Structure Analysis for Dynamic Software Architecture based on Spatial Logic*

Tingting Han Taolue Chen Jian Lu
State Key Laboratory of Novel Software Technology,
Nanjing University, Nanjing, Jiangsu, P.R.China, 210093
{hantt,ctl,lj}@ics.nju.edu.cn

Abstract

The requirement for modifying system structure during system execution is specified by dynamic software architectures. The system architecture style should remain one style or transform within a scope so that some constraints need to be imposed on during the system execution. Our work expands such an idea along two directions in the setting of formalism. The first direction is to model the system by a graph-based calculus stressing the structure. The other direction lies in that we tailor spatial logic to be a suitable logic as the system specification for structure. The model and specification are basis for the model checking algorithm that is to verify whether the system evolution satisfies some structure constraints. We invite a master-slave architecture style as a running example from the beginning and throughout the paper to demonstrate our approach. Such work can be seen as the basis of the structure analysis for architectures.

***Key words**: Dynamic Software Architecture, Spatial logic, Model Checking Algorithm*

1 Introduction

With the rapid development of Internet, the problem becomes more important of coordinating different software entities that are distributed on different locations to accomplish a computing task. To deal with this problem, *software architecture* as a blueprint of a software system at the highest level of abstraction [7] is applied to abstract the software entities to be components and the coordination between them to be connectors and then a model is extracted as the architecture on which the design, analysis and verification are based. *Dynamic software architectures* that can specify the modification of the systems during the system execution [1] is quite fit for those evolving systems that are running in the open and dynamic environment.

*Funded by NNSFC (60233010, 60273034, 60403014), 973 Program of China (2002CB312002).

From our point of view, two aspects on the dynamic software architectures are worth paying good attention to. One is the dynamic evolution of *system structures* and the other is the *coordination mechanisms* (e.g. communication protocols). The former gives a general view and evolving trace of the system while the latter cares more about the communication details among components. In this paper, we focus on the first aspect. It is widely recognized that some restrictions should be imposed on the system evolution to ensure that the system structure may remain one style or transform within a scope. These conditions, to a large extent, make the system execute under control as expected.

To further illustrate the problem of system structure, we provide a master-slave example to make clear the definition and usage of the model and specification. In the master-slave architecture (see Figure 1 and 2), a primary master (black, round-shaped) allots the computational tasks to one or more slaves (square-shaped) concerning performance or fault tolerance. The primary master can decide to add or remove a slave depending on the concurrent workloads, for example, in Figure 1, when the primary master is aware that the existent 5 slaves can no longer bear the increasing workload, it decides that the 6th slave will be added. Or, in Figure 2, the primary master can trigger the creation of secondary masters (white, round-shaped) out of consideration for load-balancing, to averagely share the workload over the region. The example is then abstracted to be two structures. The one master case (Figure 1) corresponds to the single-center-star style structure while the multiple masters case (Figure 2) corresponds to the multiple-center-star style structure. We restrict the system to evolve in the setting of these two structures, that is, any structures other than the two will be considered illegal and the evolution that may lead to illegal structures will be identified or forbidden. The example will be utilized throughout the paper.

In this paper, we deal with the problem at the level of formalism, which we believe is the basis of analysis and verification. As the main work, our formalism mainly covers three aspects:

- We model the system with a graph-based calculus in

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

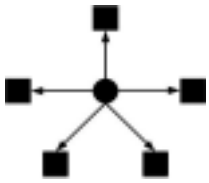

Figure 1. A single-center master-slave example

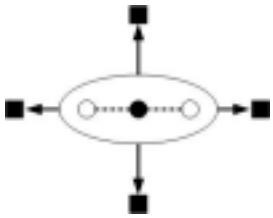

Figure 2. A multiple-center master-slave example

the flavor of graph calculus inspired by [4] for the sake of a more natural and intuitionistic way to only record the structural information but omits the behavioral information. Comparing to many of the existing works that devote more to the system evolution under behavioral changes, we explicitly separate the behavior and the topology, in which one of our novelties lies. We model private resources which is becoming aware in many of the applications by means of name hiding notions inspired by the π-calculus [9]. In Figure 2, two secondary masters are added to the primary master as private resources that are invisible to all the slaves, which illustrates the usage and importance of private resources to some extent. We refer the readers to Section 2 for more details.

- We tailor *spatial logic* [2, 3] to take on the role of specification, since it has strong expressing power for describing precisely certain properties, especially those hold at a certain location, at some location, or at every location [2]. As we will illustrate in this paper, it is quite fit for the software architecture specification. Concretely, we apply location formulas to perfectly cover the nested sub-systems; the private communication between certain components may be dealt with by the restriction formula with modal operators Ⓡ, ⊘, and quantifiers И, H and recursion has been introduced into spatial logic for clearness and conciseness.

- We also give a *model checking algorithm* for verification and automatic detection of errors in software architectures in order to locate errors and increase the reliability of these systems.

The rest of the paper is organized as follows: Section 2 and Section 3 introduce the graph model and spatial logic for software architecture respectively; Section 4 gives a model checking algorithm based on the model and logic; Section 5 offers some conclusions.

2 Models for Software Architecture

In this section, we briefly introduce the labelled graphs using a simple graph calculus [4] and a master-slave architecture is described with the graph model. We assume an infinite set $\mathcal{N}$ of nodes ranged over by $l, m, n, \cdots$, and an infinite set of edges $\mathcal{E}$ ranged over by $a, b, c, \cdots$. We use $x, y, \cdots$ to range over $\mathcal{N} \cup \mathcal{E}$. Let $\mathbb{N}$ denote natural number set, we use distinguishably z to ranged over $\mathcal{N} \cup \mathcal{E} \cup \mathbb{N}$, $\tilde{z}$ to denote a sequence of such names or natural numbers, and $|\tilde{z}|$ to denote the length of the sequence. The graph can be directed or undirected; the directed graphs are used to clarify the direction of data flow or invocation relations while the undirected graphs are used more generally when directions are not cared. In the undirected case, $a(m, n)$ denotes that node m and n are connected by an edge labelled a, while in the directed case, $a^+(m, n)$ denotes the edge from m to n and $a^-(m, n)$ denotes the edge from n to m. We use $\hat{a}(m, n)$ to represent $a(m, n)$, $a^+(m, n)$ or $a^-(m, n)$ covering either directed or undirected cases.

The set $\mathcal{G}(\mathcal{N}, \mathcal{E})$ of *graph terms* generated by $\mathcal{N}$ and $\mathcal{E}$ is given by the following BNF grammar:

G	::=	nil	empty
		$\hat{a}(m, n)$	edge
		$G \mid G$	composition
		$(\nu x)G$	hiding
		$A(\tilde{z})$	identifier

We often write $\mathcal{G}$ instead of $\mathcal{G}(\mathcal{N}, \mathcal{E})$. The *free* and *bound* names are standard: the hiding operator $(\nu x)G$ binds x in G while in $\hat{a}(m, n)$, a, m, n are free. $fn_n(G)$ is to denote the set of free names of nodes in G and $fn_e(G)$ for free names of edges and define $fn(G) = fn_n(G) \cup fn_e(G)$. The capture-avoiding substitution is $G\{y/x\}$.

A graph is *nil* when there are neither nodes nor edges in the graph, that is, it is an empty graph. $G_1 \mid G_2$ represents that graph G_1 composes with graph G_2 in some way. $(\nu x)G$ means x (node or edge) is private to G and is invisible out of G. $A(\tilde{z})$ is applied to define graph in the style of "procedure call", borrowing a term from the programming language.

Structural congruence denoted by $\equiv$, expresses basic identities on the structure of graphs, which is the least congruence satisfying the following rules:

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

node expressions	$\eta ::=$	x	node, $x \in \mathcal{N}$
		x	node variable, $\mathrm{x} \in \mathcal{V}_{\mathcal{N}}$
edge expressions	$\alpha ::=$	a	edge, $a \in \mathcal{E}$
		a	edge variable, $\mathrm{a} \in \mathcal{V}_{\mathcal{E}}$
name expressions	$\zeta ::=$	p	$p \in \mathcal{N} \cup \mathcal{E}$
		p	$\mathrm{p} \in \mathcal{V}_{\mathcal{N}} \cup \mathcal{V}_{\mathcal{E}}$

Figure 3. Node and Edge Expressions

(Zero Par)	$G \mid nil$	$\equiv$	G
(Par Assoc)	$(G_1 \mid G_2) \mid G_3$	$\equiv$	$G_1 \mid (G_2 \mid G_3)$
(Par Comm)	$G_1 \mid G_2$	$\equiv$	$G_2 \mid G_1$
(Res Res)	$(\nu x)(\nu y)G$	$\equiv$	$(\nu y)(\nu x)G$
(Zero Res)	$(\nu x)nil$	$\equiv$	nil
(Res Par)	$(\nu x)(G_1 \mid G_2)$	$\equiv$	$(\nu x)G_1 \mid G_2, x \notin fn(G_2)$

3 Spatial Logic

In this section, we present the syntax and semantics of the spatial logic, and also the master-slave architecture will be given to illustrate the expressing power of the logic. In this section, we utilize spatial logic as the specification method. More examples will also be given to illustrate the expressing power of the logic.

3.1 Syntax and Semantics

Syntax. Formulas are built from a node set $\mathcal{N}$ and edge set $\mathcal{E}$ as well as the sets of *node variables* $\mathcal{V}_{\mathcal{N}}$ and *edge variables* $\mathcal{V}_{\mathcal{E}}$, provided all the four sets are disjoint. And we assume a countable infinite set $\mathcal{X}$ of *predicate variables*, ranged over by $X, Y, Z, \cdots$. The syntax of the formula is defined by BNF in Figure 3 and Figure 4.

For the directed edges, we make it a convention that the first node is the source node and the second is the target node. In formulas of the form $\forall \mathrm{p}.A$, $\text{И}\mathrm{p}.A$, $\lambda\tilde{\mathrm{p}}.A$ and $\nu X.\Lambda$, the distinguished occurrences of p and X are binding, with scope the proposition A or predicate Λ.

We define on formulas the relation $\equiv_\alpha$ of α-congruence in the standard way. For any formula A, $fv_n(A)$ denotes the free node variables in A, $fv_e(A)$ denotes the free edge variables in A and and the free predicate variables in A are denoted $fpv(A)$. We define $fv(A) = fv_n(A) \cup fv_e(A)$ and $fn(A) = fn_n(A) \cup fn_e(A)$.

Semantics. We define the semantics of a formula A in a denotational way. The semantics of formula presented in Figure 5 is defined by assigning to each formula A a set of graphs $[\![A]\!]_{v;\xi}$, namely all the graphs that satisfy the property denoted by A. Since A may contain free node and edge variables and free occurrences of predicate variables, its denotation depends on the denotation of such variables, which is given by a valuation (node and edge name valuation and predicate valuation). A name valuation $v : \mathcal{V}_{\mathcal{N}} \cup \mathcal{N} \cup \mathcal{V}_{\mathcal{E}} \cup \mathcal{E} \to \mathcal{N} \cup \mathcal{E}$, which is identity on $\mathcal{N} \cup \mathcal{E}$, that is, $v[n/\mathrm{p}]$ is defined as $v[n/\mathrm{p}](q) \stackrel{def}{=}$ *if* $\mathrm{p} = q$ *then* n *else* $v(q)$. A predicate valuation ξ assigns to every predicate variable of arity k a function $(\mathcal{N} \cup \mathcal{E})^k \to \wp(\mathcal{G})$, that is $\xi : \mathcal{X} \to ((\mathcal{N} \cup \mathcal{E})^k \to \wp(\mathcal{G}))$.

$A, B ::=$		Propositions
	$\boldsymbol{0}$	empty
	$\alpha(\eta_1, \eta_2)$	directed edge
	$\alpha\langle\eta_1, \eta_2\rangle$	undirected edge
	$A \mid B$	composition
	$\boldsymbol{T}$	true
	$A \wedge B$	conjunction
	$\neg A$	classical negation
	$\zeta \text{®} A$	revelation
	$A \oslash \zeta$	hiding
	$\text{И}\mathrm{p}.A$	fresh quantifier
	$\forall \mathrm{p}.A$	universal quantifier
	$\zeta_1 = \zeta_2$	equalities
	$\Lambda(\tilde{\zeta})$	application
$\Lambda ::=$		Predicates
	X	recursive variable
	$\lambda\tilde{\mathrm{p}}.A$	abstraction
	$\nu X.\Lambda$	greatest fixpoint

Figure 4. Logical Formulas

The semantics is presented in the style of denotation, indeed, it can also be presented by satisfaction relation. We write $G \models_{v,\xi} A$ whenever $G \in [\![A]\!]_{v,\xi}$: this means that G satisfies formula A under name valuation v and predicate valuation ξ. Note that for a name-closed formula A, $[\![A]\!]_{v,\xi}$ does not depend on v and can be denoted by $[\![A]\!]_v$; and if A is closed, then $[\![A]\!]_{v,\xi}$ depends on neither v nor ξ thus can be denoted by $[\![A]\!]$.

We simply discuss the properties of the logic system. Due to the space restriction, most of the proofs are omitted. We refer the reader to [8].

3.2 Examples

In this part, we use the spatial logic to specify the structural properties of some common architecture styles, which usually remain unchanged or transform from one

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

$$
\begin{array}{rcl}
[\![\mathbf{0}]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv \mathbf{0}\} \\
[\![\alpha(\eta_1, \eta_2)]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv \alpha v(\eta_1 v, \eta_2 v)\} \\
[\![\alpha\langle\eta_1, \eta_2\rangle]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv \alpha v\langle\eta_1 v, \eta_2 v\rangle\} \\
[\![A \mid B]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv G_1 \mid G_2 \\
 & & \wedge G_1 \in [\![A]\!]_{v;\xi} \wedge G_2 \in [\![B]\!]_{v;\xi}\} \\
[\![\mathbf{T}]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv \mathcal{G}\} \\
[\![A \wedge B]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv [\![A]\!]_{v;\xi} \cap [\![B]\!]_{v;\xi}\} \\
[\![\neg A]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv \mathcal{G}/[\![A]\!]_{v;\xi}\} \\
[\![\zeta \circledR A]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv (\nu\zeta)G' \wedge G' \equiv [\![A]\!]_{v;\xi}\} \\
[\![A \oslash \zeta]\!]_{v;\xi} & \stackrel{def}{=} & \{G : (\nu\zeta)G \equiv [\![A]\!]_{v;\xi}\} \\
[\![\mathcal{H}p.A]\!]_{v;\xi} & \stackrel{def}{=} & \cup_{n \notin fn(A)}\{G \mid G \in [\![A]\!]_{v[n/p];\xi} \\
 & & \wedge n \notin fn(G)\} \\
[\![\forall \mathrm{p}.A]\!]_{v;\xi} & \stackrel{def}{=} & \{G : G \equiv \bigcap_{p \in \mathcal{N}} [\![A[p/\mathrm{p}]]\!]_{v;\xi}\} \\
[\![\zeta_1 = \zeta_2]\!]_{v;\xi} & \stackrel{def}{=} & \text{if } \zeta_1 v = \zeta_2 v, \text{return } \mathcal{G}; \text{ else } \emptyset \\
[\![\Lambda(\tilde{\zeta})]\!]_{v;\xi} & \stackrel{def}{=} & [\![\Lambda]\!]_{v;\xi}(\tilde{\zeta})v \\
[\![X]\!]_{v;\xi} & \stackrel{def}{=} & \xi(X) \\
[\![\lambda\tilde{p}.A]\!]_{v;\xi} & \stackrel{def}{=} & \lambda\tilde{z}.[\![A]\!]_{v[\tilde{z}/\tilde{p}];\xi} \\
[\![\nu X.\Lambda]\!]_{v;\xi} & \stackrel{def}{=} & \sqcup\{F : \mathcal{N}^k \to \wp(\mathcal{G}) | F \sqsubseteq [\![\Lambda]\!]_{v;\xi[F/X]}\}
\end{array}
$$

Figure 5. Interpretation of Formulas

style to another under some constraints as expected during evolution. We will show the examples both with and without recursion in the undirected graph style; it is easy to adapt to directed graphs. Before that, we introduce some basic properties specified in [4]. For recursively defined formulas (e.g."**in**(n, x)", "**exists_path**(x, y)"), we use the notation $\mathbf{R}(\tilde{x}) \stackrel{def}{=} \phi$, as an abbreviation for $\mathbf{R}(\tilde{\xi}) \stackrel{def}{=} (\mu\mathbf{R}(\tilde{x}).\phi)(\tilde{\xi})$:

no edge into x:	**in**$(0, x) \stackrel{def}{=} \neg\exists$y,a. a(y, x) | ***T***
n edges into x:	**in**$(n, x) \stackrel{def}{=} \exists$y,a. a(y, x) | **in**$(n-1, x)$
no edge out of x:	**out**$(0, x) \stackrel{def}{=} \neg\exists$y,a. a($x$, y) | ***T***
n edges out of x:	**out**$(n, x) \stackrel{def}{=} \exists$y,a.a($x$, y)|**out**$(n-1, x)$
unique node x:	**uni_node**$(x) \stackrel{def}{=} \forall$y. y $= x$
x in the graph:	**in_graph**$(x) \stackrel{def}{=}$ **uni_node**$(x) \vee$ ($\exists$ y,a. (a(x, y) $\vee$ a(y, x)) | ***T***
x, y has a path:	**exist_path**$(x, y) \stackrel{def}{=}$ $x = y$ $\vee(\exists$z,a. a(x, z) | **exist_path**(z, y))

We now do some remarks to interpret these examples in the setting of software architectures. x, y denote the components they stand for.

in$(0, x)$ means that x is the only component in the system or it is isolated with other components. Usually, it is one of the errors we are supposed to detect, e.g. when the component ought to be connected or integrated but failed, we should either try to reconnect it or remove it from the system.

in(n, x) means that x is connected with n other components, which is usually a quantificational constraint related with performance or other considerations. For example, a VOD (video on demand) service component can only support 100 client services simultaneously, any connection from the 101st service should be detected and blocked.

uni_node(x) is used for checking whether x is the only node in the graph. It is usually used as the initial condition for constructing a complete system.

in_graph(x) checks whether x is in the system or not. Sometimes different decisions will be made according to a certain component's existence.

exist_path(x, y) checks whether x and y are connected directly or indirectly so that there exists some invocation relations between them.

The underlying intuitions of **out**$(0, x)$ and **out**(n, x) are much the same as **in**$(0, x)$ and **in**(n, x). The main difference lies in that the initial direction of data flow may vary according to different applications.

We specify the architecture styles of ring and star as follows:

3.2.1 Ring

The ring style is used when all the components are of the same chance to take part in some activities. It is mostly used in the network system architecture, the token ring is a good case in point. **ring**$(a(x, y))$ denotes that there is a directed edge from node x to node y labelled a and **ring** denotes that it is a ring style.

- With Recursion:
  chain$(head, tail) \stackrel{def}{=}$ $\mathbf{0} \vee$ $head = tail \vee$ $\exists$a, x.($a(head,$ x) \| **chain**(x, $tail$))
  ring $\stackrel{def}{=}$ $\exists$head, tail, a. **chain**(head, tail) \| a(head,tail)

- Without Recursion:
  ring $\stackrel{def}{=}$ $\mathbf{0} \vee \exists$x.**uni_node**(x) $\vee$ (($\forall$x. **in_graph**(x) $\Rightarrow$ **in**(1, x) $\wedge$ **out**(1, x)) $\wedge$ ($\forall$y, z. **in_graph**(y)$\wedge$ **in_graph**(z) $\Rightarrow$ **exist_path**(y, z)))

In the recursive style, **ring** is denoted by means of **chain** which is recursively defined by inserting a new node in between the head node of the chain and the one directly following it.

And in the non-recursive style, **ring** denotes that either it is a void ring (denoted by $\mathbf{0}$) or there is only one node ($\exists$x.**uni_node**(x)) or for every node in the graph there is only one ingoing and one outgoing edge ($\forall$x. **in_graph**(x) $\Rightarrow$ **in**(1, x) $\wedge$ **out**(1, x)) and can reach every node in the ring ($\forall$y, z. **in_graph**(y) $\wedge$ **in_graph**(z) $\Rightarrow$ **exist_path**(y, z)).

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

3.2.2 Star

Star style is the cutting example of parallel composition. Master-slave style is a typical application of star structure, with either single center or multiple centers. The assumption is the same as in Section 1. We show both cases, among which **s_star**$(center)$ denotes the single-center-star and **m_star**$(center)$ the multiple-center-star.

- With Recursion:
  s_star$(center) \stackrel{def}{=}$ **0** $\vee$ **uni_node**$(center)\vee$
  $\exists$y,a. (**s_star**$(center)$ | a$(center, \mathrm{y})$)
  m_star$(center) \stackrel{def}{=}$ **0** $\vee$ **s_star**$(center)\vee$
  $\exists$e.Иy. y®(e$(center, \mathrm{y})$ | **m_star**$(center)$)

- Without Recursion:
  s_star$(center) \stackrel{def}{=}$ **0** $\vee$ **uni_node**$(center) \vee \forall \mathrm{y}.\ \mathrm{y} \neq$
  $center \Rightarrow$ (**in**$(0, center) \wedge$ **in**$(1, \mathrm{y}) \wedge$ **out**$(0, \mathrm{y}))$

In the single-center-star case, the recursive style is to add one edge to the graph and the non-recursive style is that the center only has outgoing edges and the peripheral nodes only has one ingoing edge and none outgoing ones.

In the multiple-center-star case, the recursive style is of great importance to demonstrate the usage of operator И and ®, which together derive a new quantifier Нx.A, i.e. the hidden-name quantifier Нx.$A \stackrel{def}{=}$ Иx.x®A. Нx.A indicates that in the graph there exists a restricted name which we shall call x and A is some property that may involve x where x is a variable that ranges over nodes or edges. This formula is meant to correspond somehow to a graph of the form $(\nu n)G$ where x denotes n. We simply give the definition of Нx.A, and we refer the reader to [5] for more details.

4 Model Checking Algorithm

In this section, we devote to provide a model checking algorithm for the graph data model and logic presented in this paper. We adapt the well-known Winskel's tag set method to our setting to deal with fixpoint operator. The correctness of the algorithm is shown in [8]. In our algorithm, the tag sets will contain pairs $\tilde{\eta}, G$ of name vector and the graph. Formally, let $T = \{(\tilde{\zeta_1}, G_1), ..., (\tilde{\zeta_l}, G_l)\}$, where, $\tilde{\zeta_i} (1 \leq i \leq l)$ are vectors of the same length, say k and for $\forall i, j.\ i \neq j$, we have $\tilde{\zeta_i} \neq \tilde{\zeta_j}$.

To deal with name restriction, as in [6], we fix the representation of the tree: using α-renaming of restricted names and the rules (ResRes) and (ResPar) of the congruence relation, we group together all name-restriction operators by transforming every graph to the form of $(\nu x_1) \ldots (\nu x_k)G$ and separate bound names by the following function *sep*. Note that all bound names are renamed apart so that they are different. For the sake of simplicity and clearness, in the following part of this section, we let $\mathcal{K} = \mathcal{N} \cup \mathcal{E}$.

Definition 1

$$\begin{cases} sep\,(nil) \stackrel{def}{=} \langle \emptyset, G \rangle, & \text{if } G \equiv nil \\ sep\,((\nu x)G) \stackrel{def}{=} \langle K \cup \{x\}, G' \rangle, & \text{if } sep\,(G) = \langle K, G' \rangle \\ sep\,(G_1|G_2) \stackrel{def}{=} \langle K \cup K', G_1'|G_2' \rangle, & \\ \quad \text{if } sep\,(G_1) = \langle K, G_1' \rangle \text{ and } sep\,(G_2) = \langle K', G_2' \rangle & \end{cases}$$

In order to decide whether a given edge belongs to a graph, which is reflected in the system to be whether a connector is in the architecture, we compute all the primitive connectors in a given architecture to be a set. Since the graph or the architecture is finite, the derived edge set is also finite, which guarantees that model checking the primitive edges is decidable. We use $\mathcal{L}$ to denote the set of edges with two connecting nodes of G.

Definition 2 *(Edge Set)*

$$\mathcal{L}(G) \stackrel{def}{=} \{a(x, y) \mid \exists G'.G_1 = (a(x, y) \mid G')\},$$
$$\text{where } (K, G_1) = sep\,(G)$$

Now, we are ready to present our model-checking algorithm. It is an extension of the algorithms from [6]. It is well known from the result of [6], for any graph G, the sets $\{G \mid G \equiv 0\}$ and $\{(G_1, G_2) \mid G \equiv G_1|G_2\}$ are decidable. For notation, we use $\dot{\cup}$ for disjoint union, that is, $A = B \dot{\cup} C$ if $A = B \cup C \wedge B \cap C = \emptyset$. Recall that all bound names in the graphs are renamed apart so that they are all different from each other and different from all free names occurring in the graphs and the formulas. Since $\mathcal{K}$ is countable, we can assume it is ordered. For a set of names V, function $new(V)$ returns the least name in $\mathcal{K} \backslash V$. The algorithm is presented in Figure 6, which gives the reader a taste of our algorithm. We leave the detailed explanation to the full paper of this abstract.

The correctness of the algorithm can be stated by the following theorem. For its detailed proof, please refer to [8].

Theorem 1 *For any graph G and closed formula A, the following properties hold:*
(i) check $(sep(G), A)$ terminates;
(ii) check $(sep(G), A) =$ true iff $G \in [\![A]\!]$.

5 Conclusion

Dynamic software architecture describes the system structure modifications at runtime. We need a mechanism to forbid the unbending evolution and thus guarantee that the system is running under control.

Our work expands along two directions which later converges at the verification problem. A graph-based calculus which can model the private resource is utilized as a simple and direct way stressing the structure. Spatial logic is

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

$$
\begin{array}{l}
\text{check}(K,G,\mathbf{0}) \stackrel{def}{=} \left\{\begin{array}{ll} \mathit{true} & \text{if } G \equiv \mathbf{0} \\ \mathit{false} & \text{o.w.} \end{array}\right. ; \\
\text{check}(K,G,\alpha(x_1,x_2)) \stackrel{def}{=} \left\{\begin{array}{ll} \mathit{true} & \text{if } \alpha(x_1,x_2) \in \mathcal{L} \\ \mathit{false} & \text{o.w.} \end{array}\right. ; \\
\text{check}(K,G,\alpha\langle x_1,x_2\rangle) \stackrel{def}{=} \left\{\begin{array}{ll} \mathit{true} & \text{if } \alpha\langle x_1,x_2\rangle \in \mathcal{L} \\ \mathit{false} & \text{o.w.} \end{array}\right. ; \\
\text{check}(K,G,A \mid B) \stackrel{def}{=} \bigvee_{K=K_1 \dot\cup K_2} \bigvee_{G \equiv G_1 \mid G_2} \text{check}(K_1,G_1,A) \wedge \text{check}(K_2,G_2,A) \\
\qquad\qquad \wedge \mathit{fn}(G_1) \cap K_2 = \emptyset \wedge \mathit{fn}(G_2) \cap K_1 = \emptyset; \\
\text{check}(K,G,\mathbf{T}) \stackrel{def}{=} \mathit{true}; \\
\text{check}(K,G,A \wedge B) \stackrel{def}{=} \text{check}(K,G,A) \wedge \text{check}(K,G,B); \\
\text{check}(K,G,\neg A) \stackrel{def}{=} \neg\text{check}(K,G,A); \\
\text{check}(K,G,p\text{®}A) \stackrel{def}{=} \bigvee_{m \in K} \text{check}(K\backslash\{m\}, G[p/m], A) \vee (p \notin \mathit{fn}(G) \wedge \text{check}(K,G,A)); \\
\text{check}(K,G,A \oslash p) \stackrel{def}{=} \text{check}(K \cup \{p\}, G, A); \\
\text{check}(K,G,\text{И}\mathrm{p}.A) \stackrel{def}{=} \text{check}(K,G,A[\text{new}(\mathit{fn}(K,G) \cup \mathit{fn}(A))/\mathrm{p}]); \\
\text{check}(K,G,\forall \mathrm{p}.A) \stackrel{def}{=} \bigwedge_{n \in \mathit{fn}(K,G) \cup \mathit{fn}(A)} \text{check}(K,G,A[p/\mathrm{p}]) \\
\qquad\qquad \wedge \text{check}(K,G,A[\text{new}(\mathit{fn}(K,G) \cup \mathit{fn}(A))/\mathrm{p}]); \\
\text{check}(K,G,(\nu X.[T]\Lambda))(\tilde{p}) \stackrel{def}{=} \left\{\begin{array}{ll} \mathit{true} & \text{if}(\tilde{p},G) \in T \\ \text{check}(K,G,\Lambda[\nu X.[T \cup \{(\tilde{p},G)\}]\Lambda/X](\tilde{p})) & \text{o.w.} \end{array}\right. ; \\
\text{check}(K,G,p_1 = p_2) \stackrel{def}{=} \left\{\begin{array}{ll} \mathit{true} & \text{if } p_1 = p_2 \\ \mathit{false} & \text{o.w.} \end{array}\right.
\end{array}
$$

Figure 6. The Model Checking Algorithm

tailored to be a fitting logic that can deal with the location relation, nested sub-system relation and private resources. The model and specification is basis for the model checking algorithm that is to verify whether the model has the specified property.

Our work is inspired by [4] which uses spatial logic to query graphs. However, it neglects the spatial operators and quantifiers that deal with the private resources. We extend their logic in this paper. And our spatial logic is designed specially to describe dynamic software architecture.

As future works, as mentioned in the paper, the specification of the non-recursive multiple-center-star architecture style is to be studied. Moreover, we intend to combine the behavior and structure analysis for dynamic software architecture in the same framework which provides a unified specification and verification approach.

References

[1] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report, Queen's University, 2004.

[2] L. Caires, A. D. Gordon. Anytime, Anywhere. Modal Logics for Mobile Ambients. In 27th ACM Symp.on Principles of Programming Languages, pages 365-377. ACM, 2000.

[3] L. Caires, L. Cardelli. A spatial logic for concurrency(Part I). In N.Kobayashi and B.C.Pierce, editors, 10th Symposium on Theoretical Aspects of Computer Science, Volume 2215 of Lecture Notes in Computer Science, pages 1-30. Springer-Verlag, 2001.

[4] L. Cardelli, P.Gardner, G.Ghelli. A spatial logic for querying graphs. In 29th Colloquium on Automata, Languages and Programming (ICALP 2002), Lecture Notes in Computer Science, pages 597-610. Springer-Verlag, 2002.

[5] L. Cardelli, A. D. Gordon. Ambient logic. Submitted to MSCS, available from the authors, 2003.

[6] W. Charatonik, J. -M. Talbot. The Decidability of Model Checking Mobile Ambient. Proc. CSL'01. LNCS 2142, pp.339-354, Springer, 2001.

[7] D. Garlan. Software Archtecture: A Roadmap. ICSE-Future of SE Track 2000:91-101.

[8] T. Han, T. Chen, J. Lu. Structure analysis for dynamic software architecture based on spatial logic. Technical report, Nanjing University, 2004.

[9] R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Process, part I/II. Journal of Information and Computation, 100:1-77, Sept.1992.

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**