A Tile Logic Based Semantics for Mobile Software Architectures

Chafia Bouanaka  
LIRE Laboratory, Mentoury University,  
Constantine, Algeria  
Chafia_Bouanaka@umc.edu.dz

Faiza Belala  
LIRE Laboratory, Mentoury University,  
Constantine, Algeria  
Faizabelala@hotmail.com

Kamel Barkaoui  
CEDRIC Research Laboratory,  
CNAM, Paris, France  
Kamel.barkaoui@cnam.fr

Abstract. The paper deals with the specification of dynamic reconfiguration of complex systems. We show how dynamic reconfiguration and especially components mobility, affects the ongoing behaviour of the reconfigured parts and its side effects on the system as a whole. Our approach is based on software architectural viewpoint of hierarchical systems, considered as sets of independent components and links. Nevertheless, the recursive description of components structure and particularly hierarchical and compositional generation of components and system behaviour, needs additional concepts. For this reason, we define a tile logic based model, where interfaces constitute basic objects that undergo the two dimensional dynamics of reconfigurable software systems. Spatial distribution of interfaces defines system / hierarchical component structural configuration. Dynamic reconfiguration is then regarded as a redistribution operation of these interfaces. On the other hand, tuples of interfaces values define system / hierarchical component behavioural configurations. The incremental and compositional definition of both system structure and behaviour is obtained thanks to the horizontal composition operator of tile logic that establishes dynamic connections between components interfaces and at the same time propagates the corresponding action effects on the environment.

Complex Systems; Software Architectures; Mobility; Tile logic; Categorical semantics.

1. INTRODUCTION

Large and complex systems are understood as assemblages of interacting components where each one may be itself a system. The recursion stops when a component is considered to be atomic; i.e., any further internal structure is not of interest and can be ignored depending on the level of abstraction considered (Besnard et al., 2006). The way system components fit together is the structure of the system. In fact, describing system structure consists of identifying its components, their functionalities, and their means of interaction. Besides, the structure of the system can be viewed as a means that allows the relevant components to interact appropriately with each other causing system global behaviour. Therefore, system structure can be considered as what enables it to generate its behaviour from the behaviours of its subcomponents. A reasonable question will be: is it possible to define system global behaviour recursively or hierarchically as it can be easily done for system structure without a need to flatten such structure. Additionally, if the system is subject to dynamic reconfigurations causing a reorganization of its structure and hierarchy, what repercussions are met on system global behaviour.

Over the past decade, software architecture discipline has emerged as an appropriate approach to augment our ability to comprehend large systems. Its main contribution regarding object oriented paradigm is that it has succeeded to detach coordination from computation by putting forward the concept of connectors. A system can then be understood as a set of independent components that can be composed in an arbitrary manner via a set of connectors (Allen, 1997). However, software architectural view of hierarchical and reconfigurable systems needs additional concepts to tackle the recursive description of components structure and particularly hierarchical and compositional generation of system behaviour.

1.1 Related Work

An abrupt work is being done in the literature to formalize the space dimension of mobile software architectures. Location-aware formalisms enrich software components structure with a location element either explicitly or implicitly. Darwin adds an integer variable to software components structure to indicate the effective machines where they are located. Similarly, an abstract data type is defined to model software components locations in CommUnity (Lopes et al., 2006). This form of...
mobility formalization is best suited for software applications where physical entities are roaming across a network of locations. The GSM cellular phones handoff that has been specified by Oliveira et al. in (Oliveira et al., 2005) using CommUnity and Road assistance application specified in (Bruni et al., 2008) represent significant examples of this category of applications focusing on physical mobility. Nevertheless, restricting component motion to a simple location update is not always the desired solution. Nowadays, applications are more interested in logical mobility of software components abstracting away their physical or effective distribution, which is actually made transparent to software applications thanks to middlewares. Therefore, another category of formalisms provide an implicit notion of location represented by composite components that host mobile components. π-ADL (Oquendo et al., 2004) does not differentiate between a composite component representing a location and an ordinary composite component. Mobis (Ciancarini et al., 1998) defines a primitive, called space, for representing nested components, locations and channels. Composite components are called locations in LAM (Xu et al., 2003), each one contains an internal connector to attach subcomponents to the physical location. Other formalisms abstract away the notion of location and exhibit dynamic topologies of software architectures via special kinds of connectors. Transient connectors have been used by several models (Fiaderio et al., 1999; Wermelinger, 1998; Bouanaka et al. 2007) to dynamically link nomadic components that wish to interact. Border connectors have been defined in C2Sadl (Medvidovic et al., 2001) to allow binding of mobile components to network hosts. Albeit, this offers more flexibility to software architectures, it constrains the underlying network to be massively interconnected.

Graphs equipped with graph transformation tools are actually emerging as a natural approach to specify software architectures and their dynamic reconfiguration. Three subclasses of graph based approaches have been identified by Bradley et al. in (Bradley et al., 2004). The first subclass considers a clear separation of concerns between individual components behaviour and software architecture dynamics, as it has been done in (Allen et al., 1998; Le Metayer, 1998; Milner, 2008). We highlight that in such case, at least two different formalisms are necessary to specify the same dynamic software architecture, without mentioning side effects of one aspect on the other one. A second subclass of models uses graphs for both aspects of dynamics. Baresi et al. (Baresi et al., 2002) consider class diagrams as software architectures and attempt to specify state evolutions, like class instantiation or object destruction, by graph transformations. This leads to additional and sometimes complex formal constructs that are usually not natural. The third subclass tries to either graft several behavioural models to graph transformation ones like constraint solving approach used in (Hirch et al., 1999) or enrich existing languages constructs (Magee et al., 1998) to support graph based concepts. Nevertheless, this is also achieved in a complicated and not natural manner.

1.2 Paper Contribution

In the context of reconfigurable and mobile software systems, two types of configurations are manipulated. Structural configurations define spatial distribution of system entities and behavioural configurations express system states. The two types of configurations are subject to dynamic evolution with eventual side effects on each other.

Our aim, in the present work, is to adopt a unified semantic framework to specify both behavioural and dynamic reconfiguration evolutions of hierarchical software architectures. Components interfaces are basic objects that undergo the two dimensional dynamics of software systems. Spatial distribution of interfaces defines system structural configurations with dynamic reconfiguration regarded as a redistribution of these interfaces. Interfaces values correspond to behavioural configurations with value changes on interfaces as behavioural evolution. Incremental and compositional definition of both system structure and behaviour is obtained thanks to the horizontal composition operator of tile logic (Bruni, 1999), the semantic framework chosen; that establishes dynamic connections between components interfaces to construct larger states and at the same time propagates actions effects on the other components.

Instead of manipulating algebraic terms on interfaces that are too syntax dependent to express topology, distribution and connectivity in software architectures, hypergraphs are used. Unlike several graph based models for software architectures (Le Metayer, 1998; Milner, 2008; Hirch et al., 1999), visible interfaces correspond to nodes of software architecture hypergraph. Components and their eventual attachments correspond to hyperedges expressing interfaces distribution. Components hierarchy is obtained by recursively defining components as being software architecture. Depending on the level of abstraction considered, hierarchical components can hide their internal structure from the outside and subcomponents functionalities are accessible only through hierarchical components interfaces. Therefore, two types of interfaces are identified. Visible interfaces function is to receive data coming from either side,
incoming or outgoing, and simply relay it to the other side. A correspondence relation is established between visible and inner interfaces that represent subcomponents visible ones. Observable functionalities are specified by affixing interfaces with their corresponding actions and actions implementations. Each one is defined by a rewrite rule expressing interface possible evolution.

Since dynamic changes on software architecture configurations may interact with ongoing computations, components interfaces are considered as the common and central structure on which mutual side effects of the two types of dynamics are recovered. A rule-based unified approach is adopted for either type of software architectures dynamics. Whereas, term rewriting rules define local possible evolutions of components states, graph transformation rules express software architecture reconfigurations.

Validating syntactic constructions by a mathematical model is a necessary task. Moreover, this allows us taking advantage of formal tools offered to correctly reason about software architectures prototypes and verify some properties. Thereby, the second part of this work is dedicated to the construction of a mathematical model for reconfigurable software architectures.

The paper is organized as follows: section 2 recalls basic concepts of tile logic and its associated semantics. An incremental presentation of syntactic concepts adopted in this work is presented in section 3. Section 4 describes our main contribution which associates a categorical semantics on tile systems to reconfigurable software architectures. Syntactic and semantic concepts introduced in this work will be illustrated via a client/Server example. The paper is outlined by some concluding remarks and perspectives.

2 FUNDAMENTAL CONCEPTS

Rewriting logic (Meseguer, 1992) was proposed as a logic of changes dedicated to formally specify concurrent systems. In its initial version, local possible evolutions of system states are defined by rewrite rules of the form \( s \rightarrow s' \) if \( C \); expressing the fact that system state \( s \) can evolve to \( s' \) if some condition \( C \) is verified. Nevertheless, ordinary format of rewrite rules lacks tools to perceive interactions with the environment. Thus, rewrite rules can be freely instantiated with any term in any context abstracting away contextual information that may guide computation or deduction in reactive systems. To overcome this problem, two main extensions of rewriting logic have been made: tile logic (Bruni, 1999) and revisited rewriting logic (Bruni et al., 2006) which introduces frozen variables and generalized rewrite theories.

Tile logic has been proposed to impose dynamic constraints on terms to which a rule may be applied by decorating rewrite rules with observations ensuring synchronizations and describing interactions. The resulting rewrite rule is called a tile. To guarantee context-aware rewriting, rewrite rules have been enriched with a new concept of (input/output) interfaces. Input interfaces express contextual conditions triggering a rule and output interfaces define rewrite rule sides effects on its environment. In this manner, rewriting process is guided by system interactions. This considerably reduces state space of each computation step. As a consequence, modular specification of concurrent systems is possible where components can be conceived separately and then composed via their interfaces.

Thereafter, rewriting logic authors have been attracted by the expressive power of tile logic. Thus, they attempted to bring back tile logic concepts. On the one hand, the replace equational basis of rewriting logic with membership equational logic. On the other hand, they introduce a new notion of frozen variables as a means to guide computation by temporary forbiding the exploration of some computation branches.

2.1 Tile system

Given that tile logic is best suited for the specification of reactive and open systems, interfaces represent basic objects manipulated within this logic. Various schemes of interface connections can be constructed thanks to parallel and horizontal composition operators of tile logic, giving rise to system configurations that express spatial distribution of system state. Contextual data captured via interfaces invocations correspond to system basic actions that may be also composed horizontally (for action synchronization) or in parallel to construct larger observations. System behaviour is then defined by a set of tiles expressing local possible evolutions of partially specified components, i.e., containing variables, called configurations and corresponding to system local states. System computations are constructed through a coordinated evolution of its local configurations due to possible interactions with the inside and outside environments only.

Formally, a concurrent system is specified by a tile system obtained by superposing two categories: horizontal category constructs system possible states or configurations, and vertical category defines system observable actions. Superposition of the two categories is realized via the set of tiles, natural transformations, defining which elementary
local transitions are possible in a distributed state (configuration) according to some observations occurrences (triggers).

A tile \( \alpha \) has the following form \( s \xrightarrow{a} t \), stating that the initial configuration \( s \) can evolve to the final configuration \( t \) via \( a \), producing the effect \( b \), which can be perceptible by the environment. Such step is allowed only if arguments (subcomponents) of \( s \) can contribute by producing \( a \), which acts as a trigger of \( \alpha \).

**Definition 1** (Bruni, 1999). A tile system is a 4-tuple \( \mathfrak{H}=(H,V,N,R) \) where \( H \) and \( V \) are monoidal categories with the same set of objects \( O_H=O_V, N \) being a set of rules names and \( R: N \rightarrow H \times V \times V \times H \) is a function where for each \( \alpha \) in \( N \), if \( R(\alpha) = \{s,a,b,t\} \), then \( s:x \rightarrow y, a:x \rightarrow z \), \( b:y \rightarrow w \), and \( t:z \rightarrow w \) for suitable objects \( x, y, z \) and \( w \).

The two categories \( H \) and \( V \) are built upon a common set of objects, called interfaces. Arrows of \( H \) and \( V \) are called configurations and observations respectively and are algebraic structures equipped with parallel and sequential operators (\( \otimes \) and ; operators respectively) to allow building larger components. \( R \) is a set of tiles expressing all basic local changes on configurations according to some observations occurrences.

### 2.2 Operational semantics of tile systems

The set of tiles \( R \) defined in a tile system, describes which elementary local changes are possible in a distributed state. Generating larger steps of computation corresponds to deduction in tile logic. It is defined by a set of deduction rules (see Figure 1) showing how global concurrent evolutions are constructed.

#### Rules generating the basic tiles:

\[
R(\alpha) = \{s,a,b,t\} \]

\[
\alpha : s \xrightarrow{a} t
\]

#### Rules generating the horizontal and vertical identities:

\[
\alpha : x \rightarrow z \in V \quad t : x \rightarrow y \in H
\]

\[
\text{id}_H : \text{id}_V \xrightarrow{a} \text{id}_V \quad \text{id}_V : t \xrightarrow{H} \text{id}_H
\]

#### Horizontal, vertical and parallel compositions:

\[
\alpha : s \xrightarrow{a} t \quad \beta : h \xrightarrow{b} f \quad \lambda : \frac{s}{a} \xrightarrow{b} \mu : \frac{h}{b} \xrightarrow{a} k
\]

\[
\alpha \ast \beta : s \xrightarrow{a \beta} t \quad \lambda \ast \mu : \frac{s}{a} \xrightarrow{b \mu} h
\]

#### Figure 1: Deduction rules of tile logic

System global behaviour is constructed upon a three dimensional composition defined by the following set of tile logic deduction rules. Horizontal dimension models components coordination according to system structure, vertical dimension (time) models state evolutions according to computation flows, and parallel (space) composition models distribution of activities and resources. Horizontal and vertical identities reflect the fact that some local states may remain idle by evolving to themselves.

### 2.3 Tile logic as an architectural model

The architecture of a software system defines that system in terms of components and of interconnections among those components (Shaw et al., 1995). It is a form of contract showing the intended correspondence between system requirements and the corresponding components, being ensured all over the ongoing stages of system development. However, one major problem with software architectures is that system specification has been toppled from the more detailed vision of lines of code to an extreme level of abstraction where components are no more than black boxes.

Architecture Description Languages (ADLs for short) are adequate enough to both define software architecture of distributed systems, in terms of line and box view, and offer necessary tools to obtain transparent boxes where a well defined semantics of components interfaces is specified via their corresponding behaviours. Nevertheless, software architecture behaviour is not completely or badly defined (Mezgari, 2004) in most existing formal ADLs (such as Wright, Rapide, Darwin, etc.). One requires an enrichment of the underlying ADLs semantics with a more complete and abstract model to allow the specification of software architectures dynamic aspects. Such formal enrichment is not always evident and needs a correctness proof of the mapping between the underlying ADL concepts and the chosen model.

Rewriting logic has been already exploited as a common semantic framework to define both abstract software architectures and their behaviours. A rewriting logic based semantics was associated to CBabel (Rademaker et al., 2005) architecture description language. CBabel is based on an object oriented approach where components and connectors are declared as classes and interactions as message passing. Like all object oriented models, component required services are totally absent and state structure is flat in CBabel. To overcome the problem of state structure flattening, authors in (Jerad et al., 2007) have defined a model to hierarchically verify Lfp software architectures. In previous work (Bouanaka et al.,...
A Tile Logic Based Semantics for Mobile Software Architectures
C. Bouanaka et al.

2008), we have promoted software architectures inherent concepts such as interfaces, required services and components assembly using tile logic, an extension of rewriting logic which puts forward the concept of interfaces intrinsic to software architectures.

Main objective of the present work is to define a tile logic based model for the specification of mobile software architectures.

3 RECONFIGURABLE SOFTWARE ARCHITECTURES SPECIFICATION

Our model differs from other architectural models by a refined architectural style, in which connectors are no more first class citizens and do not need explicit specification. Interaction schemes are realized dynamically via tile logic horizontal composition operator of interfaces. This facilitates considerably software architectures reconfiguration since only components and their interfaces have to be managed.

Algebraic approach offered by tile logic to specify basic local changes on components interfaces is too complex to define software architectures and their dynamic changes. Thus, instead of manipulating algebraic terms on interfaces to specify topology, distribution and connectivity of software components rendering their readability too difficult, a graph based approach is coupled to the algebraic one.

Three levels of abstraction are offered for component specification within a unified semantic framework: Frame view, architectural view and behavioural view.

3.1 Frame view

At a software architecture level of abstraction, components are viewed as black boxes that are connected to their environment through a set of interfaces. Accordingly, the frame view is defined upon a finite set of interfaces. Topological structure of software architectures is defined by a hypergraph where nodes correspond to interfaces and hyperedges to components and their eventual connections. Connectivity links between interfaces, superposing an input interface of one component with an output interface of another, express data flows over components.

**Definition 2.** A Software architecture is defined by a hypergraph $SA = (I_{SA}, C_{SA}, L_{SA}, s_{SA}, d_{SA})$ where:

- $I_{SA}$ is a finite set of nodes corresponding to software architecture interfaces,
- $C_{SA}$ is a finite set of hyperedges corresponding to components of the underlying software architecture,
- $L_{SA}$ is a finite set of hyperedges corresponding to the initial connections between components,
- $s_{SA}, d_{SA} : C_{SA} \rightarrow P(I_{SA})$ are total functions returning source and target nodes of a given hyperedge respectively.

To avoid linkage of input/output interfaces belonging to the same component, $s_{SA}$ and $d_{SA}$ might verify the following condition:

$$\forall i \in L_{SA}, c \in C_{SA}, (s_{SA}(i) \cap s_{SA}(c) \neq \emptyset) \Rightarrow (s_{SA}(i) \cap s_{SA}(c) \cup d_{SA}(c) = \emptyset$$

Example 1. A Client/Server system example, inspired from (Barros, 2007), will be used to clarify the proposed concepts. The system is composed of a Primary Server, responsible of executing arriving clients, a Delay Server and a Generator that generates clients and sends them to the Primary Server. Clients waiting for a long time in the Primary Server queue, can quit it in order to look for the same service in the Delay Server, a competitor, that although has a higher service time and able to provide clients with the desired services immediately. Each client is responsible of deciding whether waiting time is excessive, hence quits the Primary Server without getting the service.

Software architecture of the client/server system $SA_{CS} = (I_{CS}, C_{CS}, L_{CS}, s_{CS}, d_{CS})$ is presented graphically in Figure 2 below where squares and solid circles represent components and interfaces respectively. Arcs are oriented indicating data flows direction regarding a component. For graph simplicity, we choose to represent $L_{CS}$ hyperedges (Connections) here by a simple point indicating that an output interface of one component is superposed on the corresponding input interface of another component.

To avoid linkage of input/output interfaces belonging to the same component, $s_{CS}$ and $d_{CS}$ might verify the following condition:

$$\forall i \in L_{CS}, c \in C_{CS}, (s_{CS}(i) \cap s_{CS}(c) \neq \emptyset) \Rightarrow (s_{CS}(i) \cap s_{CS}(c) \cup d_{CS}(c) = \emptyset$$

Figure 2: Client / Server System

Following Definition 2, $I_{CS} = \{\text{ClientIn}, \text{Tick}, \text{ClientOutPS}, \text{GiveUp}, \text{ClientOutDS}, \}$$,$ $C_{CS} = \{G, \text{PS}, \text{DS}, \text{Stats}\}$$,$ $s_{CS}(PS) = \{\text{ClientIn}, \text{Tick}\}$$,$ $d_{CS}(PS) = \{\text{ClientOutPS}, \text{GiveUp}\}.$ Source and target nodes of the remaining hyperedges are defined in a similar way.
For readability reasons, each component of $C_{CS}$ is represented by a simple component name which may stand mathematically for a function between component input interfaces and output ones, which is indicated by hyperedges directions. For instance $PS$ stands for $PS$: $\{\text{ClientIn}, \text{Tick}\} \rightarrow \{\text{ClientOutPS}, \text{GiveUp}\}$. In a similar way, each link belonging to $L_{CS}$ is represented by an identity function connecting one component output interface to the corresponding input interface of another component. $\{\text{GiveUp}\} \rightarrow \{\text{GiveUp}\}$ represents a connection between $PS$ and $DS$ components that share this interface name.

Considering this abstract Frame view, the Primary Server component of Figure 2 is simply defined by its visible interfaces: $\text{ClientIn}$, $\text{Tick}$, $\text{ClientOut}$ and $\text{GiveUp}$. If we are interested on the more detailed structure of the Primary server component, the model offers another abstract view to define this type of complex components.

### 3.2 Architectural view

Hierarchical structures are allowed by nesting graphs within components hyperedges and considering inner structure of hierarchical components. Components are defined recursively by associating a body graph, having the same structure as a software architecture one, to each hierarchical component. The body graph defines inner interfaces and their interconnections via subcomponents.

**Definition 3.** A hierarchical component $HC$ is a tuple $HC = (I_{HC}, B_{HC}, m_{HC})$ where:

- $I_{HC}$ is a set of interfaces having $HC$ as hyperedge,
- $B_{HC}$ is the body hypergraph of $HC$ having the same structure as a software architecture hypergraph defined previously, except that $L(B_{HC})$ represent initial connections that will evolve due to component reconfigurations.
- $m_{HC}: I_{HC} \rightarrow I(B_{HC})$ is a partial function associating inner nodes ($B_{HC}$ vertices) to outer interface nodes ($I_{HC}$) of the hierarchical component.

A primitive component $SC = (I_{SC}, \{SC\}, \varnothing)$ is a particular hierarchical with a body hypergraph composed of a unique hyperedge $SC$ connecting its input interfaces to output ones.

Given that hierarchical components hide their inner structure from the outside, subcomponents functionalities are accessible only through hierarchical component interfaces. Therefore, visible interfaces function is to receive data coming from either side, incoming or outgoing, and simply relay it to the other side. A correspondence function $m_{HC}$ is then defined between component visible interfaces $I_{HC}$ and inner ones $I(B_{HC})$.

#### Example 2.

At the architectural view, a zoom out of $PS$ hyperedge in Figure 2 shows a hierarchical component having the body graph of Figure 3. It is obtained by assembling together a FIFO queue and a set of clients $Ci$ actually waiting for an execution time.

Formally, the hierarchical component $PS$ is defined in this architectural view by $(I_{PS}, B_{PS}, m_{PS})$ where $I_{PS} = s_{CS}(PS) \cup d_{CS}(PS) = \{\text{ClientIn}, \text{Tick}, \text{ClientOutPS}, \text{GiveUp}\}$. $B_{PS}$ corresponds to the hypergraph presented in Figure 3. The correspondence relation $m_{PS}$ between visible interfaces $(I_{PS})$ of $PS$ and subcomponents interfaces $(I_{PS})$ establishes interaction channels with the external environment. It is represented here by the four following relations:

- $\{\text{ClientIn}\} \rightarrow \{\text{ClientInFIFO}\}$, $\{\text{Tick}\} \rightarrow \{\text{TickCI}\}$,
- $\{\text{ClientOutPS}\} \rightarrow \{\text{ClientOutPS}\}$,
- $\{\text{GiveUpFIFO}\} \rightarrow \{\text{GiveUp}\}$.

For readability reasons, it is represented by directly superposing some subcomponent interface on a visible one, avoiding interfaces duplication. $\text{ClientIn}$ interface is superposed to FIFO $\text{ClientIn}$ interface, $\text{Tick}$ interface to $\text{CI}$ Tick interface and so on. $\text{Start}$, $\text{End}$, $\text{Move}$ interfaces allow Primary Server intra-interactions.

\[
\text{Figure 3 : Primary Servers Hierarchical Component}
\]

Once the static structure of software architectures was defined via the frame and architectural views, a more detailed view is possible by offering necessary tools to define behavioural semantics associated to interfaces solicitations.

### 3.3 Behavioural view

Tile logic main contribution regarding other formalisms for the specification of concurrent systems, consists of the introduction of a new composition operator. The horizontal composition operator has a twofold role. At a structural
viewpoint, it can be exploited to construct larger configurations and states by gluing compatible interfaces of different components. At a behavioural viewpoint, it is used to synchronize components evolution to construct larger computations. Consequently, it suffices to carefully specify structural and behavioural basic elements and deduction in tile logic will be charged to generate larger states and computation.

3.3.1 Computation evolution
The frame and architectural views concern the description of software components. A more detailed specification, of both observable actions and local possible changes of software components (hierarchical or primitive) is also possible. Observable actions on software components (hierarchical or primitive) visible interfaces are defined by a set of rewrite rules $R_{SC}$, expressing their local possible evolutions. Applicability conditions and side effects of actions on component state are expressed by a set of tiles $T_{SC}$.

Definition 4. The behaviour of a software component SC is defined by a pair $\text{Behav}_{SC} = (R_{SC}, T_{SC})$, where:

- $R_{SC}$ is a set of rewrite rules defining possible changes on $I_{SC}$.
- $T_{SC}$ is a set of tiles defining local possible changes of the underlying software component.

$R_{SC}$ and $T_{SC}$ are defined on algebraic terms generated upon the given set of interfaces provided with some specific operators; whose semantics will be given in the next section. The global evolution of a software component is constructed by propagating tiles local effect on its global configuration thanks to deduction rules (see section 2 for more details).

Example 3. For the Primary Server example, one can identify some of $R_{PS}$ actions on its visible interfaces:

- Receive(ClientIn) = $(\text{ClientIn} = \text{null}) \rightarrow (\text{ClientIn} = \text{msg} (c))$ evolution of ClientIn interface value from null to msg(c) means that a new client $c$ has been sent by the generator to $PS$ which receives it on ClientIn interface,

- Empty(interface) = $(\text{interface} = \text{msg} (c)) \rightarrow (\text{interface} = \text{null})$ after a message receipt on some interface given as a parameter, $PS$ component may read it by emptying the corresponding interface,

- Fail(interface) = $(\text{interface} = \text{null}) \rightarrow (\text{interface} = \text{msg} (c))$ before sending any message on some output interface, $PS$ component begins by depositing it on the given interface,

- $\text{Signal} : (\text{Tick} = \text{null}) \rightarrow (\text{Tick} = 1)$ indicates a tick arrival from the generator component.

These actions form a subset of rewrite rules belonging to the Primary Server component behaviour, i.e. $\text{Behav}_{PS}$. Other actions may be defined in a similar way for sending some client either on $\text{ClientOut}_{PS}$ interface, if it has accomplished its task, or on $\text{GiveUp}$ interface, whenever it prefers looking for the service on the $\text{Delay}$ Server. Actions associated to $\text{Start}$, $\text{End}$ and move interfaces are local to the primary server component. $\text{msg}$ is a message containing either a value or an architectural element.

As we can see later in this section, primary server hierarchical component main feature is to reconfigure itself due to some client arrival or departure. Thus, it has no proper services that it provides to its environment. Clients only transit on it in order to obtain an execution time. Thus, its principal role is to receive and forward clients. As an example of tiles contained in $T_{PS}$, we can find a tile for client transfer from the Primary Server ClientIn interface to the corresponding one of the FIFO subcomponent. Forward tile is triggered by a message receipt on the ClientIn interface of $PS$. As a reaction, the message is withdrawn from ClientIn interface and deposited on the $\text{ClientInFIFO}$ one. Left hand and right hand sides of the tile specify necessary configurations to perform this tile. To allow message transfer, ClientInFIFO might be empty.

\[
\text{Forward}(\text{msg}(c)) : (\text{ClientIn} = \text{msg}(c) ) \rightarrow (\text{ClientInFIFO} = \text{null})
\]

The set of actions $R_{SC}$ and tiles $T_{SC}$ are application dependent.

3.3.2 Dynamic reconfiguration
Hierarchical components are subject to another type of evolution: dynamic reconfiguration that causes an adaptation of the hierarchical component structure, i.e., a transformation of the body graph by creating or destructuring some interfaces and their corresponding links. Component ongoing computations are also affected by such reorganization. Adding or removing some correspondences defines or inhibits some visible actions. Consequently, component behaviour is either upgraded or relegated with those of the arriving/leaving component. Nevertheless, it is not altered by local rearrangements of component structure caused by a local redistribution of its inner interfaces (links update) only. Such local redistribution is not perceptible from the outside since the set of inner interfaces remain unchanged.
In what follows we consider component mobility as a special kind of dynamic reconfiguration since it involves component deletion/addition and links creation/suppression. While computation evolution is application-dependent, component migration scenario, from one hierarchical component to another, is generic and do not depend on specific application behaviour. Therefore, a set of generic tiles is defined to show how a hierarchical component restructures its topology as a consequence of some mobile component departure or arrival.

**Definition 5.** A reconfiguration operation, caused by a component motion, on a hierarchical component $HC = (I_{HC}, \beta_{HC}, m_{HC})$ and having a behaviour $Behav_{HC} = (R_{HC}, T_{HC})$ is defined by a generic tile:

$$
\gamma:[(I_{HC}, \beta_{HC}, m_{HC}),(R_{HC}, T_{HC})] \rightarrow \gamma_{\delta}:[(I_{HC}, \beta_{HC}, m_{HC}),(R_{HC}, T_{HC})]
$$

A primitive component, defined by a structure $SC = (I_{SC}, \beta_{SC}, m_{SC})$ and a behaviour $(R_{SC}, T_{SC})$, arrival or departure makes a tumult on the hierarchical component architecture and behaviour.

At a hierarchical component level, only mobile component arrival or departure are perceptible. The whole scenario of component motion from one hierarchical component to another is managed at the upper level of hierarchy. For component receipt by a hierarchical component $HC$, the generic tile is instantiated with the Connect tile which is triggered now by such receipt. Merging $SC$ body graph with the hierarchical component one updates both inner interfaces $I(\beta_{HC})$ and links $L(\beta_{HC})$. As already mentioned, adding new inner interfaces has a direct consequence on the hierarchical component functionalities which are upgraded with those of the arriving component. Correspondences between $HC$ visible interfaces and inner ones are enriched with new connections linking some of the newly arriving component interfaces. Thereby, some undefined actions associated to these visible interfaces are thereafter available inducing $R_{HC}$ and $T_{HC}$ update with the corresponding actions and their implementations.

In order to reconfigure the structure of hierarchical components and consequently upgrade its functionalities, some actions, defined in the trigger and effect of the reconfiguration tiles, come to augment $Behav_{HC}$.

We can write $\gamma_{\delta}(HC, Behav_{HC}) = (I_{HC}, \beta_{HC}, m_{HC}, \gamma_{\delta}(R_{HC}), \gamma_{\delta}(T_{HC}))$ where the resulting hierarchical component is obtained as follows:

- $B'_{HC} = \gamma_{\delta}(B_{HC}) = (\{(B_{HC}) \cup SC\}, C(\beta_{HC}) \cup \{(SC)\}, \gamma_{\delta}(R_{HC}), \gamma_{\delta}(T_{HC}))$
- $m'_{HC} = \gamma_{\delta}(m_{HC}, m) = m_{HC} \cup m$
- $R'_{HC} = \gamma_{\delta}(R_{HC}, R_{SC}) = R_{HC} \cup R_{SC}$
- $T'_{HC} = \gamma_{\delta}(T_{HC}, T_{SC}) = T_{HC} \cup T_{SC}$

Some mobile component departure is a reconfiguration operation that is realized by instantiating the generic tile previously defined with a Disconnect tile. The trigger is now a move request formulated either by the mobile component itself or the hierarchical one. As an effect, the Disconnect tile removes $SC$ body graph from the hierarchical component one causing some inner interfaces, links and correspondences destruction. Since some correspondences are withdrawn, meaning that some actions are no more available, restrictions are to be made on the hierarchical component behaviour by inhibiting some actions and their corresponding tiles.

**Example 4.** A reconfiguration scenario of the Primary Server component may be caused by an expiration of the execution time of a given client which generates a move request $move(ci, PS, L, m)$ on its $End_{ci}$ interface. This triggers the Disconnect tile which extracts the client hyperedge from the $PS$ one, destroying the client visible interfaces $I_{SC} = \{Start_{ci}, End_{ci}, GiveUp_{ci}\}$ and their corresponding links.

$$
L = \{Start_{ci} \rightarrow Start_{FIFO}, End_{ci} \rightarrow End_{FIFO}, GiveUp_{ci} \rightarrow GiveUp_{FIFO}\}
$$

Since client interfaces are not connected to visible interfaces ($m(\beta_{SC}) = \emptyset$) of the primary server hierarchical component, client interfaces removal has no direct repercussion on the hierarchical component functionalities.

Various scenarios of reconfiguration are possible at different levels of abstraction in the Client/Server system like client motion from the generator to the primary server, or from the Primary Server to the Delay or Stats components. They are obtained by composing Connect/Disconnect generic tiles making evident the importance of hierarchical structures.

Contrary to most existing ADLs where different formalisms intervene to specify dynamic software architectures and their behaviours, a common semantic framework is used here. It is important to highlight that mutual side effects of the two dimensional evolution of software architectures are easily recovered through interfaces; that are the basic concept on which both architectural and behavioural views of software architectures are built.
4 SEMANTICS OF RECONFIGURABLE SOFTWARE ARCHITECTURES

The ability of tile logic to implement software architectures main feature, which consists of separating coordination from computation, promotes its usage as a semantic basis for the defined model. Two main concepts have contributed to the realization of such a separation of concerns: introducing interfaces in the rewriting process abstracts away unnecessary details of component implementation. Interfaces constitute a means for components awareness of contextual data conducting the rewriting process. Moreover, the definition of an additional composition operator (horizontal composition operator for instance) allows defining various schemes of components dynamic assembly. Consequently, double categories constructed upon tile systems are chosen as the semantic model of mobile software architectures.

Tile system interpreting syntactic concepts of mobile software architectures is construed as indicated in Definition 6 below.

**Definition 6.** Semantics of a hierarchical software component \((HC, Beh_{HC})\) = \([I_{HC}, R_{HC}, m_{HC}, (I_{HC}, T_{HC})]\) is given by a tile system \(TS_{HC} = (H_{HC}, V_{HC}, N_{HC}, T_{HC})\) and conceived as follows:

- Objects of the horizontal and vertical categories \(H_{HC}\) and \(V_{HC}\) correspond to the set \(I_{HC} \cup I(B_{HC})\).
- Basic morphisms of \(H_{HC}\) generated from \(C(B_{HC}) \cup L(B_{HC})\) plus those generated by \(m_{HC}\),
- Basic morphisms of \(V_{HC}\) represent possible observed actions \(R_{HC} \cup (\bigcup_{SC \in C(B_{HC})} R_{SC})\) on visible and inner interfaces,
- \(T_{HC} \cup (\bigcup_{SC \in C(B_{HC})} T_{SC})\) contains all tiles expressing authorized changes on inner configurations interfaces \(T_{SC}\) and those on visible interfaces of the component \(T_{HC}\).
- \(N_{HC}\) is a set of tiles names.

Figure 4 below resumes tile system construction. The common set of objects of the two categories defines semantic interpretation of all interfaces, including outer interfaces as well as inner ones that represent the body graph nodes, contained in \(HC\). The set of basic morphisms of the horizontal category formalizes body graph hyperedges (subcomponents and their actual links) plus those generated by the correspondence relation \(m_{HC}\) (links between visible and inner interfaces). What is original in the present work is the systematic construction of larger morphisms; representing hierarchical component global configurations, by composing subcomponents interfaces horizontally, for subcomponents interactions, and in parallel for global state spatial distribution. Therefore, all possible configurations upon the software component hypergraph are modelled by possible proofs in this horizontal category. Basic vertical morphisms define rewrite rules associated with hierarchical component interfaces (inner and outer ones). Larger morphisms are also constructed using parallel, horizontal and sequential operators as composition laws; representing parallel evolution, synchronization and sequential execution of subcomponents actions.

<table>
<thead>
<tr>
<th>Software Architecture</th>
<th>Tile System</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hierarchical Component</strong> (HC)</td>
<td><strong>Système de tuiles:</strong> (ST(HC) = (H_{HC}, V_{HC}, N_{HC}, R_{HC}))</td>
</tr>
<tr>
<td>Structural Configurations</td>
<td>Morphisms Composition in (H_{HC}) category</td>
</tr>
<tr>
<td>Behavioural Configurations</td>
<td>Morphisms Composition (V_{HC}) category</td>
</tr>
<tr>
<td>Dynamics(reconfiguration/computation)</td>
<td>Tiles (R_{ST-C}) / including <strong>Connect</strong> and <strong>Disconnect</strong></td>
</tr>
<tr>
<td>Operational semantics</td>
<td>Deduction in tile logic</td>
</tr>
<tr>
<td><strong>Primitive Component</strong> (SC)</td>
<td><strong>Objects of</strong> (H_{SC}) and (V_{SC}) ((I_{C} = 0(H_{SC} = 0(V_{SC}))))</td>
</tr>
<tr>
<td>Interface (I_{C})</td>
<td>Object Type (interfaces)</td>
</tr>
<tr>
<td>Interface Type</td>
<td>Morphisms of (H_{SC})</td>
</tr>
<tr>
<td>state</td>
<td>Basic morphisms of (V_{SC})</td>
</tr>
<tr>
<td>Actions</td>
<td>Tiles (R_{SC})</td>
</tr>
<tr>
<td>Dynamics</td>
<td>Deduction in tile logic</td>
</tr>
<tr>
<td>Operational semantics</td>
<td><strong>Equations in the signature</strong> ((\Sigma, E))</td>
</tr>
<tr>
<td><strong>Connexion (links)</strong> (L)</td>
<td></td>
</tr>
</tbody>
</table>

**Example 5.** The semantic model associated to the Primary Server hierarchical component is a double category generated from the following tile system \(TS_{PS} = (H_{PS}, V_{PS}, N_{PS}, T_{PS})\) :

---

Figure 4. Tile system construction.
The common set of objects is \( \text{Objects}_{PS} = \{ \text{ClientIn}, \text{ClientOutPS}, \text{Tick}, \text{GiveUp} \} \cup \{ \text{Start(Ci)}, \text{End(Ci)}, \text{Move(Ci)} \} \) for \( H_{PS} \) and \( V_{PS} \). For readability reasons, we have avoided duplicating \( \text{ClientInPS}, \text{ClientInFIFO}, \text{ClientOutPS} \).

Basic morphisms of \( H_{PS} \) are FIFO: \( \{ \text{ClientIn}, \text{End}, \text{GiveUp} \} \rightarrow \{ \text{Start}, \text{ClientOut}, \text{GiveUp} \} \) and \( \{ \text{Tick}, \text{Start} \} \rightarrow \{ \text{End}, \text{Move} \} \). \( m_{PS} \) and \( L_{PS} \) are identity morphisms. A category is then generated by tupling interfaces with different forms of interconnection: in parallel for spatial distribution, horizontally for data flows between interfaces.

Basic morphisms of \( V_{PS} \) are all actions defined on the set of objects \( \text{Objects}_{PS} \) (see example 3). So, each expected action on \( PS \) interfaces is considered here and its effect is naturally propagated through this model thanks to the defined set of tiles \( T_{PS} \) and deduction rules of tile logic.

Semantic interpretation of configuration evolution is a natural transformation that connects horizontal morphisms to vertical ones.

One can note that all concepts introduced in the two first views (Frame and Hierarchical) find their adequate semantic interpretation in the above tile system. The defined semantics also successfully formalizes the computation evolution of software architectures. Besides, reconfiguration operations; that vary from software architecture reconfigurations to simple rearrangements of primitive components connections depending on the level of abstraction chosen, are naturally formalized in this semantic model too. Hypergraph modelling of software architectures facilitates enormously local reconfigurations by simply redirecting certain links to other interfaces but still maintaining the same set of interfaces. Such local rearrangements of the hierarchical component interfaces and links affect only spatial distribution of the considered interfaces. The corresponding state space is updated without modifying its visible behaviour and thus its model remains unchanged. Hence, local reconfigurations of a hierarchical component topology by rearranging subcomponents links are not perceptible by its environment given that functionalities carried out through its interfaces or objects remain unchanged. A Primitive component motion is another type of reconfiguration having the same semantics as local rearrangements if perceived at a higher level of abstraction, i.e., at the software architecture level.

Indeed, it suffices to go up to the higher order hierarchical component to realize that mobile component interfaces \( I_{SC} \) and their corresponding links, being removed from one hierarchical component \( H1 \) and inserted into another one \( H2 \), belong to the upper level hierarchical component containing both \( H1 \) and \( H2 \). Nevertheless, component motion alters source and destination hierarchical components behaviours since their inner interfaces and links have been updated. Fortunately, the abstraction property ensures that it is possible to topple over diverse levels of details depending on what we need to see.

**Example 6.** Tile logic based semantics associated to the Primary Server hierarchical component presented in example 5 may be considered to model various reconfiguration operations depending on the level of abstraction chosen. One can imagine different scenarios of local reconfigurations of the Primary Server hierarchical component \( PS \) by redirecting some existing links without disturbing its visible behaviour. It is evident to note that such operation preserves the associated model since both the common set of objects \( \text{Objects}_{PS} \) and basic morphisms \( V_{PS} \) (the corresponding actions) remain unchanged. Only basic morphisms of \( H_{PS} \) are altered expressing the fact that state space has been changed and consequently some branches of execution may or may not occur in the actual situation. So, expected actions on interfaces are still unchanged here.

If we consider now semantic interpretation of client motion from the Primary Server component to the Delay or Stats ones considered in example 4, we note that it suffices to reason at the Client/Server system level in order to treat it as local reconfiguration of interfaces links.

Unlike several formalisms (Petri Nets, automata, \( \pi \)-calculus...) used as a semantic basis for various ADLs, tile logic offers necessary assets to formalize basic concepts of software architectures, especially interfaces and composition operators defined on them, allowing incremental specification. Horizontal composition operator deserves a particular attention since it creates dynamic assembly schemes over components interfaces for the propagation of local changes on a component to the whole software architecture.

5. **CONCLUSION**

We have presented in this work a unified formal semantic framework for the specification of reconfigurable software architectures. A hierarchical and compositional approach is adopted for the definition of both structural and behavioural views of software architectures. Different facets (topology, behaviour and reconfiguration) of software architectures are easily specified in a unified semantic framework. Interactions between the different views are perceptible through the common structure defined upon visible interfaces. Spatial distribution of interfaces describes software architecture topologies. Observable actions on interfaces are grafted via a set of rewrite rules.
expressing their possible evolutions. Local possible changes on components are defined by a set of tiles expressing actions effects on component local states and eventual side effects on its environment. Transitions between software architecture/hierarchical components configurations are controlled by a set of generic rules expressing conditions and side effects of such reconfiguration operations. Software architecture/hierarchical components global state and behaviour are constructed in compositional manner by deduction in tile logic.

Semantic model associated to reconfigurable software architectures is a double category generated on the basis of the tile system defining it. Horizontal category defines possible configurations of the software architecture and vertical category describes possible reconfiguration actions on it. Our ongoing work will focus on projecting syntactic and semantic concepts introduced here into rewriting logic taking advantage of the mapping defined in (Meseguer et al., 1997). Such projection allows us exploiting the Maude environment, one implementation language of rewriting logic, and its model checker to verify some mobile software architectures inherent properties.

6. REFERENCES


