2011 16th IEEE International Conference on Engineering of Complex Computer Systems

# An Implementation of the Behavior Annex in the AADL-toolset Osate2

Gilles Lasnier, Laurent Pautet
Inst. TELECOM - TELECOM ParisTech - LTCI
Paris, F-75634 CEDEX 13, France
Email: {firstname.lastname}@telecom-paristech.fr

Jérôme Hugues
ISAE - Toulouse University
Toulouse, 31056, France
Email: jerome.hugues@isae.fr

Lutz Wrage
SEI - Carnegie Mellon University
Pittsburgh, PA, 15213, USA
Email: lwrage@sei.cmu.edu

***Abstract*—AADL is a modeling language to design and analyze High-Integrity Distributed and Real-time systems. Embedded sub-languages published as AADL annexes extend an AADL model to enhance analysis. The behavior annex specifies the behavior of an AADL application model. Thus, an implantation of this annex allows to perform behavior analysis. In addition, as there are several AADL annexes, the implementation of generic mechanisms to support each one of them is challenging. The behavior annex is a valid candidate to illustrate these challenges by combining several sub-languages. In this paper we expose our experiment to support the behavior annex in the reference AADL toolset OSATE2. This one, supports the AADL version 2 by providing a front-end and a set of analysis plug-ins to analyze an AADL model.**

***Index Terms*—AADL; AADL-BA; behavior; annex; Osate2; MDD.**

## I. INTRODUCTION

The Architecture Analysis and Design Language (AADL [1]) is a Domain Specific Modeling Language (DSML) targeting High-Integrity (HI) Distributed and Real-time (DRE) systems. It allows the modeling, the analysis and the production of software system components [2] for distributed, reconfigurable, or partitioned systems.

The AADL core language is designed to be extensible and provides capabilities for users or tools to refine the semantics of an AADL application model and analysis capabilities using additional property sets or annex languages, like the error modeling annex or the behavior annex. They integrate approved external sub-languages to enhance DRE systems analysis.

The AADL Behavior Annex [3] is an extension to specify the behavior of an AADL application model. It refines the implicit behavior specified in the core of the language by attaching a behavioral specification (e.g. a state machine) to each AADL component. Thus, an implementation of this annex allows to perform behavior analysis on HI-DRE systems.

As there are several annexes to extend the analysis capabilities of an AADL application model, implementing generic mechanisms to support each one of them is challenging.

The behavior annex is a valid candidate to illustrate many challenges. 1) it requires to parse and analyse several sub-languages and not only one. 2) the different AST produced need to be connected to be analyzed. 3) to complete its analysis, it requires to ensure the consistency with the core language. 4) the internal representation of the annex needs to be compliant with the core language internal representation in order to provide a unique representation as input in analysis.

To assist the development of HI-DRE systems with AADL, the Software Engineering Institute (SEI) has developed the AADL-toolset OSATE2. This one is an extensible open source platform which includes an AADL front-end, architecture analysis capabilities and extension mechanisms to integrate external back-ends as plug-ins.

This paper presents our experiment to support the behavior annex as an extension plug-in [4] to the reference AADL-toolset OSATE2. The challenges relate the strategy chosen to specify the AADL-BA meta-model and the design of the AADL-BA compiler for its integration in OSATE2. We explain how we re-use several OSATE2 modules to drive the analysis of an AADL model completed with AADL-BA elements.

This paper is divided in five parts. Section II shows an overview of OSATE2 and the sub-language extension process; Section III exposes a brief overview and the challenges to implement the behavior annex; Section IV presents our strategy to specify the AADL-BA meta-model; Section V describes the implementation of the behavior annex and its integration in OSATE2; and Section VI presents concluding remarks and our future works.

## II. OVERVIEW OF THE AADL TOOLSET OSATE2

### A. Tool Architecture

The Open Source AADL Tool Environment (OSATE [5]) has been developed as a set of plug-ins to the ECLIPSE platform. OSATE2 is the version supporting AADL version 2. Its main components are illustrated in the figure 1.

The OSATE2 front-end includes a parser, name resolver, and semantic checker to process textual AADL models and translate them into an internal representation. This internal representation can be serialized back into textual AADL by an unparser or saved in an XMI-based format (AAXL2 files). Textual AADL models only declare components and must be instantiated (by the AADL instantiator) for analysis. Instance models are also persisted in XMI format.

To analyze AADL models, OSATE2 provides a set of analysis plugins. An analysis plugin is an ECLIPSE plugin that uses the OSATE2 infrastructure to access models, report errors, etc. Analysis plugins are typically implemented in Java, but other implementation languages can be chosen, e.g.,

978-0-7695-4381-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ICECCS.2011.39

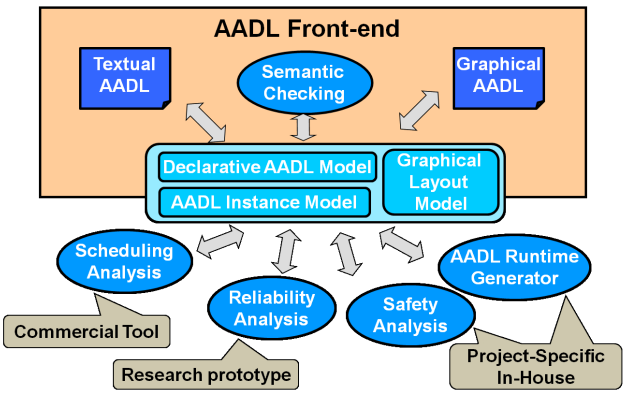


Fig. 1: OSATE2 architecture

Groovy [6]. External analysis tools can be integrated into OSATE2 by creating plugins that extract data needed by the tool from an AADL model and invoke the tool. Analysis tools may also access the XMI interchange format directly.

OSATE2 uses the AADL2 meta-model for the internal representation of AADL models (declarative and instance). This meta-model is defined in UML2 and implemented using the ECLIPSE/UML2 together with the ECLIPSE Modeling Framework (EMF). This automatically provides the XMI format and code to read and write the corresponding AAXL2 files built from AADL models.

### B. Sub-languages in AADL

AADL is extensible by (a) user defined property sets and (b) embedded sub-languages. Users can define new properties as part of a model and associate them with modeling elements. These properties can be evaluated during model analysis. The property set mechanism is used mainly to add those characteristics to modeling elements that can be expressed in the form of structured values such as strings, enumerated values, numeric values with or withour units, references to other model elements, and lists or records composed of such values. Sub-languages allow more complex structures to be added to an AADL model. A sub-language can be standardized and published as AADL annex. Several such annexes have been defined, for example, the error modeling sub-language is used to define error states and fault propagation for an AADL model, and the behavior annex allows modeling of detailed component behavior as a state machine.

A sub-language fragment is included in a textual AADL model as either an `annex library` or an `annex subclause`. An annex library is included directly in an AADL package. It contains declarations of annex elements that can be used in annex libraries and subclauses. In contrast, annex subclauses are included in classifier declarations, which makes the elements defined in the subclause part of the component. This is similar to the way in which core AADL classifiers are declared in a package and used in the declarations of other classifiers, for example, as subcomponents.

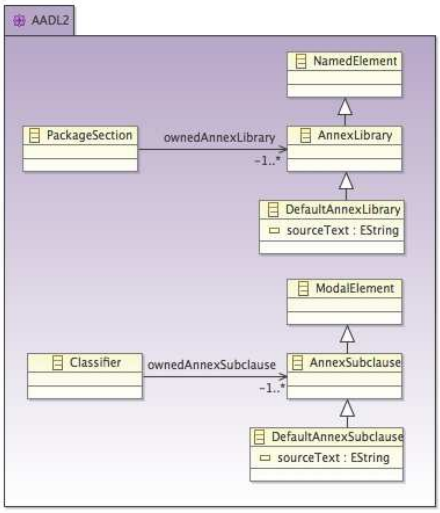


Fig. 2: OSATE2 annex extension

Annexes are separate from the core AADL in the following sense: if all annex libraries, subclauses, and annex-related property associations are removed from an AADL model, the resulting model is a valid core AADL model. Also, the different annexes are assumed to be independent of each other.

### C. Sub-languages in OSATE2

Support for a sub-language can be added to OSATE2 by creating one or more plug-ins that implement the necessary components. Typically, these include (a) an internal representation for annex libraries and subclauses, and (b) components to process the sub-language, e.g., a parser and unparser, name resolver, and semantic checker.

The core AADL meta-model contains two abstract UML classes that represent the top level annex elements: `AnnexLibrary` and `AnnexSubclause` (see figure 2). These classes must be extended by the meta-model for a sub-language. When the annex is implemented in ECLIPSE, the annex meta-model can be defined using UML2 and reference the core AADL meta-model. The UML2/EMF infrastructure in ECLIPSE can then be used to generate an implementation of the annex-metamodel that defines the internal representation for annex libraries and subclauses, and supports serialization and de-serialization in XMI.

OSATE2 provides a registry for sub-language processing components where plug-ins can register parsers, name resolvers, etc. The core language processing functions invoke the annex specific functionalities when needed, e.g., the core AADL parser invokes the behavior annex parser whenever it reaches a behavior annex subclause in the AADL source text.

As different annexes are treated as independent of each other and the core language, it is possible to ignore an annex when processing a model. OSATE2 supports this by defining a default annex meta-model, parser, and unparser. The default annex meta-model simply stores the source text of an annex library or subclause as a string, and the default parser and

unparser are trivial as they don't process the annex content. This allows processing of AADL models with annex elements even if the plug-ins for a particular annex are not installed.

## III. AADL-BA and Model-Driven Development

### A. Overview of the AADL Behavior Annex

The AADL Behavior Annex allows to attach a "behavior_specification", specified through five sub-languages, to each AADL component. Each of the sub-languages defines behavior concepts with: a syntax; a set of naming and legality rules to validate a behavior specification; a set of semantic rules to validate the specification towards its semantics; and a set of rules to ensure the consistency with the core language.

The figure 3 shows how the five sub-languages are combined to describe the behavior of the sporadic thread `Receiver`. At the reception of a data on its port `Data_Sink`, the sporadic thread increments the last value received with the new one and stores the result in a shared data.

```
1  thread Receiver
2  features
3   Data_Sink: in event data port dt;
4   Shared_Data : requires data access sd;
5  properties
6   Dispatch_Protocol => Sporadic;
7  end Receiver;
8
9  thread Receiver.impl
10 calls
11  toCall : { doUpdate : subprogram Shared_Data.Update; };
12 annex behavior_specification {**
13  variables
14   lastValue : dt;
15  states
16   stInit : initial state;
17   stExec : complete final state;
18  transitions
19   stInit_Exec : stInit -[ ]-> stExec { holdValue := 0 };
20   stExec_Exec : stExec -[ on dispatch Data_Sink ]-> stExec
21         { lastValue := lastValue + Data_Sink;
22           doUpdate!(lastValue); }
23  **};
24 end Receiver.impl;
```

Fig. 3: Sporadic thread behavior specification

The first language defines a state/transition automaton with variables, behavior condition (e.g., guards) and actions to describe the `Receiver` component execution behavior. It specifies the different kinds of states as the "complete" state `stExec` (line 17) which represents a suspend/resume state out of which the thread is dispatched.

A transition represents a change from the current state to a destination state and is activated when its behavior condition is evaluated to true. Then the action block ('{'...'}') attached to the transition is performed.

Behavior conditions are expressed as execution or dispatch conditions. Dispatch conditions are defined by a second language and refine the use of ports and subprogram calls involved as triggers in the AADL thread dispatch behavior [1]. In our example, the `Data_Sink` port defined in the AADL thread interface (line 3) is used to specify the dispatch condition in the behavior specification (line 20).

A third language defines interaction operations to refine the AADL component interactions described through AADL shared data, ports, subprogram calls, etc. Actions processed when a transition triggers are expressed through behavior actions (fourth language). This uses the interaction operations language to define actions between components as the call to the subprogram `doUpdate` (line 22) provided by the shared data to update its internal value.

The last language is an expression language which provides logical, relational and arithmetic expressions to manipulate AADL data components and behavior variables as the `lastValue` incremented by the value received on the `Data_Sink` port (line 21).

Finally, several approaches for defining or for interpreting the semantics of the AADL-BA have already been proposed [7], [8]. An implementation of an old AADL-BA specification based on AADL 1.0 has also been developed [9]. However, the several modifications made on the AADL meta-model to support its version 2.0 and the current AADL-BA draft do not allow to re-use this first experiment.

### B. Building Blocks to Implement AADL-BA

*a) MDD Approach:* A DSML such as AADL expresses concisely the common concepts of its domain. It provides several features as: an abstract syntax, described by a meta-model, which defines domain's concepts and their relations; concrete syntax(es) (textual or graphical) which define how concepts are represented; and semantics described by definition of transformations to map concepts with formal methods or target programming languages as Ada, etc. In such approach, the meta-model is elevated to a "central" and "governing" role.

AADL-BA is a DSML to express the behavior of HI-DRE systems and provides relevant elements to use an MDD approach for its implementation. As the annex defines several sub-languages we need to efficiently design its meta-model.

We observe that each of the AADL-BA sub-languages requires the use of AADL components described in the core language. Some languages as the state/transition language (resp. the behavior action) require the use of other AADL-BA sub-languages, e.g., the dispatch condition (resp. interaction operations) sub-language. These requirements are expressed as dependencies between the different sub-languages of the annex and the AADL core language. These dependencies have an impact on the strategy chosen to design the AADL-BA meta-model detailed in section IV.

*b) Rules Implementation:* The standard defines rules to verify that behavior concepts are "legal" and to verify their semantics. As a behavior concept defined in a sub-language can be combined with a behavior concept defined in another sub-language some rules can be specified in both sub-languages section. To avoid code duplication and simplify maintenance in our analyzers we select a subset of the rules which verify the whole behavior specification.

In the following sections we expose the AADL-BA meta-model and the architecture of the AADL-BA compiler.

## IV. The AADL-BA Meta-Model

In the previous section we showed a brief overview of the behavior annex which provides all the features required for an implementation using an MDD approach.

To design the AADL-BA meta-model we re-use the EMF framework utilized to specify the AADL meta-model. On the one hand its facilitates the integration of the AADL-BA in OSATE2 in terms of dependencies and embedded Java API. On the other hand, using the same formalism to specify the two meta-models eases the expression of the object dependencies and simplifies the navigation between them.

In the followings we describe the strategy and our rules to map an AADL-BA concept to an EMF meta-model element.

### A. A Single AADL-BA Meta-Model

In a classical meta-model development process a natural solution would be to define a meta-model for each AADL-BA language. As a consequence, it requires to express several dependencies between the different meta-models and to duplicate the dependencies with the AADL meta-model.

To ease the navigation between the objects of a behavior specification and the core language and to reduce the different dependencies we choose to develop a single meta-model for all languages specified in the AADL-BA annex.

### B. Mapping AADL-BA Concepts to Meta-Model Elements

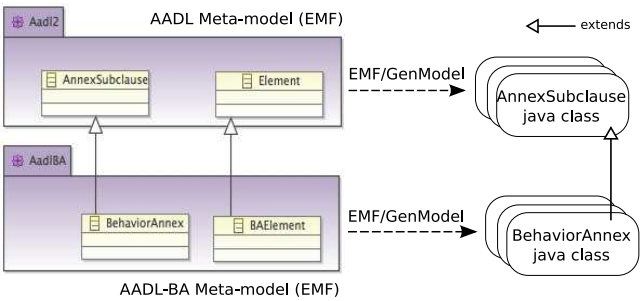


Fig. 4: AADL-BA meta-model dependencies

An AADL-BA EMF meta-model describes the structure of a behavior specification (e.g., AADL-BA model) and makes explicit all concepts expressed by the standard. Then, EMF generates the Java implementation classes corresponding to the meta-model objects (see figure 4).

To specify the AADL-BA meta-model we define some rules to map an AADL-BA concept to an object of the meta-model. These rules simplify the complexity and the generation of classes from the meta-model by focusing on 1) limiting the number of generated classes, 2) keeping only the relevant elements used to define a concept, and 3) facilitating their re-use in external analysis tools.

In the following we give the main rules:

**R1: Behavior concepts with strong semantics and concrete textual representation (e.g, a BNF) are mapped to meta-model EClass (e.g, a java class).** Several AADL-BA concepts as the `BehaviorAnnex` described in figure 5 come with a concrete textual representation and rules to describe precisely its semantic. Thus, it is quite natural to map these concepts as an EClass of the EMF meta-model.

**R2: Behavior concepts with weak or no semantic but used to clarify the concept hierarchy are mapped to abstract EClass.** Some concepts as the `BehaviorCondition` (see figure 5) do not have a concrete textual representation and a real semantic but are expressed to simplify and clarify the understanding of the annex. In this case, we map these concept as an abstract class. Their use is helpful to simplify the class hierarchy and the implementation of visitors in our analyzers and external model-based analysis tool.

**R3: Behavior concepts belonging to the same family are mapped with respect to the Java's inheritance mechanism.** Some concepts in AADL-BA as loop statements (e.g., for/forall, while, etc) share common attributes. In this case, their mapping defines parent as abstract EClass, subclasses as EClass using the Java's inheritance mechanism. This rule allows to factorize code and simplifies the compiler development and its maintenance.

**R4: Links to express that a behavior concept requires another behavior concept are mapped to EReferences.** As an EMF EReference represents one end of an association between two classes it is used to link (e.g., to reference) two behavior concepts.

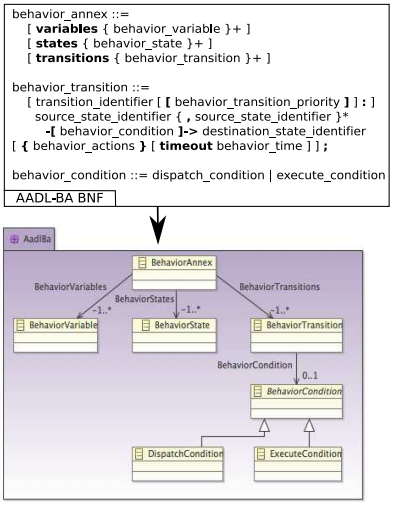


Fig. 5: AADL-BA: meta-model and BNF

### C. AADL Meta-Model Dependency

The behavior annex requires to attach a behavior specification to an AADL component and to link AADL-BA objects with AADL objects. It is reinforced by the implementation of the OSATE2 sub-language extension which requires to link a `BehaviorAnnex` object to the `AnnexSubclause` object of the AADL component (see figure 4).

In addition, the standard provides several rules to ensure the consistency between an AADL architecture component and its behavioral specification. These rules are verified by our different AADL-BA analyzers and require to navigate across the AADL-BA meta-model and the AADL meta-model.

Figure 4 shows how we use EMF extensions (e.g., Java's inheritance mechanism) to express the dependencies between the two meta-models. Thus, a `BehaviorAnnex` extends an `AnnexSubclause` and an `BAElement` extends an `Element`. The last one eases the navigation from an AADL-BA model to an AADL model by enabling to reference an AADL object in a behavior specification and to retrieve easily the corresponding AADL object during analysis.

In the next section we describe how we use the AADL-BA meta-model as backbone to build several modules involved in the architecture of the AADL-BA compiler.

## V. Implementation and Integration of the AADL-BA Compiler

The integration of the behavior annex in OSATE2 allows to re-use the AADL meta-model, the AADL front-end and the annex plug-in which implements the sub-language extension mechanism provided by OSATE2.

In this section, we detail the implementation of the AADL-BA compiler as an ECLIPSE plug-in integrated to OSATE2 and how the AADL-BA meta-model acts as backbone to build several modules of the compiler.

### A. Compiler Architecture

Figure 6 shows the "classical" AADL-BA compiler architecture with two parts: a front-end and a back-end. From the AADL-BA meta-model and EMF we generate the AADL-BA builder factory to build and manipulate AADL-BA objects used in the compiler. The front-end contains two modules: a parser and an analyzer. Error handling is managed by the OSATE2 error manager.

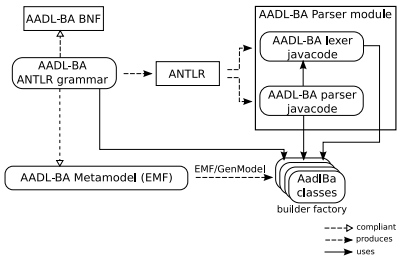


Fig. 7: AADL-BA parser

*1) Parser: AADL-BA BNF to* ANTLR *Grammar:* The figure 7 represents the architecture of the parser. We used the ANTLR framework to build the parser according to the AADL-BA BNF. The figure 8 shows that the rules defined by the BNF were easily mapped to the ANTLR grammar.

ANTLR allows to attach Java code declarations in grammar rules. Thanks to our rules R1, R2, R3 and R4 (see section IV) to specify the AADL-BA meta-model, we use only methods of the AADL-BA builder factory in the ANTLR grammar to specify how to build the abstract syntax tree (AST) with AADL-BA objects. It ensures that the AST is compliant with the AADL-BA meta-model. Finally we use the ANTLR framework to generate the Java classes of the parser and the lexer from the AADL-BA ANTLR grammar defined.

```
behavior_annex returns [BehaviorAnnex ba]
 @init{ ba = AadlBaF.createBehaviorAnnex();
        ba.setLocationReference(new LocationReference(this.getFilename(), input.get(0).getLine())) ;
} :
  ( VARIABLES ( bVars=behavior_variable { ba.getBehaviorVariables().add(bVars);} )+ )?

  ( STATES ( bStates=behavior_state { ba.getBehaviorStates().add(bStates); } )+ )?

  ( TRANSITIONS ( bTrans=behavior_transition {ba.getBehaviorTransitions().add(bTrans); } )+ )?
;
catch [RecognitionException ex] {
  reportError(ex);
  consumeUntil(input,SEMICOLON);
  input.consume();
}
```

```
behavior_annex ::=
   [ variables { behavior_variable }+ ]
   [ states { behavior_state }+ ]
   [ transitions { behavior_transition }+ ]
AADL-BA BNF
```

Fig. 8: AADL-BA BNF to ANTLR grammar

*2) Naming, Legality, Consistency and Semantic Analyzers:* The analyzer module scans the AST and checks the semantics of the AADL-BA model. First, it proceeds to a resolution phase (e.g, naming resolver) which links AADL-BA objects to their corresponding AADL objects or AADL-BA objects (see figure 3, source identifier `stExec` and transition `stExec_stExec`). To achieve this phase we use the visitors (e.g, java classes) provided by OSATE2 to retrieve AADL objects. For AADL-BA we have developed the visitors required to navigate through the AADL-BA AST. This phase adds information to the AST and makes its use easier.

Secondly, we have implemented the subset of rules sufficient to verify a whole behavior specification. The second phase proceeds to the verification of this subset of legality, semantic and consistency rules.

To verify the consistency with AADL components we have developed some specific visitors to navigate across the AADL model. The result of this analyze leads to an AST conforms to the AADL-BA semantics and consistent with the AADL model. This AST acts as internal representation of the annex and can be used as input in the back-end part or external tools.

We chose the same strategy to implement our analyzers. We consider the whole behavior specification (see figure 3). First, we analyse variables and states. Then, we analyze the structure of transitions, behavior conditions and finally the action block attached to the transition.

*3) Back-end Integration:* The AADL-BA is a formalism to describe the behavior of an AADL application model. An implementation of the standard only focuses on the verification of the syntax and the semantic of a behavior specification. Thus, it is possible to describe a "legal" thread behavior specification with deadlocks. To verify these kinds of properties we need the use of model checker.

To achieve this goal we provide a back-end registry. Thanks to the ECLIPSE plug-in extension points, we allow to register an external ECLIPSE-based plug-in as back-end and to use as input the AST produced from the AADL-BA front-end.

We have also developed an unparser back-end to produce from an AADL-BA AST the corresponding textual behavior specification. It allows us to verify the different modules of our compiler by building intermediate models. It is also used to produce the different outputs of our test-suite which is constituted by examples of the standard and AADL-BA models developed by our team.

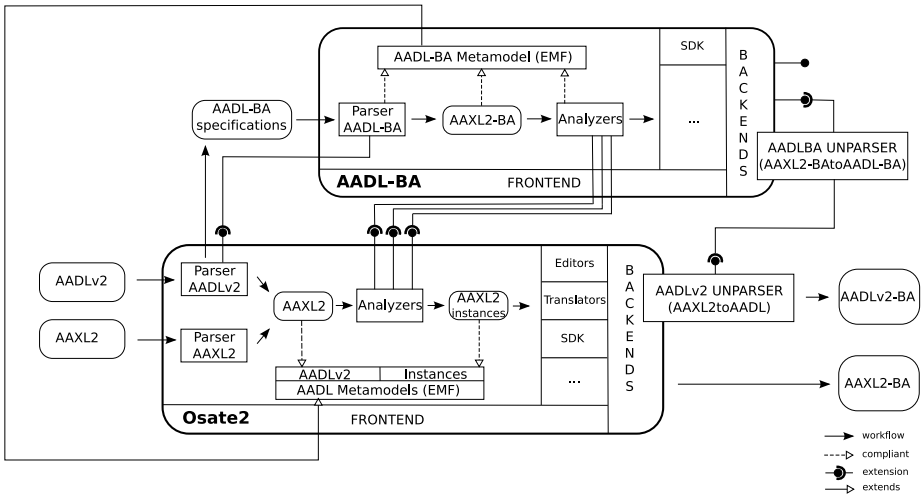


Fig. 6: AADL-BA plug-in integrated to OSATE2

### B. Integration in OSATE2

The figure 6 shows the integration of our behavior annex plug-in in OSATE2.

*1) Plug-in Extensions:* The integration of the AADL-BA plug-in is a two-steps process. First, we link the AADL-BA plug-in to the OSATE2 annex plug-in. The annex plug-in defines ECLIPSE extension points which allow plug-ins to be connected together. A plug-in extends another by declaring an extension. In our AADL-BA plug-in we declare the extension corresponding to the extension points declared by OSATE2 to extend the AADL parser and analyzers.

Second, we have to register our parser (resp. analyzers) in the OSATE2 annex registry. As the AADL-BA is a part of the AADL description, the AADL-BA plug-in is not a back-end but is directly integrated and driven by OSATE2.

*2) Outputs and Back-end Extension:* The AADL-BA AST is the internal representation of the behavior annex. As the AADL-BA meta-model uses the same formalism of the AADL meta-model it facilitates the integration of this representation in the internal representation of the AADL model.

We also support serialization and de-serialization in XMI following the same XML schema. It allows to produce a unique AAXL2 representation including the serialized behavior annex objects. Thus, we provide a persistent representation including the whole description AADL architecture + behavior annex. It can be used as inputs in different back-ends. Figure 6 describes how OSATE2 uses the AADL unparser back-end to produce an AADL textual description from an AAXL2 file. It provides the extension point to extend this back-end with the different annex unparsers. We use this extension point to link our AADL-BA unparser and to produce an AADL textual model including textual behavior specifications.

## VI. CONCLUSION AND FUTURE WORK

This paper presented our implementation of the AADL behavior annex as ECLIPSE plug-in [4]. We showed how we specified the AADL-BA meta-model used as backbone to develop several modules of our compiler. As the behavior annex defines several simple interconnected sub-languages, the design of a single meta-model allows to produce and to analyze a unique AST. This internal representation of an AADL-BA model is used as input in external back-ends which are integrated using our back-end registry.

Our plug-in is integrated in the reference AADL toolset OSATE2. This one provides an AADL front-end and an annex plug-in which drives the behavior annex analysis (parser+analyzers). The same technologies used to define the AADL and the AADL-BA meta-models ease the navigation across both meta-models and the production of a unique persistent XMI representation (AADL model+behavior elements) which facilitates its use as input in external back-ends.

Our future work will focus on the analysis of behavior automaton properties by integration of external back-ends e.g., as model checker to verify deadlock and model-based tool to enhance scheduling analysis by refining WCET estimation and blocking time on shared resources, thanks to the behavior annex capabilities.

## REFERENCES

[1] SAE, *AADL v2.0 (AS5506)*, Sep. 2008.
[2] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the Prototype to the Final Embedded System using the Ocarina AADL Tool Suite," *ACM Transactions in Embedded Computing Systems (TECS)*, vol. 7, Jul. 2008.
[3] SAE, *Annex X Behavior Annex (AS5506-X draft-2.13)*, Aug. 2010.
[4] Telecom ParisTech, "AADL-BA Project," http://eve.enst.fr/aadlba, 2010.
[5] SAE AADL, "OSATE," http://www.aadl.info, 2010.
[6] D. Koening, *Groovy in Action*. Manning Publications Co., 2007.
[7] Y. Ma, J.-P. Talpin, and T. Gautier, "Interpretation of AADL Behavior Annex into Synchronous Formalism Using SSA," *International Conference on Computer and Information Technology*, pp. 2361–2366, 2010.
[8] Z. Yang, K. Hu, D. Ma, and L. Pi, "Towards a Formal Semantics for the AADL Behavior Annex," in *Design, Automation Test in Europe Conference Exhibition - DATE'09.*, Apr. 2009, pp. 1166 –1171.
[9] R. Frana, J.-P. Bodeveix, M. Filali, and J.-F. Rolland, "The AADL Behaviour Annex – Experiments and Roadmap," in *Engineering Complex Computer Systems, 12th IEEE International Conference on*, Jul. 2007, pp. 377 –382.