Dimensions of Composition Models for Supporting Software Evolution*

In-Gyu Kim[1], Tegegne Marew[2], Doo-Hwan Bae[2], Jang-Eui Hong[3], and Sang-Yoon Min[4]

[1] Telecommunication R & D Center, Telecommunication Network Business, Samsung Electronics, Co. Ltd., Suwon, Korea
igkim.kim@samsung.com
[2] Dept. of Electrical Engineering & Computer Science, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejon 305-701, Korea
{tegegnem, bae}@se.kaist.ac.kr
[3] School of Electrical & Computer Engineering, CBNU, 12, Gaeshin-dong, Heungduk-gu, Cheongju 361-763, Korea
jehong@chungbuk.ac.kr
[4] SOLUTIONLINK, KAIST Venture Incubator 373-1, Guseong-dong, Yuseong-gu, Daejon 305-701, Korea
sang@sol-link.com

Abstract. Software systems with constrained and dynamic environments need to adapt to local and diverse computing environments by providing highly customized services at run-time. In order to address such dynamic changes effectively, composition models addressing complicated composition issues and supporting advanced composition features are required. In order to analyze and identify the required features of composition models supporting dynamic changes, we propose the dimensions of composition models by survey and analysis of existing work. Based on the dimensions, it is possible to provide a road map to improve capability of a composition model for a specific domain such as a dynamic mobile agent domain.

1 Introduction

An important emerging requirement for software systems is the ability to address dynamic requirements changes. As the competitions among enterprises become fiercer, there is a need for each enterprise to satisfy the time-to-market requirement faster. In addition, the spread of the Internet and mobile communications with constrained devices requires software systems (e.g. mobile agent systems) to adapt to local and diverse computing environments by providing highly customized services at run-time.

Composition based techniques are practical and effective approaches for supporting software evolution because of high flexibility and increased productivity.

* This work was supported by the Ministry of Information & Communication, Korea, under the Information Technology Research Center (ITRC) Support Program.

In this paper, we define compositional aspects of the techniques as composition models. Composition models enable decisions to be made on how composition units are composed and which functionality the composed one provides. According to the capabilities of composition models that applications are based on, the ability of the applications to accommodate changes is decided. Thus, in order to support software evolution more effectively, composition models addressing complicated composition issues and supporting advanced composition features are required. In this paper we propose dimensions of composition models in order to analyze and identify the required features of composition models for evolvable software systems especially with dynamic requirements changes. Through surveying existing work supporting software evolution and existing criteria for comparing composition models, the dimensions are identified, collected, classified, and refined. Based on the dimensions, it is possible to find which areas are not supported or need to be more supported in existing composition models and to provide a road map to improve composition model capability. The dimensions also enable software developers to find out required features of a composition model for a new application domain where computing conditions or environments are different from other domains.

The remainder of this paper is organized as follows. Section 2 shows and analyzes briefly some existing efforts and research projects supporting software evolution. Section 3 collects and analyzes existing criteria related to comparing composition models. Based on the analysis of Section 2 and 3, Section 4 proposes dimensions of composition models and describes each dimension in detail. Section 5 compares some existing research projects by the proposed dimensions. As a case study applying the proposed dimensions to a new domain to find out the required features for effectively supporting software evolution in the domain, Section 6 shows how the dimensions can be used for choosing the required features for a composition model supporting dynamic mobile agent applications. In Section 7 we conclude our research with further work.

2 Work Supporting Software Evolution

There exist techniques or research projects for supporting software evolution. We have classified these efforts largely into 9 categories.[1] The pros and cons of each category are explained briefly in Table 1.[2] Since our paper focuses on dimensions of composition models, we are more interested in composition based techniques (row 3 in Table 1). Some of efforts in the category (i.e. composition based techniques) are explained in detail in the following.

As basic OO composition techniques, there are association, inheritance, and delegation. Association is one of the simple composition techniques which has been widely used in OO systems. It enables an object to refer to other objects, for instance, by using object variables. In association, functionality can be changed

[1] The classification is not mutually exclusive. Some efforts belong to more than one category.

[2] For more detailed information, [18] can be referred to.

Table 1. Comparison of Efforts Supporting Software Evolution In the Large

Efforts supporting software evolution	Pros	Cons
Code Modification	any kind of adaptation, any part can can be modified, direct modification, efficient for experienced programmers.	requiring source code, error-prone, not suitable for large and complex systems.
Parameter Modification	controlled modification by parameters.	limited modification within parameter scopes.
Composition Based Techniques [1,2,5,22,23]	producing and adapting software systems fast and cost-effectively by (run-time) composition, high reuse of components.	requiring various mechanisms to support composition.
Design Pattern Based Techniques [12]	providing general solutions for addressing software evolution problems.	hard to find exact patterns for addressing given problems.
Software Architecture Based Techniques [8]	supporting high level modification by changing components, connectors, or configuration.	subjective to architectural styles, requiring further research on dynamic architecture supporting software evolution.
Transformation Based Techniques [3,4]	as powerful as code modification, supporting controlled modification by transformation templates.	requiring source code, hard to modify at run-time.
Reflection Based Techniques [11,21]	supporting run-time change by modifying meta data, used as complementing or supporting techniques for many other adaptation efforts.	requiring mechanisms to support meta-level architecture, complex to use.
Collaboration Based Techniques [16,27]	large granularity of reuse (collaboration-level), supporting separation of concerns.	requiring further research on efficient realization of the concepts and supporting mechanisms.
Industry Component Models [9,13]	providing various practical supporting tools for developing systems based on their own component models, easy to use.	not sufficiently providing component or composition models for adapting components or systems.

at run-time by changing the references. Class inheritance allows a subclass's implementation to be defined in terms of the parent class's implementation [2,26]. The advantage of class inheritance is that it is done statically at compile-time and is easy to use. The disadvantage of class inheritance is that the subclass becomes dependent on the parent class's implementation and the implementation inherited from a parent class cannot be changed at run-time. Delegation is similar to association except message handling mechanisms [2,20]. Using delegation, a method can always refer to the original recipient of the message, regardless of the number of indirections. Like association, delegation also supports dynamic composition by changing parents at run-time.

Ostermann et al. propose compound references, a new abstraction for object references, that allows to provide explicit linguistic means for expressing and combining individual composition properties on-demand [22]. They provide five composition properties to express a seamless spectrum of composition semantics in the interval between object composition and inheritance: overriding, transparent redirection, acquisition, subtyping, and polymorphism. A variety of composition mechanisms can be used by simply decorating object references with the above composition properties. A seamless transition from one composition mechanism to the other is also possible by changing composition properties, which enables applications to be adapted to have the changed functionality.

Context relation is a relation between classes which directly models dynamic evolution [23]. In Context relation, a context class defines a dynamic update for a base class. Attaching a context object to a base object alters the base object's method table based on the class updates defined by the context class. Context relation supports method-level updating.

HADAS is a decentralized framework for composition of software systems by connecting components [5]. HADAS supports dynamic adaptation, which allows for the adjustment of structure and behavior of autonomous components. Each component is split into two sections; Fixed and Extensible. Data items and methods defined in the Fixed section are not changed during the component's lifetime. In contrast, the Extensible section comprises the mutable portion of the component through which component's structure and behavior can be changed, and in which new methods can be added or removed on-the-fly. HADAS is based on 2-level method invocation mechanism which supports extensibility of the invocation mechanism itself. The mechanism partially enables "supporting multi-services" by metainvocation. Added components can access original components through "*selfObject*" construct. HADAS supports dynamic adaptation and a hybrid approach to get benefits both from class-based and instance-based changes.

DC-AOP is a platform for scalable mobile agents, which supports dynamic composition of functionality using code mobility [19]. Kim et al. categorize functionalities that mobile agents can use as follows: built-in functionality, resident functionality, carried functionality. Carried functionality enables mobile agents to add functionalities in remote nodes into their behaviors by code mobility and use the functionalities at run-time. DC-AOP supports such dynamic composition of functionality by providing four language constructs for carried functionality.

Lasagne defines a platform-independent architecture for dynamic customization of component-based systems using wrappers [28]. Lasagne introduces the concept of "Composition Policy". In Lasagne, composition logic is externalized from the code of clients, core system, and extensions by encapsulating it in a composition policy. In Lasagne, an application consists of a minimal functional core (implemented as a component-based system), and a set of potential extensions that can be selectively integrated within this core functionality. Each extension (i.e. collaboration) is implemented as a layer of mixin-like wrappers, simultaneously tailoring multiple components of the application and their interactions between each other.

GenVoca is one of program transformation approaches. GenVoca generators synthesize software systems by composing plug-compatible and interchangeable components [4]. GenVoca components are parameterized program transformations that are capable of operation refinements. The interfaces and bodies of GenVoca components are subjective (i.e. changeable). When components are composed, GenVoca checks additional constraints (e.g. precondition and postcondition) called design rules as well as type.

The CORBA Component Model (CCM) is a specification for creating server-side scalable, language-neutral, transactional, multi-user, and secure enterprise-level applications [13]. In CCM, components support a variety of interaction features, called ports. The ports includes facets, receptacles, and event sources/sinks. A component can provide multiple object references, called facets, which are capable of supporting distinct IDL interfaces. Using facets, operations can be grouped. In addition, introspection facilities associated with facets permit one to discover the set of roles provided by a component type at run-time. Other component models such as EJB ([9]) provide similar functionalities for component customization, composition, evolution, and deployment.

3 Existing Dimensions for Comparing Composition Models

This section presents and analyzes existing criteria used to compare composition models. Bosch proposes superimposition, a novel black-box adaptation technique that allows one to impose predefined, but configurable types of functionality on a reusable component [6]. He identifies the requirements that component adaptation techniques should fulfill; "transparent", "black-box", "composable", "configurable", and "reusable" requirements. Some of these requirements are useful to identify dimensions of composition models. For example, "composable" requirement implies that the adapted component should be as composable as it was without the adaptation and the adaptation should be composable with other adaptations. In addition, Bosch focuses on configurable adaptation, which is realized by a number of component adaptation types that can be configured for the specific component.

Heineman et al. present a list of requirements necessary for component adaptation techniques from surveying and analyzing some existing work and considering three additional requirements [14]. Although some requirements such as "identity" and "architectural focus" are useful as dimensions of composition models, other requirements are not suitable directly for composition models. For example, "conservative" requirement is based on the assumption that existing functionalities of components are not cancelled. However, we think that composition of two components can make a combined component with less functionalities than the sum of functionalities of two components.

Kniesel classifies component adaptation approaches according to four criteria [20]. "anticipated or unanticipated changes" and "time" are important aspects of composition models.

Buchi et al. provide requirements for a wrapping mechanism [7]. "shielding", one of the requirements, indicates that a wrapper should be able to control whether clients can directly access the wrappee or not.

Dominick et al. provide concerns which are important for extensible and configurable components [10]. "extensible and reusable extensions" concern means that components can be plugged into components recursively. They think that extensions (to components) also should have component-like properties.

Svahnberg et al. provide selection criteria of variability realization techniques for selecting an appropriate technique for implementing variability [24]. They realize variability in product line software systems through steps of identifying variant features, introducing variation points for the features, populating the variant feature with its variants (software entities), and binding variation points with specific variants. They organize variability realization techniques into 13 types by using involved software entities and binding times. They compare the 13 types of variability realization techniques in detail by five criteria: introduction times of variation points, open times for adding variants, ways of populating collection of variants, binding times, ways of binding. The criteria are focused to classify variability realization techniques especially for product line software systems.

4 Dimensions of Composition Models

Based on the analysis of Section 2 and 3, we have identified, collected, classified, and refined dimensions of composition models. The dimensions and their elements (features) are explained as follows:

Granularity: Granularity classifies composition units into attribute, method, object, component, and collaboration. Collaboration is a set of objects, together which provides a particular functionality to the application.

Composition Time: This dimension addresses when composition is performed. This dimension has the following elements:

– compile-time: Composition is performed at compile-time or before (e.g. product architecture derivation time [24]).
– deploy-time: Composition is performed at deploy-time.
– load-time: Composition is performed at load-time.
– run-time: Composition is performed at run-time.

These elements are cumulative: a later time element implies the previous time elements. For example, run-time element implies load-time, deploy-time, and compile-time elements.

Location of Delta: Where can we get "added functionality" (called as delta) at run-time? This dimension is explained in more detail from [19].

– built-in: Delta is combined into original components at compile-time.
– local: Delta in the local node (computer) is used.
– remote: Delta can be loaded and combined from remote nodes.

Elements in this dimension are cumulative.

Required Composition Information: This dimension addresses what kinds of information is necessary for composition.

– interface: It requires signature information (e.g. return type, name, parameters).
– contract: Pre&post conditions and invariants are necessary [15].
– configuration: For advanced or flexible composition such as expressing various composition semantics, more configurable composition information should be provided explicitly and could be used and manipulated by component customers. Explicit configuration information enables developers or adapters to customize components to provide different behaviors by changing the information.

Contract and configuration elements imply interface element unless explicit notes are provided.

Consistency Checking when Composition: What kind of consistency checking is performed when composing?

– signature: Signature checking is performed.
– subtype: Subtype checking is performed.
– rule: Composition rules are used for consistency checking.

Subtype element implies signature element. Usually, signature element is a minimal element to check.

Composition Capability: This dimension shows which composition semantics can be provided after composition.

– adding new services (1)
– deleting existing services (2)
– changing services (3)
– supporting multi-services (4): When a message is received, multi-services can be provided.
– overriding (5)
– wrapping (6)
– combinations (7): Combinations of composition primitives are supported. In order to provide expressive and changeable composition semantics among components, it is necessary to combine various composition operators and provide various composition semantics through the combinations.

As a note, the numbers in parentheses for the above elements are for the reference in Table 2.

Reference Primitives: When a composition unit (let's say it as a component) is composed with other components, the reference scope of the other components which the component can access is decided by reference primitives. Let's assume that two components, original and delta, are composed.

– origin (O): The delta component can access the original component.
– delta (D): The original component can access the delta component.

- identity (ID): The original and the delta are aggregated into one identity.
- based on internal structure information (ISI): This category is different from the above three categories in that the reference scope can be decided by using internal structure information of a component such as information of fixed and extensible parts. For example, an internal component of a component can access all internal components of the fixed part of the component by using "fixed" reference primitive. Reference primitives belonging to this category can be used to express more specific and various reference scopes other than original and delta components.

Identity element (ID) implies both original (O) and delta (O).

Hierarchical Composition Support: This dimension decides whether composition can be applied hierarchically or not. It is important to raise the level of abstraction in such a way that the evolution is expressed, reasoned about, and implemented. One way of raising the level of abstraction is hierarchical composition. Hierarchical composition enables component composition to be applied uniformly to both component adaptation and application assembly. It also enables components to be adapted by other components and the adapted components to be used for adapting other components as well. It increases reusability by enabling components to play both the roles: original and delta.

Composable Parts: This dimension shows which parts of a system are allowed to be composed.

- Whatever: Any part can be changed or composed.
- Designated parts: Only particular parts are allowed to be changed or composed.

Anticipation: This dimension shows whether or not unexpected functionality which original developers do not consider at design-time, can be added into the software later by using the extension mechanism which is provided by the supporting composition model.

- Expected: By using the supporting composition model, only expected functionalities which are considered at design-time are allowed to be composable.
- Unexpected: By using the supporting composition model, unexpected functionalities which are not considered at design-time can also be composed.

Who Provides Composition Codes?

- Manual (Developer): Composition logic is programmed by developers. In this manual decision of composition, anticipated composition logic is coded at compile-time and according to the fixed logic, compositions in software systems are performed.
- Automatic (AI): Reasoning engine decides what to do at run-time (e.g. which parts have to be composed and in which ways) by using inference rules together with inputs from environments. Thus, composition logic can be decided automatically at run-time by the reasoning engine.

5 Comparison of Existing Work by the Proposed Dimensions

This section compares some of work presented in Section 2 by our proposed dimensions in Section 4. The comparison results are shown in Table 2 and the detailed comparison results with respect to each dimension are explained in the following:

Table 2. Comparison of Software Composition Efforts by the Proposed Dimensions

Dimensions	Software Composition Efforts				
	HADAS	DC-AOP	Context Relation	Lasagne	GenVoca
Granularity of Composition Units	object (mainly focused on methods)	object	object	collaboration (extension)	collaboration
Composition Time	run	run	run	run	compile
Location of Delta	local	remote	local	local	local
Required Composition Information	interface	interface	interface	configuration (composition policy)	contract
Consistency Checking	signature	signature	signature	subtype	rule (design rule checking)
Composition Capability	1,2,4	1,2	1,2,3	6,7	1,6
Reference Primitive	O,D	D	O	ID	O
Hierarchical Composition	No	Yes	No	No	Yes
Composable Part	Part (Extensible part)	Part (carried-functionality)	Part (Instance-based method)	All	All
Anticipation	Expected	Expected	Expected	Expected	Unexpected
Who	Manual	Manual	Manual	Manual	Manual

Granularity of Composition Units: HADAS can add and remove items (data, methods, or objects). It mainly deals with methods. DC-AOP and Context Relation are based on objects. Lasagne wraps a set of components with extensions. GenVoca can compose collaborations of "realm".

Composition Time: In HADAS, DC-AOP, Context Relation, and Lasagne, composition is performed at run-time. GenVoca performs software composition at compile-time. Generally, compile-time composition enables better performance because it does not require intermediate invocation layers. Run-time composition enables flexible functionality change because it can change functionality at run-time.

Location of Delta: Only DC-AOP enables remote functionalities to be loaded and composed with existing functionalities.

Required Composition Information: In HADAS, DC-AOP, and Context Relation, signature information is used for composition. Lasagne and GenVoca require more information than signature. Lasagne describes components with services (interfaces), dependencies, decorators (wrappers), and intercepters. At run-time, Lasagne uses component descriptions and composition policy to combine extensions into the core system selectively. GenVoca uses contract information such as pre and post conditions.

Consistency Checking when Composition: HADAS, DC-AOP, and Context Relation perform consistency checking at the level of signature; composition is not allowed if signatures are not matched. Lasagne wraps components or uses role object patterns to compose components and deltas. Thus, it requires subtype relation between components and deltas. GenVoca performs design rule checking to detect illegal combinations of components.

Composition Capability: HADAS, DC-AOP, and GenVoca enable components to provide new services. HADAS and DC-AOP can remove existing services. Context Relation changes existing services using context objects. It also adds new services which are only invoked by the attached context objects. Those new services can be disposed by changing context objects. GenVoca can add or wraps existing services. HADAS supports multi-services by metainvocation. Lasagne supports wrapping of services and selective combination of extensions by composition policy.

Reference Primitives: In HADAS, when two components are composed, one component can access the other component through "*selfObject*" construct. Context Relation has *this* primitive to access original components. It also has *context* primitive for delta to access itself. However, it does not provide primitives for delta to access original components. DC-AOP provides "cafInvoke()" method to access deltas. However, it does not provide facilities to invoke original objects which load the deltas. Lasagne provides the notion of component identity by *variation point*. It uses *inner* primitive to access the aggregate of a component instance and its decorating wrapper instances. Lasagne enables original components and deltas to be combined into one identity. Component identity (ID) implies that deltas can access the original component (O), the original component can access deltas (D), and delta can access other deltas. In GenVoca, deltas (upper layers) can access original components (lower layers).

Hierarchical Composition Support: In HADAS and Lasagne, deltas cannot be extended by other deltas recursively. For example, in Lasagne, it is very difficult to reuse deltas because they have subtype relation with original classes. Context Relation also does not support hierarchical composition because context classes are specified only for one base class. In DC-AOP and GenVoca, deltas can be extended by other deltas recursively.

Composable Parts: In HADAS, each component has two parts; Fixed and Extensible. Functionality in only "Extensible" part can be added or deleted.

In DC-AOP, only carried-functionalities of system can be changed through the proposed language constructs. In Context Relations, instance-based methods can be updated. In Lasagne, any services of components can be wrapped. In GenVoca, any components can be composed.
Anticipation: In HADAS, DC-AOP, Context Relation, and Lasagne, developers of components have to anticipate adaptations which will be performed in the future and also provide some ways (hooks) within the components to realize the adaptations. In GenVoca, original developers do not have to anticipate future adaptations. Adapters, instead of the original developers, perform necessary adaptations. However, although GenVoca composes unanticipated functionality, it is done before run-time. In order to satisfy fast-changing requirements more fully, unanticipated adaptations should be supported at run-time as well as at compile-time.
Who Provides Composition Codes?: In all work, the change is encoded by the developers or adapters at compile-time. Specifically, in HADAS, DC-AOP, Context Relations, and Lasagne, the change is encoded by developers. In GenVoca, the change is encoded by adapters as well as developers.

6 Applying the Dimensions for Dynamic Mobile Agent Applications

As a case study applying the proposed dimensions to a new domain, this section shows how our proposed dimensions can be used to select the required features of a composition model supporting dynamic mobile agent applications.

6.1 A Testing Mobile Agent with Dynamic Requirements Changes

As an application with dynamic requirements changes, a mobile agent application is described as follows. A mobile agent, called as DTMA, navigates to various nodes (e.g. insurance company web sites) where there are insurance components differently implemented by various companies with their own business rules. At each node, DTMA tests the insurance component provided at the node. The goal of DTMA is to find the most reliable insurance component among nodes. At various nodes, DTMA performs some testing activities. At a specific node, it happens to test the insurance component in the node in more detail because the insurance component has passed all testing activities of the current DTMA. DTMA changes the existing testing functionality with a new testing functionality which has more detailed test cases, and performs new testing activities. Similarly, at another nodes, DTMA adds a new display functionality which displays texts in well formatted forms, and adds a monitoring functionality which performs some backup activities. The above scenario is shown in Figure 1.

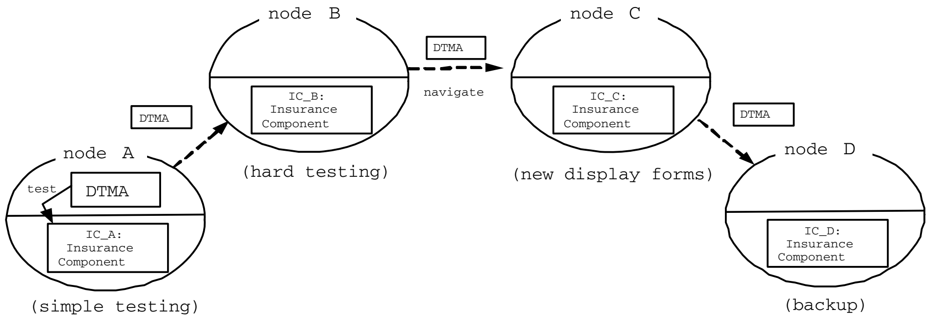
node A
node B
node C
node D
DTMA
test
IC_A: Insurance Component
IC_B: Insurance Component
IC_C: Insurance Component
IC_D: Insurance Component
navigate
(simple testing)
(hard testing)
(new display forms)
(backup)

Fig. 1. A Navigating Scenario of DTMA

6.2 Required Features of a Composition Model for Dynamic Mobile Agent Applications

For the above mobile agent application, DTMA can be programmed as having all functionalities including testing, display, and monitoring functionalities at compile-time. However, it requires more memory space and increased network bandwidth when moving to other nodes. In addition, it cannot accommodate unanticipated requirements such as a new secure communication functionality. In order to address dynamic requirements changes in the mobile agent application more sufficiently, a composition model suitable for the application should be selected and used. In order to find out the required features of the composition model, we used the proposed dimensions. As a result, we decided the following features are required for the composition model:

Granularity of Composition Units: Component Based Software Development (CBSD) enables applications to be developed fast and cost-effectively by composing existing or customized components [25]. Also, CBSD is being considered as a practical and effective approach for supporting software evolution because composing components provides high flexibility and productivity. Thus, applications with dynamic requirements changes could get benefits from CBSD. As a result, the composition unit for the composition model is decided as "component".

Composition Time: DTMA changes the existing testing functionality to a newly developed testing functionality at run-time. In addition, DTMA adds a newly developed other functionality (e.g. display) at run-time. Run-time composition is very useful for satisfying dynamic changes.

Location of Delta: It is possible for mobile agents to navigate in unexpected environments. If mobile agents can add functionalities in remote nodes into their behaviors by code mobility and use the functionalities at run-time, they can use various and timely functionalities in the Internet with high robustness [19].

Required Composition Information: Components provide services through interfaces. However, in order to address dynamic requirements changes in mobile

agents through component composition, (re)configurable composition information should be explicitly provided. Through changing the information, components can provide various behaviors. For example, let's assume that DTMA decides to move to an untrustworthy node. In order not to save travel information into the node, DTMA can change internal configuration to limit access to logging services.
Composition Capability: DTMA needs the following composition capabilities:

- add new services (1): DTMA adds display and monitoring functionalities.
- delete existing services (2): DTMA could delete the existing functionality.
- change services (3): DTMA changes the testing functionality.
- change configuration information (7): DTMA can change its architectural configuration information when it navigate to untrustworthy node. In addition, mobile agents move around nodes and perform some activities for each node. Each node may have different environments or requirements such as security levels and communication protocols. Thus, various composition semantics should be supported by combinations of composition primitives.

Reference Primitives: In DTMA, existing functionalities need to access new added functionalities (local or remote) and vice versa. In order to use different kinds of internal parts of DTMA effectively, DTMA needs to support "based on internal structural information" in this dimension.
Hierarchical Composition Support: If DTMA supports hierarchical composition, it will get the benefits of hierarchical composition such as increasing reusability of components and managing different composition levels uniformly.
Consistency Checking when Composition: In DTMA, signature checking is a minimum requirement. Subtyping checking is also necessary for hierarchical composition. Configuration checking is also required.
Composable Parts: Mobile agents need to manage their parts differently according to their goals. For example, one part has functionalities fundamental and very unlikely to change, and the other part has dynamically changeable functionalities. DTMA needs some basic functionalities such as a navigation functionality to be fixed for its proper operation. In the other hand, DTMA needs to use resources effectively because of limited memory space and network bandwidth. Thus, DTMA also needs composable or changeable part.
Anticipation: The ability to compose unexpected functionality is required to handle dynamic and diverse situations in mobile agent environments.
Who provides composition codes?: For the safe, reliable, and predictable operation of DTMA, the composition logic needs to be specified by developers explicitly.

The required features of a composition model for the dynamic mobile agent application are shown in Figure 2. The circles shows the chosen features for the composition model.

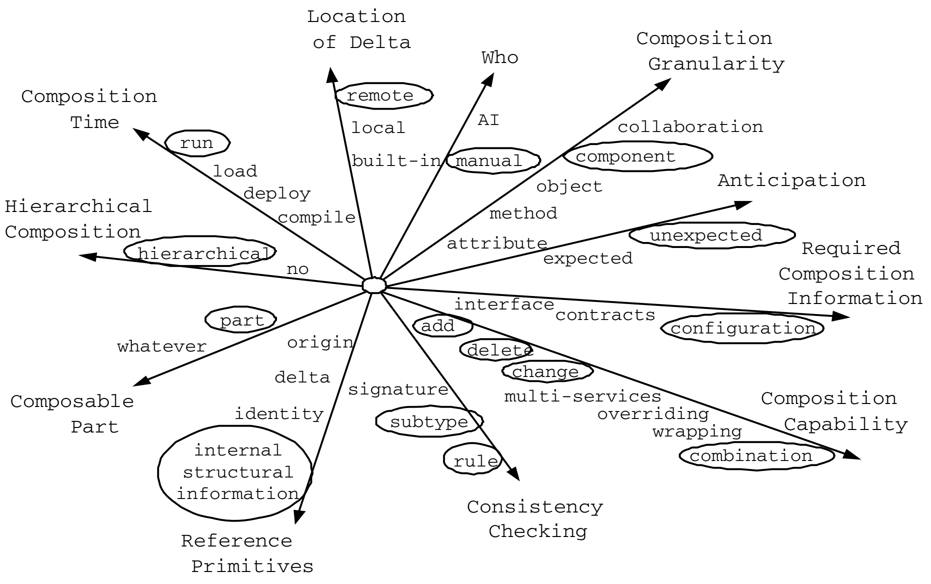

Fig. 2. Required features of a composition model for the dynamic mobile agent application

7 Conclusion and Further Work

In order to analyze and identify the required features of a composition model for software systems with dynamic requirements changes, we proposed the dimensions of composition models by survey and analysis of existing efforts supporting software evolution, especially composition based techniques, and the existing comparison criteria. The dimensions could be useful in the following areas:

– Existing work addressing dynamic requirements changes can be analyzed in various ways as shown in Section 5.
– The dimensions help to identify issues critical to improving composition capability of existing work.
– Future research directions of a specific dimension can be identified.
– When making a new composition model suitable for a specific domain such as [17], we can use, as a road map, the dimensions.

As experiments of our dimensions, first, we compared some existing software composition efforts by using the dimensions. Second, we identified the required features of a composition model supporting a dynamic mobile agent application by using our dimensions. Also, we have developed APIs for the composition model and implemented the application. For more information, please refer to "http://salmosa.kaist.ac.kr/~igkim/DCM".

While our research offers improvement in dimensions of composition models, there are some issues that are worth talking about in further research. First, it is useful to apply the proposed dimensions to various domains in order to extend the dimensions with additional features or to further refine the dimensions. We are now applying the dimensions to a hotel reservation system with continuous upgrades and changes of business requirements. Second, relations among the dimensions need to be analyzed and specified explicitly. Some of the dimensions could affect each other. They could be refined into more orthogonal dimensions, or the relations among them should be specified explicitly, for example, in documents. Finally, it is useful to identify relations between the dimensions and software quality attributes such as performance, reusability, and modifiability. For example, for modifiability, "Required Composition Information" has a higher priority than "Location of Delta". Such relations are useful to identity important dimensions that composition models should have in order to satisfy certain quality attributes or goals.

References

1. W. Aalst. "Don't go with the flow: web services composition standards exposed". *IEEE Intelligent Systems*, 18(1):72–76, 2003.
2. M. Abadi and L. Cardelli. *A Theory of Object.* Springer, 1996.
3. U. Abmann. *Invasive Software Composition.* Springer, 2003.
4. D. Batory and B. Geraci. "Composition Validation and Subjectivity in GenVoca Generators". *IEEE Transactions on Software Engineering*, 23(2):67–82, 1997.
5. I. Ben-Shaul, O. Holder, and B. Lavva. "Dynamic Adaptation and Deployment of Distributed Components in Hadas". *IEEE Transactions on Software Engineering*, 27(9):769–787, 2001.
6. J. Bosch. "Superimposition: A Component Adaptation Technique". *Information and Software Technology*, 41(5):257–273, 1999.
7. M. Buchi and W. Weck. "Generic Wrappers". In *Proceedings of ECOOP*, pages 201–225, June 2000.
8. S. Cheng, D. Garlan, B. Schmerl, J. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. "Software Architecture-base Adaptation for Pervasive Systems". In *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, pages 67–82.
9. L. DeMichiel, L. Yalcinalp, and S. Krishnan. Enterprise JavaBeans^{TM} Specification, Version 2.0. Technical report, Sun Microsystems, 2001.
10. L. Dominick and K. Ostermann. "Supporting Extension of Components with new Paradigms". In *Workshop on Advanced Separation of Concerns at OOPSLA*, 2000.
11. J. Dowling and V. Cahill. "The K-Component Architecture Meta-Model for Self-Adaptive Software". In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, LNCS 2129*, pages 81–88.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
13. O. M. Group. CORBA Components, v3.0 full specification. Technical report, OMG, 2002.
14. G. Heineman and H. Ohlenbusch. An Evaluation of Component Adaptation Techniques. Technical report, Computer Science Department, Worcester Polytechnic Institute, 1999.

15. R. Helm, I. Holland, and D. Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems". In *Proceedings of the OOPSLA/ECOOP Conference)*, pages 169–180, 1990.
16. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In *Proceedings of ECOOP*, 1997.
17. I. Kim and D. Bae. "A Dynamic Composition Model for Addressing Constrained Environments". In *OOPSLA Workshop on Reuse in Constrained Environments*, 2003.
18. I. Kim and D. Bae. Dimensions of Composition Model for Supporting Software System Evolution. Technical report, Department of Computer Science, KAIST, 2005.
19. I. Kim, J. Hong, D. Bae, I. Han, and C. Yoon. "Scalable Mobile Agents Supporting Dynamic Composition of Functionality". In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems, T. Wagner and O. Rana, eds., LNAI 1887*, pages 199–213, 2001.
20. G. Kniesel. "Type-Safe Delegation for Run-Time Component Adaptation". In *Proceedings of ECOOP*, pages 351–366, 1999.
21. P. Maes. "Concepts and Experiments in Computation Reflection". In *Proceedings of OOPSLA)*, pages 147–155, 1987.
22. K. Ostermann and M. Mezini. "Object-Oriented Composition Untangled". In *Proceedings of OOPSLA*, pages 283–299, 2001.
23. L. Seiter, J. Palsberg, and K. Lieberherr. "Evolution of Object Behavior Using Context Relations". *IEEE Transactions on Software Engineering*, 24(1):79–92, 1998.
24. M. Svahnberg, J. Gurp, and J. Bosch. "A taxonomy of variability realization techniques". *Software Practice and Experience*, 35(8):705–754, 2005.
25. C. Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
26. A. Taivalsaari. "On the Notion of Inheritance". *ACM Computing Surveys*, 28(3):438–479, 1996.
27. P. Tarr, H. Ossher, W. Harrison, and S. Jr. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In *Proceedings of ICSE*, pages 107–119, 1999.
28. E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Jorgensen. "Dynamic and Selective Combination of Extensions in Component-Based Applications". In *Proceedings of ICSE*, pages 233–242, 2001.