J. Software Engineering & Applications, 2010, 3, 1067-1079
doi:10.4236/jsea.2010.311126 Published Online November 2010 (http://www.SciRP.org/journal/jsea)

Scientific Research

A Tile Logic Based Approach for Software Architecture Description Analysis

Aïcha Choutri[1], Faiza Belala[1], Kamel Barkaoui[2]

[1]LIRE Laboratory, University Mentouri of Constantine, Constantine, Algeria; [2]CEDRIC, Conservatoire National des Arts et Métiers, Paris, France.
Email: aichachoutri@gmail.com, belalafaiza@hotmail.com, kamel.barkaoui@cnam.fr

Received July 13th, 2010; revised August 31st, 2010; accepted September 3rd, 2010.

ABSTRACT

A main advantage of Architecture Description Languages (*ADL*) *is their aptitude to facilitate formal analysis and verification of complex software architectures. Since some researchers try to extend them by new techniques, we show in this paper how the use of tile logic as extension of rewriting logic can enforce the ability of existing ADL formalisms to cope with hierarchy and composition features which are more and more present in such software architectures. In order to cover ADL key and generic concepts, our approach is explained through LfP* (*Language for rapid Prototyping*) *as ADL offering the possibility to specify the hierarchical behaviour of software components. Then, our contribution goal is to exploit a suitable logic that allows reasoning naturally about software system behaviour, possibly hierarchical and modular, in terms of its basic components and their interactions.*

***Keywords*:** *Tile Logic, LfP Model, Software Architecture, Hierarchical Composition*

1. Introduction

Software Architecture (SA) has emerged as a principle method of understanding and developing high level structures of complex software systems. Nowadays, SA artifacts are becoming the foundation for building families of such systems. Then, the problem of ensuring as early as possible the correctness of an SA occupies increasing importance in the development life-cycle of software products. Formal approaches should be used to describe software architectures and express their dynamic evolution so that one could reason on them.

Over past two decades there has been considerable research devoted to modeling and analysis of software architectures; among other, architecture description languages (or ADL) as suitable notation for SA formal specification. Their common advantage is their impressive body representation of evidence about the utility of architectural modeling and analysis [1]. Some of them attempted to provide behavioral modeling and analysis via numerous complementary behavioral formalisms. However, the most of applied approaches share the goal of defining mismatches in component composition.

Recently, some researchers in industry and academia try to extend these ADL, by new techniques to analyze and validate architectural choices, both behavioral and quantitative, complementing traditional code-level analysis technique [2].

According to this motivation, we show through this paper how the tile logic as extension of rewriting logic can support ADL artifacts allowing and enforcing formal reasoning on software system behavior and dynamics. In particular, we show that tile system is closely joined to important inherent aspects of ADL, especially hierarchical behavior of components and compositional one. We explain our proposed approach concretely through LfP architecture description language [3] as ADL offering the possibility to specify the hierarchical behavior of software components (in terms of LfP-BD diagrams) in addition to their structure that can be also of hierarchical nature (LfP-AD).

Tile Logic [4] has been introduced for modular description of open, distributed and interactive systems. It constitutes a rewriting logic [5] extension taking into account rewriting with side effects and rewriting synchronization. So, it supports modular specification of concurrent system behaviors, their interaction and synchronization semantics thanks to its particular rules of rewrite called *tiles*. These rules can be instantiated with special terms in a given context. The main idea behind this

logic is to impose dynamic restraints on terms to which a rule may be applied by decorating rewrite rules with observations ensuring synchronizations and describing interactions.

Authors of [6,7] have proposed a mapping approach of LfP architectural description into rewriting logic in order to formalize its semantics and exploit this latter for hierarchical verification of some properties using model checking. The interest of such approach is the well care in purely formal way of concurrency in a distributed configuration through executable specification.

In this paper, we use Tile logic strength the results obtained in [6,7] related to formalization and verification of LfP software architecture description via rewriting logic and its Maude language. Hence, owing to tile logic elements, we can deal with and preserve naturally the hierarchical structure and the hierarchical behavior of SA. Besides, executable specification of tile systems may be naturally defined by mapping tile logic into rewriting logic which is a semantic basis of several language implementations. In particular the proposed model implementation requires developing a set of strategies to control rewriting by discarding computations that do not correspond to any deduction in tile logic.

In the remainder of this paper, Section 2 situates our work among analogous ones in order to better surround its problematic. Section 3 is devoted to summarize basic concepts of tile logic. In Section 4, we present our approach to map an architectural description into tile logic. It is then explained and illustrated in Section 5 through LfP language. So, a successful classic case study of Producer/Consumer system is considered to show how we proceed to clutch tile logic to software architecture description language. Some comments evaluating our contribution are presented in Section 6. Last section concludes our work and gives its perspectives.

2. Contribution Setting

It is not a novel idea to give a formal specification for software architecture. Most of existing ADL concentrates on providing a precise mathematical semantics for software architecture description of a system. All well known semantic formalisms for ADL give limitation when software architectures deal not only with structural aspects but also with behavioral ones. Besides, hierarchy concept, synchronization and reconfiguration ones restrict the use of these ADL formalisms. In the case of Rapid [8] for example, based on POSets (Partially Ordered event Sets), only architectural components interactions are formalized in addition to the architectural elements. Also, Wright [9] uses a subset process algebra named CSP (Communicating Sequential Processes) formal notation to describe partially components abstract behavior in a simple manner. Recent related works focus on defining rewriting logic based model for some existing ADL. For example, authors in [10] define a mapping of CBabel concepts into rewriting logic. A particular attention has been given to specify syntactic constructs semantic of this ADL example. LfP [3] integrating UML notations and state/transition automaton has been considered by the work of to associate to each of its views (LfP-AD and LfP-BD diagrams) possibly hierarchical an appropriate semantic meaning based on rewriting logic too. In fact, rewriting logic has been recognized as a semantic framework to model software system architecture. However, complexity of such systems structure and behavior induces rewriting logic flat models that do not preserve compositional and hierarchical features of ADL, so they are often difficult to manage. In this paper, we will show that tile logic based model, even if it is more complex to grasp, is much more appropriate and efficient.

Tile logic is an extension of rewriting logic, supporting modular description of concurrent system behaviors, and their interaction and synchronization semantics. These assets delegate it as being the most suitable formalism to support semantic of more complex ADL, namely those having the possibility to describe hierarchical and modular specification of distributed and open architectural applications such as LfP descriptions.

In particular, both static and dynamic views of LfP architecture are naturally translated according to the tile system structure (space), and its computation flow (time), preserving hierarchical and compositional nature of this architectural description. Furthermore, this tile model may be then successfully exploited to formal reasoning on such descriptions and their analysis.

3. Tile Logic

In this section, we recall some fundamental concepts of the underlying semantic framework, namely tile logic. More interested readers may consult [4].

Tile Logic has been introduced for modular description of open, distributed and interactive systems. This formalism reminiscent to term rewriting and concurrency theory, constitutes a rewriting logic [5] extension taking into account rewriting with side effects and rewriting synchronization. Although, ordinary format of rewrite rules allows state changes expression and concurrent calculus in a natural manner, it lacks tools to express interactions with the environment, *i.e.*, rewrite rules can be freely instantiated with any term in any context. The main idea behind tile logic is to impose dynamic restraints on terms to which a rule may be applied by decorating rewrite rules with observations ensuring

synchronizations and describing interactions.

The obtained rules are then called tiles, defining the behavior of partially specified components, called configurations, in terms of actions related to their input and output interfaces (the possible interactions with the internal/external environment).

Each tile, having one of the forms in **Figure 1**, expresses that initial configuration s evolves to the final configuration t via the tile α, producing the effect b, which can be observable by the rest of the system. Such a rewriting local step is allowed, only if the sub-components of s (its arguments) evolve to the sub-components of t producing an effect a, which acts as a trigger for α application.

Arrows s and t of the tile α (in **Figure 1**) are called configurations (system states). They are algebraic structures equipped with operations of parallel and sequential composition. Each system configuration has both input and output interfaces responsible of system interactions with the environment.

Arrows a and b decorating tile α (in **Figure 1**) are also algebraic structures, they define observable effects (actions) for coordinating local rewrites through configuration interfaces (input and output ones).

In general, configurations and observations give rise to two categories having the same class of objects (interfaces). The former (horizontal so-called configuration) defines effects propagation; the latter (vertical so-called observations) describes state evolution. Then, double category, the superposition of these two categories of cells, should be considered as a natural model for tile system.

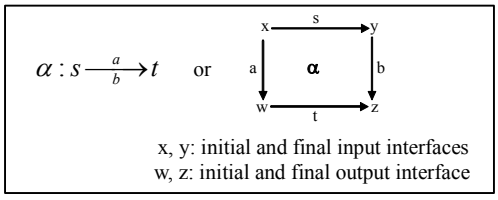

Figure 1. A tile representation.

Definition 1. *A tile system is a* 4*-tuple T = (H, V, N, R) where H, V are monoïdal categories with the same set of objects $O_H = O_V$, N being a set of rule names and R: N⟶ H x V x V x H a function where for each α in N, if R(α) = (s, a, b, t), then s:x → y, a:x → w, b:y → z and t: w → z, for suitable objects x, y, z and w; with x, y the input interfaces and, w, z the output interfaces.*

Auxiliary tiles set may be necessary to specify consistent interfaces rearrangements. A standard set of inference rules (**Figure 2**) allows building larger rewriting steps, by composing tiles freely via horizontal (through side effects), vertical (computational evolution of configuration) and parallel (concurrent steps) operations.

In the recent literature, tiles representing an extension of the SOS specification approach are designed for dealing with open states. They seem apt for many current applications [11-13]. Indeed, they have been used with success to model in detail several application classes such as coordination languages (triggers and effects represent coordination protocols) and software architecture styles [13,14]. This paper contributes to another meaningful and interesting application of Tile logic, it generalizes the approach proposed in [15] by defining a common formal model based on tile logic for architecture descriptions.

4. Tile Logic for Architecture Description Languages

Tile logic has been exploited as a common semantic framework to define both abstract software architectures and their behaviors [16]. Generally, each component in a distributed system specification is described by a set of external ports, ensuring interactions with the environment and an internal behavior, specified by a set of tiles, to

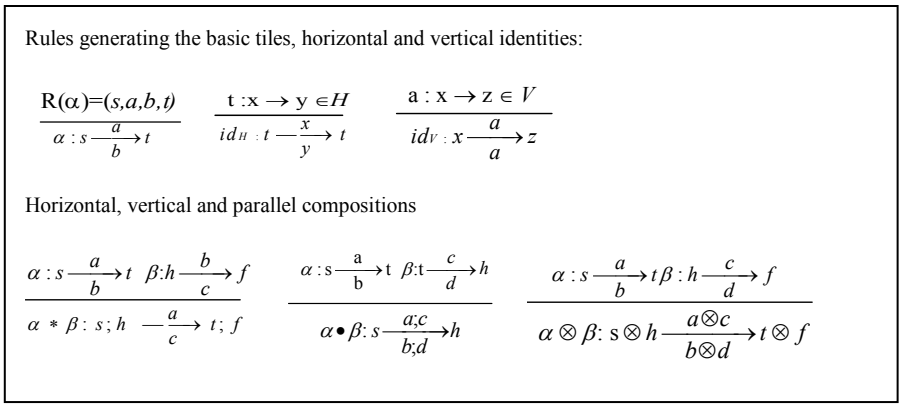

Figure 2. Inference rules.

a deliver component functionalities. It is defined by a tile system, where objects of the two categories correspond to component interfaces. Vertical category (observations) defines its possible actions corresponding to component required/provided services and horizontal category generates all component possible configurations. Gathered together via a set of tiles, horizontal and vertical categories define expected behavior of the underlying component. Starting from a basic set of tiles and deduction rules of this logic, system global behavior defined as a coordinated evolution of its subcomponents, is naturally deduced. We take in what follows all these ideas formulated through a generic semantic framework applicable for ADL either their semantics. It will be then applied on a specific case study.

4.1. Modeling Process and Fundamental Definitions

Since in existing ADL, a complete software architecture description has to consider static architectural configuration and dynamic evolution (behavior) of software system, our proposed model follows the same separation of concerns. Two distinct, but not completely independent views are considered. For the first one, expressed by a set of existing components, connectors (often considered as particular components) and connection topology, we associate a tile system integrating these components (and connectors) definitions. In the second view, state transition system, which is usually devoted to define component and system behavior, is extended and formally defined in the context of tile logic.

Then, a software component, that could be either a computation component or a connector, may have two definitions according to its granularity:

- A partial (external) description, associated mainly to its interface, evolving an abstract data type for each component type (sort, operations and their properties).
- A detailed (internal) description given by a tile system formalizing both structure and behavior component. Computation or storage component may be either primitive or composite. For later case, the associated tile system is a composition of a set of hierarchical sub tile systems. If the component is a connector (communication entity), the associated tile system defines the corresponding communication protocol.

In the following, we present the four essential steps of tile model generation for any software architecture description:

Step 1: Each software architectural *Configuration* involves an associated tile system *Configuration–TS* definition. Its horizontal and vertical categories are formed (see Definition 2) by the involved architectural elements (components, connectors, ports, etc.).

- Morphisms of the horizontal category formalize connection points (in *Configuration*) that are characterized by essentially components names and interfaces, in terms of algebraic terms.
- On the other hand, morphisms of the vertical category (observations) formalize provided/required components services.
- Both categories have the same set of objects that are algebraic terms representing hidden sub components models that will be described at the next lower level.
- The set of tiles expresses configurations evolution and its propagation over all sub-components. These tiles formalize binding connection. Some auxiliary tiles may be added to these basic tiles in order to deal with some particular ADL connection aspects like dynamic connection (components synchronization) as it is the case for parameterized dynamic connection tiles defined in [16].

Definition 2. *A tile system based model Configuration-TS for a given architectural configuration Configuration* = (*Components, toplogy-connection*) *is defined by the* 4*-tuple* ($H^{Conf\text{-}TS}$, $V^{Conf\text{-}TS}$, $N^{Conf\text{-}TS}$, $R^{Conf\text{-}TS}$) *such that:*

- $O^{Conf\text{-}TS}$ *is composed of all components abstract representation deduced from their partial description.*
- *Morphisms of* $H^{Conf\text{-}TS}$ *are formed upon algebraic terms specifying component interfaces intervening in connection points.*
- *Morphisms of* $V^{Conf\text{-}TS}$ *are simplified to design only two types of services: MsgCset, MsgCget provided or required by a component of name C under a message form.*
- $N^{ConF\text{-}TS}$ *is the set of tile names. Each one is obtained by combining a computation component name with a communication component name. If there is no explicit connector* (*i.e.*, *case of dynamic connector*)*, the second name will be that of other computation component connected to the first one.*
- $R^{Conf\text{-}TS}$: $N^{Conf\text{-}TS} \rightarrow H^{Conf\text{-}TS} \times V^{Conf\text{-}TS} \times V^{Conf\text{-}TS} \times H^{Conf\text{-}TS}$ *is a function defining tiles that formalize binding connection deduced from configuration topology-connection. Two general forms of tiles can be defined here*:

$$C1C2 : s1 \xrightarrow[MsgC2get]{MsgC1set} s1' \quad (1)$$

$$C2C3 : s2 \xrightarrow[MsgC3set]{MsgC2get} s2' \quad (2)$$

*s*1, *s*1*'*, *s*2, *s*2*'* *are configurations, i.e. morphisms of* $H^{Conf\text{-}TS}$, *C*1, *C*3 *are computation component name, C*2 *is a connector name and MsgC1set, MsgC2get are possible observation, i.e. morphisms of* $H^{conf\text{-}TS}$.

Step 2: For each computation component type in the present software architecture configuration, we associate a particular tile system (Definitions 3 and 4).

- If it is the case of a composite computation component (Composite-Component), then the tile system in question *Component-Configuration-TS* (Definition 3) formalizes its architectural configuration and has the same definition of *Configuration-TS*. Hence, it is necessary to perform all the process steps while adopting appropriate notations to avoid any confusion.

Definition3. *Tile system Composite-Component-TS modeling composite component of any ADL architecture description, is defined in the same way as for configuration-TS, by the* 4*-tuple* ($H^{CompCConf\text{-}TS}$, $V^{CompCConf\text{-}TS}$, $N^{CompCConf\text{-}TS}$, $R^{CompCConf\text{-}TS}$).

- If the software component is primitive (Primitive-Component), its associated tile system *Primitive-Component-TS* (Definition 4) is conceived on the basis of state transition system elements which is usually used to describe the component behavior. The set of common objects of its both categories (horizontal and vertical) is composed of all elementary architectural elements of the component like attributes, ports, constraints, or other eventual annotations. Horizontal (configurations) category formalizes in this case component states, vertical (observations) one specifies all its possible transition triggers given in terms of labels. Two general tile forms are then defined for either simple transition or hierarchical one.

Definition4. *Tile system Primitive-Component-TS modeling a primitive component software architecture described by a given state transition system with respect to a given ADL, is defined by the* 4*-tuple* ($H^{PrimComponent\text{-}TS}$, $V^{PrimComponent\text{-}TS}$, $N^{PrimComponent\text{-}TS}$, $R^{PrimComponent\text{-}TS}$) *such that:*

- $O^{PrimComponent\text{-}TS}$ *is composed of algebraic terms associated to all architectural elements of the component.*
- *Morphisms of* $H^{PrimComponent\text{-}TS}$ *are formed upon algebraic definitions of states in terms of well defined tuples.*
- *Morphisms of* $V^{PrimComponent\text{-}TS}$ *are formed upon algebraic definitions of all existing transition labels.*
- $N^{PrimComponent\text{-}TS}$ *is the set of tile names. Each one is either a transition name or defined to formally identify an unnamed transition.*
- $R^{PrimComponent\text{-}TS}$: $N^{PrimComponent\text{-}TS} \rightarrow H^{PrimComponent\text{-}TS} \times V^{PrimComponent\text{-}TS} \times V^{PrimComponent\text{-}TS} \times H^{PrimComponent\text{-}TS}$ *is the function defining tiles that formalize all transitions.*

  Two general tile forms are proposed for either simple (*tile*3) *or hierarchical* (*tile*4) *such as l and id denote*

$$t{:}s1 \xrightarrow[idH]{l} s2 \tag{3}$$

$$t : s1 \xrightarrow[t-TS-trigger]{l} s2 \tag{4}$$

  label transition and identity morphism respectively while t-TS-trigger denotes transition effect that acts as trigger for hidden transition tile system.
- Finally, If the component is a communication one (Communication-Component), its tile system model is defined in the same way as for primitive computation component.

Step 3: Since transitions in the state/transition model that describes primitive component may be hierarchical, they hide other sub state transition systems. So, recursively we associate to each hierarchical transition, at the next lower level, a novel tile system as we have proceed in the previous step.

Step 4: Step 3 is repeated until all transitions become simple.

It is clear that our approach for defining a formal model based on tile logic for any ADL is quite generic and too general as it provides concise and complete semantics definitions for all important common ADL concepts, mainly the hierarchy. Indeed, this may reduce considerably the semantic gap between ADL noted before. We will show through the instantiation of this modeling process to the LfP language case, that the resulting model covers entirely and naturally its formal semantics while preserving its modular and hierarchical structure. Therefore, we resolve the usual problem (flat model) still posed in previous ADL formalization approaches (such as those based on Petri nets and rewriting logic models).

4.2. Case of LfP

LfP (Language for Prototyping) [2,16] allows describing both control and structuring aspects of a SA. It is regarded as a language having characteristics of a coordination language and an ADL.

4.2.1. LfP Software Architecture

LfP, like all other ADL, provides a concrete syntax to describe SA with a declarative style of components assembly. It offers two distinct and complementary views

to allow a complete description of software system. Architectural view uses LfP-AD diagram to define system architectural configuration and its components, behavioral view specification deals with system dynamic behavior in terms of hierarchical behavior diagrams: LfP-AD. LfP constructs are well defined in [17]. In what follows, we recall the most used ones in our approach context.

Architecture diagram (LfP-AD): It is LfP static model description defining the participating components to a system definition, their links and all system global declarations. More precisely, LfP-AD describes system software architecture as a graph whose nodes are the software entities (LfP classes) and their link edges are communication entities (LfP media). The interaction point's connection (LfP ports) allows these entities to be assembled. Any connection has to respect binding contracts (LfP binders).

Behavior diagram (LfP-BD): it specifies LfP class behavior, or a communication protocol associated to LfP media or a method execution flow belonging to LfP class or LfP media. Formally, this diagram type expresses component (class or media) behavior through state/transition automaton. Communication between component automatons is achieved by message queues. Since LfP allows the description of hierarchical behaviors, two types of transitions are defined in LfP-BD behavior model: simple transitions and hierarchical ones (method transitions, block transitions).

These latter, encapsulate other behaviors also expressed by sub-LfP-BD that may be reused. A transition can be provided with annotations, a guard or a post condition which must be checked.

Research works around LfP semantics formalization are all concentrated on Petri nets formalism in order to use their well known analysis tools. But in practice, these models have already proved their insufficiencies. Indeed, the hierarchical behavior greatly expressed in LfP components is not preserved by this translated model and even those based on rewriting logic, recently introduced by [5,6].

4.2.2. Tile Model for LfP

Figure 3 summarizes graphically basic LfP methodology defined in [3] to which our approach (bold part) is transplanted as a new possible alternative formal semantic model emphasizing both software system compositional behavior and hierarchical one. LfP software architecture description is naturally translated into set of tile systems. Each tile system allows the formal specification of an LfP software component (LfP-AD, class, class instance, media) or sub-component one (methods, block transitions) including the declaration of their constraints.

The proposed generic model is instantiated to LfP language with respects to its methodology different views (Functional, properties and implementation). This will help us then to give more precise and complete definition of all different LfP architectural elements semantics in Tile logic based framework. Besides, executable specification of this new LfP model may be naturally defined by mapping [18,19] tile logic into rewriting logic which is a semantic basis of Maude language [20] and will extend the proprieties view in LfP methodology, by the specification and formal analysis of other behavior constraint kinds, strongly related to LfP features (synchronization, hierarchical behavior, etc.). Therefore, we apply to LfP software architecture description, the construction tile model process presented previously, step by step while highlighting the hierarchy preservation.

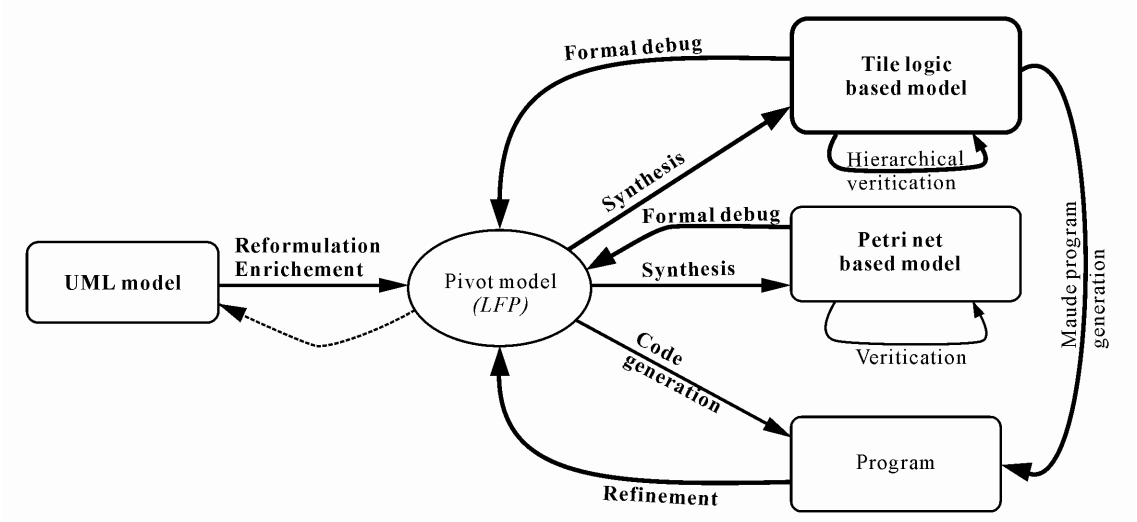

Figure 3. LfP methodology extension.

Table 1. Mapping LfP Concepts into Tile Logic.

LfP elements	Tile model elements
LfP-AD	***LfP-AD-TS***
Class	Sub-sort of *component* sort
Media	Sub-sort of *component* sort
Binder	Sorts, Configurations, Binder-tiles
Global declarations	Global definitions
Component LfP-BD	*Component* sort, ***Component-TS***
Local declarations	Objects of horizontal and vertical categories
1. Class LfP_BD	***1. Class-TS***
States	Configurations
Transition Name	Transition Name-Tile
Annotations, guards, post-conditions	Objects (interfaces or particular configurations)
Simple transitions	Simple-Transition-Tiles
Method transitions	Method-Tiles, ***Method-TS***
Block transitions	Entailed Block-Tiles, ***Block-TS***
Transition actions	Observations
2. Media LfP-BD	***2. Media-TS:*** Particular case of *Class-TS* with Simple-Transition-Tiles only.
Similar to LfP-BD class except: no media type and no method transition	

Table 1 summarizes the mapping of most LfP concepts into tile logic. It helps to dress the tile model.

LfP-AD Tile Based Model (Step 1):

Architectural view specified in LfP by an architecture diagram (LfP-AD) is formalized by a configuration tile system *LfP-AD-TS* as defined in Definition 2, enriched with global definitions as translation of global declarations of the LfP-AD (sorts for types, operations for constants and static instances). So, this Tile system is noted by the instantiated tuple ($H^{AD\text{-}TS}$, $V^{AD\text{-}TS}$, $N^{AD\text{-}TS}$, $R^{AD\text{-}TS}$) such as:

- $O^{AD\text{-}TS}$ is composed of component algebraic terms defined according to signatures as presented in (**Figure 4**). These terms are in fact abstraction of hierarchical component tile systems (*Component-TS*)
- $H^{AD\text{-}TS}$ (*configurations*) formalize binders as they support the binding connection point's semantics of *LfP* components. Their definition is based on signature given in (**Figure 5**).
- $V^{AD\text{-}TS}$ (*observations*) formalize *LfP* message sending and receiving. Each one may have one of the following form:
  MsgCset: send message by class C.
  MsgCget: receive message by class C.
  Msg-Mget: receive message by a media M.
  MsgMset: send message by media M.
- $N^{AD\text{-}TS}$ is set of tile names. Each one is defined by combining associated component and media names or the reverse noted in general: *B-CM, B-MC.*
- $R^{AD\text{-}TS}$: $N^{AD\text{-}TS} \rightarrow H^{AD\text{-}TS} \times V^{AD\text{-}TS} \times V^{AD\text{-}TS} \times H^{AD\text{-}TS}$ is a function to formalize binding contracts. For synchronous case, each resulting tile has one of the following forms with respect to general tiles (1) and (2) defined in the tile model construction process.

Where:

Conf-BCM2 = binder-CM(c, p, m, i1, i2+1, synch, po),
Conf-BMC1 = binder-MC(m, c, p, i1, i2, synch, po),
Conf-BMC2 = binder-MC(m, c, p, i1, i2+1, synch, po),
c: class-name, p: port, m: media-name, i1and i2: integer that represent binder capacity and message number respectively and po: policy.

For example, the tile *B-CM* expresses connection between any object of a class *c* that sends a message through its port *p* and the media *m* which must perform its routing towards a receiving class object. This connection can be accomplished only during execution with class and media instances, by horizontal composition (synchronization) of this tile with: on the one hand, a sender tile having MsgCset as effect, and a media tile having MsgMget as trigger, on the other hand. By analogy,

Sorts name, interface
Operations
… : string → name
(….,…): port, msg → interface

Sort component
SubSort computeComponent, connector < component
Operations
… : name, interface → component
NameC: component → name
InterfaceC: component → interface

Figure 4. Signatures for component algebraic definition.

Sort binder
Operations
Binder-CM: class-name, port, media-name, integer, integer, mode, policy → binder
Binder-MC: media-name, class-name, port, integer, integer, mode, policy → binder
Where
-Arguments of integer sort correspond respectively to binder storage capacity de and the number of messages.
-Sorts mode and policy are defined by:
Synch, Asynch: → mode
FIFO, LIFO, BAG: → policy

Figure 5. Signature of bine.

$$\text{B-CM} : \text{binder-CM(c, p, m, i1, i2, synch, po)} \otimes (\text{i2} < \text{i1}) \xrightarrow[\text{Msg.M.get}]{\text{Msg.C.set}} \text{binder-CM(c, p, m, i1, i2, synch, po)} \quad (5)$$

$$\text{B-MC} : \text{binder-MC(m, c, p, i1, i2, synch, po)} \otimes (\text{i2+1} < \text{i1}) \xrightarrow[\text{Msg.C.get}]{\text{Msg.M.set}} \text{binder-MC(m, c, p, i1, i2+1, synch, po)} \quad (6)$$

tile B-MC expresses a connection between media and any receiving class instance.

LfP-BD Tile Based Model (Step 2):
Through behavioral view, we generate *Component*-TS tile system to each LfP-BD diagram defined by the tuple ($H^{Component\text{-}TS}$, $V^{Component\text{-}TS}$, $N^{Component\text{-}TS}$, $R^{Component\text{-}TS}$) as given by Definition 4. Particularly, *Component-TS* would be either *Class-TS or Media-TS* and for simple and concise presentation we consider only primitive component which may have hierarchical behavior.

- $O^{Component\text{-}TS}$ is composed of semantic elements expressed as algebraic terms that correspond to component architectural elements like attributes, local declarations, or annotations defined in the component *LfP-BD*.
- $H^{Component\text{-}TS}$ (*configurations*) formalizes states of the underlined *LfP-BD*.
- $V^{Composant\text{-}}$ (*observations*) formalizes all transition actions (from simple action to sequential composition of actions, call method, perform block).
- $N^{Component\text{-}TS}$ is the set of transition names.
- $R^{Component\text{-}TS}$: $N^{Component\text{-}TS} \rightarrow H^{Component\text{-}TS} \times V^{Composant\text{-}TS} \times V^{Composant\text{-}TS} \times H^{Component\text{-}TS}$ is a function formalizing transitions in terms of tiles having forms (3) or (4).

In any form of tile, initial configuration could be a composition of initial transition state with eventual guards or precondition, as well as final configuration could be a composition of resulting transition state with eventual post-condition. For both cases, the used operator is $\otimes$.

Hierarchical Transition Formalization (Step 3):
All tiles associated to LfP hierarchical transitions, method transitions and bloc transitions, have to be in fact entailed by sub tile systems *Method-TS* and *Bloc-TS* associated to the hidden sub-LfP-BD respectively.

Bloc-TS = ($H^{Block\text{-}TS}$, $V^{Block\text{-}TS}$, $N^{Block\text{-}TS}$, $R^{Block\text{-}TS}$) is defined recursively in the same way as *Component-TS* replacing component name by LfP block name.

But, the tile system defined for a method transition is given by the tuple *Method-TS* = ($H^{method\text{-}TS}$, $V^{method\text{-}TS}$, $N^{method\text{-}TS}$, $R^{method\text{-}TS}$) such that:

- $O^{method\text{-}TS}$ is composed of semantic elements associated to attributes and annotations.
- $H^{method\text{-}TS}$, $V^{method\text{-}TS}$ *and* $N^{method\text{-}TS}$ are defined in similar way as for $H^{Component\text{-}TS}$, $V^{Component\text{-}TS}$ *and* $N^{Component\text{-}TS}$ respectively.
- $R^{method\text{-}TS}$: $N^{BD\text{-}TS} \rightarrow H^{BD\text{-}TS} \times V^{BD\text{-}TS} \times V^{BD\text{-}TS} \times H^{BD\text{-}TS\ TS}$ is a function formalizing only simple transitions as defined in the underlined sub-LfP-BD in terms of tiles of form (3).

Step 4: Lastly, step3 must be repeated with respect to the number of sub LfP-BD specified in LfP architecture description**.**

5. Case Study: Simple Client-Server Application

The aim of this section is to illustrate the proposed generic semantic model concepts to a classical client-sever (CS) architectural application example and show that tile logic provides effectively a powerful semantic framework for LfP description language in particular and for any ADL in general.

5.1. CS LfP Model

The architectural view (LfP-AD) of our application stated in **Figure 6** contains three components: two classes "Client" and "Server" connected via a media named "MsgPassing". Connexion is represented by interposing binders (messages queues with multiplicity and interaction semantics noted by annotations) at media binding ports level.

We will present a part of the whole LfP specification, example sufficient to deal with all architectural and behavioural LfP concepts mainly modular and hierarchical ones. We only give *server* class behaviour (**Figure 7**). The corresponding *LfP-BD* defines concurrent execution of actions requested by the client. It consists of three states *BEGIN, S1, END* and four hierarchical transitions *init, start, worker* and *daemon*.

The sub-LfP-BD (right of **Figure 7**) describes methods and blocks invoked by various associated hierarchical transitions. As it is shown in the figure, we have presented only three sub-LfP-BD. From top to bottom, the first one describes the init method role by a simple transition T1 (pseudo code). Both other sub-LfP-BD describe roles of block transitions *worker* (by a not detailed simple transition) and *daemon* (by two possible not detailed execution paths: transition methods *exec* and *notify*), respectively.

5.2. CS Tile Model

With respect to LfP definition instantiation given previously, we deduce a set of hierarchical tile systems (CS-AD-TS, Client-TS, Msg-Passing-TS, Server-TS, Start-TS, init-TS, Daemon-TS, worker-TS,…) for our client-server application following especially the proposed generation steps.

Step1:
Global definitions :
Opérations MaxClient: $\rightarrow 5$
MaxServeur: $\rightarrow 1$
CS-AD-TS = ($H^{CS\text{-}AD\text{-}TS}$, $V^{CS\text{-}AD\text{-}TS}$, $N^{CS\text{-}AD\text{-}TS}$, $R^{CS\text{-}AD\text{-}TS}$)
where:

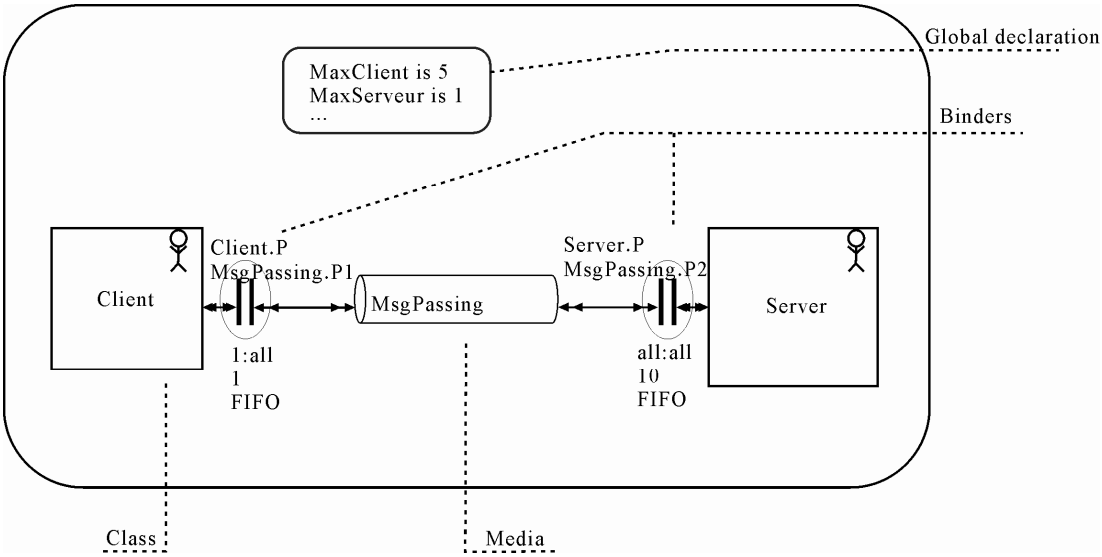

Figure 6. Client-Server LfP-AD (CS-LfP-AD).

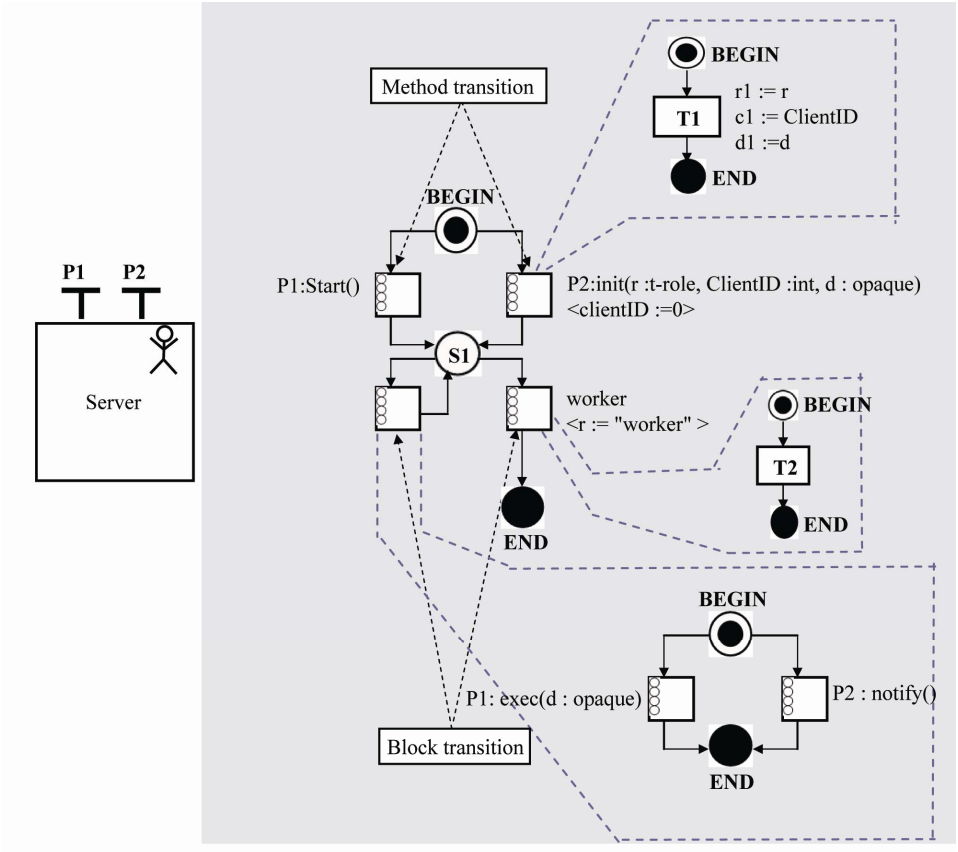

Figure 7. ClienFt-server LfP-BD (CS-LfP-BD).

- $O^{CS\text{-}AD\text{-}TS}$ = {Client-TS, Serveur-TS, MsgPassing-TS}. At this granularity level, elements of this set are name of the tile systems modeling LfP classes involved in the LfP-BD diagram.
- $H^{CS\text{-}AD\text{-}TS}$ = {binder-ClientMsgPassing (client, client.P, Msg-Passing, 1, i2, synch, fifo), binder-MsgPassingServer (MsgPassing, server, server.P, 10, i2, synch, fifo), binder-MsgPassing Client

(Msg-Passing, client.P, client, 1, i2, synch, fifo), binder-ServerMsgPassing (server, server.P, MsgPassing, 10, i2, synch, fifo)}. All binder declarations are considered by this configurations category (objects of the category).

- $V^{CS\text{-}AD\text{-}TS}$ = {Msg-ServerGet, Msg-ServerSet, Msg-ClientGet, Msg-MsgPassingGet, Msg-MsgPassing Set}. Here all observable actions are summarized.
- $N^{CS\text{-}AD\text{-}TS}$ = {B-ClientMsgPassing, B-MsgPassingClient B-MsgPassingServer, B-Server-Msgpassing}.
- $R^{CS\text{-}AD\text{-}TS}$: $N^{CS\text{-}AD\text{-}TS} \rightarrow H^{CS\text{-}AD\text{-}TS} \times V^{CS\text{-}AD\text{-}TS} \times V^{CS\text{-}AD\text{-}TS} \times H^{CS\text{-}AD\text{-}TS}$.

For synchronous case, we can define four basic tiles, with $R^{CS\text{-}AD\text{-}TS}$.

$$\text{B-Client-MsgPassing: binder-ClientMsgPassing(P1)} \otimes (i2 < 1) \xrightarrow[\text{Msg-MsgPassingGet}]{\text{Msg-ClentSet}} \text{binder-ClientMsgPassing(P1')} \otimes (i2 \leq 1)$$

with:

P1 = (client, client.P, MsgPassing, 1, i2, synch, fifo) P1' = (client, client.P, MsgPassing, 1, i2+1, synch, fifo)

$$\text{B-Server-MsgPassing: binder-ServerMsgPassing(P2)} \otimes (i2 < 10) \xrightarrow[\text{Msg-MsgPassingGet}]{\text{Msg-ServerSet}} \text{binder-ServerMsgPassing(P2')} \otimes (i2 \leq 10)$$

with:

P2 = (serveur, server.P, MsgPassing, 10, i2, synch, fifo) P2' = (serveur, server.P, MsgPassing, 10, i2+1, synch, fifo)

$$\text{B-MsgPassing-Client: binder-MsgPassingClient(P3)} \otimes (i2 < 1) \xrightarrow[\text{Msg-ClientGet}]{\text{Msg-MsgPassingSet}} \text{binder-MsgPassingClient(P3')} \otimes (i2 \leq 1)$$

With:

P3 = (MsgPassing, client, Msg-Passing.P, 1, i2, synch, fifo) P3' = (Msg-Passing, client, MsgPassing.P, 1, i2+1, synch, fifo)

$$\text{B-MsgPassing-Server: binder-MsgPassingServer(P4)} \otimes (i2 < 10) \xrightarrow[\text{Msg-ServerGet}]{\text{Msg-MsgPassingSet}} \text{binder-MsgPassingServer(P4')} \otimes (i2 \leq 10)$$

With:

P4 = (MsgPassing, server, MsgPassing.P, 10, i2, synch, fifo)

P4' = (MsgPassing, server, MsgPassing.P, 10, i2+1, synch, fifo)

For example, the tile B-ClientMsgPassing means that starting from the configuration binder-ClientMsgPassing (client, client.P, Msg-Passing, 1, i2, synch, fifo) of the connection point between the Client and the media MsgPassing, while the number i2 of messages in the binder file is less than its capacity (equals to one), the client can set its message (Msg-ClientSet trigger) and so, the configuration evolves to binder-ClientMsgPassing (client, client.P, MsgPassing, 1, i2+1, synch, fifo) while i2 will be incremented. The final effect of this rewriting (Msg-MsgPassingGet) is necessary to ensure synchronization of rewrites with a particular tile (having Msg-MsgPassingGet as a trigger) to be defined into MsgPassing-TS.

The tile B-MsgPassingServer means that starting from the configuration binder-MsgPassingServer (MsgPassing, server, server.P, 10, i2, synch, fifo) of the connection point between the media Msg-Passing and the Server, while the number i2 of messages in the binder file is less than its capacity (equals to 10), the media can set a message received from the Client (Msg-MsgPassingSet trigger) and so, the configuration evolves to binder-MsgPassingServer (MsgPassing, server, server.P, 10, i2+1, synch, fifo) while i2 will be incremented. The final effect of this rewriting (Msg-SeverGet) is necessary to ensure synchronization of rewrites with a particular tile (having Msg-ServerGet as a trigger) to be defined into Server-TS.

By analogy, the meaning of both other tiles can be easily deduced.

Step 2:

At this granularity level, we associate to each hierarchical tile systems defined in the previous step a refined model in terms of tile systems. As we have noted just the server class LfP-BD is concerned by our approach illustration. Its associated model is given by the following tile system:

Server-TS = ($H^{Server\text{-}TS}$, $V^{Server\text{-}TS}$, $N^{Server\text{-}TS}$, $R^{Server\text{-}TS}$)

where:
$O^{Server\text{-}TS}$ is a set of algebraic terms defining Server class attributes, guards and post-conditons.
$H^{Server\text{-}TS} = \{BEGIN_{Serveur}, S1_{Serveur}, END_{Serveur}\}$. Objects of this category correspond to tile systems associated to LfP-BD states.
$V^{Server\text{-}TS}$ = {P1: call start (), P2: call init (r: t-role, ClientID:int, d:opaque), Perform daemon, Perform worker}. These category objects (tile systems) are formed around actions or transitions of the LfP-BD.
$N^{Server\text{-}TS}$ = {Start, Init, Daemon, Worker}
$R^{Server\text{-}TS}$: $N^{Server\text{-}TS} \rightarrow H^{Server\text{-}TS} \times V^{Server\text{-}TS} \times V^{Server\text{-}TS} \times H^{Server\text{-}TS}$

As basic tiles example let us consider one method-transition tile: init tile (7) and one bloc-transition-tile: worker tile (8).

$$\text{init: } BEGIN_{Serveur} \otimes < ClientID\ !=0 \xrightarrow[init(r,ClientID,d)]{Msg-ServeurGet} S1_{serveur} > \quad (7)$$

$$\text{worker: } S1_{Serveur} \otimes < r: = \text{'worker'} \xrightarrow[perform\ worker]{Msg-ServeurGet} END_{serveur} > \quad (8)$$

Tile (7) expresses evolution involved by firing action Msg-ServerGet (tile trigger) of initial configuration $BEGIN_{Serveur}$ composed with a new interface<ClientID ! = 0>(precondition) that is noted by $BEGIN_{serveur} \otimes$ <ClientID ! = 0>. Effect of this tile is materialized by the init method invocation. We notice that in a lower abstraction level, we can define a more detailed model through their associated hidden tile system init-TS (step 3). The meaning of tile (8) is similar to that of tile (7).

Step 3:
For each transition method or Bloc-transition is associated a hidden sub-LfP-BD, its corresponding sub-tile system is naturally deduced preserving hierarchical behaviour view, let us consider the init-TS and the worker-TS respectively:
init-TS = ($H^{init\text{-}TS}$, $V^{init\text{-}TS}$, $N^{init\text{-}TS}$, $R^{init\text{-}TS}$)
$O^{init\text{-}TS}$ = {r, clienID, d, r_1, c_1, d_1}
$H^{init\text{-}TS}$ = {$BEGIN_{init}$, END_{init}}
$V^{init\text{-}TS}$ = {r_1:= r; c_1:= ClientID; d_1:= d; ! (r⊗ClientID⊗d)}
Sequential composition of T1 actions with local context destroy ! (r⊗ClientID⊗d).
$N^{init\text{-}TS}$ = {T1}
$R^{ini\text{-}TS}$: $N^{init\text{-}TS} \rightarrow H^{init\text{-}TS} \times V^{init\text{-}TS} \times V^{init\text{-}TS} \times H^{init\text{-}TS}$

$$\text{T1: } BEGIN_{init} \xrightarrow[r_1:= r;\ c_1:= ClientID;\ d_1:=d;\ !(r\otimes ClientID\otimes d)]{init(r,ClientID,d)} END_{init} \quad (9)$$

Similarly, we define the following detailed tile system:
worker-TS = ($H^{worker\text{-}TS}$, $V^{worker\text{-}TS}$, $N^{worker\text{-}TS}$, $R^{worker\text{-}TS}$)
$O^{worker\text{-}TS} = O^{Server\text{-}TS} \cup$ {<r:= worker}
$H^{worker\text{-}TS}$ = {$BEGIN_{worker}$, END_{worker}} with END_{worker} = S1 a Server state
$V^{worker\text{-}TS}$ = {T2 actions}
$N^{worker\text{-}TS}$ = {T2}
$R^{worker\text{-}TS}$: $N^{worker\text{-}TS} \rightarrow H^{worker\text{-}TS} \times V^{worker\text{-}TS} \times V^{worker\text{-}TS} \times H^{worker\text{-}TS}$

$$\text{T2: } BEGIN_{worker} \otimes < r = \text{'worker'} > \xrightarrow[T2\ actions]{perform\ worker} END_{worker} \quad (10)$$

6. Comments and Evaluation

The proposed Tile model for a given software architecture description is in fact a set of hierarchical tile systems. Each one denotes and formalizes a particular architectural description element (architecture configuration, composite component, primitive component). Our defined modeling process is then applied to a particular ADL, LfP allowing modular and hierarchical description of not only software structure, but also its behavior. Besides, LfP language formalization has some limits provided by the based flat model since it doesn't preserve the concepts of hierarchy and modularity [3,6,7]. Through detailed evaluation of our approach, we highlight and summarize the important features of our proposed tile model and that strength rewriting model defined in [6].

It is very interesting and very useful to define semantics of both architectural and behavioural description in the same formalism. This advantage has been well enough provided by rewriting logic but, it is even better, if the formalism in question could be self-sufficient to provide in a natural way a complete compositional model thanks to some owner theoretical and practical characteristics. This extraordinary privilege is exactly and easily offered by so called tile logic; a particular extension of rewriting logic. This unique formal support gives good solution to extend ADL notation at a Meta level that will facilitate formal analysis and verification.

Tile logic as extension of rewriting logic supports not only static and dynamic aspects of any ADL, but also practical ones, such as hierarchical behaviour composition and synchronization. Its specific inference rules are in

turn instantiated not freely as for rewriting logic, but in specific context and then composed implicitly to deduce further possible behaviours. For example, in the case of CS-tile model, the effect (*call init* (r, ClientID, d)) of *init* tile (7), defined in *Server-TS*, will be the trigger of the tile T1 (9), defined in the sub tile system *init-TS*. During a computation, if *init* tile is executed, the system instantiates T1 automatically (rewriting synchronization rule) and then applies the horizontal tiles composition (*init* $*$ T1) that involves T1 execution which creates its local context at the beginning and destroys it at the end. In the same way, *worker* tile (8) effect (*perform worker*), defined in *server-TS*, triggers the tile T2 (10) defined in the sub tile system *worker-TS*. Though these concrete examples, we remark that compositional and hierarchical behaviour semantics are really and naturally considered with a good rewriting synchronization; what is not so evident in ordinary rewriting logic.

7. Conclusions

The main contribution of this paper is to propose tile logic based modelling process of a system architectural description preserving the modularity and the hierarchy of initial specification, avoiding its flat form. We highlight the interest of our approach through LfP description language since it is equipped with rich notation allowing modular and hierarchical specification of software systems.

Tile logic taking into account state changes with side effects and rewriting synchronization, has been proven as a high level (Meta) semantic framework, more appropriate to deal naturally with important ADL features that are more frequent, namely their structural and behavioural hierarchy as well as any components composition or synchronization. This particular advantage is due to the theoretical and practical characteristics of tile logic: categorical structures, guided rewriting via observations, flexible formats of configurations, tiles composition through interfaces, exploitation of three dimensional views (horizontal for structure, vertical for behaviour evolution and the third dimension for distribution).

It is obvious that defining ADL semantics within a complete semantic framework facilitates formal executing and analyzing of software system specification. This work enforces and offers new possibilities for formal debugging, checking and executing the obtained tile logic model by mapping it into rewriting logic [18,19]. We note here that the executable specification to be obtained consequently, has the advantage to discard useless deductions thanks to guided rewritings in particular. Hence, our proposed model for LfP architecture description can provide an executable specification in Maude system [20] and its practical checking tools [21,22]. Our model also opens way to several other perspectives such as semantics enrichment of LfP component behaviour under particular constraints such dynamic reconfiguration.

REFERENCES

[1] D. Garlan and B. Schmerl, "Architecture-Driven Modelling and Analysis," *Proceedings of the* 11*th Australian Workshop on Safety Related Programmable Systems* (*SCS'*06), Vol. 69, 2006, pp.3-17.

[2] P. Zhang, H. Muccini and B. X. Li, "A Classification and Comparison of Model Checking Software Architecture Techniques," *The Journal of Systems and Software*, Vol. 83, No. 5, May 2010, pp. 723-744.

[3] D. Regep, "LfP: un langage de spécification pour supporter une démarche de développement par prototypage pour les systèmes répartis," thèse de doctorat, Université de Paris VI, 2003.

[4] R. Bruni, "Tile logic for Synchronized Rewriting of Concurrent Systems," PhD. Thesis, Computer Science Department, University of Pisa, Pisa, 1999.

[5] J. Meseguer, "Conditional Rewriting Logic as a Unified Model of Concurrency," *Journal of Theoretical Computer Science*, Vol. 96, No. 1, April 1992, pp. 73-155.

[6] C. Jerad and K. Barkaoui, "On the Use of Rewriting Logic for Verification of Distributed Software Architecture Description Based LfP," 16*th IEEE International Workshop on Rapid System Prototyping* (*RSP'*05), June 2005, pp. 202-208.

[7] C. Jerad, K. Barkaoui and G. Touzi, "A. Hierarchical Verification in Maude of LfP Software Architectures," Lecture Notes in Computer Science, Vol. 4758, 2007, pp. 156-170.

[8] D. C. Lukham, "Rapide: A language Toolset for Simulation of Distributed System by Partial Ordering of Events," *DIMACS Workshop on Partial Order Methods in Verification* (*POMIV*), July 25-26, 1996, pp. 329-358. http://pavg.stanford.edu/rapide/rapide-pubs.html

[9] R. J. Allen, "Formal Approach to Software Architecture," PhD. Thesis, University of Carnegie Mellon, Pittsburgh, May 1997.

[10] C. Braga and A. Sztajnberg, "Towards a Rewriting Semantics for a Software Architecture Description Language," *Proceedings of the Brazilian Workshop on Formal Methods*, Vol. 95, May 2004, pp. 149-168.

[11] R. Bruni, J. Fiadeiro, I. Lanese, A. Lopes and U. Montanari, "New Insights into the Algebraic Properties of Architectural Connectors," *International Federation for Information Processing*, Vol. 155, 2004, pp. 367-380.

[12] R. Bruni, U. Montanari and U. Sassone, "Observational Congruences for Dynamically Reconfigurable Tile Systems," *Theoretical Computer Science*, Vol. 335, No. 2-3, May 2005, pp. 331-372.

[13] D. Hirsch and U. Montanari, "Consistent Transformations

for Software Architecture Styles of Distributed Systems," *Workshop on Distributed Systems*, Vol. 28, 1999.

[14] F. Arbab, R. Bruni, D. Clarke, I. Lanese and U. Montanari, "Tiles for Reo," *Lecture Notes in Computer Science*, Vol. 5486, 2009, pp. 37-55

[15] A. Choutri, F. Belala and K. Barkaoui, "Towards a Tile Based LfP Semantics," *Second International Conference on Research Challenges in Information Science* (*RCIS* 2008), June 2008, pp. 9-16.

[16] C. Bouanaka, A. Choutri and F. Belala, "On Generating Tile System for Software Architecture: Case of a Collaborative Application Session," *ICSOFT'*07, Barcelone, 2007, pp. 123-126.

[17] F. Gilliers, "Développement par prototypage et Génération de Code à partir de LfP, un langage de modélisation de haut niveau," thèse de doctorat, Université P. & M. Curie, Paris, 2005.

[18] R. Bruni, J. Meseguer and U. Montanari, "Tiling Transactions in Rewriting Logic," *Electronic Notes in Theoretical Computer Science***,** Vol. 71, April 2004, pp. 90-109.

[19] J. Meseguer and U. Montanari, "Mapping Tile Logic into Rewriting Logic," *Lecture Notes in Computer Science*, Vol. 1376, 1998, pp.62-91.

[20] M. Clavel, F. Duran, E. Eker, S. N. Marti-Oliet, Lincoln, P. J. Meseguer and C. Talcott, "Maude 2.0 Manual," June 2003. http://maude.cs.uiuc.edu

[21] S. Eker, J. Meseguer and A. Sridharanarayanan, "The Maude LTL Model Checker and Its Implementation," *Proceedings of the* 10*th International Conference on Model Checking Software*, 2003, pp. 230-234.

[22] R. Bruni, A. LIuch and U. Montanari, "Hierarchical Design Rewriting with Maude," *Electronic Notes Theoretical Computer Science*, Vol. 238, No. 3, June 2009, pp. 45-62.

