K-Maude Definition of Dynamic Software Architecture

Sahar Smaali, Aïcha Choutri, Faïza Belala
LIRE Laboratory, Constantine 2 University, Algeria
sahar.smaali@gmail.com, aichachoutri@gmail.com, belalafaiza@hotmail.com

Abstract – One of the complex issues in developing architectural models of software systems is the capturing of architectures dynamics, i.e., systems for which composition of interacting components, changes at run time. In this paper, we argue that it is possible and valuable to provide a Dynamic Software Architecture Meta-model (DySAM) that accounts for interactions between architectural components and their reconfiguration. The key to the proposed approach is to use a graphical notation, according to MDA approach, and a Maude semantic basis using the K framework for both dynamic software architecture elements reconfiguration and steady-state behavior.

Keywords –DySAM, Dynamic Software Architecture (DSA), Operational semantics, K framework

1. INTRODUCTION

Dynamic software architectures (DSA) are those architectures that modify their structure and behavior and enact the modifications during the system's execution without stopping the application. This behavior, commonly known as run-time evolution or dynamism (reconfiguration), is widely considered one of the most crucial features of modern software systems. It needs a flexible development strategy so that the underlying systems preserve their well-functioning over time by managing themselves their evolution [1].

In a previous work [2], we proposed a solution to address the above challenges in the meta-modelling context according to the MDA approach. It is an alternative definition of Software architecture in terms of a complete meta-model called DySAM (Dynamic Software Architecture Meta-model) that promotes the use of interfaces as the primary artefact to be specified and maintained. It offers to architects familiar modelling concepts and notations to define their applications' structure, behavior and dynamism. Indeed, it defines the interactions between architectural elements in terms of data transactions as well as their evolution strategies in terms of evolution rules. However, despite its completeness to cover all DSA aspects, DySAM model, like any other meta-model, is not self-sufficient to support its behavioral semantics (usually called operational semantics). Indeed, this meta-model supports an observational (structural and static) semantics only (via associations' multiplicities, constraints) and lacks a built-in support for defining semantics of both behavior and evolutionary changes.

On the other hand, integrations between formal and visual (graphical) modelling specifications are attracting increasing interest due to their benefits. Combining these two approaches, may show how the advantages of one approach can be exploited to cover or weaken the disadvantages of the other. In fact, combined models can make formal methods easier to apply and informal ones more precise [3].

In this paper, we aim to define DySAM operational semantics by integrating the meta-model in K framework |4]. This later is a semantic framework in which, programming languages, calculi, as well as type systems or formal analysis tools can be defined. It is based on Maude [5], a rewriting logic based language, dedicated to specify concurrent state changes.

The main advantages of our approach are:

- It provides lightweight and more intuitive graphical notations, which are more familiar and user-friendly.
- It gives formal specification and verification tools, which are easy to use in the development lifecycle of the software architecture.
- It provides DSA models with formal and rigorous semantics thanks to K framework use. This important advantage defines a meta-transformation closely conform to the MDA model transformation principle.
- It allows transparent execution of DSA models in Maude, so that model validation and verification can be performed using transparently its tools such as Model-Checker.

The remainder of the paper is organized as follow: In section 2, we position our approach among existing ones. Section 3 presents an overview of our meta-model DySAM for DSA description and basic concepts of K framework. In section 4, we explain how we integrate DySAM in K and define its operational semantics. The paper is then concluded in section 5 by drawing some comments and outlining some perspectives for future work.

2. Related Work

Some recent research efforts have adapted existing modeling notations and formal specification techniques (Process algebra, Petri nets, etc.) to specify dynamic software architecture of a given system, while others have developed new languages specifically for the purpose new architecture description languages. In this section, we divide the existing work into modelling/meta-modelling approaches, and those merely based on formal methods.
The first ones specify DSA, as a model of software systems or a set of features of the models themselves. They provide familiar and comprehensible notations (usually diagrams and graphical notations) to model all software architecture aspects (UML [6] and Palladio [7]). But, few interest is dedicated to model behavior and dynamics.

The second class of existing formal specification techniques have also been applied to formalize SA elements and its behavior in a given logic (we cite here for instance, formal ADL) [8, 9]. They provide a basis for rigorous properties analysis of the software system at an architecture abstract level. However, they remain uncommon and they are not well appreciated by designers and engineers.

Besides, a third class of hybrid approaches [10], to which our work belongs, may also be considered, they specify DSA as models having two possible forms, a graphical one intended to visualize SA elements and a theoretical form, usually used to allow reasoning and formal analysis of DSA. Thus, hybrid approaches take benefits of both models and formal methods.

In the same thought, our contribution consists in defining DSA according to the MDA meta-modeling approach and then, integrating this definition in the formal K framework. We define DySAM operational semantics with K-Maude tool, in order to facilitate execution and verification of the DSA by users not familiar with Maude language concepts. In fact, K allows a transparent passage from the SA description to its Maude based executable model.

3. Prerequisites

In this section, first, we present our developed meta-model for dynamic software architecture (DySAM) [2], based on interfaces as first class entity, then, we introduce some basic concepts of K system.

3.1. DySAM Model

In all proposed SA definitions (namely those given by ADL or related to component based models); the interface concept is the key element characterizing or even identifying an architectural element [2]. In fact, components are composed only through their interfaces and connector interfaces specify participant roles in an interaction. Interfaces support a part of architectural elements semantics and behavior allowing the description of dynamic aspects as well as their constraints.

In previous work [2], we have suggested a new definition of architectural description, primarily based on the interface concept (figure1). Thus, the DySAM model considers it as first class entity while leaving components and connectors as grey boxes. This will offer more flexibility and a loose coupling of architectural elements.

DySAM maintains both structure and dynamic behavior of SA. The model elements are not only

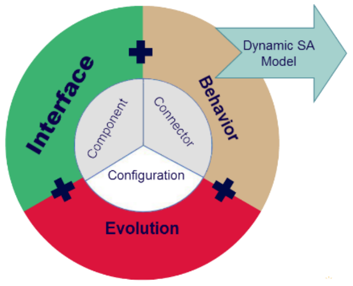

Figure 1: DySAM model motivation.

structure constructs, comparable to those of architecture description languages (ADL), but also:

- The interfaces' behavior as a state transition system. An interface may be either active or passive allowing shutting down a part of the system in order to perform some architectural changes without altering its consistency.
- The interactions between the architectural elements in terms of information exchange, in order to analyze and verify the system behavior in the early phases of the software development cycle.
- The architecture dynamic evolution as a set of strategy rules that allow adding, destroying, or even replacing architectural entities. An evolution manager is responsible of applying this rules and maintaining the coherence of the system.

DySAM is an Ecore model based on EMF (Eclipse Modeling framework) [11], a sophisticated meta-modeling framework. Textual and graphical visualization or editors can be built on top of the meta-model thanks to GMF [12] and Xtext [13] tools.

3.2. K framework

K was initiated by Grigore Rosu in 2003 and completely developed in 2010 [4]. It is a semantic framework based on rewriting logic. It provides executable Maude specifications for programming languages 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 [14].

Figure 2 presents 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 [5] 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 containing 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 back-ends: executable/analyzable Maude modules, which can serve as a basis for formal analysis, or LATEX files for documentation reasons.

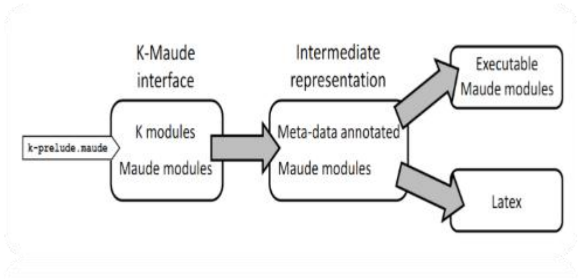

Figure 2: The K-Maude architecture.

K language definitions are given as K-modules, entirely inspired from Maude modules. Each K-module includes two sub-modules: one for the syntax definition and another for the semantics one. 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 [15], with terminals enclosed in quotes and non-terminals starting with capital letters. The syntax is similar to that of standard Maude syntax (sorts, sub-sorts and mix fix 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 some arguments of a language construct must be evaluated first (and their effects on the global state are propagated) before giving a semantic to the construct itself.

A language semantics specification in K consists of three parts:

- Providing evaluation strategies, otherwise K strategies specify the evaluation order of the arguments.
- Giving the structure of the configuration that holds the program state. Configurations are structured as nested labeled cells (using an

XML-like notation) containing various computation-based data structures.
- Writing K-rules to describe transitions between configurations.

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 DySAM model. Therefore, we use in a transparent manner the rewriting logic, recognized as a unified semantic framework, as a dynamic software architecture basis.

4. DySAM MODEL INTEGRATION IN K FRAMEWORK

To make formal methods more user-friendly (expand their use), we may combine them with informal design techniques. In this work, we integrate the DySAM model in K, while providing an intuitive modeling notation, supporting a graphical view, but still having a rigorous syntax and semantics.

Thus, we follow the same steps as for the programming languages. Otherwise, we have to define its syntax under BNF notation. Then, we define the syntactic K-rules for interpreting DySAM proposed syntax. On the other hand, we specify its operational semantics thanks to the common configurations and rules sets. K-rules define particularly, the architecture evolution strategies and interaction.

4.1. Syntax Module

In the K method application, we have to provide DySAM syntactic definition under BNF notation. This is integrated in K tool as a syntax module that represents a formal meta-model defining all concepts of DySAM. Table 1 describes a part of the DySAM syntax with an illustrative example. A system architecture description starts with the keyword **SystemArchitecture** and an identifier **Id** followed by a set of interfaces, attachments, an evolution manager and interactions between braces. Each interface may be a **ComponentInterface**, or **ConnectorInterface**.

Table 1: A part of DySAM syntax in K

Architecture Description	Syntax K-Definition
SystemArchitecture Arch{	SystemArchitecture ::= “SystemArchitecture" Id "{" Interfaces "Attachments{" Attachments "}" EvolutionManager "Interactions{" Interactions"}" Interfaces ::= Interface >Interfaces Interfaces[left] Interface ::= PrimitiveComponentInterface | PrimitiveConnectorInterface |…
PrimitiveComponentInterface A { Component is C1 ; State is Active ; Port In AIn uses as1 ; Port In AOut uses as2 ; }	PrimitiveComponentInterface::= "PrimitiveComponentInterface" Id "{"Component State Ports "}" Component ::= "Component is " Id ";" State ::= "State is" StateValue";" StateValue ::= "Active" | "Passive" Port ::= "Port" Mode PortId "uses" Id ";" Ports::= Port > Ports Ports [left] Mode ::= "In"| "Out"| "InOut"
PrimitiveConnectorInterface C { Connector is RPC1 ConnectorType is RPC State is Active ; Role In CIn ; Role Out COut ; }	PrimitiveConnectorInterface::= "PrimitiveConnectorInterface" Id"{" Connector ConnectorType State Roles "}" Connector::="Connector is " Id ";" ConnectorType ::="ConnectorType is" Id ";" Role ::= "Role" Mode RoleId";" Roles ::= Role > Roles Roles [left]

Attachments {
Attachment A.AIn to C.COut
Attachment B.Out to C.CIn }
	Attachment ::= "Attachment" Id.PortId "to" Id.RoleId
	Attachments ::= Attachment > Attachments Attachments [left]

EvolutionManager manager{
EvolutionStrategy S1(X:A ,Y:C){
 	
EvolutionRule R1{
Actions{ AddComponentInterface X2;}
 	
Transitions{
Set State Y to Passive; }
}
	EvolutionManager ::= "EvolutionManager" Id "{" EvolutionStrategies "}"
	EvolutionStrategies ::= EvolutionStrategy >EvolutionStrategiesEvolutionStrategies [left]
	EvolutionStrategy ::= "EvolutionStrategy" Id "(" List ")" "{" Rules "}"
	Rules ::= Rule > Rules Rules [left]
	Rule ::="EvolutionRule" Id "{"Actions Conditions Events Transitions"}"
	Actions ::= Action > Actions Actions [left]
	Action ::= ActionName Id ";" ...
	ActionName := "AddComponentInterface" | "DeleteComponentInterface"...

Interactions{ Interaction "Msg" between B.BOut and A.AIn through C.CIn and C.COut ;	Interactions ::= Interaction > Interactions Interactions [left] Interaction::="Interaction" Data "between" PortId "and" PortsId "through" RolesId "and" RolesId ";" | ...

The basic definition of each interface is related to the associated primitive element (component or connector). The declaration of the first one (respectively second) contains a set of **Ports** (respectively **roles**). The keyword **uses** specifies provided or required services through a port. An **Attachment** relates ports and roles to build a given structure of SA.

The interface abstract behavior is defined by a state transition system (**State** and **StateValue** keywords). The **State** attribute of an interface define its state at a given time. We consider two kinds of state: **Active** (e.g. sending or receiving a message or an event) and **Passive** (e.g. nothing happen). The Transition construct specifies the possible state changes by providing the new final state using the keywords **Set State**.

The **evolutionmanager** contains a set of evolution **Strategies (rules' sets**) to determine when and how architectural elements will change. A **Rule** is defined by a set of change **Actions**, resulting **Events**, **Conditions** and state's **Transitions**.

Interactions models **data** exchanges between interfaces that may be a **message**, an **event**, a **session**, or even a database access. It specifies the **Source Port** that initiates the interaction (output port); **target Ports** that designates one or more receiving ports (input port) and **Source/Target Roles**. Interactions allow also an interface state change according to the architecture behavior.

We can elaborate a DySAM model of any system by instantiating all its architectural element. Figure 3 describes an application that contains: instances of **A**, **B** and **C** interfaces denoted by the key word **New**, 2 attachments instances and a **notification** to the evolution manager for adding a new interface instance **A2** of type **A** and attaching it to the connector interface instance **C1**.

```
SystemSys1{
   New A1 : A ;
   New B1 :B ;
   New  C1 : C ;
   Attachment instance A1. AIn to C1.COut ;
   Attachment instance B1.BOut to C1.CIn ;
   Notification S1( A2 : A , C1 : C ) from B1 to M;   }
```

Figure 3: A simple example of a system

4.2. Configuration Definition

A configuration in K is a structure of the computations context. It is represented as nested cells containing standard items as environments, stores or other specific items (corresponding to the given semantics). The program state is typically represented as a configuration term [16].

The rewrite mechanism of K updates the given configuration repeatedly by using all possible K-rules. The configuration abstraction process allows one to specify only the minimal context needed for applying a K-rule [16].

Figure 4 presents a part of the initial configuration of DySAM. It contains three main cells. The **<k>** cell contains the whole description used by K tool to start the rewriting process. For instance, the cell **<systemArchitecture>** describes the software architecture elements, evolution strategies and interactions. The interfaces cell contains a set of (represented by a star) **<primitiveComponentInterface>** and **<primitiveConnectorInterface>** to represent the structure of each interface, while the **<attachments>** cell specifies attachments in a configuration. The value of the attributes is set to **no name** (i.e. undefined).

The cells **<ports>** and **<roles>** store respectively ports and roles in a map. The **<evolution>** cell stores the evolution strategies and rules to be applied. Finally, the sources/targets ports and roles, in addition to the data type used in an interaction, are stored in <interaction> cells.

The **<system>** cell stores the application configuration at a given time. It may be considered as an instance of the architecture. It contains cells to describe instances of interfaces and attachments, in addition to evolution notifications.

4.3. Semantics Module

The semantics of the DySAM model is defined in a separate module with rewriting rules. K-rules may be computational rules or structural rules. Structural rules capture the structural rearrangement of SA, so they allow to reorganize the configuration.

Computational rules define semantics of the computational steps while executing the defined system, as the actual state transitions. These rules are represented (in a latex file for documentation purposes) by colored cells and a

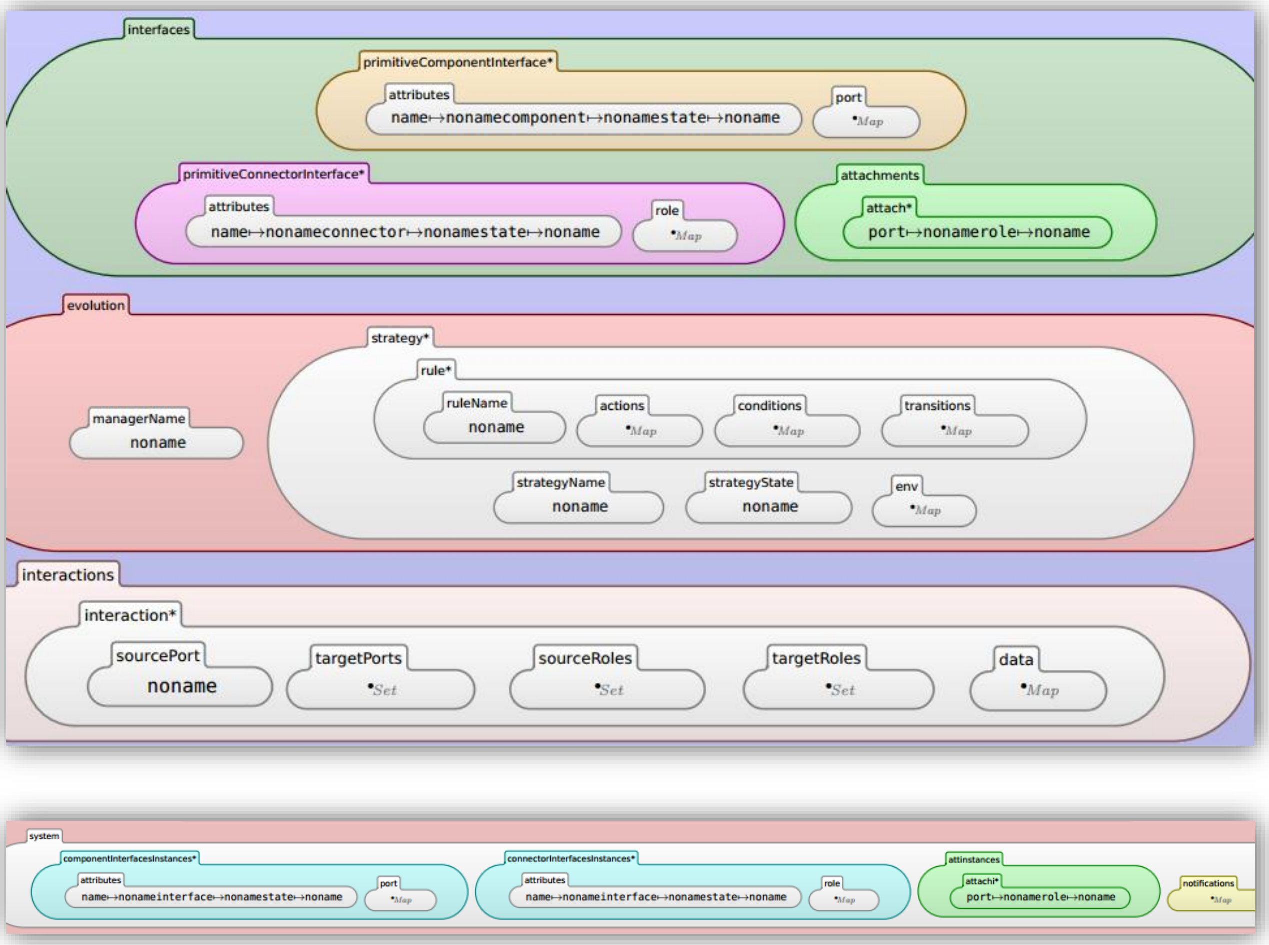

Figure 4: The initial DySAM Configuration

black horizontal line separating the right from the left side of the rewriting rule.

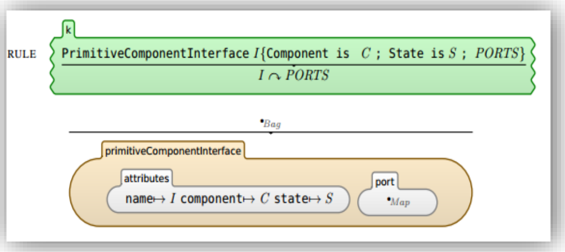
k
RULE
PrimitiveComponentInterface I{Component is C ; State is S ; PORTS}
I ↷ PORTS
•Bag
primitiveComponentInterface
attributes
name↦ I component↦ C state↦ S
port
•Map

Figure 5: A structural K-rule

The structural rule in figure 5 describes the semantics attributed to the declaration of a component-Interface. If component-interface **I** declaration is the next thing to be evaluated in the **<k>** cell then the rule replaces **I** declaration by the Variable **PORTS** (to be evaluated next), and creates a new **<primitiveComponentInterface>** and sets its internal structure described by **name**, **component**, **state** and the empty **< port>** cell.

Once all the declarations are rearranged in the configuration, computational rules may be applied. In the next section, we explain some of these rules. For instance, Figure 6 describes how a new component interface instance **I** is added to the system's configuration. **I** is added by running the action **AddComponentInterface|->N**. This action is given by the evolution rule **R** of the evolution strategy **ES**. It creates a new instance of interface **A** and sets its name to **I**, its state to **passive** and fills its ports map (if the name of this instance cannot be found in the names map).

Figure 7 shows the K-rule that performs a state transition of an interface. This transition is triggered by the evolution rule **R**. It will change the state of the interface (**A**) instance **I,** from the value **SVN** to **SV**. The interface state changing should block all incoming transactions to maintain the architecture in a coherent state, without intercepting the system too long.

In the same way, the last rule (Figure 8) details a component and connector interfaces interaction. It transmits a message **Msg** from an output port **P** of the component interface **I** to the input role of the connector interface **J.** This rule is applied, only if an attachment instance is maintained between the port and the role, and the interaction is already defined in the configuration. The interaction cell specifies ports and roles evolved in this interaction and the transmitted data (message) type.

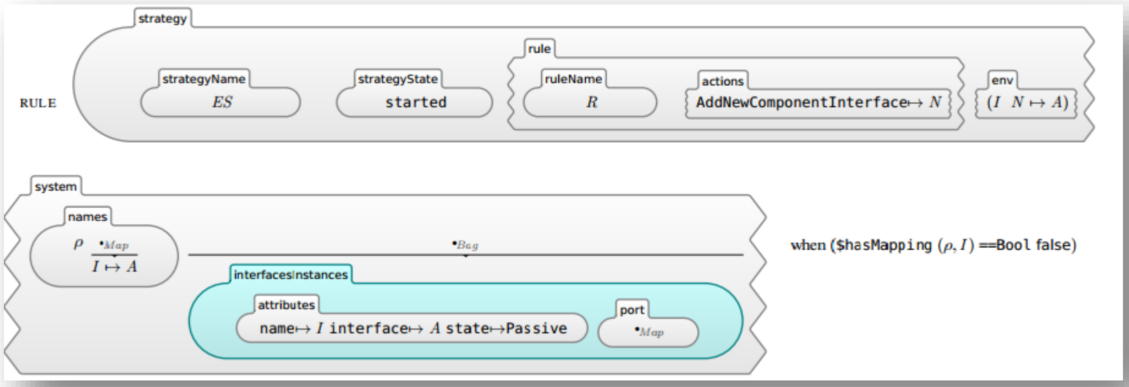

Figure 6: A K-rule describing a new interface instance creation

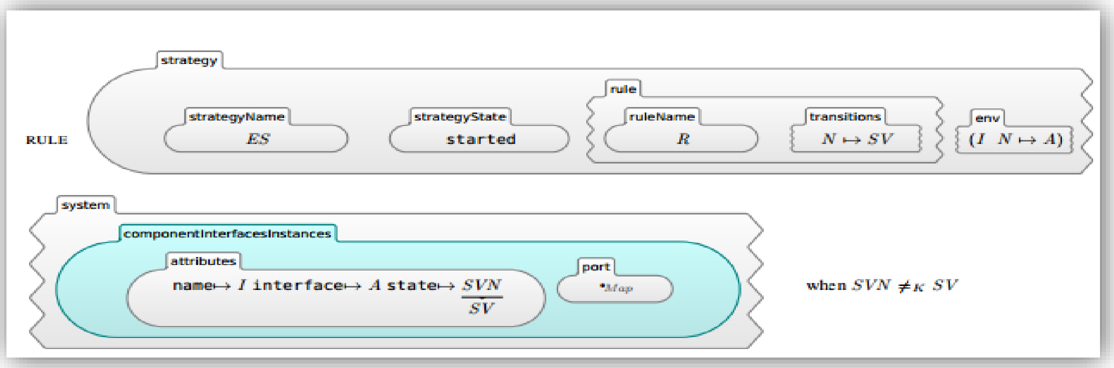

Figure 7: A K-rule modeling state transition of an interface instance.

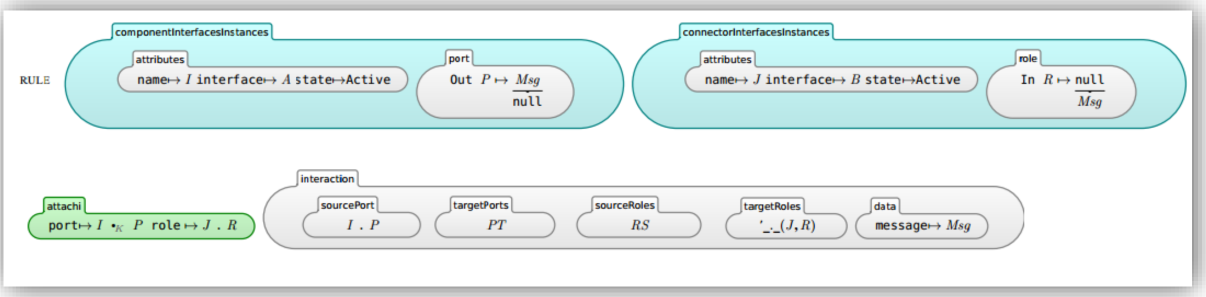

Figure 8: A K-rule describing an interaction.

5. CONCLUSION

K is a rewrite-based executable semantic framework designed for programming languages definition. In our paper, we have shown how it could also be used in the field of dynamic software architectures to fit with semantics, their description definition given in a previous work in terms of a general meta-model (DySAM). In fact, we have combined MDA development techniques with the formal K-method in order to provide the DySAM model with both an intuitive modeling notation, supporting a graphical view, and rigorous syntax and semantics. Any DySAM syntactical artifact (interface, attachments, ports, oles …), has a K-based semantic interpretation. This valuable conjunction has the advantage to make possible the execution and analysis of an architecture description in Maude system in a transparent manner. Moreover, any architecture dynamic evolution or interaction may also be well handled. Our approach could be easily exploited even by users not familiar with rewriting logic concepts. They have just to give a model conform to DySAM. Thus, in this context, K can be considered as a meta-transformation of friendly graphical and textual notations to unfriendly formal notations easy to exploit.

Because the transparency is offered by K framework even for analysis and verification, as ongoing work, we aim to integrate and simplify the use of model-checker in system validation and verification in order to guarantee that the system model, built according to DySAM, satisfies global properties during its evolution.

6. REFERENCES

[1] T. Mens, S. Demeyer, Software Evolution, ISBN 978-3-540-76440-3, Springer, 2008.

[2] S. Smaali, A. Choutri et F. Belala, “Towards a Meta-Model for Dynamic Applications,” in CBSE’14, Lille, France , 2014, in press.

[3] S.-K. Kim, D. Burger and D. Carringt, “An MDA Approach Towards Integrating Formal and Informal Modeling Languages,” J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki (Eds.): FM 2005, LNCS 3582, pp. 448–464, 2005.

[4] K semantic framework website, http://www.kframework.org/index.php/Main_Page, 2014.

[5] Clavel M., Duràn F., Eker S., Lincoln P., Marti-Oliet N., Meseguer J., and Talcott C. L. “All about Maude”, A High-Performance Logical Framework, vol 4350 of lecture Notes in Computer Science. Springer, 2007.

[6] OMG Unified Modeling LanguageTM (OMG UML), Superstructure, 2011. http://www.omg.org/spec/UML/2.4.1

[7] About Palladio, http://www.palladio-simulator.com/about_palladio/, 2014.

[8] R. Allen and D. Garlan, “A Formal Basis for Architectural Connection.” ACM Transactions on Software Engineering and Methodology, vol. 6, no. 3, pages. 213-249, 1997.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. “Specifying Distributed Software Architectures.” In Proceedings of the Fifth European Software Engineering Conference (ESEC’95), Barcelona, September 1995.

[10] D. Regep, “LfP : Un Langage de Spécification pour Supporter une Démarche de Développement par Prototypage pour les Systèmes Répartis”, PhD thesis. , Paris VI University, 2003 .

[11] Eclipse Modeling Framework, (emf). http://www.eclipse.org/modeling/emf/, 2014

[12] Graphical Modeling Framework (gmf), http://eclipse.org/gmf-tooling/, 2014.

[13] Xtext, http://www.eclipse.org/Xtext/, 2014.

[14] Serbanuta T., Rusoaie A., Lazar D., Ellison C., LucanuD., Rosu G., “The K Primer (version 2.5)”, Technical Report, January 2012.

[15] Garshol L. M., “BNF and EBNF: What are they and how do they work?” http://www.garshol.priv.no/download/text/bnf.html#id2, 2008.

[16] A. Arusoaie and T. F. Serbanuta, “a Contextual transformations in K Framework”, K 2011: 2nd International Workshop on the K Framework and its Applications, 2011.