# Vers l'Intégration des Propriétés non Fonctionnelles dans le Langage SADL

Faiza BELALA[*], Fateh LATRECHE[*], Malika BENAMMAR[**]
[*]Département d'Informatique, Université Mentouri de Constantine, Algeria
belalafaiza@hotmail.com
[**]Département d'Informatique, Université de Batna, Algeria

**Résumé.** La notion d'*architecture logicielle* est apparue aux alentours des années 1990 et est maintenant présentée comme le cœur d'une discipline à part entière. De nombreux langages de description d'architecture (ADLs) ont été proposés dans la littérature. Ils offrent des capacités complémentaires pour le développement et l'analyse architecturale d'un système logiciel. Comme l'objectif principal d'un ADL est de fournir les moyens pour une spécification explicite, plusieurs caractéristiques pour les composants et les connecteurs sont souhaitables. Parmi ces caractéristiques, les *propriétés non fonctionnelles* spécifiant les besoins des utilisateurs, sont très importantes et favorisent une implantation correcte du logiciel. SADL est un langage de description d'architecture dont la vocation n'étant pas l'analyse, nous l'avons alors étendu pour prendre en compte les spécifications non fonctionnelles des systèmes architecturaux. Nous proposons dans cet article, un cadre formel, basé logique de réécriture, permettant la description des propriétés non fonctionnelles des composants et des connecteurs du langage SADL ainsi que l'analyse d'une application architecturale garantissant ces propriétés.
**Mots clés.** Langage de Description d'Architecture, Approches Formelles pour la Validation des Architectures Logicielles, Logique de réécriture, Le langage Maude.

## 1 Introduction

Face à la croissance de la taille et de la complexité des systèmes logiciels, il est devenu de plus en plus nécessaire l'adoption de la description architecturale dans leur processus de développement. La description architecturale a joué toujours un rôle important dans le développement de logiciels complexes et maintenables à moindre coût et avec respect des délais. Elle permet de prendre en compte les descriptions haut niveau de systèmes complexes et de raisonner sur leurs propriétés. Elle repose généralement sur les modèles à base de composants, de connecteurs et de configurations architecturales. Les ADLs, ou encore langages de description d'architectures sont des notations qui servent à définir et à exprimer la structure haut niveau de l'architecture du logiciel en termes d'une collection de composants et de connecteurs et de leur structure d'interconnexion globale (Medvidovic et Taylor (2000))**.** Ils fournissent un modèle abstrait indépendant des détails d'implémentation permettant au développeur de raisonner correctement sur les propriétés du logiciel.

Les travaux menés jusqu'à présent sur le développement centré architecture de logiciels portent essentiellement sur la spécification formelle des architectures. La vérification et l'analyse des architectures logicielles restent un point ouvert de recherche. En effet, il est indispensable d'associer à une architecture logicielle des spécifications précises tant

fonctionnelles que non fonctionnelles. Ces spécifications décrivent des services offerts par les différents composants de cette architecture, mais aussi ceux dont ils ont besoin pour fonctionner. L'environnement possède des propriétés qui ont une influence sur le fonctionnement du logiciel. Il peut s'agir de propriétés temporelles, des propriétés sur le mode de panne ou sur les attaques possibles en termes de sécurité, etc. Ces spécifications sont rarement complètes et formelles.

En général, le terme contrat est de plus en plus utilisé pour décrire les propriétés d'un composant architectural. Les propriétés non fonctionnelles, devant être exprimées à part, permettent de caractériser le degré de satisfaction d'un logiciel, c'est-à-dire comment ses fonctionnalités sont réalisées. Elles sont associées à différents concepts comme par exemple la performance, la qualité de service, la sécurité, la robustesse, la portabilité, etc. De par leur complexité et leur haut niveau d'abstraction, elles sont rarement prises en compte dans le processus de développement d'un logiciel, particulièrement sa conception architecturale. En effet, le concepteur doit pouvoir considérer ce type de propriétés dès le début du processus, et les estimer avant la phase d'implémentation du système.

En fait, les ADLs qui sont censés décrire de façon formelle l'architecture d'un système à un niveau d'abstraction élevé, peuvent être dotés de mécanismes pour l'expression et la vérification de ce type de propriétés. Dans ce contexte, nous suggérons de fournir un support formel permettant l'expression et la vérification de ces propriétés au niveau des architectures logicielles décrites dans le langage SADL, tout en exploitant l'expressivité et la puissance de la logique de réécriture via son langage Maude.

La logique de réécriture a été introduite initialement par José Meseguer (Meseguer (2002)), comme une conséquence de son travail sur les logiques générales, pour décrire les systèmes concurrents. Dans cette logique, les aspects statique et dynamique des systèmes sont représentés à l'aide des théories de réécriture. La logique de réécriture bénéficie aussi de la présence de nombreux langages et environnements opérationnels, le plus connu étant Maude, créé par le laboratoire SRI (Stanford Research Institute) aux Etats-Unis. C'est un langage déclaratif très expressif autour duquel se trouve un environnement composé de plusieurs outils.

SADL (Structural Architecture Description Language) est un langage de description d'architectures proposé aussi par le laboratoire SRI (Moriconi et Riemenschneider (1997)). Il est basé comme tous les autres ADLs sur les concepts de composant, de connecteur et de configuration. La particularité de ce langage est qu'il est dédié à la description structurale des hiérarchies d'architecture à différents niveaux d'abstractions grâce à un mécanisme de raffinement explicite.

L'objectif de ce travail et d'élargir le modèle formel, basé logique de réécriture, du langage SADL présenté dans (Belala et al. (2007)), pour permettre la spécification des propriétés non fonctionnelles des composants et des connecteurs de SADL, ainsi que l'analyse d'une application architecturale garantissant ces propriétés.

Dans la suite de l'article, la section 2 présente un état de l'art relatif à notre problématique. La section 3 décrit brièvement les concepts fondamentaux de la logique de réécriture et le langage de description d'architecture SADL. Dans la section 4, nous détaillons notre approche d'intégration des propriétés non fonctionnelles dans le langage SADL. Nous commençons tout d'abord par rappeler le modèle formel, basé logique de réécriture, associé à une architecture SADL, puis nous montrons comment étendre ce modèle pour permettre d'une part, la spécification des propriétés non fonctionnelles des composants et des connecteurs du langage SADL, et d'autre part, l'analyse d'une application

architecturale garantissant ces propriétés. Finalement, la conclusion propose une synthèse du travail réalisé et des perspectives liées à la poursuite de ce travail.

# 2 Etat de l'art

La définition d'une architecture logicielle est une étape importante dans la conception d'un logiciel. Elle permet d'avoir un niveau d'abstraction élevé des applications conçues en se détachant des détails techniques propres à l'environnement, et respectant les contraintes des futurs utilisateurs. Pour répondre à ce besoin de description d'architectures, différents langages de description d'architecture ont été développés, chacun offre des capacités complémentaires pour le développement et l'analyse architecturale (Garlan et al. (1997)), à titre d'exemple, Aesop supporte l'utilisation des styles architecturaux, METAH offre la possibilité de conception des systèmes de contrôle avionique temps réel, C2 supporte la conception des interfaces graphiques, Wright supporte la spécification et l'analyse comportementale des éléments architecturaux, Rapide permet la vérification et la validation des architectures logicielles par simulation, SADL offre un mécanisme explicite de raffinement d'architectures.

Cependant, dans ces ADLs, la prise en compte des spécifications non fonctionnelles est incomplète ou mal traitée. Il existe un manque notable de supports et de techniques permettant l'expression, l'évaluation, et la prédiction de ce type de propriétés dans les ADLs existants. Les ADLs ACME, Aesop, et Weaves permettent la spécification des propriétés et/ou des annotations arbitraires sur les éléments architecturaux, mais malheureusement ils n'offrent aucun outil permettant l'interprétation de ces propriétés (Medvidovic et Taylor (2000)).

D'un autre côté, UniCon et Rapide offrent un support partiel de modélisation des propriétés non fonctionnelles. UniCon permet la spécification des propriétés liées à certains aspects comme le ″scheduling″ de l'application, tandis que Rapide offre la possibilité de la modélisation des informations temporelles dans son langage de contraintes (Medvidovic et Taylor (2000)).

D'autre part, les auteurs de (Franch et Botella (1998)) ont présenté une approche qui porte sur l'intégration de l'information non fonctionnelle dans l'architecture logicielle des systèmes, leur objectif est de mesurer et vérifier les propriétés non fonctionnelles du produit logiciel final. L'approche adoptée utilise une notation ad hoc, indépendante de tout ADL, pour décrire les composants et les connecteurs, chacun avec une partie spécification et une partie implémentation. La modélisation de l'aspect non fonctionnel est réalisée via un autre langage appelé *NoFun,* basé sur les trois concepts clés suivants: *Non-functional attribute (NF-attribute)* représentant n'importe quel attribut du logiciel permettant de décrire et/ou évaluer ce dernier. Les attributs les plus connus dans ce cas, sont l'efficacité en temps d'exécution et espace mémoire, la réutilisation, la fiabilité; *Non-functional behavior (NF-behaviour)* correspond à toute valeur assignée à un attribut non fonctionnel lié à un élément architectural particulier; *Non-functional requirements (NF-requirement)* permettant de spécifier les valeurs permises des attributs non fonctionnels.

Par ailleurs, un langage appelé *ProcessNFL* dédié à l'expression des besoins non fonctionnels de logiciels a été proposé dans (Rosa et al. (2002)). Ce langage porte plutôt sur la modélisation des aspects de conflit et de corrélation entre les propriétés non fonctionnelles. Un exemple de deux propriétés non fonctionnelles qui peuvent être en conflit

sont la sécurité et la performance, un haut niveau de sécurité peut conduire à une perte de performance et vice versa. Dans ce langage trois abstractions sont aussi utilisées pour modéliser une propriété non fonctionnelle: *NF-Attribute*, *NF-Property* et *NF-Action*. *NF-Attribute* modélise les propriétés non fonctionnelles qui peuvent être simples ou dérivées à partir d'autres *NF-Attribute*. *NF-Action* modélise les aspects logiciels (algorithmes, structures de données) ou les mécanismes matériels (les ressources disponibles) qui ont un effet sur les *NF-Attributes*. L'abstraction *NF-Property* permet d'exprimer des contraintes sur les *NF-Attribute*.

Dans sa thèse, Aagedal (Aagedal (2001)) a défini aussi un langage de modélisation de la qualité de service, appelé CQML. Ce langage permet la spécification des offres et des exigences en termes de qualité de service pour les systèmes à base de composants. Une caractéristique importante de ce langage est ses notations sont purement syntaxiques.
Une autre approche plus prometteuse, visant l'intégration d'une combinaison de notations CSP, de phrases structurées et de notations WRIGHT pour spécifier des propriétés non fonctionnelles dans l'ADL WRIGHT a été proposée dans (Christopher et al. (2005)). L'idée de base derrière cette approche est que la satisfaction des besoins non fonctionnels d'une unité architecturale dépend de la satisfaction d'un ensemble de besoins fonctionnels liés à d'autres unités.

L'approche que nous proposons dans cet article s'inspire de cette dernière. Elle fournit un cadre formel unique pour la spécification des propriétés non fonctionnelles d'une application architecturale en SADL ainsi que leur vérification en simulant le comportement de cette architecture. L'apport principal de cette contribution se situe au niveau de la satisfaction d'une propriété non fonctionnelle, qui nécessite au préalable la satisfaction d'une autre propriété non fonctionnelle, ou/et d'un ensemble de besoins fonctionnels d'une autre unité architecturale.

# 3 Concepts fondamentaux

## 3.1 La logique de réécriture

Dans cette section nous présentons les concepts de base de la logique de réécriture permettant de faciliter la compréhension de notre travail. Pour plus de détails, le lecteur peut se référer à (Marti-Oliet et Meseguer (1996)) ou (Meseguer (2002)).

La logique de réécriture est une logique de changement permettant l'expression du calcul concurrent non déterministe d'une manière très convenable, dans cette logique l'aspect statique des systèmes est représenté par une logique sous-jacente appelée logique équationnelle d'appartenance. L'aspect dynamique d'un système est représenté par des théories de réécritures décrivant les transitions possibles entre les états du système concurrent. Elle peut constituer un cadre sémantique prometteur offrant une base formelle rigoureuse pour la description des architectures logicielles (Meseguer et Talcott (1997)).

Une théorie de réécriture $\boldsymbol{R}$ est un quadruplet $(\sum,E,L,R)$, où $(\sum,E)$ désigne la signature définissant la structure des états du système, L est un ensemble d'étiquettes et R est un ensemble de règles de réécritures (notées $[t] \rightarrow [t']$) modélisant les transitions possibles entre les états du système concurrent.

Etant donné une théorie de réécriture, nous disons que $\boldsymbol{R}$ implique une formule $[t] \rightarrow [t']$ si et seulement si elle est obtenue par une application finie des règles de déduction suivantes:

1. La réflexivité : pour chaque [t] $\in T_{\Sigma,E}(X)$, $\overline{[t] \rightarrow [t]}$

2. La congruence : pour chaque $f \in \sum_n$ , $n \in \mathbb{N}$ ,

$$\frac{[t_1] \rightarrow [t_1'] \ \ldots \ [t_n] \rightarrow [t_n']}{[f(t_1, \ldots, t_n)] \rightarrow [f(t_1', \ldots, t_n')]}$$

3. le remplacement : pour chaque règle $t(x_1, \ldots, x_n) \rightarrow t'(x_1, \ldots, x_n)$ dans R :

$$\frac{[w_1] \rightarrow [w_1'] \ \ldots \ [w_n] \rightarrow [w_n']}{[t(w/x)] \rightarrow [t'(w'/x)]}$$

4. la transitivité :

$$\frac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

Un autre aspect important de la logique de réécriture est qu'elle représente un cadre logique et sémantique général dans lequel des langages et des modèles de calcul de nature largement différentes ont été représentés. Dans ce contexte, nous pouvons citer sans être exhaustifs, le lambda calcul, les systèmes de transitions étiquetés, les réseaux de Petri, la machine chimique abstraite, CCS, LOTOS, E-LOTOS (Meseguer (2002)).

D'autre part, les concepts théoriques de la logique de réécriture sont implémentés à travers le système Maude de haute performance, défini par J. Meseguer (Clavel et al. (2002)). Ce langage a été grandement influencé par le langage OBJ3, plus précisément la partie équationnelle de Maude inclut OBJ3 comme sous langage. Un programme écrit dans le langage déclaratif Maude représente une théorie de réécriture de la logique de réécriture, c'est-à-dire, une signature et un ensemble de règles de réécriture. Le calcul dans ce langage correspond à la déduction en logique de réécriture. En plus, Maude supporte la programmation fonctionnelle et la programmation orientée objet. Il a été donc utilisé pour la spécification, le prototypage, et la vérification d'un large éventail d'applications. Un autre aspect qui favorise l'utilisation de ce langage dans la vérification des systèmes est la présence d'un ensemble d'outils tels que son model-checker LTL, son analyseur ITP, son outil de complétion Knuth-Bendix et son analyseur de terminaison et de cohérence.

## 3.2 Présentation du langage SADL

SADL (Structural Architecture Description Language) est un langage de description d'architectures proposé par le laboratoire SRI (aux Etats-Unis). Il est basé comme tous les autres ADLs, sur les concepts de composant, de connecteur, et de configuration. Pour mieux présenter la syntaxe de ce langage, nous allons considérer une portion de l'architecture SADL `compiler` (figure 1a) décrivant l'exemple du compilateur (figure 1b) repris de (Moriconi et Riemenschneider (1997)).

Cette architecture inclut la déclaration de deux composants `lexical_analyzer` et `Parser`, après le mot clés `COMPONENTS`, et la déclaration d'un connecteur `token_channel` après le mot clés `CONNECTORS`.

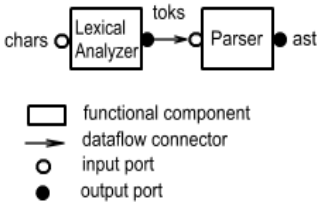
Un composant dans SADL peut être de différents types : une fonction, un module, un processus, ou encore une variable. Il est caractérisé aussi par une interface définie par un

ensemble de ports. Chaque port identifie un point d'interaction entre le composant et son environnement. Ces ports peuvent être, soit des ports d'entrées (comme par exemple le port `char_iport` du composant `lexical_analyzer`), soit des ports de sorties (comme le port `token_oport` du même composant).

```
compiler: 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
```

*a: L'architecture SADL*




*b: Le diagramme Box-and-arrow*

FIG. 1 – *L'architecture SADL du compilateur*

Un connecteur dans SADL est caractérisé par le type d'éléments portés. La structure globale d'une architecture SADL est décrite en termes d'une configuration, qui peut contenir deux types d'éléments: les connexions (`CONNECTION`) permettant d'associer des connecteurs à des ports de type compatibles et les contraintes (`CONSTRAINT`) qui sont utilisées pour lier des objets nommés ou pour mettre en place des restrictions sémantiques sur de telle liaison dans une architecture.

Une architecture SADL peut avoir une structure modulaire, cela est réalisable grâce à un mécanisme d'importation. Dans l'exemple considéré, les trois premières lignes servent à importer les différents types utilisés dans l'architecture `compiler` : les types `character`, `token` et `ast` sont importés d'une autre spécification appelée `compiler_types`; Le prédicat du type `Function` est importé du style `Functional_Style;` Les prédicats `Dataflow_Channel` et `connects` sont alors importés à partir du style `Dataflow_Style`.

La particularité du langage SADL est qu'il est dédié à la description structurale des hiérarchies d'architecture, à différents niveaux d'abstractions, grâce à un mécanisme de raffinement explicite. Ce mécanisme permet la transformation systématique d'une architecture abstraite vers une autre, contenant beaucoup plus de détails, en respectant un ensemble explicite de règles de transformations. L'aspect raffinement n'est pas du sort de cet article, pour d'amples détails le lecteur peut se référer à (Belala et al. (2007)), ou (Moriconi et Riemenschneider (1997)).

Comme l'objectif du langage SADL, ou tout autre ADL, est la définition et l'analyse formelle d'architectures logicielles, nous lui avons associé, dans un travail antérieur (Belala et al. (2007)), un modèle mathématique adéquat pour décrire le comportement d'un composant architectural et vérifier certaines de ses propriétés fonctionnelles via le model checker LTL de l'environnement Maude. Or, les propriétés non fonctionnelles spécifient

dans une architecture logicielle, les besoins qui viennent s'ajouter à ceux déjà considérés avec les propriétés fonctionnelles. Elles favorisent une implantation correcte du logiciel. La prise en compte de ces propriétés dans une architecture SADL sera réalisée en enrichissant le modèle, basé logique de réécriture, déjà proposé dans (Belala et al. (2007)).

# 4 Les propriétés non fonctionnelles dans le langage SADL

Les systèmes informatiques sont essentiellement construits pour satisfaire certains besoins fonctionnels. Pourtant, ces systèmes ont aussi des besoins additionnels qui caractérisent leur aptitude d'exister et de s'adapter aux variations de l'environnement dans lequel ils sont situés. Ces besoins sont dans la plupart du temps qualifiés d'être non fonctionnels. Deux types d'approches sont utilisées pour traiter l'aspect non fonctionnel dans les systèmes informatiques, les approches orientées produit (″product-oriented″) et les approches orientées processus (″process-oriented″) (Mylopoulos et al (1992)). Le premier type (orientées produit) porte sur l'évaluation des propriétés non fonctionnelles dans le produit logiciel final, c'est-à-dire déterminer s'il satisfait ou non certains besoins non fonctionnels par la définition généralement d'un ensemble d'attributs quantitatifs. Les approches orientées processus, prennent en considération les propriétés non fonctionnelles, en même temps que les propriétés fonctionnelles, durant le processus du développement de logiciel.

Notre contribution se situe dans ce deuxième type d'approches et son objectif est de fournir au concepteur de l'application des notations formelles permettant de caractériser l'aspect non fonctionnel de son logiciel cela au cours de la phase de conception architecturale.

En fait, les ADLs qui sont censés décrire de façon formelle l'architecture d'un système à un niveau d'abstraction élevé, peuvent être dotés de mécanismes pour l'expression et la vérification de ce type de propriétés. Actuellement, les recherches sur les ADLs, issus de milieux académiques, changent d'orientation et visent l'extension de ces langages avec des notations syntaxiques (généralement connues sous le nom de contrats) décrivant les services offerts par un élément architectural (composant, connecteur ou configuration), et aussi ceux dont il a besoin pour fonctionner. Dans ce contexte, nous suggérons un support mathématique formel, permettant l'expression et la vérification des propriétés non fonctionnelles au niveau des architectures logicielles décrites en SADL, tout en exploitant l'expressivité et la puissance de la logique de réécriture via son langage Maude. La présentation de ce modèle se fera en deux étapes pour une meilleure compréhension. Nous rappelons d'abord le modèle générique et extensible, basé logique de réécriture, associé à une architecture SADL quelconque, à travers un exemple. Nous procédons ensuite à l'extension de ce modèle avec des concepts bien définis de la logique de réécriture, pour la prise en compte de l'aspect non fonctionnel dans une architecture SADL.

## 4.1 Un modèle sémantique pour une architecture SADL

SADL est un langage de description d'architecture dont la vocation n'étant pas l'analyse, ou la description comportementale d'une architecture logicielle, nous l'avons alors étendu dans (Belala et al. (2007)), pour prendre en compte ces concepts. Nous lui avons défini un modèle extensible, basé logique de réécriture, permettant de raisonner formellement sur

l'architecture d'un système écrite dans ce langage et de l'implémenter. Le modèle théorique associé à une architecture SADL représente une théorie équationnelle de la logique équationnelle d'appartenance (`"membership equational logic"`) qui est une sous logique de la logique de réécriture. Il est noté : $(\sum, E \cup A)$, où $\sum$ est la signature du modèle, c'est-à-dire la spécification de l'ensemble de sortes, de sous sortes et de l'ensemble des opérations utiles pour décrire statiquement une architecture, E représente l'ensemble des équations du modèle, et enfin A représente l'ensemble des attributs équationnels des opérations.

En effet, nous avons adopté une approche générique qui consiste à spécifier séparément, chaque élément architectural SADL dans un module fonctionnel Maude, par conséquent, nous avons cinq modules génériques Maude présentés dans la figure 2. Le module fonctionnel `Architecture` est unique et il représente un modèle générique d'une architecture SADL. Elle reste valide quelque soit l'exemple d'architecture considéré. Dans le cas de la spécification `compiler` de la figure 1a, sa transcription en logique de réécriture se fait via le module Maude `Compiler` (figure 3) en enrichissant le module `Architecture` avec des opérations spécifiques instanciant, dans la partie équation (eq) du module, les ports, les composants, le connecteur, la connection et l'architecture globale avec successivement les noms suivants : `char-iport`, `token-iport`, `token-oport`, `base-ast-oport`, `lexical-analyzer`, `parser`, `token-channel`, `token-flow`, et `compiler`.

A travers cette formalisation, il est clair que la spécification d'une architecture SADL est réalisée de façon modulaire et lisible. L'expressivité et la flexibilité de la logique de réécriture ont permis la déclaration très naturelle des opérateurs définis afin de préserver la syntaxe du langage SADL (voir la clause équation de la figure 3). De plus, cette approche de formalisation peut directement servir à l'extension de l'architecture logicielle SADL. En particulier, nous pouvons ajouter d'autres éléments architecturaux pour décrire le comportement d'un composant par exemple, nous pouvons aussi formaliser le concept de raffinement utilisé dans ce langage.

## 4.2 Extension du modèle avec l'aspect non fonctionnel

Actuellement, aucune suggestion efficace et complète n'a été faite pour considérer l'aspect non fonctionnel d'une architecture logicielle dans un ADL. Le but de cette section est de proposer une extension du modèle sémantique précédent orientée dans ce sens.
De par leur complexité et leur haut niveau d'abstraction, les propriétés non fonctionnelles sont habituellement exprimées d'une manière informelle sans langages ni outils supportant leur expression ou leur validation. La prise en compte de ce type de propriétés dans un ADL constitue une tâche importante pour son succès, elles doivent être considérées dès le début du processus pour pouvoir les estimer avant la phase d'implémentation de ces systèmes. Nous procédons à leur intégration dans SADL selon deux niveaux : la spécification ensuite la validation. Notre approche est présentée dans la section suivante, à travers l'exemple simple et assez générique considéré jusqu'ici : architecture SADL du compilateur (figure 1a). Notons que le formalisme logique de réécriture via son langage Maude offre des spécifications exécutables et prêtes à l'analyse en tirant profit des outils de l'environnement Maude.

```
fmod Port is
/permet la spécification de la notion de port d'entrée/sortie d'une
architecture SADL.
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 /pour spécifier la structure d'un composant SADL.
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
endfm                                                prec 24] .
                                        ***
fmod Connector is /pour modéliser l'interaction entre les composants.
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
endfm                                                prec 24] .
                                        ***
fmod Configuration is
/ce module décrit la relation entre un connecteur et les ports de type
compatible.
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 / permet la définition d'une architecture SADL.
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
```

FIG. 2 – *Les modules Maude pour formaliser une architecture SADL*

```
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 : -> Architecture .
eq compiler = ARCHITECTURE
          [ char-iport : SEQ-character -> base-ast-oport : ast ]
COMPONENTS
    lexical-analyser : 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
```

FIG. 3 – *L'architecture "compiler" en Maude*

### 4.2.1 La spécification

Pour pouvoir décrire l'aspect non fonctionnel des architectures SADL, nous proposons d'intégrer une nouvelle partie, appelée configuration non fonctionnelle, à la structure globale d'une architecture SADL. Ceci est réalisable en modifiant l'opération principale (2ème opération du module `ARCHITECTURE` dans la figure 2), qui génère une architecture SADL comme suit:

```
op  ARCHITECTURE_COMPONENTS_CONNECTORS_CONFIGURATION_NFCONFIGURATION_
: Head SetComponent SetConnector SetCon NfConf -> Architecture [ctor
prec 25] .
```

où `NFCONFIGURATION` est l'entête indiquant le début de la spécification non fonctionnelle et `NfConf` dénote la sorte du terme algébrique spécifiant la structure statique de l'information non fonctionnelle dans l'architecture logicielle.

En fait, notre approche de modélisation des propriétés non fonctionnelles repose sur un principe très général qui déclare que les besoins de satisfaction d'une propriété non fonctionnelle peuvent être fonctionnels ou non fonctionnels. Pour cela nous introduisons une autre construction aussi importante que la précédente:

`NfReq` **`REQUIRE`** `{ SetReq }`

Où `NfReq` représente une propriété non fonctionnelle liée à un port de sortie (point d'accès au service) ou à un connecteur (support du service) et `SetReq` représente une liste des besoins fonctionnels et/ou non fonctionnels permettant de garantir cette propriété.

La spécification des offres en terme de services fonctionnels est définie par la construction :

`element1` **`PROVIDE`**`{ element2 . FReq }`

Pour exprimer que l'élément architectural `element1` fournit le service fonctionnel `FReq` à l'élément architectural `element2`.

Ces constructions supplémentaires utiles pour décrire l'information non fonctionnelle dans une architecture SADL sont implémentées dans un module fonctionnel Maude présenté dans la figure 4 :

```
fmod NfConfiguration is
extending Component Connector .
sorts NfProp FProp NfReq FReq SetReq NfStatement NfConf .
subsort NfReq FReq < SetReq .
subsort NfStatement < NfConf .
op _._ : ConnectorName NfProp -> NfReq [ctor prec 21 ] .
op _._ : OPortName NfProp -> NfReq [ctor prec 21] .
op _._ : ConnectorName FProp -> FReq [ctor prec 21] .
op _._ : OPortName FProp -> FReq [ctor prec 21] .
op _._ : IPortName FProp -> FReq [ctor prec 21] .
op none : -> SetReq [ctor] .              op none : -> NfConf [ctor] .
op __ : SetReq SetReq -> SetReq [ctor assoc id: none comm prec 22] .
op _REQUIRE{_} : NfReq SetReq -> NfStatement [ctor prec 23 ] .
op _PROVIDE{_} : IPortName FReq -> NfStatement [ctor prec 23 ] .
op _PROVIDE{_} : ConnectorName FReq -> NfStatement [ctor prec 23] .
op __ : NfConf NfConf -> NfConf [ctor assoc id: none comm prec 24 ] .
endfm
```

FIG. 4 – *Le module Maude pour gérer l'information non fonctionnelle*

Pour expliciter notre approche de conception du module `NfConfiguration`, nous l'illustrons à travers l'exemple du compilateur (figure 1). Une configuration non fonctionnelle possible peut être décrite au niveau de la figure 5.

```
fmod CompilerNfConf is
including Compiler NfConfiguration .
op nfconf : -> NfConf [ctor] .
eq nfconf =
    token-oport . NoTokenLost REQUIRE{ token-channel . BufferSize }
    token-iport PROVIDE{ token-channel . ReadSpeed }
    base-ast-oport . NoAstLost REQUIRE{ token-channel . NoTokenLost }
    token-channel PROVIDE{ token-oport . BufferSize }
    token-channel . NoTokenLost REQUIRE{ token-oport . NoTokenLost
           token-iport . ReadSpeed } .
   endfm
```

FIG. 5 – *Configuration non fonctionnelle pour l'architecture "compiler"*

Ainsi, dans ce module (figure 5), l'information non fonctionnelle est définie par des propriétés non fonctionnelles ainsi que des besoins fonctionnels et/ou non fonctionnels garantissant ces propriétés. L'opération constante `nfconf` décrit, à l'aide d'une équation, l'information non fonctionnelle associée à la configuration de l'architecture `Compiler`. Dans cette équation, sont spécifiés deux propriétés non fonctionnelles `NoTokenLost` et `NoAstLost` et deux services fonctionnels `BufferSize` et `ReadSpeed`. La propriété non fonctionnelle `NoTokenLost` est liée, dans la première ligne, au port de sortie `token-oport` du composant `lexical-analyzer`, et dans la cinquième ligne, au connecteur `token-channel`. La satisfaction de cette propriété nécessite, dans la première ligne, un service fonctionnel `BufferSize` de la part du connecteur `token-channel`, et dans la cinquième ligne, un besoin non fonctionnel `NoTokenLost` attaché au port de sortie `token-oport` du

composant `lexical-analyser` plus un besoin fonctionnel `ReadSpeed` lié au port d'entrée `token-iport` du composant `Parser`. La deuxième ligne de l'équation spécifie un service fonctionnel `ReadSpeed` offert par le port d'entrée `token-iport` au connecteur `token-channel`, et dans la quatrième ligne le connecteur `token-channel` fournit le service fonctionnel `BufferSize` au port de sortie `token-oport`. La propriété non foctionnelle `NoAstLost` est attachée au port de sortie `base-ast-oport` du composant `parser`, la satisfaction de cette propriété nécessite un besoin non fonctionnel `NoTokenLost` de la part du connecteur `token-channel`.

### 4.2.2 La vérification

L'extension d'une architecture SADL par l'expression de l'information non fonctionnelle facilite aisément l'analyse des propriétés fonctionnelles ou non fonctionnelles qui y lui sont reliées. Dans notre cas le processus de vérification consiste à tester l'inclusion de l'ensemble des besoins requis par une propriété non fonctionnelle dans l'ensemble des besoins contenus dans l'information non fonctionnelle. La mise en oeuvre de ce processus peut être réalisée par un module système Maude appelé `validate,` donné au niveau de la figure 6, contenant l'ensemble des règles de réécritures formalisant ce processus.

```
mod validate is   including CompilerNfConf .
op check : NfConf NfReq -> Bool .
op _IN_ :  NfStatement NfConf -> Bool .
var oprt  : OPortName .               var fp : FProp .
var iprt : IPortName .                var nfp : NfProp .
var setrq : SetReq .                  var conf : NfConf .
var nfs : NfStatement .
vars nfr1 nfr2 : NfReq .
eq nfs IN nfs conf = true .
eq nfs IN conf = false [owise] .
crl [1] : check(oprt . nfp REQUIRE{ concn . fp setrq }  conf , oprt . nfp) =>
check(oprt . nfp REQUIRE{ setrq } conf , oprt . nfp) if concn PROVIDE{ oprt .
fp } IN conf .
crl [2] : check(concn . nfp REQUIRE{ iprt . fp setrq } conf , concn . nfp) =>
check(concn . nfp REQUIRE{ setrq } conf , concn . nfp) if iprt PROVIDE{ concn
. fp } IN conf  .
crl [3] : check(nfr1 REQUIRE{ nfr2 setrq } conf, nfr1) => check(nfr1 REQUIRE{
setrq } conf, nfr1) if check(conf , nfr2) => true .
crl [4] : check(oprt . nfp REQUIRE{ concn . fp setrq }  conf , oprt . nfp) =>
false if not( concn PROVIDE{ oprt . fp } IN conf ) .
crl [5] : check(concn . nfp REQUIRE{ iprt . fp setrq } conf , concn . nfp) =>
false if not( iprt PROVIDE{ concn . fp } IN conf ) .
crl [6] : check(nfr1 REQUIRE{ nfr2 setrq } conf, nfr1) => false if check(conf
, nfr2) => false .
rl [7] : check( nfr1 REQUIRE{ none } conf, nfr1) => true .
endm
```

FIG. 6 – *Le module "validate"*

L'opération principale dans ce module est appelée `check.` Elle est utilisée pour vérifier si une propriété non fonctionnelle est garantie ou non par rapport à une information non fonctionnelle décrite. Alors le processus de vérification consiste à éliminer chaque besoin requis s'il est garanti. Plus précisément, nous éliminons successivement les besoins fonctionnels s'ils sont offerts par d'autres éléments (`crl [1]` et `crl [2]` de la figure 6) et

les besoins non fonctionnels s'ils sont valides par rapport aux éléments qui lui sont associés (`crl [3]`). Si un des besoins fonctionnels et/ou non fonctionnels requis par une propriété non fonctionnelle n'est pas établi, le résultat de vérification se réécrit à false (les règles `crl [4]` et `crl [5]` pour les besoins fonctionnels, et la règle `crl [6]` pour les besoins non fonctionnels). La dernière règle de réécriture sert donc à confirmer la validité d'une propriété non fonctionnelle.

Un exemple d'utilisation de ce module est donné dans la figure 7, pour garantir la propriété non fonctionnelle `token-oport . NoTokenLost,` déclarée dans l'information non fonctionnelle associée à la configuration de l'architecture `Compiler`. Notons que le modèle théorique étendu est assez générique, il peut être appliqué à n'importe quelle architecture SADL et quelque soit la propriété non fonctionnelle considérée. Nous projetons dans un travail futur, d'appliquer cette modélisation à un exemple significatif de grandeur naturelle.

```
Maude> rew check(nfconf , token-oport . NoTokenLost )  .
rewrite in validate : check(nfconf, token-oport . NoTokenLost) .
rewrites: 4 in 1625750370999ms cpu (0ms real) (0 rewrites/second)
result Bool: true
```

FIG. 7 – *Un exemple d'utilisation de l'opération "check"*

# 5 Conclusion

Nous nous sommes intéressés aux ADLs dans le but de disposer de moyens d'analyse des propriétés non fonctionnelles pour des systèmes en cours de conception. Souvent, les propriétés fonctionnelles d'une architecture logicielle sont considérées dès le début du processus de développement. Elles ont été bien définies et précisément spécifiées. Cependant, peu d'investigations ont été consacrées aux propriétés non fonctionnelles qui sont aussi importantes que les propriétés fonctionnelles. Dans cet article, nous avons proposé une approche permettant l'intégration de l'information non fonctionnelle dans l'architecture logicielle. Nous avons enrichi le langage SADL par quelques constructions permettant de qualifier l'aspect non fonctionnel des architectures SADL. L'idée de base derrière cette approche, est que la validation d'une propriété non fonctionnelle nécessite au préalable l'assurance de certains besoins fonctionnels et/ou non fonctionnels.

Le modèle proposé, qui est à base de la logique de réécriture, s'avère un cadre théorique approprié pour décrire d'une part, les propriétés non fonctionnelles des éléments architecturaux et d'autre part, l'assemblage de ces éléments pour garantir les propriétés non fonctionnelles spécifiées. En plus, la réalisation de cette spécification en Maude offre une spécification exécutable, lisible et extensible, du à la flexibilité et la puissance de ce langage. Ainsi, nous avons étudié une première solution au problème de la prise en compte des propriétés non fonctionnelles à un niveau haut et abstrait dans le processus de conception des systèmes logiciels. Nos futurs travaux dans ce cadre, porteront sur l'utilisation de ces résultats pour élargir l'éventail des exemples à des applications significatives de la réalité (systèmes embarqués, transactionnels, etc.). Nous nous intéresserons en particulier aux architectures logicielles reconfigurables.

## Références

Aagedal, J. (2001), *Quality of Service Support in Development of Distributed Systems*. Thesis doctor Scientiarum, Department of Informatics, University of Oslo.

Belala, F., F. Latreche, M. Benammar. (2007), *A Formal Semantic Framework for SADL Language*. In Proc. of ACIT'07, 26-28 Nov. 2007, Lattakia, Syria, pp.44.

Christopher, V. E., H. Osama, K. Khaled (2005), *Addressing Non-Functional Properties in Software Architecture using ADL*. Proc. 6th Australasian Workshop on Software and Systems Architectures (AWSA 2005), Brisbane.

Clavel, M., F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada (2002). *Maude: Specification and programming in rewriting logic*. Theoretical Computer Science, 285:1, pp.87–243.

Franch, X., P. Botella (1998), P*utting non-functional requirements into software architecture*. Proceedings of the 9th International Workshop on Software Specification and Design, pp. 60 – 67.

Garlan, D., R. Monroe, D. Wile (1997). *Acme: An Architecture Description Language*. Proceedings of CASCON'97.

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

Medvidovic, N., R. M. Taylor (2000). *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, Vol 26, n°1, pp. 70-93.

Meseguer, J., C. Talcott (1997), *Formal Foundations for Compositional Software Architectures*. Position Paper, OMG-DARPA-MCC Workshop on Compositional Software Architectures.

Meseguer, J. (2002), *Rewriting Logic Revisited*. Université Illinois de Urbana-Champaign, USA.

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

Mylopoulos, J., L. Chung, and B. Nixon (1992). *Representing and Using Non-Functional Requirements: A Process-Oriented Approach*, IEEE Transactions on Software Engineering, 18(6), pp. 483-497.

Rosa, N. S., P. R. F. Cunha, R. R George (2002), *Process$^{NFL}$: A language for describing non-functional properties*. Proceedings of the 35th Annual Hawaii International Conference (HICSS), pp. 3676-3685.

## Summary

Software architecture notion is appeared around the years 1990 and presented as a fully-fledged discipline. Many architecture description languages (ADLs) have been proposed in the literature. They offer complementary capacities for the development and the architectural analysis of a software system. As the main objective of an ADL is to provide an explicit and precise specification, several features for the components and the connectors are desirable. Among these features, the non functional properties specifying user's needs, are very important and encourage a correct implantation of the software. SADL is an architecture description language not devoted to architecture analysis, then, we extend it to take in account the non functional specifications of architectural system. In this paper, we propose a formal framework based rewriting logical, permitting non functional properties description of the SADL components and connectors as well as the architectural application analysis which guarantees these properties.
**Key words.** Architecture Description Language, Formal Approaches for the Software Architecture Validation, Rewriting Logic, Maude language.