL metamodel

In this example, we consider a simplified part of the AADL metamodel which describes the concept of a flow. A flow can be declared in the specification of a component as a connexion between an input port and an output port. In the implementation, the flow can traverse sub-components. Three kinds of flows exist: flow paths, flow sources and flow sinks. We consider here flow paths which can be defined using the following regular expression:

```
flow_path ::= port (connection flow_path)* connection port
            | port port
```

Iterated (connection / flow_path) couples express sub-component traversals. They are defined in the metamodel using an ordered sequence of flow elements referring one connection and one flow path. Moreover, the couple `connexction flowpath` is represented by an instance of class `FlowElement` and the end ports are represented by an instance of the `Connection` class referenced as `srcDst` (fig: 1).

The metamodel is translated into B constants, abstract variables and invariants using the principles explained before.

4.2 Transformation example

Let us consider the transformation which introduces a *filter* on a given flow path of a component. Such a transformation is illustrated by figure 2.

Applying this transformation leads to the creation of several objects such as the filter component, its input and output ports and the flow path between them. Then the filter is added as a sub-component of the implementation and inserted in the flow path given as parameter.

The transformation must allocate new objects. They are chosen in the set of unallocated elements of their class. Objects of the same class are explicitly declared as different. For example, the following code fragment declares two new objects taken among the free instances of `FlowElement` and `Connection`.

```
ANY fe, cnx WHERE
  fe : FlowElement - FlowElementinst ∧
```

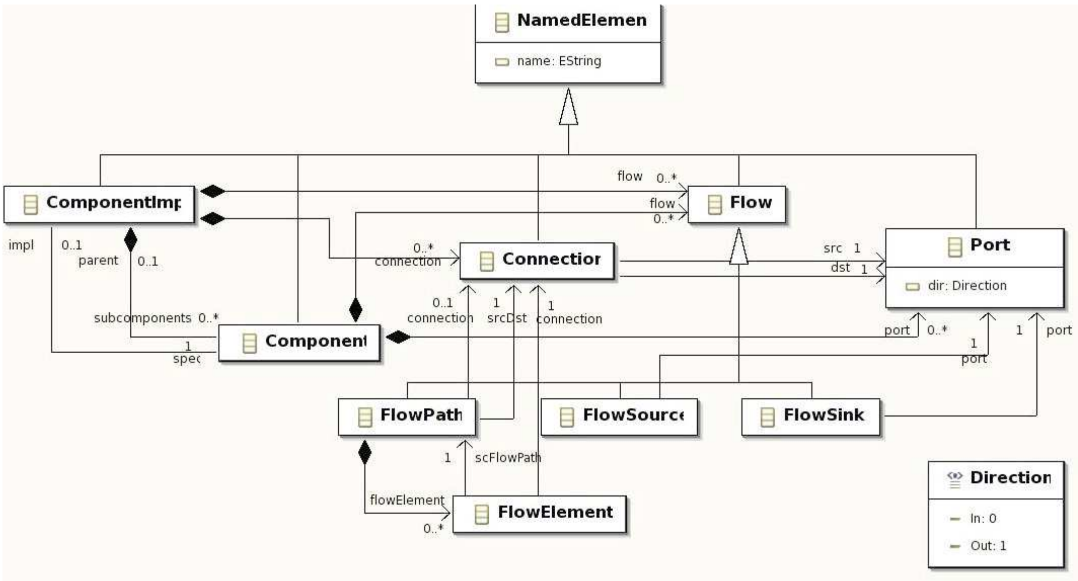

Fig. 1. Simplified AADL flow model

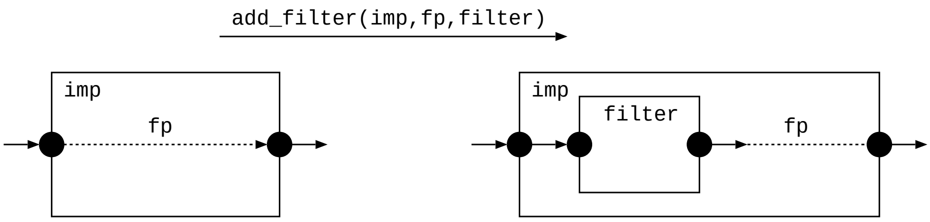

Fig. 2. Adding a filter

```
  cnx : Connection - Connectioninst
THEN
  ...
END
```

Declared objects are then added to the corresponding set of instances:

```
   FlowElementinst :=  FlowElementinst ∪ {fe}
|| Connectioninst := Connectioninst ∪ {cnx}
```

Finally, connections between objects must be established. For this purpose, (functional or non functional) relations are updated. For example, the flow element is initialised by the flow path of the filter sub-component and by a connection linking the input port of the component to the input port of the filter.

```
   FlowElementscFlowPath(fe) := f_fp
|| FlowElementconnection(fe) := cnx
```

The full code for the transformation is given above. The preconditions of the operations assert that the given filter has exactly one input port and one output port, with one flow path between them. This flow path will be inserted into the flow path fp of the given component implementation.

```
Addfilter(imp, fp, filter) =
  PRE
      imp : ComponentImplinst ∧
      fp: Componentflow(imp) ∧ fp : dom(FlowPathconnection) ∧
      filter : Componentinst ∧ card(Componentport(filter)) = 2 ∧
      Portdir[Componentport(filter)] = {In,Out} ∧
      Componentflow(filter) ⊆ FlowPath ∧
      card(Componentflow(filter)) = 1
  THEN
    ANY p_in, p_out, fe, cnx, f_fp WHERE
            p_in : Componentport(filter)
      ∧ p_out : Componentport(filter)
      ∧ Portdir(p_in) = In
      ∧ Portdir(p_out) = Out
      ∧ fe : FlowElement - instances
      ∧ cnx : Connection - instances
      ∧ f_fp : Componentflow(filter)
    THEN
       instances := instances ∪ {fe,cnx}
/* initialisation of the flow element */
    || FlowElementscFlowPath(fe) := f_fp
    || FlowElementconnection(fe) := cnx
    || Connectiondst(cnx) := p_in
    || IF FlowPathflowElement(fp) ≠ [] THEN
         LET fe1,cnx1 BE
            fe1 = first(FlowPathflowElement(fp)) ∧
            cnx1 =  FlowElementconnection(fe1) IN
          Connectionsrc :=  Connectionsrc <+ {cnx1 ↦ p_out,
                                  cnx ↦ Connectionsrc(FlowPathsrcDst(fp))}
         END
       ELSE Connectionsrc(cnx) := Connectionsrc(FlowPathsrcDst(fp)) END
/* insertion of the filter */
    || FlowPathflowElement(fp) := fe -> FlowPathflowElement(fp)
    END
  END
END
```

This specification of the transformation allows the verification of static properties such as the preservation of wellformedness properties of the model specified by invariants. The properties considered here are those directly expressed by the metamodel. They could easily be extended at the B level: flow path must be well build so that the extremity of connections and of the subcomponent flow paths match. Furthermore, implementation and specification of components must be compatible, which means that they have the same ports and that the origin and destination of flow paths are the same. These invariants ensure that the transformation preserve the flow-based semantics of the model. In order to go one step further, an abstract specification of the transformation should express that the specified flow is implemented by traversing the filter.

5 Conclusions

We have presented approaches of translating the Ecore language into different formalisms (set based, higher-order logic). In particular, we are interested in a representation of the AADL meta model, which permits us to specify and prove correct transformations of AADL models. A major concern of our further studies will be the scalability of such embedding approaches.

An avenue that we have not further explored is the following: perceive the AADL class structure as the class structure of an object-oriented program, and transformations as appropriate methods of these classes. In order to show the correctness of transformations, prove that the methods are correct with respect to a particular specification. The proof could be carried out with tools geared towards Java [JMR04,MPMU04,Mül02,?] or towards OO specification mechanisms such as Object-Z or variants [?] . Even though we do not have concrete evidence, we suspect that verification of an OO program is more heavy-weighted than the approach we have chosen and that, even worse, we cannot naturally express many interesting (e.g. temporal) properties in the given specification formalism.

Acknowledgement

We thank Nicolas Laleve for a careful reading.

A The B encoding

This section gives a sample of B code. Only the text of the transformation, i.e., the B operation `Addfilter` (section 4) has been written. The machine clauses: **SETS, CONCRETE_CONSTANTS, DEFINITIONS, PROPERTIES, ABSTRACT_VARIABLES, INVARIANT** and **INITIALISATION** have been obtained automatically from an Ecore description of the simplified fragment of the AADL model for flows (The translator has been written as a combination of CDuce [BCF03] and Ocaml).

MACHINE flows_toy
SETS
  String; AObject;
  Direction = {In, Out}
CONCRETE_CONSTANTS
  FlowSource, FlowSink, FlowElement, NamedElement, Flow, Connection,

```
    Component, Port, ComponentImpl, FlowPath
DEFINITIONS
  FlowSourcesub ≜ (FlowSource);
  FlowSinksub ≜ (FlowSink);
  FlowElementsub ≜ (FlowElement);
  NamedElementsub ≜ (
      NamedElement ∪ Flowsub ∪
          Portsub ∪ Connectionsub ∪ ComponentImplsub ∪ Componentsub);
  Flowsub ≜ (Flow ∪ FlowSourcesub ∪ FlowSinksub ∪ FlowPathsub);
  Connectionsub ≜ (Connection);
  Componentsub ≜ (Component);
  Portsub ≜ (Port);
  ComponentImplsub ≜ (ComponentImpl);
  FlowPathsub ≜ (FlowPath)
PROPERTIES
  FlowSource ⊆ AObject ∧
  FlowSink ⊆ AObject ∧
  FlowElement ⊆ AObject ∧
  NamedElement ⊆ AObject ∧
  Flow ⊆ AObject ∧
  Connection ⊆ AObject ∧
  Component ⊆ AObject ∧
  Port ⊆ AObject ∧
  ComponentImpl ⊆ AObject ∧
  FlowPath ⊆ AObject ∧
  Flow ⊆ Flowsub - (FlowSourcesub ∪ FlowSinksub ∪ FlowPathsub) ∧
  FlowSourcesub ⊆ Flowsub - (Flow ∪ FlowSinksub ∪ FlowPathsub) ∧
  FlowSinksub ⊆ Flowsub - (Flow ∪ FlowSourcesub ∪ FlowPathsub) ∧
  FlowPathsub ⊆ Flowsub - (Flow ∪ FlowSourcesub ∪ FlowSinksub) ∧
  NamedElement ⊆ NamedElementsub -
      (Flowsub ∪ Portsub ∪ Connectionsub ∪ ComponentImplsub ∪ Componentsub) ∧
  Flowsub ⊆ NamedElementsub -
   (NamedElement ∪ Portsub ∪ Connectionsub ∪ ComponentImplsub ∪ Componentsub) ∧
  Portsub ⊆ NamedElementsub -
   (NamedElement ∪ Flowsub ∪ Connectionsub ∪ ComponentImplsub ∪ Componentsub) ∧
  Connectionsub ⊆ NamedElementsub -
   (NamedElement ∪ Flowsub ∪ Portsub ∪ ComponentImplsub ∪ Componentsub) ∧
  ComponentImplsub ⊆ NamedElementsub -
      (NamedElement ∪ Flowsub ∪ Portsub ∪ Connectionsub ∪ Componentsub) ∧
  Componentsub ⊆ NamedElementsub -
   (NamedElement ∪ Flowsub ∪ Portsub ∪ Connectionsub ∪ ComponentImplsub) ∧
  NamedElement = ∅ ∧
  Flow = ∅
ABSTRACT_VARIABLES
  instances, FlowSourceport, FlowSinkport, FlowElementconnection,
    FlowElementscFlowPath, NamedElementname, Connectionsrc, Connectiondst,
    Componentimpl, Componentport, Componentparent, Componentflow, Portdir,
```

```
    ComponentImplunnamed, ComponentImplspec, ComponentImplsubcomponents,
    ComponentImplflow, ComponentImplconnection, FlowPathflowElement,
    FlowPathconnection, FlowPathsrcDst
INVARIANT
  instances ⊆ AObject ∧
  FlowSourceport ∈ (FlowSourcesub ∩ instances) ⟶ (Portsub ∩ instances) ∧
  FlowSinkport ∈ (FlowSinksub ∩ instances) ⟶ (Portsub ∩ instances) ∧
  FlowElementconnection ∈
    (FlowElementsub ∩ instances) ⟶ (Connectionsub ∩ instances) ∧
  FlowElementscFlowPath ∈
    (FlowElementsub ∩ instances) ⟶ (FlowPathsub ∩ instances) ∧
  NamedElementname ∈ (NamedElementsub ∩ instances) ⟶ String ∧
  Connectionsrc ∈ (Connectionsub ∩ instances) ⟶ (Portsub ∩ instances) ∧
  Connectiondst ∈ (Connectionsub ∩ instances) ⟶ (Portsub ∩ instances) ∧
  Componentimpl ∈
    (Componentsub ∩ instances) ⇸ (ComponentImplsub ∩ instances) ∧
  ComponentImplspec = Componentimpl⁻¹ ∧
  Componentport ∈ (Componentsub ∩ instances) ⟷ (Portsub ∩ instances) ∧
  Componentparent ∈
    (Componentsub ∩ instances) ⇸ (ComponentImplsub ∩ instances) ∧
  ComponentImplsubcomponents = Componentparent⁻¹ ∧
  Componentflow ∈ (Componentsub ∩ instances) ⟷ (Flowsub ∩ instances) ∧
  Portdir ∈ (Portsub ∩ instances) ⟶ Direction ∧
  ComponentImplunnamed ∈
    (ComponentImplsub ∩ instances) ⟷ (Connectionsub ∩ instances) ∧
  ComponentImplspec ∈
    (ComponentImplsub ∩ instances) ⟶ (Componentsub ∩ instances) ∧
  ComponentImplsubcomponents ∈
    (ComponentImplsub ∩ instances) ⟷ (Componentsub ∩ instances) ∧
  ComponentImplflow ∈
    (ComponentImplsub ∩ instances) ⟷ (Flowsub ∩ instances) ∧
  ComponentImplconnection ∈
    (ComponentImplsub ∩ instances) ⟷ (Connectionsub ∩ instances) ∧
  FlowPathflowElement ∈
    (FlowPathsub ∩ instances) ⟶ seq((FlowElementsub ∩ instances)) ∧
  FlowPathconnection ∈
    (FlowPathsub ∩ instances) ⇸ (Connectionsub ∩ instances) ∧
  FlowPathsrcDst ∈
    (FlowPathsub ∩ instances) ⟶ (Connectionsub ∩ instances)
INITIALISATION
  instances := ∅ || FlowSourceport := ∅ || FlowSinkport := ∅ ||
  FlowElementconnection := ∅ || FlowElementscFlowPath := ∅ || NamedElementname := ∅ ||
  Connectionsrc := ∅ || Connectiondst := ∅ || Componentimpl := ∅ || Componentport := ∅ ||
  Componentparent := ∅ || Componentflow := ∅ || Portdir := ∅ ||
  ComponentImplunnamed := ∅ || ComponentImplspec := ∅ ||
  ComponentImplsubcomponents := ∅ || ComponentImplflow := ∅ ||
  ComponentImplconnection := ∅ ||
```

FlowPathflowElement := ∅ || FlowPathconnection := ∅ || FlowPathsrcDst := ∅
OPERATIONS
Addfilter(imp, fp, filter) ≜ /* see section IV. */
END

References

[Abr96] J.R. Abrial. *The B-Book Assigning programs to meanings*. Cambridge University Press, 1996.

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 396–409, 1996.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce an XML-Centric general purpose language. In *Proc. of the ACM International Conference on Functional Programming*, 2003. http://www.cduce.org/papers.html.

[BN00] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Proc. TYPES Working Group Annual Meeting 2000*, LNCS, 2000.

[BRV+03] B. Berthomieu, P.-O. Ribet, F. Vernadat, J.-L. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.-L. Lambert. Towards the verification of real-time systems in avionics: The Cotre approach. In *Eigth International workshop for industrial critical systems, ROROS*, pages 201–216. Thomas Arts, Wan Fokkink, 5-7 juin 2003.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[Com95] Adriana B. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, University of Nijmegen, The Netherlands, January 1995. ISBN 90-9007860-6.

[Con] Worldwide Web Consortium. Extensible markup language (XML). http://www.w3.org/XML/.

[EMF] EMF: Eclipse Modeling Framework. http://download.eclipse.org/tools/emf/scripts/home.php.

[FLV03] Peter H. Feiler, Bruce Lewis, and Steve Vestal. The SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *RTAS Workshop 2003*, pages 1–10, May 2003.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[HP96] Martin Hofmann and Benjamin Pierce. Positive subtyping. *Information and Computation*, 126(1):186–197, 1996.

[Int04] SAE International. *Architecture Analysis & Design Language (AADL)*, August 2004.

[JMR04] Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. 2004.

[Lam02] Leslie Lamport. *Specifying Systems:The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Met97] MetaH. *http://www.htc.honeywell.com/metah/*. 1997.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. http://krakatoa.lri.fr.

[Mül02] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.

[NW98] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics, TPHOLs'98*, volume 1479 of *LNCS*, pages 349–366. Springer, 1998.

[Ohe01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. http://www4.in.tum.de/~oheimb/diss/.

[Pau04] Lawrence Paulson. Isabelle's logics: FOL and ZF. Technical report, University of Cambridge, Computer Laboratory, 2004. Available from http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[SW03] Jerome Simeon and Philip Wadler. The essence of XML. In *Proc. POPL'2003*, 2003. http://homepages.inf.ed.ac.uk/wadler/topics/xml.html.