Towards formalising AADL in Proof Assistants

Jean-Paul Bodeveix¹  David Chemouil²  Mamoun Filali³  Martin Strecker⁴

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.

Keywords: AADL, Program Transformation, Theorem Proving

1 Introduction
This paper discusses design decisions for a formal meta-model of AADL (Architecture Analysis and Design Language) [10], a language that has become an official standard of the SAE⁵ in October 2004. AADL is the synthesis of previous architecture description languages, such as MetaH [17], Acme [11] and Cotre [5]. As an architecture description language, AADL emphasizes as

¹ Email: bodeveix@irit.fr
² Email: david.chemouil@cnes.fr
³ Email: filali@irit.fr
⁴ Email: strecker@irit.fr
⁵ Society of Automotive Engineers
well static aspects like partitioning into packages and dynamic aspects related
to active entities, communication between them and timing aspects. In fact,
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 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,
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 (“acyclicity 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 (≈ 200 classes, often with a large number of at-
  tributes) 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 [6], EMF is characterized as “essentially the Class Diagram subset of UML”. Thus, Ecore itself is presented as an XML schema [8]; 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:

```xml
<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/members" 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 seman-
tics for XML [25] 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 [9,6], 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 [14].

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 eStructuralFeatures 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 [1] syntax. The translation to other set based frameworks like TLA [16], or Isabelle-ZF [23] 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 C_{sub} 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\(^6\):

\[
\begin{align*}
\text{SETS} \\
AObject /* universe of AADL objects */ \\
\text{CONSTANTS} \\
/* containers */ \\
\text{ComponentClassifier, ComponentImpl, ComponentType} \\
\text{DEFINITIONS} \\
\text{ComponentClassifiersub} == \\
(\text{ComponentClassifier} \cup \text{ComponentImplsub} \cup \text{ComponentTypesub}); \\
\text{PROPERTIES} \\
\text{ComponentClassifier} \subseteq AObject \\
/* containers are disjoint */ \\
\land \text{ComponentClassifier} \subseteq \\
\text{ComponentClassifiersub} - (\text{ComponentTypesub} \cup \text{ComponentImplsub}) \\
\land \text{ComponentImpl} \subseteq \\
\text{ComponentClassifiersub} - (\text{ComponentClassifier} \cup \text{ComponentTypesub}) \\
\land \text{ComponentType} \subseteq \\
\text{ComponentClassifiersub} - (\text{ComponentClassifier} \cup \text{ComponentImplsub})
\end{align*}
\]

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:

\[
\begin{align*}
\text{VARIABLES} \\
/* instances actually created */ \\
\text{instances}
\end{align*}
\]

\(^6\) In B, FIN(A) is the set of finite subsets of A; a DEFINITIONS introduce (parameterized) macros.
INVARIANT
instances ⊆ AObject

Remarks.
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.

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 ∈ Classsub ∩ instances → typeattribute ∧ ...

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:

classattribute($o$) := 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[20], 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 [4] 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 [22]. We would get a definition like

\[
\begin{align*}
\text{attrib_decl} &= \text{attrib_name} \times \text{attrib_type} \\
\text{class} &= \text{cname} \times \text{cname} \times \text{attrib_decl list}
\end{align*}
\]

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:

\[
\text{inst} = (\text{attrib_name} \times \text{value}) \text{ list}
\]

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 [21]. 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

\[
\begin{align*}
datatype & \quad \text{'a t}\_A = \text{c}\_A \text{ nat } \text{'a} \\
datatype & \quad \text{'b t}\_B = \text{c}\_B \text{ bool } \text{'b}
\end{align*}
\]

\[\text{types}\]
\[ \begin{align*} 
\text{a A} &= \text{A t}_A \\
\text{b B} &= \text{B t}_B t_A \\
\end{align*} \]

so a term \( \text{c}_A \mathbin{3} (\text{c}_B \text{True} ()) \) would be of type \text{unit} \ B. A function \text{get}_a, selecting the \text{a} attribute and defined as

\begin{verbatim}
consts
  get_a :: 'a A => nat
primrec
  get_a (c_A a x) = a
\end{verbatim}

would be applicable to elements of type \text{A} and type \text{B}.

This approach is elegant, but:

- The type system does not permit to express the fact that \text{A} only has specific subclasses, say \text{B} and \text{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 \[2,13,7\]. 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 \text{A} and \text{B}, we would get

\begin{verbatim}
types
  A = nat
  B = nat * bool
\end{verbatim}

which would make us lose any connection between \text{A} and \text{B}. In particular, we could not define an accessor function like \text{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:

\begin{verbatim}
consts
  get_a :: 'a => nat
defs (overloaded)
  get_a_A: get_a == id
  get_a_B: get_a == fst
\end{verbatim}

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 \( \text{Unit} \)) 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 \( \text{ComponentClassifier} \) class mentioned in Section 3.1, we get the type definitions

datatype
\[
  \text{s\_ComponentClassifier} = \\
  \quad \text{cs\_ComponentType} \; \text{t\_ComponentType} \\
  \quad | \; \text{cs\_ComponentImpl} \; \text{t\_ComponentImpl}
\]
datatype
\[
  \text{t\_ComponentClassifier} = \\
  \quad \text{ct\_ComponentClassifier} \; \text{oid} \; \text{s\_ComponentClassifier}
\]

where \( \text{s\_ComponentClassifier} \) codes the fact that this class has subclasses \( \text{ComponentType} \) and \( \text{ComponentImpl} \). The class is not abstract, otherwise we would have another constructor \( \text{cs\_Unit} \; \text{unit} \). We will return to the \( \text{oid} \) 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, \( \text{AObject} \) 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 \( \text{is\_ComponentClassifier} \). 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 \( \text{EReference} \) in Ecore). In order to model object identities, we introduce a type \( \text{oid} \) of object identifiers, which we implement as natural numbers to ease certain operations, such as creation of a “new” \( \text{oid} \):

constdefs
\[
  \text{new\_oid} :: \text{oid list} \Rightarrow \text{oid} \\
  \text{new\_oid os} == (\text{foldl max 0 os}) + 1
\]
We augment elements of the root class of the hierarchy, AObject, with an oid attribute

\[ t_{\text{AObject}} = \]
\[ \text{ct}_{\text{AObject}} \text{ oid } s_{\text{AObject}} \]

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

\textbf{types} AADL\_Model = t_{\text{AObject}} \textbf{ 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 \texttt{new\_Component} \( M \) \( \text{nm} \).

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 \( \text{im} \) is a ComponentImpl, all of its ports \( \text{po} \) are well-typed, and so are its subcomponent and flow components:

\textbf{consts} \\
\texttt{wt\_Component} :: [AADL\_Model, t\_Component] \rightarrow \text{bool} \\
\textbf{primrec} \\
\texttt{wt\_Component} \( M \) (ct\_Component \( \text{im} \) \( \text{po} \) \( \text{pa} \) \( \text{fl} \)) = \\
(\text{option\_lift (reftype} \( M \) \text{ is\_ComponentImpl) im}) \land \\
(\text{list\_all} \text{ is\_Port po}) \land \\
(\text{option\_lift (reftype} \( M \) \text{ is\_ComponentImpl) pa}) \land \\
(\text{list\_all} \text{ 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:

\[
\text{flow\_path} ::= \text{port} \ (\text{connection} \ \text{flow\_path})^* \ \text{connection} \ \text{port} \\
| \ \text{port} \ \text{port}
\]

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 connexion 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).

![Fig. 1. Simplified AADL flow model](image)

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.

\[
\text{ANY } fe, \text{ cnx WHERE} \\
\quad fe : \text{FlowElement - instances } \wedge \\
\quad \text{cnx : Connection - instances} \\
\text{THEN} \\
\ldots \\
\text{END}
\]

Declared objects are then added to the set of instances:

\[
\text{instances} := \text{instances} \cup \{fe,\text{cnx}\}
\]

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.

\[
\text{FlowElements}c\text{FlowPath}(fe) := \text{f.fp} \\
\mid \mid \text{FlowElementconnection(fe)} := \text{cnx}
\]

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 \text{fp} of the given component implementation.

\[
\text{Addfilter(imp, fp, filter) } = \\
\quad \text{PRE} \\
\quad \text{imp : (ComponentImpl } \cap \text{ instances) } \wedge
\]
fp: Componentflow(imp) \land fp : \text{dom}(\text{FlowPath}\text{connection}) \land 
filter : (\text{Component} \cap \text{instances}) \land 
card(\text{Componentport}(\text{filter})) = 2 \land 
\text{Portdir}[\text{Componentport}(\text{filter})] = \{\text{In,Out}\} \land 
\text{Componentflow}(\text{filter}) \subseteq \text{FlowPath} \land 
card(\text{Componentflow}(\text{filter})) = 1

\text{THEN}

\text{ANY } p\text{\_in, p\_out, fe, cnx, f\_fp, sc } \text{WHERE}
\quad p\text{\_in} : \text{Componentport}(\text{filter})
\quad \land p\text{\_out} : \text{Componentport}(\text{filter})
\quad \land \text{Portdir}(p\text{\_in}) = \text{In}
\quad \land \text{Portdir}(p\text{\_out}) = \text{Out}
\quad \land fe : \text{FlowElement} - \text{instances}
\quad \land cnx : \text{Connection} - \text{instances}
\quad \land f\_fp : \text{Componentflow}(\text{filter})
\quad \land sc : \text{SubComponent} - \text{instances}

\text{THEN}

\quad \text{instances} := \text{instances} \cup \{\text{fe,cnx,sc}\}

/* initialisation of the flow element */
\quad || \text{FlowElements}\text{cFlowPath}(fe) := f\_fp
\quad || \text{FlowElement}\text{connection}(fe) := cnx
\quad || \text{FlowElement}\text{sc}(fe) := sc
\quad || \text{Connection}\text{dst}(cnx) := p\text{\_in}

/* subcomponent insertion */
\quad || \text{SubComponent}\text{type}(sc) := filter
\quad || \text{SubComponent}\text{parent}(sc) := imp
\quad || \text{ComponentImpl}\text{subComponent} := \text{ComponentImpl}\text{subComponent}
\quad \quad \{\text{imp}\rightarrow sc\}

\quad || \text{IF } \text{FlowPath}\text{flowElement}(fp) \neq [] \text{ THEN}
\quad \quad \text{LET } fe1,cnx1 \text{ BE}
\quad \quad \quad fe1 = \text{first}(\text{FlowPath}\text{flowElement}(fp)) \land
\quad \quad \quad cnx1 = \text{FlowElement}\text{connection}(fe1) \ \text{IN}
\quad \quad \quad \text{Connectionsrc} := \text{Connectionsrc} \leftrightarrow \{cnx1 \leftrightarrow p\text{\_out},
\quad \quad \quad \quad cnx \leftrightarrow \text{Connectionsrc}(\text{FlowPath}\text{srcDst}(fp))\}\}
\quad \quad \text{END}
\quad \text{ELSE}
\quad \quad \text{Connectionsrc}(cnx) := \text{Connectionsrc}(\text{FlowPath}\text{srcDst}(fp))
\quad \text{END}

/* insertion of the filter */
\quad || \text{FlowPath}\text{flowElement}(fp) := fe \rightarrow \text{FlowPath}\text{flowElement}(fp)
\quad \text{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 sub-component 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. It seems that the set based approach is well suited to our concerns. However, due to the power of the frameworks usually available with higher order logics proof assistants (Coq, HOL, Isabelle, PVS), a pragmatic approach would be to work on top of an embedding of set theory in higher order logics. 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 [15,18,19,12] or towards OO specification mechanisms such as Object-Z or variants [24]. 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. We believe that the abstraction mechanisms usually available in logical frameworks are better suited.

acknowledgement: We thank Nicolas Lalevée for a careful reading.
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, INARIANT 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 [3] and Ocaml).

MACHINE flows_toy
SETS
  String; AObject;
  Direction = {In, Out}
CONCRETE_CONSTANTS
  FlowSource, FlowSink, FlowElement, NamedElement, Flow, Connection,
  Component, SubComponent, Port, ComponentImpl, FlowPath
DEFINITIONS
  FlowSourcesub ≜ (FlowSource);
  FlowSinkssub ≜ (FlowSink);
  FlowElementsub ≜ (FlowElement);
  NamedElementsub ≜ (NamedElement ∪ Flowsub ∪ Portsub ∪ Connectionsub ∪ ComponentImplsub ∪ Componentsub);
...
PROPERTIES
  FlowSource ⊆ AObject ∧
  FlowSink ⊆ AObject ∧
  FlowElement ⊆ AObject ∧
  NamedElement ⊆ AObject ∧
  Flow ⊆ AObject ∧
...
ABSTRACT_VARIABLES
  instances, FlowSourceport, FlowSinkport, FlowElementconnection,
  FlowElementsFlowPath, NamedElementname, Connectionsrc, Connectiondst,
  Componentimpl, Componentport, SubComponentparent, Componentflow, Portdir,
  ComponentImplnamed, ComponentImplspec, ComponentImplSubcomponent,
  ComponentImplflow, ComponentImplpconnection, FlowPathflowElement,
  FlowPathconnection, FlowPathsrcDst, SubComponentType
INARIANT
  instances ⊆ AObject ∧
  FlowSourceport ∈ (FlowSourcesub ∩ instances) → (Portsub ∩ instances) ∧
  FlowSinkport ∈ (FlowSinkssub ∩ instances) → (Portsub ∩ instances) ∧
  FlowElementconnection ∈
    (FlowElementsub ∩ instances) → (Connectionsub ∩ instances) ∧
...
INITIALISATION
  instances ::= ∅ || FlowSourceport ::= ∅ || FlowSinkport ::= ∅ ||
  FlowElementconnection ::= ∅ || FlowElementsFlowPath ::= ∅ || NamedElementname ::= ∅ ||
  Connectionsrc ::= ∅ || Connectiondst ::= ∅ || Componentimpl ::= ∅ || Componentport ::= ∅ ||
  Componentparent ::= ∅ || Componentflow ::= ∅ || Portdir ::= ∅ ||
...
OPERATIONS
  Addfilter(imp, fp, filter) ≜ /* see section IV. */
END
References


