International Journal of Hybrid Information Technology
Vol.8, No.4 (2015), pp. 177-184
http://dx.doi.org/10.14257/ijhit.2015.8.4.20

Implementation of AADL Interpreter Based on K

Fan Zhang, Weining Su, Yang Gang and Tianfang Wang

School of Computer Science and Engineering
Northwestern Polytechnical University
Xi'an China
zhangfan@nwpu.edu.cn

Abstract

By using model-driven architecture (MDA), most of errors can be discovered and solved at the early stage of system design. AADL lacks formal semantics which are essential for real-time embedded systems with high safety requirements. In this paper, we design and implementation the AADL Interpreter based on K semantics framework, and this lays a solid foundation for formal analysis and verification of AADL model.

Keywords: *MDA; AADL; Formal Semantics; Interpreter*

1. Introduction

Model is an abstract description of the systems, and modeling becomes the basic means of controlling complexity of system. Model Driven Architecture (MDA) [1] makes information systems based on MDA model can be implemented on different platforms through the mapping standards of models. Using modelling techniques, MDA effectively integrate a full range of cross-platform applications, and support heterogeneous platforms on interaction between systems and make the system having strong robustness and evolvability.

UML [2] and SysML [3] as MDA method are widely used in the Unified Modeling Language. And AADL [4] is a MDA-completed architecture modeling language, which can be used to describe the analysis of real-time and high-confidence embedded systems. System designers can make reliability, security and real-time performance for various analysis and evaluation.

Formal methods are a particular kind of mathematically based techniques for the specification, development and verification of software and hardware systems. K framework [5] is a formal semantic platform based on rewriting logic, and the platforms supported language K is a structured form of modular languages. K makes up for the limitations of existing analysis methods of existing formal semantics, and defines the concurrency features of programming though a structured and modular feature. In this paper, we use K to describe the AADL model and analysis the AADL model. This lays a solid foundation for the correctness verification of the transformation from system model to program code.

2. Related Work

AADL has been the common concerns by academia and industry since its introduction and AADL research is mainly used in aviation, aerospace systems.

Open Source AADL Tool Environment (OSATE) [6] provides analysis tools for error analysis, model analysis and scheduling analysis, and it can be used to analyze and simulate the scheduling for AADL models. Furthermore, Furness [7] developed by Fremont Associates company and Cheddar [8] developed by France Brest University also analyze and simulate the scheduling for AADL models. Ocarina [9] tool supports code

ISSN: 1738-9968 IJHIT

generation for AADL models currently. Ocarina is an open source tool which developed in Ada and AADL models can be generated to Ada or C code by this tool. However, it requires the user to comply with Ocarina AADL format when writing code. Northwestern Polytechnical University, the Embedded Laboratory has developed Embedded Software Model Evaluation and Analysis Tool (ESME) [10] which provides the ability to use AADL for system models design and high-assurance property analysis.

Currently, there are mainly two types of formal semantics for program language: operational semantics [11] and axiomatic semantics [12]. Operational semantics is easy to define and understand, therefore, defining operational semantics for languages can be thought of as defining a "formal interpreter". But it is difficult for operational semantics to verify a program. Axiomatic semantics can formally analyze high-level language directly, but axiomatic semantics can't run and test the formal definition of program. In order to make up for the two types of semantics' flaws, Grigore Rosu and his team from University of Illinois at Urbana-Champaign design and realize the K framework [16] which is a formal definition framework.

K, supported by K framework, is a language for describing the formal semantic. And it is based on match logic [13]. K can be used to define the formal semantic for programming languages, calculi, as well as type systems or formal analysis tools. And they has defined the formal semantics for C [14], Python [15], Scheme [16] and Java [17]. Configuration, computations and rules are three main modules in K. Configurations organize the system/program state in units called cells, which are labeled and can be nested. Computations carry “computational meaning” as special nested list structures sequentializing computational tasks. Rules generalize conventional rewrite rules by making explicit which parts of the term they rewrite. K supports to describe languages or system with a context-free grammar and follows these three rules while formally describe the languages:

- Put the definition of abstract grammar into computations. So it can reduce the semantics of refocusing [18] in evaluating the context structure.
- Put the status of executable program or system into configurations which contains the nested units called cells. This model has been widely used [19] in the chemical abstract machine (CHAM)
- Put the semantic of executable program or system into rules. This contributes to giving the semantic structure of a concurrent language accurately.

3. AADL Interpreter

AADL describes the software and hardware architecture though some concepts such as component and connection, and defines systemic-functional and non-functional characters though features and properties, also can use annex to extend the AADL model with user-defined attributes. Through model transformation, AADL can describes runtime architecture evolution. AADL defines three categories of components: software components, platform components and system components. Software components is used to model the software architecture, and platform components is to model hardware architecture, and system components combine all the components to organize hierarchical systems architecture. In this section, we describe how to use K to build the AADL interpreter, and this lays a solid foundation for defining AADL formal semantic.

In the AADL standards, component is defined in two parts: component type and component implementation. A component has one type but has zero or more related implementations. Component type describes the function interface such as input/output ports and component implementation describes the component's internal structure such as subcomponent and connections. The features of component include: identity, interface with other components, inherent properties, as well as the relationship between subcomponent and their interactions.

In Figure 1, we use AADL graphic symbols to illustrate the expansion of the hierarchical architecture.

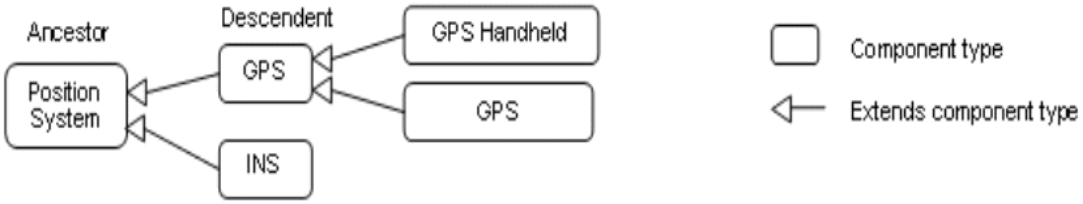

Figure 1. Extends Component Type Hierarchy

The GPS expands the Position System component type by inheriting the ports declared in Position System. It can add a port, refine a declared data type classifier in the Position System ports which will cover one or more property values. Figure 2 is a component type system defines the specification grammar for AADL.

```
component_type ::=
component_category defining_component_type_identifier
[ prototypes ( { prototype }+| none_statement ) ]
[ features( { feature }+| none_statement ) ]
[ flows( { flow_spec }+| none_statement ) ]
[ modes_subclause | requires_modes_subclause ]
[ properties(
{ component_type_property_association | contained_property_association }+
| none_statement ) ]
{ annex_subclause }*
end defining_comonent_type_identifier;
```

Figure 2. AADL Component Type System

Blue font are keywords and corner marked "+" indicates that this structure occurs at least once, and corner marked "*" indicates 0 or more times. For example, prototype in prototypes occurs one or more times, and annex_subclause structure can occur 0 or more times. The structure in "[]" may be omitted, for example in Figure 2, an instance of component type maybe has not prototypes, features flows or properties. The structure ”(term1 | term2)”means that here may be term1 or term2. We use a top-down approach to define AADL models, firstly defining the integral structure of component type, and then gradually defining individual properties. Figure 3 is the integral structure of the component type definition.

```
syntax ComponentType ::= ComponentCategory ComponentTypeIdentifier ComponentAssocDefination "end"
ComponentTypeIdentifier ";"
```

Figure 3. AADL Component Type System Definitions

Figure 4 is the refinement definition of AADL component.

```
syntax ComponentAssocDefination ::= "prototypes" ProtoType
                                  | "prototypes" NoneStatement
                                  | "features" Feature
                                  | "features" NoneStatement
                                  | "flows" FlowSpec
                                  | "flows" NoneStatement
                                  | ModesSubclause
                                  | RequiresModesSubclause
                                  | "properties" ComponentTypePropertyAssociation
                                  | "properties" ContainedPropertyAssociation
                                  | "properties" NoneStatement
                                  | AnnexSubclause
```

Figure 4. Refinement Definition of AADL Component

In Figure 5 we refines the definition of a ProtoType based on the standard AS5506B AADL, More detailed grammar definitions see: https://github.com/FormalADL/kAADL

```
syntax ProtoType ::= Id Mcolon ProtoTypeDef MSemicolon
syntax ProtoTypeDef ::= CmpProtoType
                      | FetGroupTypeProtoType
                      | FetProtoType
syntax CmpProtoType ::= CmpCategory
syntax CmpProtoType ::= CmpCategory
                      | CmpCategory MDoubleBracket
                      | CmpCategory UniCmpClassifierRef
                      | CmpCategory UniCmpClassifierRef MDoubleBracket
syntax CmpCategory ::= AbstractCmpCategory
                      | SoftwareCategory
                      | ExecPlateformCategory
                      | CompositeCategory syntax SoftwareCategory ::= Mdata
syntax SoftwareCategory ::= Mdata
                      | Msubprogram
                      | Msubprogram Mgroup
                      | Mthread
                      | Mthread Mgroup
                      | Mprocess
syntax Mdata ::= "data"
syntax Msubprogram ::= "subprogram"
syntax Mgroup ::= "group"
syntax Mthread ::= "thread"
syntax Mprocess ::= "process"
```

Figure 5. Part Definition of Component

4. Test of Interpreter

4.1 Abstract Syntax Tree

AADL interpreter tests are divided into single modules tests and integral module tests. And a single modules test is to test syntax errors for single module definitions while an integral module test is to test the interactive relationship between multiple modules. K framework offers tools called KAST that can generate abstract syntax tree which support for interpreter test.

The program is interpreted as an abstract syntax tree which includes detailed information in the program such as strings and comments. The abstract syntax tree in K framework includes the node name, definition lists, and other information. After defining the specific AADL model, the main objective is to generate abstract syntax without ambiguity. And if the specific syntax of AADL model is unambiguous, the abstract syntax tree is single. Instead, if the specific syntax of AADL model is ambiguous, there will be listing the different abstract syntax tree at ambiguous position. The abstract syntax tree provides the location and causes of the ambiguity, which is useful for ambiguity

elimination. Figure 6 is a part of an abstract syntax trees with ambiguity while testing component implementations model.

```
PpeAssocList ::= PpeAssoc PpeAssocList
  ((B :: Dispatch_Protocol) => constant Periodic ;)
  amb(
          ((B :: Period) => constant (10 ms) ;) ((B :: Source_Name) => constant
            CCActive applies to CruiseActive ;),
          ((B :: Period) => constant (10 ms) ;) ((B :: Source_Name) => constant
            CCActive applies to CruiseActive ;)
              )
```

Figure 6. The Ambiguity of the Abstract Syntax Tree (partial)

The ambiguity indicates that there are two different methods to derive the same sentence from the product "PpeAssocList ::= PpeAssoc PpeAssocList".

4.2 Eliminate Ambiguity

AADL grammar as a context-free grammar (CFG) whose ambiguous is undecidable, so removing ambiguity is an essential part in the system testing. There are a lot of ambiguity in the testing process and some of which are innate ambiguity. Below is a typical congenital ambiguity:

syntax PpeValue ::= SinglePpeValue | PpeListValue

PpeValue in the abstract syntax tree has two branches: SinglePpeValue and PpeListValue and the two branches should not derive the same sentence. However,Figure 7 is one path in the abstract syntax tree to derive the product:syntax PpeValue ::= MBracketLeft BooleanTerm MBracketRight

```
syntax PpeValue ::= PpeListValue
syntax PpeListValue ::= MBracketLeft PpeListValueMidSymbol MBracketRight
syntax PpeListValueMidSymbol ::= PpeExpression
syntax PpeExpression ::= BooleanTerm
```

Figure 7. One Path of PpeValue in the Ambiguity Syntax Tree

The path in Figure 8 also gets the product:syntax PpeValue ::= MBracketLeft BooleanTerm MBracketRight.

It is hard to find out when we define AADL grammar inherent ambiguity terms from AADL syntax definition. But it will come out at testing phases. In above, BooleanTerm scope of production is much less than the PpeValue and we remove sub-product from BooleanTerm grammar definition: syntax BooleanTerm ::= MBracketLeft BooleanTerm MBracketRight. And then eliminate the effect of this change on the overall. At this point we need to consider the scope of BooleanTerm which only appears in the definitions of the nonterminal PpeExpression. Therefore, we should add a sub-product, MBracketLeft BooleanTerm MBracketRight, into the scope of PpeExpression.

```
syntax PpeValue ::= SinglePpeValue
syntax SinglePpeValue ::= PpeExpression
syntax PpeExpression ::= BooleanTerm
syntax BooleanTerm ::= MBracketLeft BooleanTerm MBracketRight
```

Figure 8. Other Path of PpeValue in the Ambiguity Syntax Tree

For example, adding the sub-product into DefaultPpeExpression:

```
syntax DefaultPpeExpression ::= PpeExpression
```

change into:

```
syntax DefaultPpeExpression ::= PpeExpression
                              | MBracketLeft BooleanTerm MBracketRight
```

5. Conclusion and Future Work

In this paper, we introduce how to describe AADL grammar with the K framework platform, and realize the AADL grammar definition as well as tests and eliminates the inherent ambiguity.

AADL lacks formal semantics which are essential for real-time embedded systems with high security requirements. Therefore, it is significant to define the formal semantics for model validation. Besides, the AADL model without formal semantic is not executable which limits the formal analysis of the safety and effectiveness of the model. It easy for K framework platform to define formal semantic of program language and configuration provides a lot of flexibility for defining formal semantic. In the future, we will focus on AADL formal semantics definition based on Kframework, and it can be used in the formal analysis of AADL models.

Acknowledgements

This work is supported by the National Science Foundation of China (No. 61472327). We gratefully acknowledge the financial and technical support from the committee of NSFC. We also wish to thank the anonymous reviewers for their constructive comments.

References

[1] A. G. Kleppe, J. Warmer and W. Bast, "The model driven architecture: practice and promise", & Explained, M. D. A. **(2003).**
[2] C. Larman, "Patterns: an introduction to object-oriented analysis and design and iterative development", Applying, U. M. L. **(2004)**.
[3] S. Friedenthal, A. Moore and R. Steiner, "A practical guide to SysML: the systems modeling language [M]", Elsevier, **(2011).**
[4] SAE, Architecture Analysis & Design Language (AADL), SAE Standards AS5506, **(2004)**.
[5] G. Roşu and T F. Şerbănuţă, "An overview of the K semantic framework", The Journal of Logic and Algebraic Programming, **(2010)**, vol. 79, pp. 397-434.
[6] M. Kerboeuf, A. Plantec, F. Singhoff, A. Schach and P. Dissaux, "Comparison of six ways to extend the scope of Cheddar to AADL v2 with Osate', In Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference, IEEE, **(2010)** March 22-26; Oxford, English.
[7] B. Sun, Y. Dong and H. Ye, "On Enhancing Adaptive Random Testing for AADL Model", In Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2012 9th International Conference on (pp. 455-461). IEEE. (2012), September 4-7 Fukuoka, Japan.
[8] F. Singhoff, J. Legrand, L. Nana and L. Marcé, "Cheddar: a flexible real time scheduling framework", In ACM SIGAda Ada Letters, vol. 24, **(2004),** pp. 1-8.
[9] J. Hugues, B. Zalila, L. Pautet and J. Kordon, "From the prototype to the final embedded system using the Ocarina AADL tool suite", ACM Transactions on Embedded Computing Systems (TECS), vol. 7, **(2008)**, pp. 42-65.
[10] Y. Dong, Y. Cheng, T. Wu and H. Ye, "On Schedulability Analysis for Embedded Systems with AADL Model", In Quality Software (QSIC), **(2013)** July 29-30; Nanjing, China.
[11] A W. Roscoe, "Operational Semantics [M]", Understanding Concurrent Systems. Springer London, vol. 191, no. 228, **(2011)**.

[12] N. Soundararajan, "Axiomatic semantics of communicating sequential processes", ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 6, **(1984),** pp. 647-662.
[13] G. Roşu and A. Ştefănescu, "Matching logic: A new program verification approach", in: ICSE (NIERTrack), **(2011)**, pp. 868,871.
[14] Ellison, C. and G. Roşu, "An executable formal semantics of C with applications", in: POPL **(2012),** pp. 533–544.
[15] D. Guth, "A Formal Semantics of Python 3.3", Master's thesis, University of Illinois at Urbana Champaign **(2013)**.
[16] P. Meredith, M. Hills and G. Ros, "An executable rewriting logic semantics of K-Scheme", in: SCHEME, vol. 91, no. 103, **(2007)**.
[17] D. Bogdanas, G. Rosu , "K-Java: A Complete Semantics of Java", appears in POPL'15, ACM. **(2015)**.
[18] O. Danvy and L. R. Nielsen, "Refocusing in reduction semantics", Technical Report BRICS RS-04-26, University of Aarhus **(2004)**.
[19] J.-P. Banâtre and D. L. Métayer, "The GAMMA model and its discipline of programming", Sci. Comput. Programming, vol. 15, **(1990),** pp.55–77.

Authors

Fan Zhang, he was born in 1979. He is a PH.D service in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include modeling, verification and analysis for Large-Scale Complex Embedded system.

Weining Su, he was born in 1989. He is a master in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include modeling, verification and analysis for Large-Scale Complex Embedded system.

Yang Gang, he was born in 1974. He is a PH.D service in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include distributed embedded computing system modeling technology and the CPS system collaborative design technology.

Tianfang Wang, he was born in 1992. He is a master in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include modeling, verification and analysis for Large-Scale Complex Embedded system.