K Semantics for Dynamic Software Architectures

Sahar SMAALI , Aicha CHOUTRI , Faiza BELALA
Lire Laboratory, University of Constantine, Algeria
sahar.smaali@gmail.com, aichachoutri@gmail.com, belalafaiza@hotmail.com

Abstract: Recently, several research works propose multi formalisms based models to define rigorously the static and dynamic concepts of a software architecture. Further, these models are used to assess the architectural quality and to predict final system characteristics. The objective of this paper is to address both structural and behavioral viewpoints, involved in software architecture specification, with a unique formal model. Our contribution is twofold; on the one hand, it gives an operational semantics definition based on rewriting logic, to the architecture structural constructs and their behavior actions. On the other hand, we define a transparent ADL-Maude translation. Thus, the resulting Maude architectural specification is executable and analyzable even if the code is unfamiliar to the user.

Keywords: Architecture Description Language; Behavior; Operational semantics; Maude; K-Framework, Model checking.

1. Introduction
Software architecture denotes a high-level abstraction of complex software systems. From a runtime perspective, its description is defined as a set of software elements, the relations between these elements, and all associated properties needed to reason about the system. Further, a dynamic software architecture description must also specify how the architecture can change during system evolution.

The common advantage of Architecture Description Languages (ADL), defined as formal notations, is their impressive body representation of evidence about the architectural modeling and analysis utility. In fact, studies mainly based on [13], which is a reference, in the domain, have raised some flaws as the behavior description lack and semantics basis weakness. This makes hard the formal analysis and verification of description models [5]. Therefore, enabling specification of software architecture behavior is actually a large challenge for an ADL.

Many recent research works addressed that question. They generally propose multi formalisms based models, to define rigorously the semantics of complete software architecture (i.e. its static and dynamic viewpoints). These models are used to assess the architectural quality and to predict final system characteristics. However, several challenges remain unsolved. The most important are:

1) Modeling concurrent, distributed and time-dependent behavior of software architecture components.
2) Modeling components communication with delays.
3) Modeling components and connectors changes or moves at runtime.
4) Analyzing behavior properties with respect to some declared constraints.
5) Providing automated tools for specification and verification of software architecture.

In this paper, we aim to solve some of these challenges by integrating in K-framework a unique semantic model so that we can formally reason on dynamic software architectures. K-framework [10] was introduced as a semantic framework in which programming languages, calculi, as well as type systems or formal analysis tools can be defined. Our contribution is twofold; on the one hand, it gives an operational semantics definition to the architecture structural constructs and their actions behavior (which may be time-dependent). On the other hand, we define a transparent translation from a given ADL to Maude [6], a rewriting logic based language, dedicated to specify concurrent state changes. The main advantage of this translation is the possibility of executing and analyzing the resulting Maude architectural specification, even if the code is unfamiliar to the user.

The remainder of this article is organized as follows. Section 2 presents an overview on K tool, focusing on its fundamental elements and notations, through examples of a simple arithmetic language definition. In Section 3, we explain our general K based ADL definition approach; then we illustrate our ADL-Maude translation through an extended ACME [7] case study. We show how static architecture could be specified and additional dynamic aspects could be also defined, even if the considered ADL neglects the software architecture behavior. Section 4 is devoted to exploit our formal model to simulate and analyze dynamic architecture descriptions. Section 5 compares our proposed approach with related work. Finally, section 6 summarizes the main contributions of this paper.

2. The K Framework

K was initiated by Grigore Rosu in 2003 and completely developed in 2010 [10]. It is a semantic framework based on rewriting logic. It provides executable Maude specifications for programming
languages, type systems and formal analysis tools using configurations, computations and rules. Its general objective is to demonstrate that a formal specification language for these systems can be simple, expressive, analyzable, and executable at the same time [15].

Figure 1 represents the K-framework architecture. The gray arrows denote translator tools implemented as part of the K-Framework toolkit. The file “k-prelude.maude” contains several Maude [6] modules that are handy in most language definitions, such as ones for defining computations, configurations, environments, stores, etc. The “K-Maude” interface is what the user typically sees: besides usual Maude module (K-Maude fully extends Maude), one can also include K modules comprising syntax, semantics or configuration definitions using the K notation. A first component of K-Maude tool translates K modules to Maude modules. These later encode K-specific features as meta-data attributes and serve as an intermediate representation of k-definitions. This intermediate representation can be further translated to various backends: executable/analyzable Maude modules, which can serve as a basis for formal analysis, or LATEX files for documentation reasons.

K language definitions are given as K-modules, entirely inspired from Maude modules. Each K-module imbricates two sub-modules: one for the syntax definition and another for semantics. This separation reduces ambiguities and allows parsing a large variety of programs. Syntax in K is defined using a variant of the familiar BNF (Backus Naur Form) notation [8], with terminals enclosed in quotes and non-terminals starting with capital letters.

Example 2.1: In Figure 2, the module EXP~SYNTAX is a K-definition of calculator language syntax with input/output actions (read, print) [15]. Arithmetic expressions are constructed from integers with addition, multiplication, and division operations. The syntax is too similar to that of standard Maude (sorts, subsorts and mixfix operation declarations). However, in addition to Maude’s attributes (such as precedence and gathering), specific K-attributes can be added, such as strict, which is used to specify that (certain) arguments of a language construct must be evaluated first (and their effects on the global state will be propagated) before giving a semantic to the construct itself. More complex examples can be found in [1].

```
module EXP~SYNTAX
//@ Arithmetics Syntax
syntax Exp ::= #:Int | Exp ^* Exp [strict] //addition
| Exp ^* Exp [strict] //multiplication
| Exp */ Exp [strict] //division
| Exp "?" Exp ?: Exp [strict(1)]
| Exp ":" Exp [sequstrict]
//@ Input / Output Syntax
syntax Exp ::= "read"| "print" Exp [strict]
end module
```

Figure 2. K-syntax module of a calculator language with I/O

A language semantics specification in K consists of three parts:

- Providing evaluation strategies. They specify order in which the arguments of a language construct should be evaluated.
- Giving the structure of the configuration to hold program state. Configurations are structured as nested labeled cells (represented using an XML-like notation) containing various computation-based data structures.
- Writing K-rules to describe transitions between configurations. A K-rule describes how a term or a sub-term in the configuration can change into another term.

```
module EXP
imports EXP~SYNTAX
syntax KResult ::= #Int
configuration
<color="green" multiplicity="* ">PGM: K<k>
<streams>
in color="magenta" stream="stdin"> .List </in>
<out color="Fuchsi" stream="stdout"> .List </out>
</streams>
//@ Arithmetics Semantics
rule I:#Int + I2:#Int => I1 + Int I2
rule 11:#Int * I2:#Int => 11 * Int I2
rule 11:#Int / I2:#Int when I2 = / = Bool 0 => 0
rule 0 : . => E: Exp => E
rule 1:#Int ? E:Exp : . => E when I = / = Bool 0
rule _:#Int ; I2:#Int => I2
//@ Input / Output Semantics
rule<in> read => I1:#Int <k>
<in> ListItem(I) => , . . .</in>
rule<in> print I1:#Int => I1</in>
end module
```

Figure 3. K-semantic module of an arithmetic language with I/O

Example 2.2: Figure 3 defines the semantics of the above arithmetic language. Both arguments of the addition operator “+_” must be evaluated before computing their sum, whereas for the conditional operator “?_:”, only the first argument should be evaluated. Arithmetic language configuration consists
of a cell labeled $\langle k \rangle$ which is meant to hold the running program (denoted here by the variable SPGM of type $K$), and a cell $\langle$streams$\rangle$ holding in/out cells, modeling program input/output streams. The seventh rule replaces each statement read in the cell $\langle k \rangle$ by the variable I given in the cell $\langle$in$\rangle$.

The K-Maude tool is designed to well define operational semantics of programming languages. In our paper, we exploit it to define the operational semantics of an ADL. Our proposed model will cope with dynamic aspects of software architectures. Therefore, we use in a transparent manner the rewriting logic, which is recognized as a unified semantic framework for concurrent and distributed computation.

3. A K based definition of software architecture

An architecture description, from a runtime perspective, should provide a formal specification of both structure and behavior of a software system. The structural viewpoint may be specified in terms of components, connectors and how they are assembled. The behavioral viewpoint may be specified in terms of actions, that a system executes or participates in, and relations among these actions. In this section, we explain how we consider these two architecture viewpoints in order to define a K-based operational semantics for any ADL. We illustrate our approach through the ACME language.

Before applying k-method to define a formal semantics for dynamic software architecture, we have to put the syntax of the ADL (in which it is described) under BNF notation. Then, we follow the same steps as for the programming languages. In other words, we define the ADL syntax according to the syntactic K-rules. Then, we define its operational semantics by defining the architecture evolution strategies at runtime, the common configurations and the set of rules defining the architecture behavior at a given time.

3.1. Structural Concepts

To apply the K method, the first step consists on providing for any considered ADL, syntactic definition under BNF notation. The result of this step is more complex as the ADL shows more rich set of syntactical constructions. We note the following common and necessary elements in any software architecture description. Then, we show, through an ACME [7] example (see figure 4), how these syntactic definitions will be integrated in K tool as a module that represents a formal meta-model defining all standard concepts of this ADL.

Component: is a computation unit (e.g. producer and consumer ACME-components in figure 4) or a data store of a system. It is defined by its provided and required interfaces (input/output ports, offered/required services) supporting the component interaction with its environment. In figure 4, components’ interfaces are denoted put and get.

Connector: is a communication support used to model interactions among components and rules that govern these interactions, as for instance ACME-connector buffer of figure 4.

Configuration: is a topology or a graph of components and connector that describes complete or partial application architecture defining its structure and its behavior (see figure 4 for a complete configuration of Producer/Consumer software architecture example).

3.1. Structural Concepts

To apply the K method, the first step consists on providing for any considered ADL, syntactic definition under BNF notation. The result of this step is more complex as the ADL shows more rich set of syntactical constructions. We note the following common and necessary elements in any software architecture description. Then, we show, through an ACME [7] example (see figure 4), how these syntactic definitions will be integrated in K tool as a module that represents a formal meta-model defining all standard concepts of this ADL.

Component: is a computation unit (e.g. producer and consumer ACME-components in figure 4) or a data store of a system. It is defined by its provided and required interfaces (input/output ports, offered/required services) supporting the component interaction with its environment. In figure 4, components’ interfaces are denoted put and get.

Connector: is a communication support used to model interactions among components and rules that govern these interactions, as for instance ACME-connector buffer of figure 4.

Configuration: is a topology or a graph of components and connector that describes complete or partial application architecture defining its structure and its behavior (see figure 4 for a complete configuration of Producer/Consumer software architecture example).

3. A K based definition of software architecture

An architecture description, from a runtime perspective, should provide a formal specification of both structure and behavior of a software system. The structural viewpoint may be specified in terms of components, connectors and how they are assembled. The behavioral viewpoint may be specified in terms of actions, that a system executes or participates in, and relations among these actions. In this section, we explain how we consider these two architecture viewpoints in order to define a K-based operational semantics for any ADL. We illustrate our approach through the ACME language.

Before applying k-method to define a formal semantics for dynamic software architecture, we have to put the syntax of the ADL (in which it is described) under BNF notation. Then, we follow the same steps as for the programming languages. In other words, we define the ADL syntax according to the syntactic K-rules. Then, we define its operational semantics by defining the architecture evolution strategies at runtime, the common configurations and the set of rules defining the architecture behavior at a given time.

3.1. Structural Concepts

To apply the K method, the first step consists on providing for any considered ADL, syntactic definition under BNF notation. The result of this step is more complex as the ADL shows more rich set of syntactical constructions. We note the following common and necessary elements in any software architecture description. Then, we show, through an ACME [7] example (see figure 4), how these syntactic definitions will be integrated in K tool as a module that represents a formal meta-model defining all standard concepts of this ADL.

Component: is a computation unit (e.g. producer and consumer ACME-components in figure 4) or a data store of a system. It is defined by its provided and required interfaces (input/output ports, offered/required services) supporting the component interaction with its environment. In figure 4, components’ interfaces are denoted put and get.

Connector: is a communication support used to model interactions among components and rules that govern these interactions, as for instance ACME-connector buffer of figure 4.

Configuration: is a topology or a graph of components and connector that describes complete or partial application architecture defining its structure and its behavior (see figure 4 for a complete configuration of Producer/Consumer software architecture example).
This example shows how to extend the ACME syntax with type and message concepts, in order to allow description of architecture evolution by message exchange. Furthermore, it may be possible to define additional syntactic constructs, even those related to time expression to model components communication with delays, and thus be able to address challenges 1 and 2 cited above.

3.2. Behavior Concepts

Most ADL such as ACME, focus on the software architecture structure and disregard its behavior. Recently some ADL pay more attention to the dynamic aspect description. These language semantics are based on formal models, they may be either partial, considering only structural evolution (case of Rapide [11], Wright [2] and Darwin [12]), or more detailed, (case of LF/P [14]).

Theoretically, to define the dynamic behavior of an architecture description in K, we have just to express the architecture evolution strategies as a set of constraints or proprieties, defined by k-functions and k-rules on description elements. Therefore, we propose to complete the syntactic module with another k-module containing the configuration definition and k-rules.

Example 3.2: For the ACME case, we define a configuration as shown in figure 6 in terms of system state at a given moment. We define also this new concept of component or connector state thanks to message exchanges scenarios between components and connectors (see figure 7b). The first rule describes how to transfer a message S from a component’s output port P to a connector role J. In the same way, the two later rules model a message passing from an input role of a connector to its output roles and from a connector’s output role to a component input port.

This work result gives a good example of formal executable specification of components and connectors changes or moves at runtime. Formal
verification may be also possible, taking benefit from the Maude engine system and its formal analysis tools.

4. Formal analysis

The proposed formal semantic framework is used to describe software architectures and express their dynamic evolution, so that one could reason on them. In particular, ensuring as early as possible the correctness of a software architecture is the most important preoccupation of this work. This section will explain how, under appropriate assumptions, we can model check in K-framework behavior properties of an architectural system such as reachability, safety and liveness, expressed as LTL properties.

We can analyze a given software architecture description to define the behavior of each declared component and the possible components’ connections. For instance, the Maude executable model specifying the static and dynamic aspects of the Producer/Consumer application of figure 4 is shown in figure 8. We identify the execution result of this system, which gives the final configuration after having transmitted the message «item » from the producer to the consumer via the buffer.

Besides, our model may be used to check LTL properties in a transparent way, even if the code is unfamiliar to the user. We have just to define, another module that combines the k-definitions of the syntax and the semantics of the given ADL. Then, we add the LTL formulas written in the module LTL-HOOKS and the model-checker given by the module MODEL-CHECKER-HOOKS. The module LTL-HOOKS is a K interface of the Maude module defining the LTL syntax. We have also to define the atomic propositions and their intended semantics.

```plaintext
module ACME-MC
imports ACME
imports LTL-HOOKS
imports MODEL-CHECKER-HOOKS

rule KItem(B:Bag) LTL|= sending ( X:ComponentId ,I:PortId ,S:String )
to (Y:ConnectorId ,J:RoleId ) => true
when val(B, X, I) ==K S
andBool val(B, Y, J ) ==K "" [anywhere]

rule KItem(B:Bag) LTL|= receiving(X:ConnectorId ,A:RoleId , I: String )
from Y:ComponentId ,C:PortId ) => true
when val(B, X, A) ==K I
andBool val(B, Y, C) ==K "" [anywhere]

syntax #ModelCheckerState ::= "KItem" (" Bag ")
syntax #Prop ::= "sending" ("ComponentId "," PortId "," String ")
to ("ConnectorId "," RoleId ")

|"receiving" ("ConnectorId "," RoleId "," String ")"
from (" ComponentId "," PortId ")

syntax String ::= "val" (" Bag "," Id "," Id ") [function]
rule val(<generatedTop>...<comp>...<compName> X<compName>
<ports>... I | -J:String ...</ports>
=> J [anywhere]}
rule val(<generatedTop>...<con>...<conName> X<conName>
<roles>... I:String ...</roles>
=> I [anywhere]
```

Figure 09. ACME Model checker module.

For our running example, we may formally analyze the Producer/Consumer ACME architecture. Indeed, we define first the module ACME-MC as indicated in Figure 9. Then, we check two properties.

The first one (P1) checks whether the Consumer will surely receive the message item from the Producer. The second property (P2) verifies that from the initial configuration, the final state of sending or receiving message actions is reached at a given time. These properties are expressed in LTL notation as shown in figure 10. The semantics of both sending and receiving propositions used to define these properties is specified by rules of figure 9b. The sending proposition for instance, is satisfied in a configuration (state) B if and only if the value of the Port I of the Component X is equal to S, and the value of the attached role J of the connector Y is empty.

Figure 11 shows the result of these two properties analysis. The first one is satisfied. The result is true,
while the second one returns false with a counter example. This negative result means that the Consumer and the Producer may be stopped, which is the intended behavior. We deduce that K-framework provides a simple and efficient way to give a meta-model of ADL concepts. Therefore, ADL syntax and semantics is always extensible. Besides, the most benefit of our k-based approach is the possible execution of the ADL semantic model. We can naturally define system architecture evolutions or reconfigurations.

Authors in [4] propose object oriented rewrite theories to define a software architecture noted in CBabel language. While in [9], the authors explore the possibility of using Real-Time Maude as a formal notation to extend L/fP language for software architecture description of real time systems. In the same context, authors of [3], define a new behavior annex, ABARcL for AADL language, based on object-oriented real-time rewrite theories. The above formal models permit not only a precise description of the semantics of interconnected components, but also makes possible the formal verification of behavioral properties via LTL model checker tool of Maude system.

However, all these works are related to a given ADL, so each proposed approach has its own motivation and requires a good mastering of Maude system to exploit its code efficiently; which is not evident for any user.

Our approach is quite similar, since it proposes a Maude-based formal framework to define an ADL operational semantics, but it differs from the others, as it is a generic one. It may be applied to any ADL. Moreover, it will facilitate execution and verification of the deduced software architecture model for users not familiar with rewriting logic concepts or model checking ones. The use of K-framework allows a transparent passage from the ADL in question to its Maude based semantic model.

7. Conclusion

K is a rewrite-based executable semantic framework designed for programming languages definition. In our paper, we used it in another context. We have proposed a formal k-based approach to define ADL syntactic aspects as well as its semantic ones. We have attributed to every ADL’s syntactical artifact (component, connector, configuration), a k-based semantic interpretation in order to execute and analyze an architecture description in Maude system in a transparent manner. The resulting mathematical model can be extended with other syntactical constructions. Moreover, the architecture possible evolutions may be easily described. We have shown through our approach how we can enrich an ADL like ACME with ports typing concept and their behavior evolution. In a future work, we will generalize more our contribution with more realistic and complex examples while addressing all expected challenges. We will define a meta-model for ADL syntax and

---

**Figure 10. Producer/Consumer LTL properties.**

**Figure 11. Model checking results of the Producer/Consumer system.**
model transformations for its operational semantics definitions.

References


