A FORMAL SEMANTIC FRAMEWORK FOR SADL LANGUAGE

F. BELALA[1], F. LATRECHE[1], M. BENAMMAR[2]

[1]Department of Computer Science, Mentouri University, Constantine, Algeria
[2]Department of Computer Science, University of Batna, Algeria

Tel/ Fax: 213 (0) 31 81 88 88
Belalafaiza@hotmail.com

ABSTRACT

The primary purpose of an ADL (Architecture Description Language) is to specify the structural composition of a software system in terms of system's components and connectors through the means of a formal representational language. Many ADLs have emerged recently, none of them addresses formal analysis and verification of distributed architecture with a tractable model and an efficient mechanisable technique. In this paper, we explore the possibility of using Rewriting Logic model (via its Maude language) for specifying SADL architectural systems, showing how to conceive a behavior specification of systems using Maude concepts and rules. With them we do not only obtain a high level specification of SADL architecture system behavior, but we are also in a position to formally reason about and prototype the specification design produced and prototype it.

Keywords: *ADLs, Properties Formal Analysis, Rewriting Logic.*

1 INTRODUCTION

Nowadays, formal models proposed for software systems are too complex to understand, and to ensure a correct analysis. In addition, parts of such systems can be reused within another similar system or replaced by others. *Architecture Description Languages* (ADLs) [15] have opened the way to number of applications concerning the development of complex software systems and their maintenance. These languages serve to software architect since they provide a well defined semantics, at an architectural level, which is not limited to "boxes and arrows" description; they permit the architecture analysis, either by the respecting style constraints, or by formal techniques (for example, verifying the deadlock absence); and finally they help in system implantation, for example by allowing automatic code generation of interactions between system components.

Some existing formal ADL, as Wright [1], Rapide [8] and Darwin [9], based mainly on process algebra as the π-calculus, CSP or FSP, support the expression and the analysis of components, connectors and topologies (configurations) behaviors in architectures. In counterpart, these ADL are more complex to use since they make resort to several formalisms which are judged insufficient to describe formally the entire system architecture.

The experience showed that semantic formalisms used currently, present some limits concerning formulation of some inherent ADL concepts as synchronization and dynamic connection between architectural components. The objective of the present work is to propose a unique semantic formalism, *Rewriting Logic*, to well describe configuration of a software system, and to analyze their behaviors according to functional or non functional properties.

Rewriting logic has been introduced by José Meseguer [10, 12], as a consequence of its work on general logics to describe concurrent systems. In this logic, a concurrent system is represented by a rewriting theory describing its static and dynamic structures. Several languages were created on the basis of rewriting logic, the most known is Maude (SRI laboratory, United States), a declarative language where several dynamic and concurrent applications have been considered.

In this paper, we propose to formalize SADL (Structural Architecture Description Language) [14] to encourage its extension in a domain which is not yet covered behavior expression, and then this will make it very suitable for properties analysis.

SADL is an architecture description language, proposed by the SRI laboratory (United States) [14] and based like all other ADLs, on the concepts of component, connector and configuration. The particularity of this language is its explicit *refinement* mechanism of architectures at different abstraction levels. In fact, this mechanism makes possible the systematic translation of an abstract architecture to a concrete one containing more details.

Our contribution is then double. Firstly, we define a formal semantic framework, based on rewriting logic, specifying all SADL key concepts (component, connector, configuration, refinement, etc.), and allowing SADL architectures analysis. On the static level, it is possible to verify formally the respect of connection constraints between ports and roles of components (or the architectural style in general). Secondly, other verification mechanisms based on the behavior of architectural elements constitute possible complementary analysis which can be naturally addressed in this framework. We are interested in this

paper in functional properties verification (as the non deadlock) of SADL architecture, using the LTL model-checker of Maude environment; it constitutes the first step in this domain of research.
The present paper is organized as follows. Section 2 is dedicated, first of all to some related works presentation. In section 3, we present briefly rewriting logic, the formal setting used to support our SADL language formalization, and the syntactic concepts of this architecture description language. The generic approach of the SADL architecture transformation toward an equational rewrite theory is given in section 4. The proposed approach is then extended to describe the refinement mechanism of SADL architectures. In section 5, some functional properties analysis of an extended SADL architecture is made using the LTL model-checker of Maude environment. Finally, a conclusion and perspectives round off the paper are presented.

2 RELATED WORKS

The definition of some ADLs is based on well known formalisms: Wright [1], a CSP based ADL, is designed to specify components interaction using connectors and architectural styles. Rapide [8] is based on Partial Ordered event Sets and emphasizes the behaviour of software architectures and simulation to produce refinements. However, most of them focus on the software architecture description where component semantics is in part expressed by its interface, and system behavior is not completely defined. Indeed, two compatible components according to their services names can nevertheless be blocked if their behavioral protocols (the order in which services are to be used) are incompatible. Therefore, software architecture concepts need to be associated to formal theories, clarifying them or providing rules to determine whether a given architecture is well-formed. In [13] authors specify in rewriting logic the semantics for several typical architectural patterns. Bragal and Sztajnberg in [3] provide a formal model, in rewriting logic of Cbabel, a specific ADL. In [2], authors attempt to extend also the CBabel language by defining a new notion of components mobility. SADL [14] is another ADL whose semantic model was constructed in PVS. Authors' objective in [7] was to translate an SADL design into the core structured semantics (for components, connectors and their interconnections), plus appropriate semantic layers. Each semantic layer describes related semantic properties. Our approach is complementary to all these researches. In particular, it defines an alternative semantic model to SADL architecture, based on *rewriting logic*. Thus, in our proposal, the SADL software architecture, designed to facilitate designer's job, is systematically transformed to a formal and unified rewriting theory, which can be extended to manage components behavior, prototyped or model checked. This facilitates integration of formal specifications in the traditional life-cycle of an application development. Additionally, we benefit from the presence of rewriting logic operational environment *Maude* [4, 5]. The proposed semantic model is then executable under this environment (version 2.3).

3 BASIC CONCEPTS

3.1 REWRITING LOGIC

Rewriting logic is a good semantic framework in which concurrent and distributed systems can naturally and simply specified. It has been used to formalize several applications, languages and environments. This section objective is to present rewriting logic elementary concepts, useful to present our semantic setting associated to SADL. For more details, it will be necessary to refer to [10] or [12].
In rewriting logic, a concurrent system is described by a rewrite theory (Σ, E, L, R). The signature (Σ, E) defines the structure of system states. The set R of rewrite rules (of the form t→t'), axiomatizes the basic local transitions that are possible in the concurrent system. The process of concurrent rewriting describes concurrent evolution of the system by performing such local transitions modulo structural laws E satisfied by the system.
Computation in the concurrent system is a deduction in rewriting logic by finite application of the following set of deduction rules

- **Reflexivity.** For each term [t] in $T_{\Sigma,E}(X)$,
$$\overline{[t] \rightarrow [t]}$$
- **Congruence.** For each f in Σ_n, $n \in N$,
$$\frac{[t_1] \rightarrow [t_1'] \; \dots \; [t_n] \rightarrow [t_n']}{[f(t_1, \dots, t_n)] \rightarrow [f(t_1', \dots, t_n')]}$$
- **Replacement.** For each rewrite rule r: $t(x_1, \dots, x_n) \rightarrow t'(x_1, \dots, x_n)$ in R,
$$\frac{[w_1] \rightarrow [w_1'] \; \dots \; [w_n] \rightarrow [w_n']}{[t(w/x)] \rightarrow [t'(w'/x)]}$$
- **Transitivity.**
$$\frac{[t_1] \rightarrow [t_2] \qquad\qquad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

The deduction rules above allow us to infer all possible finite concurrent computations of a system specified as a rewrite theory as follows: i) reflexivity is the possibility of having idle transitions, ii) congruence is a general form of parallel composition, iii) replacement combines an atomic transition at the top using a rule with nested concurrency in the substitution, and iv) transitivity is sequential composition.
A significant consequence of the rewriting logic definition is that concurrent rewriting, instead of emerging as an operational concept, corresponds exactly to the deduction in this logic. Several languages were conceived on the basis of rewriting logic, the most known ones are: CafeOBJ (Japan), ELAN (France) and Maude defined by Meseguer (SRI, United States).

3.1. MAUDE SYSTEM

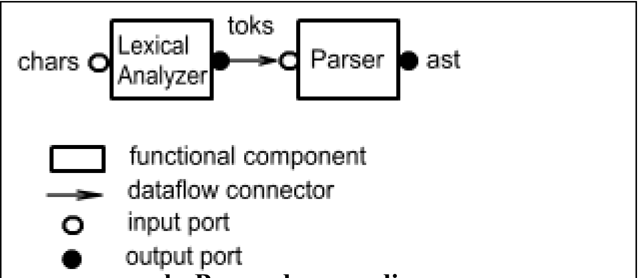
Maude is a declarative language based on rewriting logic, used as a meta-language to create different environments. It regroups three types of modules mainly: functional modules that define the static aspects of a system, they form a Maude sub-language (extension of OBJ3) based on the equational logic; system modules specify the dynamic aspect of the system using rewriting rules; while object oriented modules specify the objects oriented systems. The fact that specifications in rewriting logic are executable makes possible to have a flexible formal model of system which can constitute a prototype for the analysis and validation phase. In particular, the Maude system [4, 5] offers a powerful model checker (LTL) for checking systems properties. It acts as follows: it takes as input a system model (the module "M") expressed in rewriting logic formalism, and a specification (the module "M-Preds") which expresses a system specification property written in linear temporal logic. For a given initial state of the system (expressed in the module "M-Check"), it performs a calculus using the "on the fly" local methods principle to produce two possible results. The result is positive, and all the model executions satisfy the specification, or the result is negative and at least one execution of the model does not satisfy the specification, and in this case the Model-Checker gives this execution or a simplification of it as a counter example. From this counter example, the user corrects the source of the problem and then re-executes a new checking of the model.

3.2. SADL LANGUAGE

SADL (Structural Architecture Description Language) is a language of architectures description proposed by the SRI laboratory (United States) [14] and based like all other ADLs, on the concepts of component, connector and configuration. To present SADL's syntax let us consider a portion of standard dataflow architecture for the well known compiler example, taken from [11], which comprise two components and one connector (figure 1b).

```
compiler_L1: ARCHITECTURE
 [char_iport:         SEQ(character)         ->
base_ast_oport: ast]
    IMPORTING  character,  token,  ast  FROM
compiler_types
    IMPORTING Function FROM Functional_Style
    IMPORTING  Dataflow_Channel,  Connects  FROM
Dataflow_Style
BEGIN
COMPONENTS
 lexical_analyzer: Function
  [char_iport:  SEQ(character)  ->  token_oport:
SEQ(token)]
 Parser: Function
  [token_iport:  SEQ(token)  ->  base_ast_oport:
ast]
CONNECTORS
 token_channel: Dataflow_Channel<SEQ(token)>
CONFIGURATION
 token_flow: CONNECTION
     =   Connects(token_channel,   token_oport,
token_iport)
END compiler_L1
```

a: The SADL architecture



b: Box-and-arrow diagram

Figure 1: Example Compiler architecture

In SADL architecture of figure 1a, denoted "compiler_L1", component (as "lexical_analyzer") or connector (as "token_channel") declaration has been defined. Internally, a component can have one or more ports. A port in SADL defines an interface through which a component can provide a service (in port, as "char_iport"of "lexical_analyzer" component) or require a service (out port, as "token_oport" of the same component).

Initial topology of the architecture can be described in terms of a configuration, which can contain two kinds of elements:

– Connections: statements to link out ports of a component to in ports of another component mediated by a connector. In figure 3a, one connection denoted "token_flow" is declared, it expresses that the connector "token_channel" relates the out port "token_oport" of the component "lexical_analyzer" to the in port "token_iport" of the "Parser" component.

– Constraints: used to relate named objects or to place semantic restrictions on how they can be related in an architecture.

In the above example, the three lines of the architecture head design define the different type predicates used in this architecture:

– The type predicates: "character", "token" and "ast" are imported from another specification module named "compiler_types",

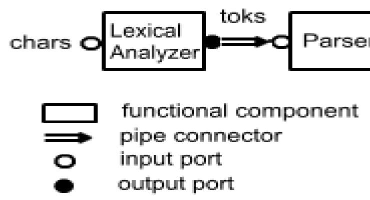
– The "Function" predicate is imported from the "Functional_Style" style,

– The "Dataflow_Channel" predicate and "Connects" are imported from "Dataflow_Style" style.

We note here that the internal architecture in this specification is invisible, if we observe closely the "Functional_Style" style, we can find, for example, the following declaration: `Function: TYPE <= COMPONENT`, which expresses that "Function" is one subtype of the predefined type"COMPONENT", in "Dataflow_Style" we recover the declaration in the same way:

```
compiler_L2: ARCHITECTURE
[char_iport: SEQ(character) ->  base_ast_oport:
ast]
IMPORTING character, token, ast FROM
compiler_types
IMPORTING Function FROM Functional_Style
IMPORTING Pipe Finite_Stream Connects FROM
Process_Pipeline_style
BEGIN
COMPONENTS
   lexical_analyzer: Function
[char_iport: SEQ(character) -> token_oport:
SEQ(token)]
   Parser: Function
[token_iport: SEQ(token) -> base_ast_oport:
ast]
CONNECTORS
 pipe_channel:   Pipe< Finite_Stream(token)>
CONFIGURATION
 token_pipe: CONNECTION
   = Connects(pipe_channel, token_oport,
token_iport)
END compiler_L2
```

a: The refined SADL architecture



b: Compiler refined structure

```
compiler_map: MAPPING FROM compiler_L1 TO
compiler_L2
BEGIN
token_channel   - -> (pipe_channel)
token_flow      - -> (token_pipe)
END compiler_map
```

c : A mapping example

Figure 2: Refinement example in SADL

`Dataflow_Channel : TYPE <= CONNECTOR`, as well as the declaration of a ternary predicate:
`Connects : PREDICATE(3).`
SADL is dedicated to structural description of architectures hierarchies at different levels of abstraction thanks to an explicit refinement mechanism. In fact, this mechanism makes possible the systematic transformation of an abstract architecture to a concrete one containing more details, according to an explicit set of model transformation rules. The figure 2a presents a refined SADL architecture of the compiler example. The mapping is clarified in figure 2c.
In this second architecture level of the compiler: "`compiler_L2`" of figure 2a, we note a new architectural style:
"`Process_Pipeline_style`", its role is to provide more concrete solutions for some architectural elements. In this example, the connector "`token_channel`" and the connection "`token_flow`" are replaced respectively by the connector "`pipe_channel`" and the connection "`token_pipe`", offering a more deterministic and comprehensible implementation of these two architectural elements.
The architectural description language SADL is intended for the definition of software architecture hierarchies that are to be analyzed formally. SADL language can be used to specify both the structure and the semantics of architecture, but untill now, the main focus has been on the former. Thus, in this paper, we associate an adequate mathematical model to SADL architecture in order to analyze it and to verify some of its properties via the Maude Model Checker [6]. Practical realization of the SADL parser and analyser tools is inspired from this specification prototype which is designed and tested under the Maude environment [4, 5].

4. SEMANTIC MODEL OF A SADL ARCHITECTURE

Our main contribution consists of defining a formal semantic framework, based on rewriting logic, to describe all SADL key concepts (component, connector, configuration, refinement, etc.), and to analyze such a static architecture. At this level, merely static, rewriting logic through its Maude language offers an adequate semantic setting to verify the respect of connection constraints between architecture ports and roles (or architectural style in general). Besides, the refinement mechanism in SADL is naturally integrated in the considered formalism. In our on going works, we plan to reconsider this formalization to prove the refined architectures equivalence. Since architecture behavior is not even covered by SADL language, the main interest of this approach is to encourage formal extension of the language with this concept to make it suitable for properties analysis.

4.1. ARCHITECTURAL OBJECTS FORMALIZATION

The theoretical model that we associate to SADL architecture is an equational theory of the membership equational logic, one rewriting logic subclass. This model is noted: $(\sum, E \cup A)$, where $\sum$ is our model signature, the useful set of sorts, and operators to statically describe an architecture, E represents the set of our model equations, and finally A represents the set of operators equational attributes.
Indeed, we adopt a generic approach that associates to each architectural object of SADL, a functional Maude module (implementation of equational theory). Therefore, we have five generic Maude modules mentioned in figure 3.
The proposed approach is general enough since the generated functional theory "architecture" is unique and a generic model of SADL architecture; it remains valid for any architecture example. So, in order to transcript a specification architecture example, as "`compiler-L1`" (figure 1a), in rewriting logic, we declare a new functional Maude module "`Compiler`" (figure 4) extending the module "`Architecture`" and it will contain the constant operators specification to identify in this case, the ports (`char-iport, token-iport, token-oport, base-ast-oport`), the components (`lexical-analyzer, parser`), the connector (`token-`

`channel`), the connection (`token-flow`), and the architecture (`compiler-L1`) names. Indeed, only one equation is included in this module to specify clearly and in a global manner each SADL architecture; this represent a typical instance of the generic model.
Through the presented modules of this section, we achieved a modular and legible specification of SADL architecture.
Rewriting logic flexibility permits declaration of user defined operators persevering SADL architecture syntax (see eq clause in figure 4).

In the same way this specification can be easily enriched, particularly, we can add other elements to specify architectural components behavior. Another theoretical extend of this model will permit deduction of Meta functions in rewriting logic that will formalize our mapping. Implementation of these functions in Maude produces an executable environment for SADL specifications that should simplify their parser process.

```
fmod Port is
 / permits the specification of the in/out port notion of a SADL architecture.
sort DataType .
sorts IPortName OPortName IPort OPort SetIPort SetOPort .
subsort OPort < SetOPort .
subsort IPort < SetIPort .
op none : -> IPort [ctor] .
op none : -> OPort [ctor] .
op _:_ : IPortName DataType -> IPort [ctor prec 21] .
op _:_ : OPortName DataType -> OPort [ctor prec 21] .
op _;_ : SetIPort SetIPort -> SetIPort [ctor assoc id: none comm prec 22] .
op _;_ : SetOPort SetOPort -> SetOPort [ctor assoc id: none comm prec 22] .
endfm
```

```
fmod Component is
 / to specify the structure of a SADL component .
extending Port .
sorts ComponentName ComponentType Component SetComponent.
subsort Component < SetComponent .
op none : -> Component [ctor] .
op _:_`[_->_`]  : ComponentName ComponentType SetIPort SetOPort -> Component [ctor prec 23] .
op __ : SetComponent SetComponent -> SetComponent [ctor assoc id: none comm prec 24] .
endfm
```

```
fmod Connector is
 / to modelize the interaction between the components .
extending Port .
sorts ConnectorName ConnectorType Connector SetConnector .
subsort Connector < SetConnector .
op _:_<_> : ConnectorName ConnectorType DataType -> Connector [ctor prec 23] .
op none : -> Connector [ctor] .
op __ : SetConnector SetConnector -> SetConnector [ctor assoc id: none comm prec 24] .
endfm
```

```
fmod Configuration is
 / this module describe the relation between two components ports and a compatible connector.
extending Component .
extending Connector .
sorts  ConnectionName  ConnectionRelation  Connection  ConstraintName  ConstraintRelation  Constraint
SetCon .
subsorts Connection Constraint < SetCon .
op _:CONNECTION_`(_`,_`,_`) : ConnectionName ConnectionRelation ConnectorName OPortName IPortName ->
Connection  [ctor prec 23] .
op   _:CONSTRAINT_`(_`,_`)   :  ConstraintName  ConstraintRelation  ComponentName  ComponentName  ->
Constraint [ctor prec 23] .
op none : -> SetCon [ctor] .
op __ : SetCon SetCon -> SetCon [assoc id: none  comm prec 24] .
endfm
```

```
fmod Architecture is
 / to define a SADL architecture thanks to the second declared operator .
extending Configuration .
sorts Head Architecture .
subsort Architecture < Component .
op `[_->_`]  : SetIPort SetOPort -> Head [ctor] .
op  ARCHITECTURE_COMPONENTS_CONNECTORS_CONFIGURATION_  :  Head  SetComponent  SetConnector  SetCon  ->
Architecture [ctor prec 25] .
endfm
```

Figure 3: Maude modules formalizing SADL architectural objects

```
fmod Compiler is
extending Architecture .
ops SEQ-character SEQ-token ast : -> DataType
[ctor] .
ops char-iport   token-iport   : -> IPortName
[ctor] .
ops token-oport base-ast-oport : -> OPortName
[ctor] .
ops lexical-analyzer parser : -> ComponentName
[ctor] .
op Function : -> ComponentType [ctor] .
op token-channel : -> ConnectorName [ctor] .
op Dataflow-Channel : -> ConnectorType [ctor] .
op token-flow : -> ConnectionName [ctor] .
op Connects : -> ConnectionRelation [ctor] .
op compiler-L1 : -> Architecture .
eq compiler-L1 = ARCHITECTURE
      [ char-iport  : SEQ-character  -> base-
ast-oport : ast ]
COMPONENTS
    lexical-analyzer : Function
         [  char-iport  :  SEQ-character  ->
token-oport : SEQ-token ]
    parser : Function
         [ token-iport  :  SEQ-token  -> base-
ast-oport : ast ]
CONNECTORS
    token-channel  :  Dataflow-Channel  <  SEQ-
token >
CONFIGURATION
    token-flow  :CONNECTION  Connects  (  token-
channel , token-oport , token-iport ) .
endfm
```

Figure 4: "`compiler-L1`" architecture in Maude

4.2. REFINEMENT FORMALIZATION

An explicit refinement of architectures is supported in SADL language according to a mapping which is specified by a set of pairs associations (see figure 2c) of the form: `architectural-element1 --> architectural-element2`

In this section, we present an approach for the refinement mechanism formalization. In figure 5, a system Maude module "`REFINER`" allows achieving such relation between two any abstracts SADL architectures. The most important operator of this module is "`Refine`", which acts on two arguments: a term of sort "`Architecture`" and another term of sort "`Mapping`", and generates an abstract refined architecture using explicit architectural elements refinements according to a set of conditional rewrite rules, we present in figure 5 an example of such rules ("`refine-component`").

```
mod Refiner is
protecting Architecture .
sorts MapComp Mapping .
subsort MapComp < Mapping .
op nil : -> Mapping [ctor] .
op _-->_ : Component Component -> MapComp [
prec 26] .
op __ : Mapping Mapping -> Mapping  [assoc
prec 27 id: nil] .
op  Refine  :  Architecture  Mapping  ->
Architecture .
var arch : Architecture .
var map : Mapping .
var head : Head .
var setcom : SetComponent .
var setcon : SetConnector .
var setcx : SetCon .
vars comp1 comp2 comp3 : Component .
crl  [refine-component]  :  Refine(ARCHITECTURE
head COMPONENTS comp1 setcom CONNECTORS setcon
CONFIGURATION setcx, comp2 --> comp3 map) =>
Refine(ARCHITECTURE  head  COMPONENTS  comp3
setcom CONNECTORS setcon CONFIGURATION setcx,
map) if comp1 == comp2 .
rl [no-refinement] : Refine(arch, nil) => arch
.
endm
```

Figure 5: A system Maude module for the refinement mechanism

In the following Maude code, we take advantage of the inherent rewriting mechanism in Maude system modules to achieve architectures refinement. More precisely, we show in figure 6, how to use the "`Refine`" operator to refine the connector "`token-channel`" and the connection "`token-flow`" of the "`compile-L1`" architecture. The result of "`refine`" application is displayed directly in the Maude environment window (figure 6).

```
              \||||||||||||||||||/
            --- Welcome to Maude ---
              /||||||||||||||||||\
     Maude 2.3 built: Feb 21 2007 14:55:47
     Copyright 1997-2007 SRI International
           Fri May 25 21:10:21 2007
Maude> rew Refine ( compiler-L1 ,  token-
channel : Dataflow-Channel < SEQ-token > -->
pipe-channel : Pipe < Finite-Stream-token >
token-flow :CONNECTION Connects ( token-channel
, token-oport , token-iport ) --> token-pipe
:CONNECTION Connects ( pipe-channel , token-
oport , token-iport ) ) .
rewrites: 6 in 1625750371000ms cpu (0ms real)
(0 rewrites/second)
result Architecture: ARCHITECTURE [char-iport :
SEQ-character -> base-ast-oport
    : ast] COMPONENTS lexical-analyzer :
Function[char-iport : SEQ-character ->
    token-oport : SEQ-token] parser :
Function[token-iport : SEQ-token ->
    base-ast-oport : ast] CONNECTORS pipe-
channel : Pipe < Finite-Stream-token
    > CONFIGURATION token-pipe :CONNECTION
Connects(pipe-channel,token-oport,
    token-iport)
```

Figure 6: A use example of "`Refine`" operator

In addition, the use of rewriting logic via its Maude language, will offer an executable and analyzable specification while taking advantage of tools around this environment as using a model-checker for the linear temporal property verification.

We clarify our approach through a very simple and generic example that specifies two processes (`procA` and `procB`) in mutual exclusion, the shared variable "`turn`", having the value 0 initially, permits to keep trace of the process that is going to enter in the critical section, which examine or change the value of this variable. The SADL architecture of the system example is presented in figure 7.

The Maude modules describing the structure and behavior aspects of the `Mutex` system is given in figure 8b.

```
Mutex: ARCHITECTURE [ -> ]
IMPORTING nat FROM Mutex_types
IMPORTING     Variable     Read\Write     FROM
Shared_Memory_style
IMPORTING Process FROM Process_style
BEGIN
COMPONENTS
     procA: Process [ -> ]
     procB: Process [ -> ]
     turn: Variable(nat)[ -> ]
CONFIGURATION
     rwA-var:  CONSTRAINT  Read\Write  (procA,
turn)
     rwB-var:  CONSTRAINT  Read\Write  (procB,
turn)
END Mutex
```

Figure 7 : Architectural system Mutex

The structure aspect of the `Mutex` system is defined as in the previous section (figure 8a). We notice the use of an enriched version of the `Component` module with "`ComponentState`" sort that permits to define the state of a component. Now, the operator generating SADL component has the following form:

```
op     _:_`[_->_`]`(_`):     ComponentName
ComponentType        SetIPort        SetOPort
ComponentState ->  Component.
```

Therefore, we declare in the `Mutex` module, with constructors operators, the process state (`wait` or `critical`) as well as the state associated to the shared variable `turn` (0 or 1). Reading or updating operations of the shared variable `turn` (by the two processes procA and procB), will be defined using the two constraints: `rwA-var` and `rwB-var`. The behavior of the `Mutex` SADL architecture will be mainly defined by the rewriting rules of the Maude system module `Behav-Mutex`

```
fmod Mutex is
extending Architecture .
ops  wait  critical  0  1  :  ->  ComponentState
[ctor] .
ops procA procB turn : -> ComponentName [ctor]
.
ops Process Variable : -> ComponentType [ctor]
.
op Read\write : -> ConstraintRelation [ctor] .
ops rwA-var rwB-var : -> ConstraintName [ctor]
.
op Mutex : -> Architecture .
eq Mutex = ARCHITECTURE [ none -> none  ]
 COMPONENTS
    procA : Process
         [ none -> none ](wait)
    procB : Process
         [ none -> none ](wait)
    turn : Variable [ none -> none ](0)
 CONNECTORS
    none
 CONFIGURATION
   rwA-var  :CONSTRAINT  Read\write  (  procA  ,
turn )
   rwB-var  :CONSTRAINT  Read\write  (  procB  ,
turn ) .
endfm
```

a: Maude module Mutex

```
mod Behav-Mutex is
protecting MutexG .
var setc : SetCon .
var st : ComponentState .
rl [A-enter] : ARCHITECTURE [ none -> none ]
COMPONENTS
  procA : Process [ none -> none ](wait)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](0)
  CONNECTORS none CONFIGURATION setc =>
  ARCHITECTURE [ none -> none  ] COMPONENTS
  procA : Process [ none -> none ](critical)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](0)
  CONNECTORS none CONFIGURATION setc .
rl  [A-exit]  :  ARCHITECTURE  [  none  ->  none  ]
COMPONENTS
  procA : Process [ none -> none ](critical)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](0)
  CONNECTORS none CONFIGURATION setc =>
  ARCHITECTURE [ none -> none  ] COMPONENTS
  procA : Process [ none -> none ](wait)
  procB : Process [ none -> none ](st)
  turn : Variable [ none -> none ](1)
  CONNECTORS none CONFIGURATION setc .
          . . .
endm
```

b: An extended module with behavior specification

Figure 8 : The component behavior expression

(see figure 8b). Then, the behavior of the Mutex architecture is essentially described by the four following rewriting rules:

- `A-enter/B-enter`: the process `procA` (respectively `procB`) examines the value of `turn` variable and if it is equal to 0 (respectively 1) it enters in its critical section.
- `A-exit/B-exit`: the process `procA` (respectively `procB`) decides to leave its critical section, it modifies the value of `turn` variable to 1 (respectively 0).

Besides, these modules have been tested syntactically and analysed formally with the LTL model checker of Maude. In particular, we have been interested in verifying the mutual exclusion property of the two processes `procA` and `procB` (see figure 8) expressed as a linear temporal logic property: `[] ~(crit(procA) /\ crit(procB)`. A theoretical extend of our proposed model, based rewriting logic will permit the deduction of a Meta model that serves to the development and analysis of architectural components. A set of functional properties has been already considered, while non functional ones will constitute our next research axis.

```
Maude> red modelCheck(init, [] ~(crit(procA) /\
crit(procB))) .
reduce in Behaviour-CHECK :
modelCheck(init, []~ (crit(procA) /\
crit(procB)))    .
rewrites: 23 in 7714480714ms cpu (3ms real) (0
rewrites/second)
result Bool: true
```

Figure 8: The mutual exclusion property analysis

5. CONCLUSION

Since the beginning of the years 90, the community of software engineering developed several languages for architectures description, their main goal is to develop and maintain complex software systems. SADL presents a particular interest since it proposes a rich type system and allows one to describe designs at various levels of abstraction. In this paper, we firstly presented a semantic framework for the SADL architecture description language based on rewriting logic. In fact, we presented how each element of a SADL architecture will be transformed towards one rewriting logic term while preserving their initial syntax, thanks to the expressive power of this logic. This formalization approach can also be adapted to other ADLs.

We have then proceeded to the refinement formalization offered by the SADL language; more precisely we indicated how rewriting logic power especially the rewriting mechanism will be used for the refinement propagation. Indeed, the interest of such approach is to offer on the one hand, the possibility to formally verify architectural properties, and on the other hand, to add other elements to specify new architectural concepts. We have enriched the proposed model to allow behavior modeling in a SADL architectural component. In one ongoing work, we plan to formalize our mapping as meta-functions in rewriting logic. The implementation of these functions in Maude produces an executable environment for SADL specifications that should ease the verification process of SADL specifications.

REFERENCES

[1] Allen R., "A Formal Approach to Software Architecture", PhD Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, 1997.

[2] Bouanaka C., Belala F., "On Adding Some Mobility Primitives to an Architecture Description Language", *In CSIT'06 Proceeding, the 4th International Multiconference on Computer Science and Information Technology*, Amman, Jordan, 2006.

[3] Braga1C., Sztajnberg A., "Towards a Rewriting Semantics for a Software Architecture Description Language", *in: A. Cavalcanti and P. Machado, editors, Proceedings of 6th Workshop on Formal Methods (WMF)*, Campina Grande, Brazil, vol. 95 , pp.148-168,2003.

[4] Clavel M., Duran F., Eker S., Marti-Oliet N., Lincoln P., Meseguer J. and Talcott C.., "Maude 2", http://maude.cs.uiuc.edu , 2003.

[5] Clavel M., Duran F., Eker S., Martı-Oliet N., Lincoln P., Meseguer J., and Quesada J., "Maude: Specification and Programming in Rewriting Logic", SRI International Lab., http://maude.csl.sri.com, (1999).

[6] Eker S. and Mesguer J. and Ambarish S., "the Maude LTL model-Checker", *Electronic Notes in Theorical Computer Science*, vol. 71, 2002.

[7] Herbert J., Dutertre B., Riemenschneider R. and Stavridou V., "A Formalisation of Software Architecture", J. Wing, J. Woodcock, J. Davies (Eds): FM'99, Vol.1, LNCS 1708, pp. 116-133, 1999.

[8] Luckham D. C., Kenny J. J., Augustin L. M., Vera J., Bryan D. and Mann W., "Specification and Analysis of system Architecture Using Rapide", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336-355,1995.

[9] Magee J., Dualy N., Eisonbach S. and Kramer J., "Specifying Distributed Software Architectures", *In Proceedings of the Fifth Symposium on the Foundations of Software Engineering (FSE4)*, vol. 989,pp. 137 - 153,1995.

[10] Marti-Oliet N., Meseguer J., "Rewriting logic as a logical and semantic framework", *Electronic Notes in Theoretical Computer Science*, Vol. 4, no.1, pp.1-36, 1996.

[11] Megzari K., "REFINER : Environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture", Thèse de Doctorat préparée au Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance, ESIA – Université de Savoie, 2004.

[12] Meseguer J., "Conditional Rewriting as a Unified Model of Concurrency", *Theoretical Computer Science 96*, pp. 73-155, 1992.

[13] Meseguer J. and Talcott C., "Semantic Models for Distributed Object Reflection", *In ECOOP 2002 Object-Oriented Programming 16th European Conference*, Malaga, Spain, vol. 2374, pp. 1–36, 2002.

[14] Moriconi M. and Riemenschneider R. A., "Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies", *Technical Report SRI-CSL-97-01*, Computer Science Laboratory, SRI International, 1997.

[15] Nenad M., Richard M., "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, Vol. 26, no. 1, pp. 70-93, 2000.