 Electronic Notes in Theoretical Computer Science 141 (2005) 153–169 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Towards formalising AADL in Proof Assistants

# Jean-Paul Bodeveix[1](#_bookmark0) David Chemouil[2](#_bookmark0) Mamoun Filali[3](#_bookmark0) Martin Strecker[4](#_bookmark0)

*IRIT*

*Universit´e 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

# Introduction

This paper discusses design decisions for a formal meta-model of AADL (Ar- chitecture Analysis and Design Language) [[10](#_bookmark19)], a language that has become an official standard of the SAE [5](#_bookmark1) in October 2004. AADL is the synthesis of previous architecture description languages, such as MetaH [[17](#_bookmark26)], Acme [[11](#_bookmark20)] and Cotre [[5](#_bookmark14)]. As an architecture description language, AADL emphasizes as

1 Email: [bodeveix@irit.fr](mailto:bodeveix@irit.fr)

2 Email: [david.chemouil@cnes.fr](mailto:david.chemouil@cnes.fr)

3 Email: [filali@irit.fr](mailto:filali@irit.fr)

4 Email: [strecker@irit.fr](mailto:strecker@irit.fr)

5 Society of Automotive Engineers

1571-0661 © 2005 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2005.05.008

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](#_bookmark2). 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](#_bookmark3).
* 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](#_bookmark6) shows where we are heading. The paper concludes in Section [5](#_bookmark9) with a discussion of related work and possible extensions.

# Language Definitions in Ecore

Ecore is the meta-model of EMF, the Eclipse Modelling Framework. In [[6](#_bookmark15)], 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 [[8](#_bookmark17)]; 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 indi- cated by the negative *upperBound* ), NamedElement has a name attribute, apart from the comment attribute inherited from AObject. A Connection, with su- pertypes ModeMember and ReferenceElement, has three (non-inherited) at- tributes, 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](#_bookmark34)] 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](#_bookmark18),[6](#_bookmark15)], but again not in a systematic way.

Altogether, we are left on our own to give a precise meaning to the con- structs of Ecore, based on an informal understanding of the underlying con- cepts. This translation, which we will explore for different target formalisms in Section [3](#_bookmark3), 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](#_bookmark23)].

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 per- mits mutual dependencies between elements across package boundaries, which would not be acceptable for the systems used in Section [3](#_bookmark3).

# 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.

* 1. *Set based encodings*

In this section, we illustrate how an ecore description can be encoded in set theory. We use the B [[1](#_bookmark10)] syntax. The translation to other set based frameworks like TLA [[16](#_bookmark25)], or Isabelle-ZF [[23](#_bookmark29)] should be similar. In this section, we sketch the principles of data modeling. In section [4](#_bookmark6), 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 [6](#_bookmark5) :

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](#_bookmark6), 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

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 *n*×(*n*−1) derived predicates, such assertions should be auto- matically generated only for classes with a small number of derivatives.

2

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 over- loading notation of B:

classattribute(*o*) := new attribute value

* 1. *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](#_bookmark2), in Isabelle/HOL[[20](#_bookmark30)], 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; sec- ondly, we remain in the realm of the type systems of traditional programming languages such as ML, so that we get verified code executable [[4](#_bookmark11)] 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](#_bookmark31)]. 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 struc- ture 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](#_bookmark32)]. 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 [[2](#_bookmark12),[13](#_bookmark22),[7](#_bookmark16)]. Most of them are based on exten- sions 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](#_bookmark4), 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 Com- ponentClassifier. These are used for expressing some consistency condi- tions 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 Sec- tion [4](#_bookmark6)), 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.

# 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.

* 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 traver- sals. They are defined in the metamodel using an ordered sequence of flow elements referring one connection and one flow path. Moreover, the couple connexcion 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](#_bookmark7)).

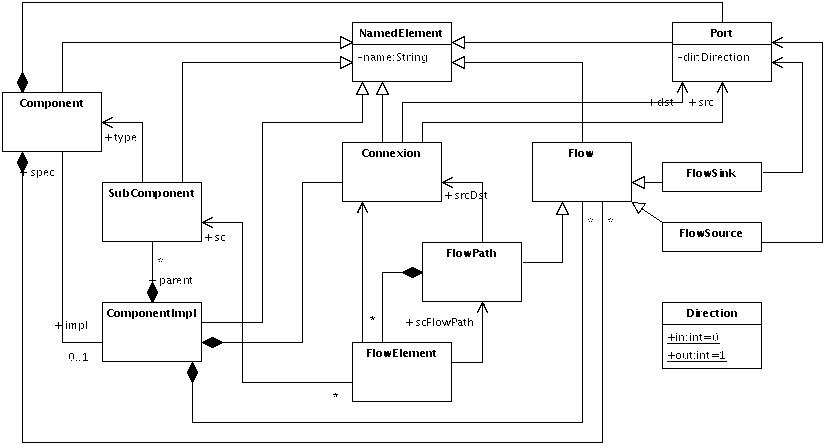


Fig. 1. Simplified AADL flow model

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

* 1. *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](#_bookmark8).

add\_filter(imp,fp,filter)

imp

fp

filter

imp

fp

Fig. 2. Adding a filter

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 - instances ∧

cnx : Connection - instances

THEN

...

END

Declared objects are then added to the set of instances:

instances := instances ∪ {fe,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 : (ComponentImpl ∩ instances) ∧

fp: Componentflow(imp) ∧ fp : dom(FlowPathconnection) ∧ filter : (Component ∩ instances) ∧ 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, sc 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)

∧ sc : SubComponent - instances

THEN

instances := instances ∪ {fe,cnx,sc}

/\* *initialisation of the flow element* \*/

|| FlowElementscFlowPath(fe) := f\_fp

|| FlowElementconnection(fe) := cnx

|| FlowElementsc(fe) := sc

|| Connectiondst(cnx) := p\_in

/\* *subcomponent insertion* \*/

|| SubComponenttype(sc) := filter

|| SubComponentparent(sc) := imp

|| ComponentImplsubComponent := ComponentImplsubComponent

{imp'→sc}

|| 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 ex- pressed 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.

# 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 embeddding 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](#_bookmark24),[18](#_bookmark27),[19](#_bookmark28),[12](#_bookmark21)] or towards OO specification mechanisms such as Object-Z or variants [[24](#_bookmark33)] . 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 mechanismes usually available in logical frameworks are better suited.

acknowledgement: We thank Nicolas Lalev´ee for a careful reading.

# A The B encoding

This section gives a sample of B code. Only the text of the transforma- tion, i.e., the B operation Addfilter (section [4](#_bookmark6)) has been written. The ma- chine clauses: SETS, CONCRETE CONSTANTS, DEFINITIONS, PROPERTIES, ABSTRACT VARIABLES, INVARIANT and INI-

TIALISATION 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](#_bookmark13)] 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); FlowSinksub (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, FlowElementscFlowPath, NamedElementname, Connectionsrc, Connectiondst, Componentimpl, Componentport, SubComponentparent, Componentflow, Portdir, ComponentImplunnamed, ComponentImplspec, ComponentImplSubcomponent, ComponentImplflow, ComponentImplconnection, FlowPathflowElement, FlowPathconnection, FlowPathsrcDst, SubComponentType

INVARIANT

instances ⊆ AObject ∧

FlowSourceport ∈ (FlowSourcesub ∩ instances) −→ (Portsub ∩ instances) ∧ FlowSinkport ∈ (FlowSinksub ∩ instances) −→ (Portsub ∩ instances) ∧ FlowElementconnection ∈

(FlowElementsub ∩ instances) −→ (Connectionsub ∩ instances) ∧

...

INITIALISATION

instances := ∅  FlowSourceport := ∅  FlowSinkport := ∅ 



FlowElementconnection := ∅  FlowElementscFlowPath := ∅  NamedElementname := ∅  Connectionsrc := ∅  Connectiondst := ∅  Componentimpl := ∅  Componentport := ∅  Componentparent := ∅  Componentflow := ∅  Portdir := ∅ 

...

OPERATIONS

Addfilter(imp, fp, filter) /\* see section IV. \*/

END

# References

1. J.R. Abrial. *The B-Book Assigning programs to meanings*. Cambridge University Press, 1996.
2. Mart´ın 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.
3. 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>.
4. Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In *Proc. TYPES Working Group Annual Meeting 2000*, LNCS, 2000.
5. 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.

1. Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.
2. Adriana B. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, University of Nijmegen, The Netherlands, January 1995. ISBN 90-9007860-6.
3. Worldwide Web Consortium. Extensible markup language (XML).

<http://www.w3.org/XML/>.

1. EMF: Eclipse Modeling Framework.

<http://download.eclipse.org/tools/emf/scripts/home.php>.

1. 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.
2. 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.
3. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In *Proc. Theorem Proving in Higher-Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 301 – 319. Springer Verlag, 2000.
4. Martin Hofmann and Benjamin Pierce. Positive subtyping. *Information and Computation*, 126(1):186–197, 1996.
5. SAE International. *Architecture Analysis & Design Language (AADL)*, August 2004.
6. Bart Jacobs, Claude March´e, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. 2004.
7. Leslie Lamport. *Specifying Systems:The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
8. MetaH. <http://www.htc.honeywell.com/metah/>. 1997.
9. Claude March´e, 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](http://krakatoa.lri.fr/).

1. Peter Mu¨ller. *Modular Speciﬁcation and Veriﬁcation of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.
2. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.
3. 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.
4. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universit¨at Mu¨nchen, 2001. [http://www4.in.tum.de/∼oheimb/diss/](http://www4.in.tum.de/~oheimb/diss/).
5. 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/>.
6. Thomas Santen. A theory of structured model-based specifications in Isabelle/HOL. In *Proc. Theorem Proving in Higher-Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 243–258. Springer Verlag, 1997.
7. Jerome Simeon and Philip Wadler. The essence of XML. In *Proc. POPL’2003*, 2003.

<http://homepages.inf.ed.ac.uk/wadler/topics/xml.html>.