A Framework for Unanticipated Software Changes

Nguyen Truong Thang
School of Information Science
- Japan Advanced Institute of
Science and Technology
thang@jaist.ac.jp

Katayama Takuya
School of Information Science
- Japan Advanced Institute of
Science and Technology
katayama@jaist.ac.jp

February 16, 2003

Abstract

Software evolution effectiveness depends on two factors, namely how to determine the optimized path of changing software; and which software development paradigm guarantees low costs for evolving operations in such a path. A general software evolution framework has been proposed to deal with the first factor in the most abstract level of system specification. In particular, this framework can provide the best evolution path from a system to its evolved version. It is based on two concepts. They are *evolutionary domain* and *evolutionary development process* [2]. Because of its capability to handle changes at high level of abstraction, the proposed framework is considered very robust to many kinds of changes, possibly unanticipated.

Collaboration-based designs [5, 8, 9] are complementary to the framework due to its low cost in terms of software development and evolution. This design type is the answer for the second factor. The evolution of collaborative systems are examined and performed according to the path proposed by the framework. We believe that way - combination of the evolution framework and resilient designs - provides an effective framework to the evolution for many changes, possibly unanticipated. This paper proposes a solid theoretical background for the combination. It includes a formal specification for collaborative systems and formal definitions of evolution operators acting on that specification domain. Based on that background, various evolution activities are performed to illustrate the advantages of our approach. Observation in the illustrating examples also partially confirms our claim.

1 Introduction

Software evolution generally means that software can change its structure and functions to tolerate changes of its specification and operating environment in which it is used. A significant amount of work has been done so far, however, software evolution problem is still a challenge. One of the most difficult problems in evolving software is to deal with as many kinds of changes as possbile, especially unanticipated ones. In our opinion, there are two important factors affecting evolution cost. Firstly, actual evolution activities are not performed properly. Therefore, people tends to get a longer road from existing system to the target. Moreover, even with the same path, actual evolution cost are different because individual operations such as extraction, composition and transformation of software artifacts deliver different costs depending on software designs.

We believe that many unanticipated changes could be managed if software are built from resilient design methods. One of such methods is collaboration-based design [5, 8, 9]. This idea is a solution for the second evolution factor mentioned above. On the other hand, unanticipated changes should be handled at more abstract level than that of foreseeable changes. Therefore, the design method and evolution environment should involve an abstract model, which allows to capture changes at abstract levels matching the degree of unanticipatedness. The first cost factor can be partly solved by this idea. The combination of these two ideas, namely ability to capture changes at high level of abstraction; and low evolving cost for actual concrete changes at software by resilient designs, is the key to software evolution success, especially in unanticipated situations.

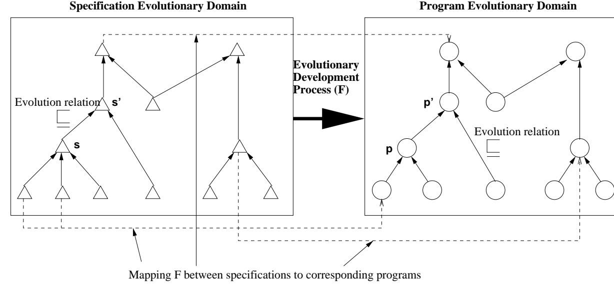

Figure 1: A simple visualization of evolutionary domain framework and the internal semi-lattice structure within each domain.

For the first factor, we propose in Section 2 a general framework which characterizes software artifact set under consideration according to a lattice structure. In the framework, changes are captured and formalized in a systematic analysis process. Certainly, by some formal analysis work with respect to simplified lattice structure, the best evolution path can be systematically determined. The general schemes to handle changes are discussed in Section 3. These schemes ensure minimum amount of work to be performed at high level of abstraction. On the other hand, each operation can be handled efficiently by well-designed software architecture. Experience has shown that an *orthogonal* design is good in terms of software maintenance and evolution [4]. Collaboration-based designs are orthogonal. The formalization and evolution of collaborative designs with respect to the proposed abstract framework are then discussed in Section 4. Its associated evolutionary development process is also formally introduced in Section 5. This development process is essentially *mixin-based.* A partial code generation to C++ is also presented in this section. Illustrating examples about handling variety of changes are mentioned in Section 6. Section 7 devotes for the limitation of our approach and sketches further work from current stage. We conclude the paper in Section 8. Hereafter, two terms role-based designs and collaboration-based designs are equivalent.

2 A Lattice-Theoretic Approach for Software Evolution

Let S and P be sets of all the specifications and programs respectively which will appear in the evolution problem under consideration. s and s', p and p' are specifications and derived programs respectively. When s changes to s', p has to transform to the corresponding p'. Of course, the evolved program p' has to be constructed reusing as much part of p as possible.

The proposed software evolution framework consists of three major parts: *evolutionary domains* on specification and program sides respectively; plus *evolutionary development process* between those domains as illustrated in Figure 1.

2.1 Evolutionary Domain

To make sound and effective discussions of the evolution problem, we need to restrict the way specification and programs may change. To this end, *evolution relations* $\sqsubseteq_s$ and $\sqsubseteq_p$ are introduced in the sets S and P to express the relationship between s and s', and, p and p' respectively. We assume that the changes $s \Rightarrow s'$ and $p \Rightarrow p'$ are possible only if $s \sqsubseteq_s s'$ and $p \sqsubseteq_p p'$. In the following, we write, for the sake of simplicity, both relations by $\sqsubseteq$. We also assume that S and P have the following structure with respect to $\sqsubseteq$.

Usually, s' is more detailed or has richer functions than s. Though, what the relation $\sqsubseteq$ will take depends on the evolution we consider, we are still able to pose a general and acceptable restrictions on it.

1. S is a partially ordered set with respect to $\sqsubseteq$ and there exists the greatest lower bound $s \sqcap s'$ for any $s, s' \in S$. Mathematically, S is called a *lower semi-lattice.* The structure is illustrated in Figure 1.

2. Two operators *difference* $\ominus$ and *composition* $\oplus$ are introduced with a set of *tags* T_S associated with S.

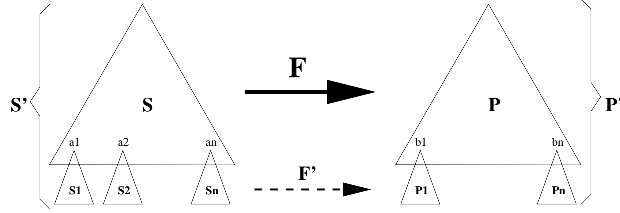

Figure 2: A simple visualization of evolutionary development process.

$$\ominus : S \times S \to (T_S \to S)$$
$$\oplus : S \times (T_S \to S) \to S$$

The operators are required to satisfy:

$$s \sqsubseteq s' \text{ implies } s' = s \oplus (s' \ominus s)$$

We call a set S satisfying the above properties 1 and 2 an *evolutionary domain*, denoted as: $(S, \sqsubseteq, \ominus, \oplus)$. In the evolutionary domain, we can determine whether one object is more evolved than the other, and, if so, we can extract their difference as a set of objects in the domain together with their associated tags.

2.2 Evolutionary Development Process

Suppose the sets of specifications and programs are formulated as evolutionary domains $(S, \sqsubseteq_s, \ominus_s, \oplus_s)$ and $(P, \sqsubseteq_p, \ominus_p, \oplus_p)$ respectively. We examine the evolutionary development process between them.

We assume here for the sake of simplicity, the program development process could be represented by a mapping $\mathcal{F}$ between the above evolutionary domains, and assume that a unique program $\mathcal{F}(s)$ is obtained from the specification $s \in S$ [1]. That is, $\mathcal{F}$: $S \to P$. This mapping $\mathcal{F}$ is illustrated in Figure 2.

Suppose $s \sqsubseteq_s s'$, and denote $\Delta s = s' \ominus_s s$. An evolutionary development process $\mathcal{F}$ should possess the following four properties:

- Realizability: For any $s \in S, \mathcal{F}(s) \in P$.
- Monotonicity: $\mathcal{F}(s) \sqsubseteq_p \mathcal{F}(s')$
- Incrementality: $\mathcal{F}(s \oplus_s \Delta s) = \mathcal{F}(s) \oplus_p \mathcal{F}'(\Delta s)$ for some $\mathcal{F}'$: $(T_S \to S) \to (T_P \to P)$. More specifically, the fragment mapping $\mathcal{F}'$ is derived from $\mathcal{F}$ as:

$$\mathcal{F}'(\{(a_1, s_1), ..., (a_n, s_n)\}) = \{(b_1, \mathcal{F}(s_1)), ..., (b_n, \mathcal{F}(s_n))\} \text{ for } b_1, ..., b_n \in T_P.$$

- Locatability: $p = \mathcal{F}(s)$ could be obtained as a substructure of $p' = \mathcal{F}(s')$ and be located in p' by a *locator* $\mathcal{L}$ as $p = \mathcal{L}(s, s', p')$.

The tag sets correspondence between two domains are expressed by a tracing function $\mathcal{C}$: $T_S \to T_P$, and $\mathcal{C}(a_i) = b_i$. With respect to collaborative designs, tags a_i, b_i are explicitly corresponded in the specification and program domains. They are both specified by a pair of class and collaboration labels (a, o) (details in Section 5). This clear mapping between a_i and b_i is the root of high traceability of role-based designs. Comparing with traditional object-oriented (OO) approaches, this type of design can trace to a finer granularity than objects, i.e. to each role inside an actor object. As a result, evolving a role, if any change arises, would be less costly than doing so to the whole object as in conventional OO technology.

The idea of the evolutionary development is to express that the evolved program should be obtained by merging the original program with program fragments which are implemented from specification difference.

[1] It is not difficult to extend the mapping $\mathcal{F}(s)$ to multiple programs. In this case, $\mathcal{F}$ is formulated as a relation and the subsequential discussions on the development process can be carried out based on the same concepts introduced.

The last property is about the ability to extract a sub-program p from p'. This ability is required to obtain a less evolved program, which corresponds to a less evolved specification, within a given program. This is needed in the second scheme of evolution stated in the next section. It is expressed by a function $\mathcal{E}$ extracting from a program all fragments associated with given program tags. Mathematically, it has the form:

$$\mathcal{E}\colon P \times (T_P \to P) \to P.$$

For example, $\mathcal{E}(p', \{(b_1, p_1), ..., (b_n, p_n)\}) = p$ which is the reversal of evolution operator $\oplus_p$. In this way, we can think the set $\{(b_1, p_1), ..., (b_n, p_n)\}$ is actually the program difference mapping for specification difference Δs. Therefore, locator $\mathcal{L}$ can be expressed as follows:

$$\mathcal{L}(s, s', p') = \mathcal{E}(p', \mathcal{F}'(\Delta s))$$

3 General Evolution Schemes for Software Changes

Under the existence of a lattice-structured specification and program domains, we propose two fundamental schemes for software evolution. We think that in either case, the evolution path is guaranteed to be minimum in number of evolution activities.

- First scheme: Given an existing specification of s, the intended version s' is more evolved than s, i.e. $s \sqsubseteq s'$.
- Second scheme: There is no evolution relationship between s and the destined specification s', i.e. $s \not\sqsubseteq s'$ and $s' \not\sqsubseteq s$. We just know that, there exists a specification t in the same lattice-structured domain which is less evolved than both s and s', namely $t \sqsubseteq s$ and $t \sqsubseteq s'$.

The first scheme is simple. The evolution step is direct upward path from s to s'.

1. Find the specification difference: $\Delta s = s' \ominus_s s$.
2. Map the corresponding program fragments for that specification difference: $\Delta p = \mathcal{F}'(\Delta s)$.
3. Compose those fragments with existing program: $p' = p \oplus_p \Delta p$.

On the other hand, the second scheme can be performed as follows:

1. Extract the common specification t between s and s', preferably t is the *greatest lower bound* of s and s': $t = s \sqcap s'$. This is possible due to the semi-lattice properties of evolutionary specification domain.
2. By locator $\mathcal{L}$, locate the sub-program implementing t from p. That is, $r = \mathcal{L}(t, s, p)$.
3. Apply the first scheme for evolution path from $r \Rightarrow p'$ according to specification evolution $t \Rightarrow s'$.

In this framework for software evolution, the problem of coping with changes, anticipated or unanticipated, is reduced to how possible specification changes are captured in the specification set S and reflected on the evolution relation $\sqsubseteq_s$. Really, a specification s can be evolved only to another s' which is in the set S. If the new specification s' is not in S, evolution activity is just to construct a brand-new program p' corresponding to s' from the scratch. This limitation of the framework will be clarified further in case of collaborative designs later in Section 7. More specifically, the admissibility of changes in S could be reduced to the possibility of performing the following activities in the case of the second evolution scheme.

- Testing if $s' \in S$ and $s \sqsubseteq_s s'$
- Finding the greatest lower bound $t = s \sqcap s'$
- Making difference $s' \ominus_s t$

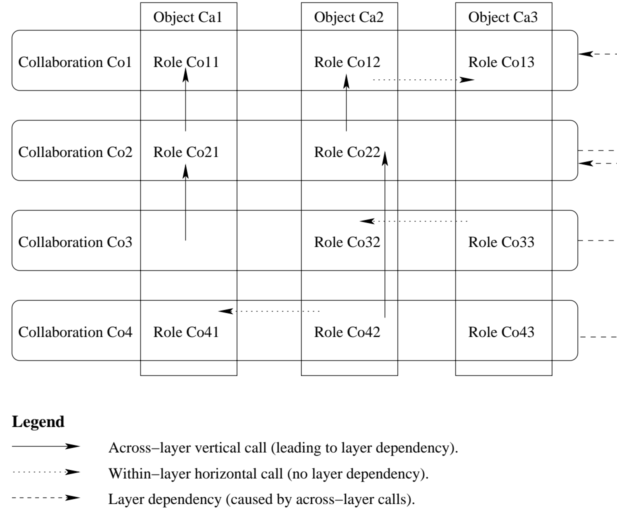

Figure 3: Example of collaboration decomposition. Horizontal round rectangles are collaborations, while vertical rectangles are objects. Their intersections represent roles.

If the set of all the specifications under consideration are defined by some formal system K, together with the evolution relation, the above will be all performed within K, possibly by mechanical means. Even in case specifications are not given formally, we need to do them informally using our experience and knowledge about the application domain, possibly with the support of evolution environment. And the possibility of performing the above activities determines the degree of preparation to unanticipated changes. As later shown, the second scheme is more robust than the first which is simply an incremental evolution scheme. It can cope with higher degree of unanticipateness.

4 Collaborative Static Structure Modeling

4.1 Collaboration-Based Software Design

A key objective in designing evolvable and reusable software modules is to encapsulate within each module a single and orthogonal aspect of application design [4, 6]. According to [4], an orthogonal system is constructed in layers. Moreover, function calls and messages are routed either horizontally inside layers or vertically across layers. Layer-crossing calls must be wrapped within an object class though. Interfaces between layer components are small and hence resilient to changes. One view on the system for such orthogonal aspects is based on collaborations. Each collaboration consists of some classes and the interaction between them. Each actor class encapsulates several roles each of which represents that class under one of the collaborations.

Figure 3 displays a system consisting of a set of object classes Ca_1, Ca_2, Ca_3 together with a set of collaborations Co_1, Co_2, Co_3, Co_4. Between collaborations, there may exist some dependencies. As a rule, if collaboration A is dependent on B then A must be instantiated after B. The intersection between a collaboration and a class is a role. A role could be empty if the class does not participate in that collaboration.

Our discussion will start on how a mixin-based system is formally specified. In this formal specification, each role is treated as a primitive (via *mixin term*). In reality, a mixin term is the interface of a role. As a result, a mixin term should contain suitable methods and data fields (i.e. attributes) associated with that role. How to determine such entities for a role is left for future work. At this stage, this paper assumes such interfaces are known in advance.

Later we define the evolution operators such as $\ominus$, $\oplus$ etc on the above formally defined systems. These operators operate on mixin terms, and hence on role interfaces. Such formal evolution operators will change the contents of mixin terms, i.e. role interfaces change. As a result, system evolves.

4.2 A Small Case Study

This section presents a simple case study which will be used to illustrate the power of our proposed framework. Our example is initially a simple data structure of binary tree [5]. This data structure allocates memory for an item into a container whose management scheme is like a binary tree.

From this data structure, some illustrating evolution tasks are performed. The first change requires this data structure to store the creator's name of each item (Section 6.2). On the other hand, the original data structure evolves by encapsulating an additional time-related functionality (Section 6.3). Whenever a node in this binary tree is accessed, its associated timestamp is updated.

The previous two evolution tasks are incremental changes to the existing system, namely new system is able to handle richer functions than its predecessor. Our last requirement change happens when the latest data structure (i.e. time-related binary tree) undergoes a major change. The management scheme fundamentally changes from binary tree to linked list, while its timestamp functionality is kept as before (Section 6.4). The *degree of unanticipateness* of this change is much higher than the previous two incremental changes.

4.3 Formal Model of Static Structure

4.3.1 Roles as Primitives of the Model

As shown in Figure 3 and Section 4.1, a typical role-based design has two sets of classes and collaborations. Regarding to classes of the system, each class encapsulates attributes for interactions with other classes to accomplish collaborations. This view is inherently object-oriented. The improvement over that OO view is that our model reveals more about internal structure of each class. Each class is vertically composed of several roles, while each collaboration is a horizontal suite of roles. From that observation, roles are treated as primitives in system model.

Originally, some of implementation techniques for role-based design are rooted at techniques using mixin classes [1]. Mixin is usually named as abstract subclass [5]. It represents a mechanism for specifying classes that will eventually inherit from a super-class while this super-class is not yet specified at mixin's definition. According to [5], each collaboration is implemented by a *outer mixin* or *mixin layer.* Each outer mixin, in turn, encapsulates several roles inside. Each role is mapped to a *inner mixin* during implementation phase. Thus, a role is represented by a *mixin term.* Hereafter, the mixin term is used to mean inner mixins. In case of outer mixins, they will be explicitly explained to avoid ambiguity.

In general, a system contains several collaborations. A class plays a role in a collaboration, but it may not in another (i.e. empty role). Let $\mathcal{M}$ be the set of mixin terms including ϵ - empty mixin for empty role. As a mixin term is used to express a role, it is indeed an inner mixin. In other words, $\mathcal{M}$ is a set of all inner mixins.

4.3.2 Basic Object-Oriented Class Domain: Evolution Relation and Operators

In our model, primitive mixins are equally treated as usual OO classes. Like a class, a mixin encapsulates all methods and variables, i.e. *attributes.* These attributes, to some extent, represent evolution aspect of mixins when compared with each other.

Definition 1 *Given two mixins $m_1, m_2 \in \mathcal{M}$, from OO perspective, m_2 is said to be more evolved than m_1, denoted as $m_1 \sqsubseteq_R m_2$, if all attributes in m_1 are also encapsulated in m_2.*

This relation is a partial order relation. Certainly, the empty mixin ϵ is less evolved than any mixin because it contains no attribute, i.e. $\forall m \in \mathcal{M} : \epsilon \sqsubseteq_R m$.

The basic operators are defined over two mixins. They are *intersection*, *difference*, *composition* and *initialization* operators, denoted as $\cap_R$, $\oplus_R$, $\ominus_R$, $\circ$ respectively. $\cap_R$ returns a new mixin whose attributes are shared by those initial mixins. The semantic of $\oplus_R$ is to create a new mixin whose attributes are from the union set of attributes of two initial disjoint mixins (i.e. having no attribute in common). On the other hand, $\ominus_R$ returns a new mixin having attributes in the first but not in the second, given the second operand is less evolved than the first in terms of $\sqsubseteq_R$. Finally, $\circ$ takes two operands. It initializes the second as supermixin of the first.

Definition 2 *Given $m_1, m_2 \in \mathcal{M}$, the intersection, composition, difference and initialization operators between these two are defined as:*

- $\cap_R : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$, *finds the intersection of m_1 and m_2.*
- $\oplus_R : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$, *returns the mixin formed by adding two mixins together.*
- $\ominus_R : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$, *returns the difference between two mixins.*
- $\circ : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$, *initializes the second mixin as supermixin of the first.*

The subscript R in the above means role as mixin terms are actually formal representations of roles - the basic elements in formal model discussed in the next section.

4.3.3 Formal Model of Static Structure

This section gives the details of formal specification of a typical role-based system. According to arguments in Section 4.1, a collaboration-based system contains four essential parts which altogether define uniquely a system. They are: set of collaboration labels; set of class labels; set of roles (mixin terms) formed by intersection of a class and a collaboration from the previous two sets respectively; and collaboration dependencies. From a standpoint within a collaboration, the order of roles is not important as long as a role is properly assigned to correct class during composition process. On the contrary, class is sensitive to order of roles; and in a bigger scale, system is dependent on order of collaborations as the order is not commutative [5]

Let $\mathcal{CA}$ and $\mathcal{CO}$ be universal sets of class and collaboration labels. A typical role-based design could be then structured from the following: a set of classes $Ca \subseteq \mathcal{CA}$; a set of collaborations $Co \subseteq \mathcal{CO}$; a function δ representing the role (mixin term) of a class with respect to a collaboration; and collaboration dependency function realized by ω. The formal specification can be defined below:

Definition 3 *The* static structure model *of a role-based system s is formally defined as a tuple of (Ca, Co, δ, ω) in which:*

- $Ca \subseteq \mathcal{CA}$ *is a set of constituent class labels.*
- $Co \subseteq \mathcal{CO}$ *is a set of collaboration labels.*
- $\delta : \mathcal{CA} \times \mathcal{CO} \rightarrow \mathcal{M}$ *is a role mapping which assigns a mixin in place for the role of a class in a collaboration.*
- $\omega : \mathcal{CO} \rightarrow 2^{\mathcal{CO}}$ *is the dependency constraints between collaborations in the system. 2^A is the powerset of A.*

Concerning with ω function, if a collaboration $o \in Co$ is independent of all other collaborations in Co, $\omega(o) = \emptyset$. The universal set of all such formally defined s is denoted as $\mathcal{S}_f$.

The above formal definition only addresses the static structure of a role-based system. Dynamic behavior is used to determine the contents of mixins. The formal dynamic behavior model of roles is left for future work.

4.4 Evolutionary Domain of Role-Based Static Structure Specification

From the above formal definition, an evolution relationship $\sqsubseteq$ is defined to express how two systems are related to each other.

Definition 4 *Given two role-based formal specifications $s_1 = (Ca_1, Co_1, \delta_1, \omega_1)$ and $s_2 = (Ca_2, Co_2, \delta_2, \omega_2)$. The latter specification is more evolved than the former, denoted as $s_1 \sqsubseteq s_2$, if and only if the following are satisfied:*

1. $Ca_1 \subseteq Ca_2$,
2. $Co_1 \subseteq Co_2$,

3. $\forall a \in Ca_1,\ \ o \in Co_1 : \delta_1(a,o) \sqsubseteq_R \delta_2(a,o)$,

4. $\forall o \in Co_1,\ \omega_1(o) \subseteq \omega_2(o)$.

The third condition confirms that all roles in the second system are more evolved than their counterparts in the former. The fourth condition ensures that although each collaboration may evolve separately, the composition orders between those collaborations in the former system are preserved in the latter. Composition dependency of the former are completely maintained in the latter.

Concerning with evolution schemes mentioned in Section 3, the greatest lower bound of two specifications, namely $t = s_1 \sqcap s_2$ is constructed as follows:

Lemma 5 *Given specifications* $s_1 = (Ca_1, Co_1, \delta_1, \omega_1)$ *and* $s_2 = (Ca_2, Co_2, \delta_2, \omega_2)$. *The* greatest lower bound *of these two specifications,* $s_1 \sqcap s_2 = t = (Ca_t, Co_t, \delta_t, \omega_t)$*, is constructed as follows*

1. $Ca_t = Ca_1 \cap Ca_2$,

2. $Co_t = Co_1 \cap Co_2$,

3. $\forall a \in Ca_t,\ \ o \in Co_t : \delta_t(a,o) = \delta_1(a,o) \cap_R \delta_2(a,o)$,

4. $\forall o \in Co_t,\ \omega_t(o) = \omega_1(o) \cap \omega_2(o)$.

Lemma 6 *The tuple* $(\mathcal{S}_f, \sqsubseteq)$ *forms a lower semi-lattice.*

The proof of this lemma is skipped due to space limitation.

Concerning with $\ominus$ and $\oplus$ operators, we need to define the tag set corresponding to *specification fragments* during evolution process. We assigns the tag set T_S (specified in Section 2) to be exactly the product of $\mathcal{CA} \times \mathcal{CO}$.

- $\ominus : \mathcal{S}_f \times \mathcal{S}_f \to ((\mathcal{CA} \times \mathcal{CO}) \to \mathcal{S}_f)$.
- $\oplus : \mathcal{S}_f \times ((\mathcal{CA} \times \mathcal{CO}) \to \mathcal{S}_f) \to \mathcal{S}_f$.

Definition 7 *Let* $s_1, s_2 \in \mathcal{S}_f$ *and* $s_1 \sqsubseteq s_2$. *The difference between* s_1 *and* s_2 *is given by*

$$s_2 \ominus s_1 = \{(a,o) \mapsto s_{@(a,o)} \mid \ s_{@(a,o)} = (Ca_2, Co_2, \delta_{@(a,o)}, \omega_{@(a,o)}), \forall a \in Ca_2, o \in Co_2\}.$$

The notation $s_{@(a,o)}$ is the specification fragment at (a,o), i.e. it could be represented by a 4-tuple of class and collaboration label sets, role mapping and collaboration dependencies. In this definition, $\delta_{@(a,o)}(ca, co)$ is defined as:

- $\delta_2(a,o) \ominus_R \delta_1(a,o)$, if $(ca = a) \wedge (co = o)$,
- ϵ, otherwise.

On the other hand, $\omega_{@(a,o)} = \{o \mapsto \omega_2(o)\}$. Obviously, $s_{@(a,o)} \in \mathcal{S}_f$ and it represents a mixin fragment specification from the perspective of a role-based design.

For the composition operator, there are two cases to consider. The first is simpler when this addition does not cause any new class or collaboration creation. On the contrary, the second case occurs when new class or collaboration is required. For simplicity, the following definition only deals with a mapping D whose cardinality is one.

Definition 8 *Let* $s = (Ca, Co, \delta, \omega) \in \mathcal{S}_f$. *The composition of* s *with specification mapping* $D = \{(a,o) \mapsto s_d\}$*, where* $s_d = (Ca_d, Co_d, \delta_d, \omega_d)$*, is*

$$s \oplus D = s' = (Ca', Co', \delta', \omega').$$

1. *if* $a \in Ca, o \in Co$:
   - $Ca' = Ca$,
   - $Co' = Co$,

- $\delta'(ca, co) = \delta(ca, co) \ \forall ca \in Ca, co \in Co : (ca \neq a) \vee (co \neq o)$,
- $\delta'(a, o) = \delta(a, o) \oplus_R \delta_d(a, o)$,
- $\forall co \in Co, co \neq o : \omega'(co) = \omega(co)$,
- $\omega'(o) = \omega(o) \cup \omega_d(o)$.

2. *if* $(a \notin Ca) \vee (o \notin Co)$
   - $Ca' = Ca \cup \{a\}$,
   - $Co' = Co \cup \{o\}$,
   - $\delta'(ca, co) = \delta(ca, co) \ \forall ca \in Ca', co \in Co' : (ca \neq a) \vee (co \neq o)$,
   - $\delta'(a, o) = \delta_d(a, o)$,
   - $\forall co \in Co', co \neq o : \omega'(co) = \omega(co)$,
   - $\omega'(o) = \omega(o) \cup \omega_d(o)$, *if* $o \in Co$,
   - $\omega'(o) = \omega_d(o)$, *otherwise.*

In general, if $D = \{d_1, ..., d_n\}$, we have:

$$s \oplus D = ((s \oplus \{d_1\}) \oplus ... \oplus \{d_{n-1}\}) \oplus \{d_n\}$$

Theorem 9 *The set of $\mathcal{S}_f$ together with above $\sqsubseteq$ evolution relation, $\ominus$ and $\oplus$ operators forms an evolutionary domain $(\mathcal{S}_f, \sqsubseteq, \ominus, \oplus)$.*

This domain serves as a basis to analyze the evolution of role-based systems in evolutionary development process discussed in Section 5.

5 Evolutionary Development Process in Role-Based Designs

Based on previous formal specification, this section presents a formal description of evolutionary development process for role-based designs. Because the formal specification so far only mentions static structure, the mapped programs are only represented as class (mixin) skeletons. They could be composed or woven from their corresponding formal specifications as given in Section 5.2. This development process is inherently mixin-based.

5.1 Linear Ordering of Mixin Layers

This section presents the way collaborations are composed during system development. The collaborations are sequentially composed according to their significance to the system. Based on the $\omega : \mathcal{CO} \rightarrow 2^{CO}$ factor, the ordering dependency between collaborations are specified. Our software development framework only deals with acyclic dependencies.

We represent the dependency between collaborations by a directed graph. In that graph, each collaboration is shown as a node, and a directed edge starts from a collaboration to the collaboration it depends on. Each node is then associated by an *in-degree* identifying how many layers are dependent on it. There are some properties involving with dependency acyclic graph.

Property 10 *In an acyclic directed graph, there is at least one node with zero in-degree.*

Property 11 *Given an acyclic directed graph, if one or more nodes are taken away from the graph together with their outward directed edges, the resulting subgraph is still acyclic.*

Before going to the formal development process definition, some notations are shown with respect to tuple (Ca, Co, δ, ω), collaboration o and class a.

- $\mathcal{G}(\omega)$ is the associated directed graph of ω;
- $\delta_{\dashv o} = \{(a, o, m) | (a, o, m) \in \delta\}$ - the (horizontal $\dashv$) projection of mapping δ on collaboration o;

- $\delta_{\neg o} = \delta \setminus \delta_{\dashv o}$ - the mapping δ after taking away elements related to collaboration o;
- $\omega_{\neg o} = \{(co, s) | (co, s) \in \omega \wedge (co \neq o)\}$ - the dependency mapping ω after collaboration o is taken away;
- $\mathcal{H}_{\perp a}$ - the projection of some entity $\mathcal{H}$ onto class a.
- Let α be a specification entity, $\sharp\alpha$ is its respective implementation in the program domain.

5.2 Formal Development Aspects

The formal development process is defined as: $\mathcal{F}$: $S \to P$, where S is the set of all role-based specifications (Ca, Co, δ, ω); P is the set of programs implementing those specifications.

Definition 12 *Given $s = (Ca, Co, \delta, \omega) \in S$, the mapping of s to program domain P is defined as:*

- $\mathcal{F}(s) = \sharp\epsilon$ *if* $Co = \emptyset$,
- $\mathcal{F}(s) = \mathcal{F}_o(Ca, o, \delta_{\dashv o}) \circ \mathcal{F}(Ca, Co \setminus \{o\}, \delta_{\neg o}, \omega_{\neg o})$, *where o is a layer with zero in-degree in $\mathcal{G}(\omega)$, otherwise.*

Of course, the dependency subgraph $\mathcal{G}(\omega_{\neg o})$ is acyclic due to Property 11. It should contain at least one zero in-degree node because of Property 10. That node is selected for recursive call up to the top layer. During this recursive call, the specification s gets simpler and simpler due to the decreasing cardinality of collaboration label set. This process stops when it contains no more collaboration. As a consequence, this development process certainly halts.

$\mathcal{F}_o$ is a mapping for outer mixins. Because an outer mixin is a suite of inner mixins each of which is assigned with a class label in Ca, an inner mixin is the "projection" of the outer mixin onto the associated label. Instead of dealing with a suite of mixins at a time, $\mathcal{F}_o$'s formal definition is then simplified via its (vertical $\perp$) projection onto a class. The overall $\mathcal{F}_o$ mapping for a collaboration is taken as suite of all projections onto class labels in Ca.

Definition 13 *The projection of $\mathcal{F}_o$ for an outer mixin onto a class $a \in Ca$ is defined as:*

$$[\mathcal{F}_o(Ca, o, \delta_{\dashv o})]_{\perp a} = \sharp\delta_{\dashv o}(a, o) \equiv \sharp\delta(a, o).$$

The right hand side of expression is inner mixin mapping assigned to role of class a with respect to collaboration o.

By Definition 13, Definition 12 is simplified via system projection on a class a.

Definition 14 *Given $s = (Ca, Co, \delta, \omega)$,*

- $\mathcal{F}(s)_{\perp a} = \sharp\epsilon$ *if* $Co = \emptyset$,
- $\mathcal{F}(s)_{\perp a} = [\mathcal{F}_o(Ca, o, \delta_{\dashv o})]_{\perp a} \circ [\mathcal{F}(Ca, Co \setminus \{o\}, \delta_{\neg o}, \omega_{\neg o})]_{\perp a}$, *where o is a layer with zero in-degree in $\mathcal{G}(\omega)$, otherwise.*

In this simplified definition, only inner mixins are to be dealt with. According to Definition 13, in case of $Co \neq \emptyset$, the right hand side can be simplified as:

$$\mathcal{F}(s)_{\perp a} = \sharp\delta(a, o) \circ [\mathcal{F}(Ca, Co \setminus \{o\}, \delta_{\neg o}, \omega_{\neg o})]_{\perp a}$$

This is essentially recursive on $\mathcal{F}$ with respect to collaboration cardinality. Let the linear layer ordering be $o_1, o_2, ..., o_n$. Since the $\circ$ operator instantiates the second operand as supermixin of the first operand (Section 4.3.2), fully expanded form of system projection on class a is:

$$\begin{aligned}
\mathcal{F}(s)_{\perp a} &= \sharp\delta(a, o_1) \circ (\mathcal{F}(Ca, Co \setminus \{o_1\}, \delta_{\neg o_1}, \omega_{\neg o_1})]_{\perp a}) \\
&\ldots \\
&= \sharp\delta(a, o_1) \circ (\sharp\delta(a, o_2) \circ (\ldots \circ \sharp\delta(a, o_n)(\sharp\epsilon))) \\
&= \sharp\delta(a, o_1) \circ \sharp\delta(a, o_2) \circ \ldots \circ \sharp\delta(a, o_n) \circ \sharp\epsilon \\
&\equiv \sharp\delta(a, o_1) \circ \sharp\delta(a, o_2) \circ \ldots \circ \sharp\delta(a, o_n)
\end{aligned}$$

Theorem 15 *Let $o_1, o_2, ..., o_n$ be the linear layers ordering from $\mathcal{G}(\omega)$. The formal projection of $\mathcal{F}(s)$ onto a class $a \in Ca$ is a composite mixin,*

$$\mathcal{F}(s)_{\perp a} = \sharp\delta(a, o_1) \circ \sharp\delta(a, o_2) \circ ... \circ \sharp\delta(a, o_n)$$

This theorem provides the basic class structure for a role-based design from OO perspective. $\mathcal{F}(s)_{\perp a}$ is essentially the mixin of class a for participating in all collaborations. The concrete class for a is formed by instantiating that mixin with an empty class (`NULL`).

5.3 A Typical Mixin-Based Implementation via C++ Template

There are several languages supporting mixin technology. One of them is C++. In C++, mixin mechanism is accomplished via template [5]. Given the formal specification of a collaborative design, we can transform such a specification to C++ codes.

Let co be a collaboration among many in a system. ca is a class participating in that collaboration. The role of ca with respect to co is specified by mixin term m which is essentially a role interface. The C++ implementation of the collaboration co and class ca would look like:

```
template <class Para> class co:  public Para{// collaboration co declaration
   ...// Other role declarations
   class ca:  public Para::ca{// role ca declaration
   ...// Content of m comes here, i.e.  role interface definition.
   };
   ...
};
```

Let o_1, o_2, ..., o_n be collaborations in a given system and the linear ordering of those collaborations is in such an order. Individually, each o_i can be represented in C++ as co above. The overall application class `SYS_CLASS` of the system can be formed by composing those collaborations in that linear order. That is:

```
typedef o1 <o2 <...<on> >...> SYS_CLASS;
```

In addition, individual actor class ca can be retrieved by projecting `SYS_CLASS` to the appropriate actor name, such as:

```
typedef SYS_CLASS::ca CA_CLASS;
```

This paper focuses on the theoretical part of mixin-based implementation for collaborative designs rather than to be specific to any programming language. In the subsequent examples, the C++ classes of roles are only sketched for illustration only. Automatic code generation from formal specification to C++ is closely related with dynamic behavior model. That code generating process is not difficult once the dynamic model is completed.

5.4 Formal Evolution Aspects

The previous section deals with development aspects during software life cycle. This section formalizes evolution-related activities according to the framework in Section 2.

This section formalizes the way on how programs are incrementally composed (via $\oplus_p$) and how a sub-program is located within another (via $\mathcal{E}$). These two operators are involved with program modification during evolution process. They are based on tracing mapping $\mathcal{C}$ between specification tags (a_i's) and program tags (b_i's). In our model, a_i, b_i are both specified by a pair of class and collaboration labels (a, o). Hence, $\mathcal{C}$ is essentially an identity mapping - an advantage of collaborative design in terms of traceability. Note that even though a_i and b_i look the similar, their meanings are different in specification and program domains.

Let $p = \mathcal{F}(s)$, $s' \ominus_s s = \Delta s$ and $p' = \mathcal{F}(s') = p \oplus_p \mathcal{F}'(\Delta s)$ and $p = \mathcal{E}(p', \mathcal{F}'(\Delta s))$. According to Section 2 when tags a_i, b_i are expressed by a pair (a, o), we have the following:

$$\mathcal{F}'(\Delta s) = \{(ca, co) \mapsto \mathcal{F}(s_{@(ca,co)}) | (ca, co) \mapsto s_{@(ca,co)} \in \Delta s\}.$$

Its projection onto class a is:

$$\mathcal{F}'(\Delta_s)_{\perp a} = \{(ca,co) \mapsto \mathcal{F}(s_{@(ca,co)})_{\perp a} | (ca,co) \mapsto \mathcal{F}(s_{@(ca,co)}) \in \mathcal{F}'(\Delta s)\}.$$

According to Definition 7, it is obvious that in specification $s_{@(a,o)}$, its associated role mapping $\delta_{@(a,o)}$ is defined at (a,o) only. At other places, the mapping is ϵ - empty mixin. Applying Theorem 15, we can derive the value of $\mathcal{F}(s_{@(a,o)})$ as follows:

- $\mathcal{F}(s_{@(ca,co)})_{\perp a} = \sharp\epsilon \circ ... \circ \sharp\delta_{@(ca,co)}(ca,co) \circ ... \circ \sharp\epsilon = \sharp\delta_{@(ca,co)}(a,co)$, if $ca = a$. (Refer to Definition 7 for the meanings of $s_{@(ca,co)}$ and $\delta_{@(ca,co)}$)
- $\mathcal{F}(s_{@(ca,co)})_{\perp a} = \sharp\epsilon = \sharp\delta_{@(ca,co)}(a,co)$, if $ca \neq a$.

Theorem 16 *$\mathcal{F}(s_{@(ca,co)})_{\perp a} = \sharp\delta_{@(ca,co)}(a,co)$, $\forall a \in Ca$.*

By Theorem 16, the projection of program difference $\mathcal{F}'(\Delta s)$ onto class a is essentially:

$$\mathcal{F}'(\Delta s)_{\perp a} = \{(ca,co) \mapsto \sharp\delta_{@(ca,co)}(a,co) | (ca,co) \mapsto \mathcal{F}(s_{@(ca,co)}) \in \mathcal{F}'(\Delta s)\} \subset ((\mathcal{CA} \times \mathcal{CO}) \to \mathcal{M})$$

On the other hand, the operators $\oplus_p$ and $\mathcal{E}$ are based on two basic operators $+_m$ and $-_m$ of the forms: [2]

- $-_m : \mathcal{M} \times ((\mathcal{CA} \times \mathcal{CO}) \to \mathcal{M}) \to \mathcal{M}$.
- $+_m : \mathcal{M} \times ((\mathcal{CA} \times \mathcal{CO}) \to \mathcal{M}) \to \mathcal{M}$.

Let cm be a composite mixin for class a formed by development process $\mathcal{F}$ as in Theorem 15, namely

$$cm = \sharp\delta(a,o_1) \circ \sharp\delta(a,o_2) \circ ... \circ \sharp\delta(a,o_n).$$

The composition or extraction operations of a fragment $\{(a,o) \mapsto im\}$ with respect to cm are valid only if the tags are compatible. The tag of fragment (a,o) should be among tags (a,o_i) of cm. In this definition, we assume that the second operand has only one element. For multiple cardinality, we just pick out one by one and iterate this definition until all members in the second operand are done. Of course, if the second operand is an empty set or the tags are not compatible, the first operand is not affected in both $+_m$ and $-_m$.

Definition 17 *Suppose $o = o_i$, the semantics of $-_m$ and $+_m$ are:*

- $\sharp\delta(a,o_1) \circ \sharp\delta(a,o_2) \circ ... \circ \sharp\delta(a,o_n) +_m \{(a,o) \mapsto im\} = \sharp\delta(a,o_1) \circ ... \circ (\sharp\delta(a,o_i) \oplus_R im) \circ ... \circ \sharp\delta(a,o_n)$.
- $\sharp\delta(a,o_1) \circ \sharp\delta(a,o_2) \circ ... \circ \sharp\delta(a,o_n) -_m \{(a,o) \mapsto im\} = \sharp\delta(a,o_1) \circ ... \circ (\sharp\delta(a,o_i) \ominus_R im) \circ ... \circ \sharp\delta(a,o_n)$.

Like previous section, the definitions of $\mathcal{E}$ and $\oplus_p$ are expressed by the respective projections of two operands onto class a.

Definition 18 *Given s, s', p, p' as previously defined, the formal definitions of $\oplus_p$ and $\mathcal{E}$ are given as follows.*

- $p' = p \oplus_p \mathcal{F}'(\Delta s)$ *iff* $\forall a \in Ca$, $p'_{\perp a} = p_{\perp a} +_m \mathcal{F}'(\Delta s)_{\perp a} \equiv \mathcal{F}(s)_{\perp a} +_m \mathcal{F}'(\Delta s)_{\perp a}$.
- $p = \mathcal{E}(p', \mathcal{F}'(\Delta s))$ *iff* $\forall a \in Ca$, $p_{\perp a} = p'_{\perp a} -_m \mathcal{F}'(\Delta s)_{\perp a} \equiv \mathcal{F}(s')_{\perp a} -_m \mathcal{F}'(\Delta s)_{\perp a}$.

6 Examples

6.1 Binary Tree Problem

This section explains briefly the ideas discussed so far for role-based static structure. Our example is initially a simple data structure of binary tree [5]. This data structure allocates memory for a node into a container with binary tree management scheme. By analysis, this simple data structure consists of two collaborations: `Alloc` and `BinTree`. In addition, for each collaboration, there are only two classes: `Node` and `Container`. This system is shown in Figure 4. Note that the `BinTree` collaboration requires the existence of `Alloc` for its operations.

According to the Definition 3, this system $s_1 = (Ca_1, Co_1, \delta_1, \omega_1)$ is formalized as follows:

[2] m stands for mixins

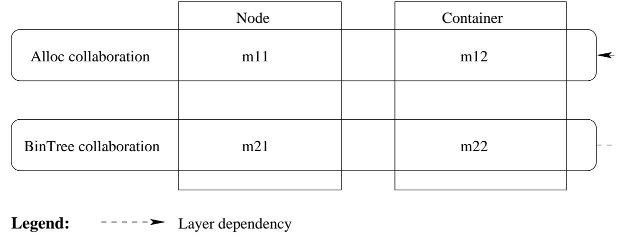

Figure 4: A simple binary tree data structure from the collaboration-based perspective.

- $Ca_1 = \{Node, Cont\}$,
- $Co_1 = \{Alloc, BinTree\}$,
- $\delta_1 = \{(Node, Alloc, m_{11}), (Cont, Alloc, m_{12}), (Node, BinTree, m_{21}), (Cont, BinTree, m_{22})\}$,
- $\omega_1 = \{(BinTree, \{Alloc\})\}$.

In terms of concrete definition, m_{11} defines a role for `Node` class in the context of `Alloc` collaboration. It holds the actual value of data item. On the other hand, m_{12} declares the memory allocation method for each node. It is used in place of `Container` in the `Alloc` collaboration. In C++, they could be expressed in a simplified form as:

```
class Node {// ♯m11 definition
   EleType m_tElement;// The actual stored data
public:
   ...// method declarations
};

class Container {// ♯m12 definition
public:
   // The actual type of stored data
   virtual Node* NodeAlloc();
   // memory allocation call must be virtual
   // etc.
};
```

Mixins m_{21} and m_{22} are very similar with their respective roles in `BinTree` layer.

Let p be a program formed by $\mathcal{F}(s_1)$. By Theorem 15, we have:

- $p_{\perp Node} = \sharp m_{21} \circ \sharp m_{11}$
- $p_{\perp Cont} = \sharp m_{22} \circ \sharp m_{12}$

6.2 Local Role Change

Suppose we need a change to the binary tree problem above. Besides storing data items, this binary tree is required to store the creator's name of each item. As the owner name and data value are coupled together while the binary tree management scheme is not affected, it seems that changes are local to `Alloc` layer. In that layer, new `Node` class should include one more attribute for owner's name. On the other hand, new `Container` class should refine the old `NodeAlloc()` method to handle new `Node` class. Both roles have proceeded to a higher level of evolution. This new data structure is shown in Figure 5. Mixins m'_{11}, m'_{12} inherit m_{11}, m_{12} respectively.

This new system $s_2 = (Ca_2, Co_2, \delta_2, \omega_2)$ has:

- $Ca_2 = \{Node, Cont\}$,
- $Co_2 = \{Alloc, BinTree\}$,

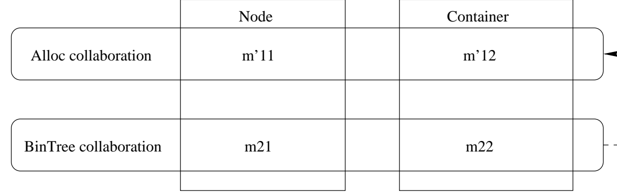

Figure 5: An identical binary tree data structure but different mixins in Alloc layer: mixins m'_{11}, m'_{12} replace m_{11}, m_{12}.

- $\delta_2 = \{(Node, Alloc, m'_{11}), (Cont, Alloc, m'_{12}), (Node, BinTree, m_{21}), (Cont, BinTree, m_{22})\}$,
- $\omega_2 = \{(BinTree, \{Alloc\})\}$.

Furthermore, as the mixins evolve from their counterparts, we have:

- $m_{11} \sqsubseteq_R m'_{11}$ and $m'_{11} \ominus_R m_{11} = \Delta_{11}$ (In C++, the mixin difference is possibly the addition of `char *m_sName` attribute for creator's name to old `Node` class).
- $m_{12} \sqsubseteq_R m'_{12}$ and $m'_{12} \ominus_R m_{12} = \Delta_{12}$ (the mixin difference is possibly the refinements of `NodeAlloc` method and destructor call).

By Definition 4, $s_1 \sqsubseteq s_2$. And their specification difference is expressed by:

$$s_2 \ominus s_1 = \{(Node, Alloc) \mapsto s_{@(Node,Alloc)}, (Cont, Alloc) \mapsto s_{@(Cont,Alloc)}\}$$

where:

- $s_{@(Node,Alloc)} = (Ca_2, Co_2, \{(Node, Alloc, \Delta_{11}), (Cont, Alloc, \epsilon), (Node, BinTree, \epsilon), (Cont, BinTree, \epsilon)\}, \emptyset) \equiv (Ca_2, Co_2, \{(Node, Alloc, \Delta_{11})\}, \emptyset)$,
- $s_{@(Cont,Alloc)} = (Ca_2, Co_2, \{(Node, Alloc, \epsilon), (Cont, Alloc, \Delta_{12}), (Node, BinTree, \epsilon), (Cont, BinTree, \epsilon)\}, \emptyset) \equiv (Ca_2, Co_2, \{(Cont, Alloc, \Delta_{12})\}, \emptyset)$.

We can apply the first scheme of evolution mentioned in Section 3 to perform this evolution process. The program difference (Δp) can be obtained as

$$\mathcal{F}'(s_2 \ominus_s s_1) = \{(Node, Alloc) \mapsto \mathcal{F}(s_{@(Node,Alloc)}), (Cont, Alloc) \mapsto \mathcal{F}(s_{@(Cont,Alloc)})\}$$

We then compose Δp with p and get p'. Firstly, look at intermediate result by the composition of

$$p \oplus_p \{(Node, Alloc) \mapsto \mathcal{F}(s_{@(Node,Alloc)})\}$$

The system has only two classes. We project this intermediate result onto both of them.

- $p_{\perp Node} \ +_m \ \{(Node, Alloc) \mapsto \mathcal{F}(s_{@(Node,Alloc)})\}_{\perp Node}$
  $= \sharp m_{21} \circ \sharp m_{11} \ +_m \ \{(Node, Alloc) \mapsto \sharp\delta_{@(Node,Alloc)}(Node, Alloc)\}$ (by Theorem 16)
  $= \sharp m_{21} \circ \sharp m_{11} \ +_m \ \{(Node, Alloc) \mapsto \sharp\Delta_{11}\}$
  $= \sharp m_{21} \circ (\sharp m_{11} \oplus_R \sharp\Delta_{11})$
  $= \sharp m_{21} \circ \sharp m'_{11}$.
- $p_{\perp Cont} \ +_m \ \{(Node, Alloc) \mapsto \mathcal{F}(s_{@(Node,Alloc)})\}_{\perp Cont}$
  $= \sharp m_{22} \circ \sharp m_{12} \ +_m \ \{(Node, Alloc) \mapsto \sharp\delta_{@(Node,Alloc)}(Cont, Alloc)\}$ (by Theorem 16)
  $= \sharp m_{22} \circ \sharp m_{12} \ +_m \ \{(Node, Alloc) \mapsto \sharp\epsilon\}$
  $= \sharp m_{22} \circ \sharp m_{12}$ (due to incompatible tags).

Repeat the same composition step for the second fragment $\{(Cont, Alloc) \mapsto \mathcal{F}(s_{@(Cont,Alloc)})\}$ into the above intermediate program, we should have the result of p':

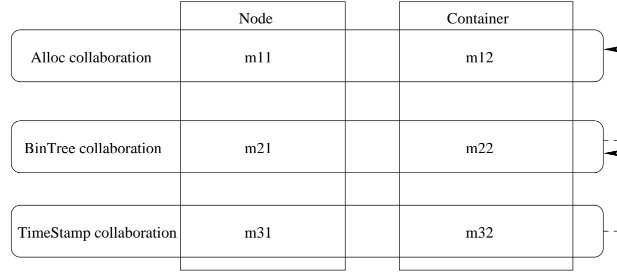

Figure 6: The data structure is extended with a TimeStamp layer.

- $p'_{\perp Node} = (\sharp m_{21} \circ \sharp m'_{11}) \ +_m \ \{(Cont, Alloc) \mapsto \mathcal{F}(s_{@(Cont,Alloc)})_{\perp Node}\}$
  $= (\sharp m_{21} \circ \sharp m'_{11}) \ +_m \ \{(Cont, Alloc) \mapsto \sharp\epsilon\}$
  $= \sharp m_{21} \circ \sharp m'_{11}$ (due to incompatible tags).
- $p'_{\perp Cont} = (\sharp m_{22} \circ \sharp m_{12}) \ +_m \ \{(Cont, Alloc) \mapsto \mathcal{F}(s_{@(Cont,Alloc)})_{\perp Cont}\}$
  $= (\sharp m_{22} \circ \sharp m_{12}) \ +_m \ \{(Cont, Alloc) \mapsto \sharp\Delta_{12}\}$
  $= \sharp m_{22} \circ (\sharp m_{12} \oplus_R \sharp\Delta_{12})$
  $= \sharp m_{22} \circ \sharp m'_{12}$.

The result p' is exactly what we expected.

6.3 Additinal Collaboration Change

Suppose the data structure s_1 should handle an additional time-related function. It updates the timestamp associated with a node whenever that node is accessed. In our view, this functionality is essentially a new collaboration `TimeStamp` between two classes `Node` and `Container`. Moreover, its insertion, deletion, search operations should rely on data management scheme, i.e. `BinTree` in this case. Therefore, after specification analysis, new version $s_3 = (Ca_3, Co_3, \delta_3, \omega_3)$ should look like the one in Figure 6:

- $Ca_3 = \{Node, Cont\}$,
- $Co_3 = \{Alloc, BinTree, TimeStamp\}$,
- $\delta_3 = \{(Node, Alloc, m_{11}), (Cont, Alloc, m_{12}), (Node, BinTree, m_{21}), (Cont, BinTree, m_{22}), (Node, TimeStamp, m_{31}), (Cont, TimeStamp, m_{32})\}$,
- $\omega_3 = \{(BinTree, \{Alloc\}), (TimeStamp, \{BinTree\})\}$.

Of course, $s_1 \sqsubseteq s_3$ and Δs can be specified by a pair of specifications defined below (irrelevant ϵ mixins in $\delta_{@(Node,TimeStamp)}$ and $\delta_{@(Cont,TimeStamp)}$ are dropped):

- $s_{@(Node,TimeStamp)} = (\{Node, Cont\}, \{Alloc, BinTree, TimeStamp\}, \{(Node, TimeStamp, m_{31})\}, \{TimeStamp \mapsto \{BinTree\}\})$,
- $s_{@(Cont,TimeStamp)} = (\{Node, Cont\}, \{Alloc, BinTree, TimeStamp\}, \{(Cont, TimeStamp, m_{32})\}, \{TimeStamp \mapsto \{BinTree\}\})$.

They are parts of an implementation of `TimeStamp` collaboration. We still apply the first scheme to evolve s_1 to s_3. The development and evolution activities could be explained similarly as in previous section. At the end, the projection of new program onto `Node` and `Container` classes are:

- $p'_{\perp Node} = \sharp m_{31} \circ \sharp m_{21} \circ \sharp m_{11}$.
- $p'_{\perp Cont} = \sharp m_{32} \circ \sharp m_{22} \circ \sharp m_{12}$.

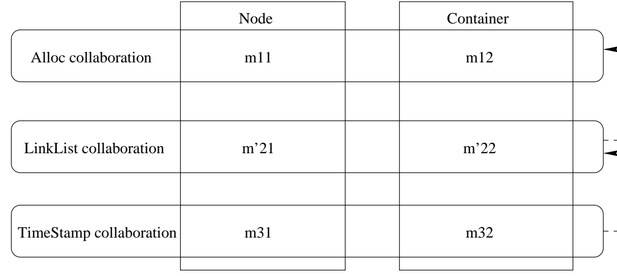

Figure 7: The new data structure is formed by replacing binary tree with linked list management scheme.

6.4 A Fundamental Change in Management Scheme

Up to this stage, our system s_3 essentially consists of three collaborations, namely `Alloc`, `BinTree` and `TimeStamp`; and two actor classes `Node` and `Container`. We have performed two incremental changes according to the first scheme in Section 3. The next evolution step is quite unexpected. We restructure the management scheme of this data structure from binary tree style to linked list style s_4. Obviously, there is no evolution relation between s_3 and s_4. We need to apply the second evolution scheme mentioned in Section 3. The work involves mostly with composition and difference operators over mixin terms whose details are quite similar to the previous two sections. Due to space limitation, we skip such detailed parts and focus on major steps of how s_3 evolves to s_4. The greatest lower bound glb of s_3 and s_4 is found as:

- $Ca_{glb} = \{Node, Cont\}$,
- $Co_{glb} = \{Alloc, TimeStamp\}$,
- $\delta_{glb} = \{(Node, Alloc, m_{11}), (Cont, Alloc, m_{12}), (Node, TimeStamp, m_{31}), (Cont, TimeStamp, m_{32})\}$,
- $\omega_{glb} = \emptyset$.

From glb, we evolves to s_4 by composing the difference $\Delta s = s_4 \ominus glb$ with glb. The specification of Δs can be briefly specified as:

- $s_{@(Node,LinkList)} = (\{Node, Cont\}, \{Alloc, LinkList, TimeStamp\}, \{(Node, LinkList, m'_{21})\}, \{TimeStamp \mapsto \{LinkList\}, LinkList \mapsto \{Alloc\}\})$,
- $s_{@(Cont,LinkList)} = (\{Node, Cont\}, \{Alloc, LinkList, TimeStamp\}, \{(Cont, LinkList, m'_{22})\}, \{TimeStamp \mapsto \{LinkList\}, LinkList \mapsto \{Alloc\}\})$.

After the composition, we get system s_4 which is depicted in Figure 7:

- $Ca_4 = \{Node, Cont\}$,
- $Co_4 = \{Alloc, LinkList, TimeStamp\}$,
- $\delta_4 = \{(Node, Alloc, m_{11}), (Cont, Alloc, m_{12}), (Node, LinkList, m'_{21}), (Cont, LinkList, m'_{22}), (Node, TimeStamp, m_{31}), (Cont, TimeStamp, m_{32})\}$,
- $\omega_4 = \{(LinkList, \{Alloc\}), (TimeStamp, \{LinkList\})\}$.

In addition, the actor classes `Node` and `Container` are:

- $p'_{\perp Node} = \sharp m_{31} \circ \sharp m'_{21} \circ \sharp m_{11}$.
- $p'_{\perp Cont} = \sharp m_{32} \circ \sharp m'_{22} \circ \sharp m_{12}$.

Clearly, the evolution cost in this step is small due to its localized change in collaboration transformation, i.e. `BinTree` to `LinkList`. In other words, we only evolves $\sharp m_{21}$ and $\sharp m_{22}$ to $\sharp m'_{21}$ and $\sharp m'_{22}$ respectively. This evolution cost is much less than evolving $\sharp m_{31} \circ \sharp m_{21} \circ \sharp m_{11}$ to $\sharp m_{31} \circ \sharp m'_{21} \circ \sharp m_{11}$, or $\sharp m_{32} \circ \sharp m_{22} \circ \sharp m_{12}$ to $\sharp m_{32} \circ \sharp m'_{22} \circ \sharp m_{12}$ as in conventional OO technology.

7 Discussion

This paper presents a general evolution framework from the perspective of collaborative designs. The formal theory behind this framework delivers many advantageous aspects of this approach in case of requirement changes, especially with respect to unanticipated events. In our opinion, there is no way to deal with changes without some *sense of anticipation*. A particular software development methodology can not handle all kinds of changes. It could be very efficient in some kind of changes but it can display some serious pitfalls in other groups out of its anticipation range. We believe that one software engineering approach is better than another if it can handle more variety of changes than the other with lesser cost. Our concept about the term "unanticipated software evolution" fits to this claim. The proposed evolutionary domain framework is universally applicable to all formal specification types. In addition, many changes can be captured by this framework as long as the changes are expressable within specification domain. However, because of its abstractness, some particular formal specification is required. In our paper, formal notation for collaborative designs is chosen since this approach is a special case [3] of multi-dimensional separation of concerns (MDSOC) [6] or aspect-oriented programming (AOP) [7]. AOP-based software development paradigm is well-known for its promising effectiveness and efficiency in software development, maintenance and evolution. The advantage of collaborative designs then clearly outperforms traditional OO technology in all stages of software life cycle due to its inherited advanced characteristics from AOP.

Because the proposal is inherently collaboration-based, clearly its power is limited within collaborative systems. All requirements must be (or should be) describable in terms of collaborations and roles so that later these requirements can be transformed into role-based formal specifications. Any change outside this boundary can not be handled by our framework. AOP paradigm classifies two types of *aspects*, namely *static* and *dynamic*. In our point of view, collaboration is a static aspect. Changes involved with dynamic aspects are out of scope at current stage. We plan to extend this work into larger world of AOP in the future.

As stated at the beginning of the paper, we assume the contents of mixin terms (i.e. role interfaces) are known in advance. This preliminary formal model of collaboration-based designs rather focuses on static structure in terms of collaborations and roles. The interface of a role is defined after analyzing dynamic behavior of each actor class within a particular collaboration context such as: how actors react to external events, how messages are sent between actors and in which order, how the role below overrides/refines the pre-defined behavior of the role above in the same actor etc. The formal dynamic behavior model is another point of future work. Once that dynamic behavior model is completed, it is fairly straightforward to transform formal specification to a specific language supporting mixin class such as C++ as partially shown in Section 5.3. Such an automatic code generating tool is then not a major problem.

8 Conclusion

Handling unanticipated changes during software development is regarded as one of the most difficult problems. In our opinion, there are two important factors behind that problem. Firstly, in reality, evolution environment is not robust enough in terms of capturing many kinds of changes. If the environment can handle changes at very abstract level, it would be very likely that unanticipated changes are actually refinements of such an abstract change. Based on such a model of high abstraction, actual evolution activities are guided in a proper manner with minimum number of steps. We propose an abstract evolution framework with a lattice theory at its basis. This framework contains two basic concepts: evolutionary domain which is organized in a lattice structure; and evolutionary development process - essentially a correspondent mapping between specification and program domains.

We believe that, this framework of high abstraction level only helps to captures many kinds of changes. It requires the support of underlying software architecture to perform evolution activities in an efficient way. A design method resilient to changes is the second factor in our proposal. Collaborative designs are one solution to that factor. In collaborative designs, each collaboration is encapsulated in a separate and orthogonal module. As a result, this kind of design is very resilient to changes. Furthermore, changes are usually local in one module. Thus, the evolution cost is quite small.

We claim that the combination of two above factors is the key to software evolution success, even in unanticipated situation. This paper presents a unified software evolution framework with two factors

at the same time. This framework is the application of evolutionary domain from collaboration-based design perspective. The formalization of role-based static structure is introduced. That formalization is then superimposed on evolutionary domain. Furthermore, the formal development process between specification and program is also considered. We also discuss the strength and weakness of the current work. The foremost drawback of our proposal is the boundary defined by only the set of collaborative systems. We intend to extend the coverage into a more general software paradigm of AOP. Examining the assumption about role interfaces is another topic in future work.

References

[1] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP/OOSPLA*, pages 303–311, 1990.

[2] T. Katayama. Evolutionary domains: A basis for sound software evolution. In *Proc. IWPSE*, 2001.

[3] T. T. Nguyen and T. Katayama. Collaboration-based evolvable software implementations: Java and Hyper/J vs. C++ templates composition. In *Proc. IWPSE*, pages 29–34, 2002.

[4] V. Rajlich and J. H. Silva. Evolution and reuse of orthogonal architecture. *IEEE Transactions on Software Engineering*, 22(2):153–157, Febuary 1996.

[5] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP*, 1998.

[6] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N-degrees of separation: Multi-dimensional separation of concerns. In *Proc. ICSE*, pages 109 – 117, 1999.

[7] The AspectJ Team. *The AspectJ(TM) Programming Guide.* Xerox Corporation., 2001.

[8] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.

[9] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. OOSPLA*, 1996.