Formalization and Evolution of Collaboration-Based Object-Oriented Methodology

By Nguyen Truong Thang

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Takuya Katayama

September, 2002

Formalization and Evolution of Collaboration-Based Object-Oriented Methodology

By Nguyen Truong Thang (010203)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Takuya Katayama

and approved by
Professor Takuya Katayama
Associate Professor Katsuhiko Gondow
Professor Koichiro Ochimizu

August, 2002 (Submitted)

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview of Software Evolution Problem

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. It ranges from the very practical software maintenance problem to the design of sophisticated software which can adapt its behavior autonomously according to changes.

The significant amount of work has been done so far, however, software evolution problem is still a challenge. It is not only due to the inherent difficulty of changing complex softwares, but also it comes from the fact that evolution activities are not performed properly. Aiming at establishing a sound and scientific basis for software evolution, a general framework of software evolution has been proposed. It is based on the concepts of evolutionary domain and evolutionary development process.

The evolutionary domain and evolutionary development process are parts of a general software framework. This framework assumes a rather ideal software development process. The core of this framework is a transparent mapping from specification domain to program (implementation) domain. Each specification is mapped to an unique program [1]. Therefore, if the specification and program domains behave in similar manners, the change in specification will then result in a very similar change in its program. We call this property as *transparency* between specification and program. In reality, these two domains usually behave in that way as transparency property has been observed in most well-designed systems. The evolution job is then reduced to manage the correspondences between specifications and their counterparts in program domain.

Another assumption in this framework is about program *composition simplicity.* If a system can be specified in term of the composition from several specification fragments, its implementation can be regarded as a composition of program fragments respectively associated with the initial specification fragments. As system evolves, some specification

[1]In evolutionary development process, it is assumed that the mapping between specification and program sets is one-to-one. That assumption is only for formalization simplicity purpose as there could be more than one program satisfying a given specification. Fortunately, it is not difficult to extend the model for dealing with one-to-many mapping.

fragments are either added to or removed from the system. Due to transparency property assumption, program fragments of those specifications are composed or removed from the original implementation according to the specification domain. Moreover, if a single specification fragment changes its internal structure during evolution process, the change does not propagate to other parts as long as the interfaces between them are kept and the implementation change within the fragment compromises the interface. Fortunately, that is usually the case.

The above software evolution framework may rely on some ideal assumptions. However, its evolutionary software development process is a good guide for practical software evolving tasks. In fact, it is really effective in case software specification of a system is regarded as a composition of rather orthogonal goals. The whole system is simply a composition of several separate components. And the composition step is simple and dynamic.

In brief, ideal software evolution framework relies on two properties: transparency and composition simplicity.

1. System is specified in terms of several independent concepts or goals.
2. There exists a scheme to maintain the transparency between concept domain and program domain.
3. System implementation is regarded as composition of program components implementing above goals.

Software development and evolution in this way are guaranteed to cost little as the process consists of two main steps: concept-to-program mapping and programs composition. The first is certainly simple due to transparency property, while the second is ensured by composition simplicity.

Any software paradigm satisfying these two properties are very promising in software development and evolution. The paradigm we investigate partially in this thesis seems to handle those two properties well. It is *Aspect-Oriented Software Development.* We investigate this paradigm partially because we only consider a special case of this approach. The software design is based on system partition according to collaborations. The aspects in system are expressed by collaborations.

1.2 Aspect-Oriented Software Development - AOSD

Due to its emphasis on the identity associated with the object, the classical design of object-oriented (OO) methodology is trapped in the so-called "tyranny of the dominant decomposition". It permits the separation and encapsulation of classes only. Other concepts, goals of the software are not encapsulated and they are scattered across the classes. The proposed paradigm called Aspect-Oriented Software Development (AOSD) tries to overcome the shortcoming in traditional OO approach. This paradigm is constructed on top of OO technology. In some perspective, it is an extension of current OO methodology. In AOSD community, there are two major research streams which differ on the way dealing with aspect granularity. Both streams are rooted at the so-called *subject-oriented programming* (SOP). Its recent derivation *multi-dimensional separation of concerns* (MDSOC) is the first of those two mentioned streams. The other trend *aspect-oriented programming* (AOP), despite of deriving from SOP much of its characteristics, takes a different approach in defining and composing concerns. Comparing with MDSOC, AOP is substantially different from SOP.

1.2.1 SOP and MDSOC

In SOP model, each *subject* is an object-oriented program or program fragment that models its domain in its own subjective way. A system, instead of focusing on participating classes, is structured from many subjects. Each subject can be implemented in a relatively independent manner. And any two subjects are loosely coupled. Once the implementation of subjects are completed, a composition designer can use a compositor program to compose subjects into larger subjects, and eventually the whole system. During composing process, corresponding classes in different subjects are merged in the manner that their methods, data members can mix or cancel each other out depending on the composing decisions. That composition process repeats for all subject components until entire system is constructed. The final system then behaves as a combination of member subjects.

The recent enhancement of SOP leads to a new model of MDSOC. *Concern*, *hyperslice* and *hypermodule* are the central concepts of MDSOC. The concept of concern is wider than that of subject. The idea behind the separation of concerns is to identify *concerns* of importance, and seek to localize units representing concepts associated with each concern into a module. Common dimensions of concern are object (leading to data abstraction) and function (leading to functional decomposition). Other concerns may be feature (both functional and non-functional), role (collaboration) etc. A hyperslice is a set of conventional modules to encapsulate concerns in dimensions other than the dominant one. Hyperslices may overlap in the sense that a given unit may occur in multiple hyperslices. A system is written as a collection of hyperslices, thereby separating all the concerns of importance in that system. To some extent, hyperslice in MDSOC and subject in SOP are very similar. A hypermodule is a set of hyperslices, together with a composition rule that specifies how those must be composed to form a single, new hyperslice that synthesizes and integrates their units.

1.2.2 Aspect-Oriented Programming: AOP

Though inheriting much of its merit from SOP, AOP takes a lower level approach to separating concerns. In AOP, there are only two main dimensions of concerns: dominant class and aspect. Aspects encapsulate all concerns but classes. They look and behave much like classes. AOP's top job is to weave aspects in aspect dimension into classes. In this sense, each aspect crosscuts all participating classes. In SOP model (eg. Hyper/J), primitive units are data members and methods. Hence, weavable units are only classes. In contrast, AOP allows a much more powerful programming mechanism in controlling execution flow. The key concept in AOP is *join point* at which some interesting event will occur. A join point could then be nested at any depth in a class's function call. An aspect crosscuts a class at a join point. A *point cut* defines a set of related join points in many classes. Hence, to our view, a point cut is essentially similar to a concern in MDSOC. AOP attempts on aspects associated with dynamic behavior. On the other hand, MDSOC focuses on static structure of system composed from many aspects.

The difference between aspect and concern, in my opinion, are in their granularity. Concerns are usually defined in top-down approach and hence, they are coarse. On the contrary, aspects are associated with very small join points during thread execution. Therefore, aspects are, in general, finer than concerns. In a concern, there could be several aspects defined on special events during concerns execution. I think, concern concepts are more goal- or feature-oriented, while aspect concepts are biased toward events during execution.

1.3 Purpose and Scope of This Research

AOSD is a promising approach as it seems to satisfy two essential properties: transparency and composition simplicity as discussed in Section 1.1. As a result, software evolution in this paradigm is inherently easier compared with many traditional approaches such as OO.

Due to its pre-mature state and great scope, I do not intend to investigate the evolution process of AOSD paradigm in the most general form. Instead, we take a special case, namely collaboration-based software, for investigating under general software evolution framework. The collaborative system is initially formalized for adapting to *evolutionary specification domain*. This formal model needs to address both static structure as well as dynamic behavior of system specification. After that, as such a formal system evolves, we are interested in how the changes in specification affect implementation part. That is the mapping task of *evolutionary development process*. This step is used to clarify that collaboration-based approach can map easily the change in specification to implementation (i.e. transparency). Further, program fragments in this method can be composed in a relatively simple manner (i.e. composition simplicity). In general, evolutionary development process is a mapping between specification and program domains. Chapters 3, 4 attempt to formalize role-based static and dynamic behaviors. They are all on the specification side. Based on these formal definitions, an evolutionary domain on role-based system specification is derived. On the program side, as program is dependent on the underlying languages, the real formalization for evolutionary program domain, if any, is different between languages. In this thesis, the formalization of evolutionary domain on program side is not presented in details. Its existence is assumed independent from programming languages. By assuming the existence of evolutionary program domain regardless programming languages, the evolutionary development process is mentioned in Chapter 5. This chapter presents some basic mapping framework from specification to codes written in two selected languages, namely C++ and Java. Their selection are rooted at their popularity and above all, their program fragment composition mechanisms. Those mechanisms are at two extremes. In case of C++, it directly facilitate mixin layer - an elegant way to represent collaboration, while Java does not. However, Java relies on a powerful AOSD programming paradigm to compose collaboration layers.

The above clarification, in essence, confirms that collaboration-based approach possesses two important properties: transparency and composition simplicity. As a result, the thesis partially confirms the usefulness of AOSD software paradigm, especially collaboration-based designs, in developing and evolving softwares.

The background and basic explanations are shown in Chapter 2. This part is a stepping-stone for later arguments in Chapters 3, 4. Finally, we discuss some related work and future direction of this research at Chapter 6.

Chapter 2

Background

2.1 Evolutionary Domain - An Evolution Software Framework

2.1.1 Software Evolution Problem

Let S and P be sets of all the specifications and programs respectively which will appear in the evolution problem under consideration. The expression $p \vdash s$ denotes the fact that a program $p \in P$ satisfies a specifications $s \in S$. p is usually constructed from s by applying a evolutionary software development process for p [1]. Or p could be derived mechanically by, say, a theorem proving techniques if the specification is given formally enough.

Software evolution problem can be formulated as follows. Let p be derived from s. When s changes to s', denoted as $s \Rightarrow s'$, p has to transform to the corresponding p'. That is,

$$\text{Suppose } p \vdash s \text{ and } s \Rightarrow s'$$
$$\text{Find } p'\text{: } p \Rightarrow p' \text{ and } p' \vdash s'$$

Of course, the evolved program p' has to be constructed reusing as much part of p as possible.

2.1.2 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

[1]The evolutionary development process is shown later. In essence, it is a mapping between specification and program domains, namely: $F : S \to P$

$s \sqsubseteq_S s'$ 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 $\sqsubseteq$ [2].

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. We suppose 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* [3].

2. We assume one more property of S. Two operators *difference* $\ominus$ and *composition* $\oplus$ are introduced with a set of *tags* A_S associated with S.

$$\begin{array}{l} \ominus : S \times S \to 2^{(A_S \times S)} \\ \oplus : S \times 2^{(A_S \times S)} \to S \end{array}$$

where 2^T is the set of subsets of T. The operators are implicitly assumed satisfying:

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

In the above, we require that if the specification s' is an evolution of s, then (i) we are able to find their difference $s' \ominus s$ as a tuple of specifications and (ii) adding this difference back to s gives s'.

We call such a set S an *evolutionary domain.* That is, the evolutionary domain is defined as:

$$(S, \sqsubseteq, \ominus, \oplus)$$

satisfying the above properties 1 and 2. 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.

2.1.3 Evolutionary Development Process

Suppose the sets of specifications and programs are formulated as evolutionary domain D_S and D_P respectively, and consider evolutionary development process between them.

$$D_S = (S, \sqsubseteq, \ominus, \oplus) \text{ and } D_P = (P, \sqsubseteq, \ominus, \oplus)$$

[2]In fact, the $\sqsubseteq$ relations in both domains are different. Both domains require a separate definition to each of those. However, this thesis focuses only on specification side. Intuitively, there is an assumption that evolutionary domain on program side does exist and its structure is similar to that on specification side.

[3]A lower semi-lattice structure is formed over a set if there exists an order relation over that set. That relation must be reflective, asymmetric and transitive. In addition, for any two items in that set, there always exists a greatest lower bound of two. Finally, that set should include a minimum item which is less than any other item in the set.

We assume here for the sake of simplicity, the program development process could be represented by a mapping F between the above evolutionary domain, and assume that a unique program $F(s)$ is obtained from the specification $s \in S$.

$$F : S \to P$$

Suppose $s \sqsubseteq s'$. An evolutionary development process F should possess the following four properties:

- Realizability: For any $s \in S, F(s) \in P$.
- Monotonicity: $F(s) \sqsubseteq F(s')$
- Incrementality: $F(s \oplus \Delta s) = F(s) \oplus G(F'(\Delta s))$ for some $G : 2^{(A_P \times P)} \to 2^{(A_P \times P)}$ and $F' : 2^{(A_S \times S)} \to 2^{(A_P \times P)}$.
- Locatability: $p = F(s)$ could be obtained as a substructure of $p' = F(s')$ and the be located in p' by a *locator* L as $p = L(s, s', p')$.

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. It characterizes the most desirable software evolution process in terms of evolutionary domain.

2.1.4 Evolution Types

Two types of evolution is considered. The first occurs when $s \sqsubseteq s'$, while in the other case, there is no evolution relation between s and s'. Instead, s and s' have a common specification to be evolved from, i.e. $t \sqsubseteq s$ and $t \sqsubseteq s'$ for some t.

In the first evolution type, the evolutionary development process is executed in the following manner:

1. Make the difference $\Delta s = s' \ominus s$
2. Construct incremental program fragment by $\Delta p = G(F'(\Delta s))$
3. Merge Δp with p to create p': $p' = p \oplus \Delta p$.

In the second evolution type, the evolutionary development process is executed in the following manner:

1. Extract the common specification t between s and s'. The most desirable t will be the greatest lower bound of them: $t = s \sqcap s'$.
2. According to the degeneration of specification s to t, the program $r = F(t)$ is needed. By the locatability of F, $r = L(t, s, p)$.
3. Evolve r to p' corresponding to the specification evolution t to s' as in the first scheme.

Step 1 is simply the phase to find the common specification between two unrelated specifications. The lower bound could be the intersection of two collaboration sets. The step 2 requires to reverse the usual evolution direction. The last step is then interpreted in the same manner as the first evolution type.

2.2 Collaboration-Based Software Design - A Special Case of AOSD paradigm

This is a design aiming to break a system functionality into several orthogonal layers [4] of object collaboration. Within each outer mixin, there are a number of objects participating with some role in the collaboration. Each object's role is called inner mixin. As a collaboration is a software concept, it is quite capable to apply the argument of AOSD to this design. In other words, collaboration-based design is only a special software design in AOSD paradigm. Hereafter the terms of collaboration-based design, role-based design are used interchangeably.

2.2.1 Role-Based and Layered Designs

A key objective in designing reusable software modules is to encapsulate within each module a single and mostly orthogonal aspect of application design. One view on the system for such orthogonal aspects is based on collaborations. Each collaboration consists of some classes and the interaction between them. The collaborations are very much independent of each other. The systems under such design are broken down into several orthogonal collaborations. Hence, an object-oriented application simply consists of a set of classes and a set of collaborations. Each application class encapsulates several roles where each role represents that class under one of the collaborations. In other words, a collaboration is a cooperating suite of roles corresponding to participating classes.

If viewed in two-dimensional coordinate of object classes and collaborations axes as in Figure 2.1, the design is necessarily constructed in a layered manner.

This type of design eases effort in implementing the software. Each layer can be implemented in a relatively independent manner. Those layers are then composed together for a synthesized layer and eventually, a complete system. The development process has shifted from traditional construction of OO classes to layers. Each layer encapsulates collaborative protocol between objects in that collaboration. Change in the protocol only results in local modification in that layer. In this design style, concerns are separated. As a result, softwares built from this style are quite simple to maintain and evolve.

[4]As later shown, these layers are outer mixins during implementation phase.

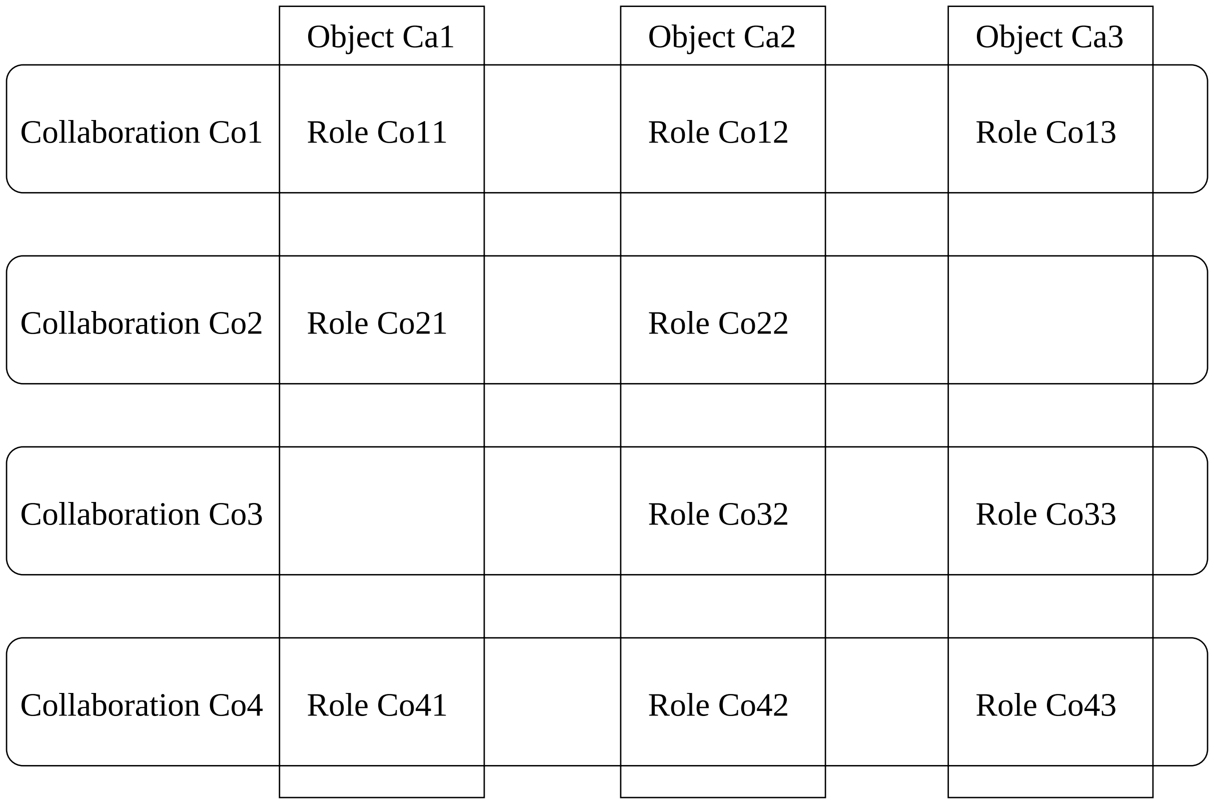

Figure 2.1: Example of collaboration decomposition. Round rectangles represent collaborations, while rectangles represent objects. Their intersections represent roles.

2.2.2 Implementation of Role-Based Designs by Mixins and Mixin Layers

There are several implementation techniques that transform the collaboration-based designs into the respective implementations. Many of them are rooted at techniques using mixin classes. Mixin is a facility compensating for some drawbacks in traditional single inheritance in object-oriented languages. Mixin is a particularly useful facility specifying a uniform extension of many family-related classes with one set of fields and methods (i.e. interface). Mixin is actually an abstract subclass. 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. The characteristic behind mixin programming is linearization of interfaces.

In the real world, collaborations between participating entities are generally stable in the sense that the interfaces between objects are rarely changed. The changes in these collaborations are usually in the form of either independent role evolution or the introduction of new collaboration into the application. The former is regarded as fine-grained evolution and the later is considered as large-grained evolution. Referring to Figure 2.1, the large granularity evolution could be expressed as an uniform extension of family of existing collaboration layers with one new collaboration. For example, the new collaboration Co_4 in Figure 2.1 could be introduced into the existing family of collaborations Co_1, Co_2, Co_3. More specifically, let G be the application class defining and implementing the collaborations Co_1, Co_2, Co_3 from their corresponding mixins and having been instantiated by some concrete classes. By defining a mixin which provides the implementation to the collaboration Co_4 and instantiating class G as the super-class of the mixin, we can theoretically create a new application class H which defines and implements all the collaborations Co_1, Co_2, Co_3, Co_4. Such a mixin is called *mixin layer.*

Mixin layer is implemented as an abstract subclass which in turn contains many inner classes to accommodate the scalability problem. In the Figure 2.1, those inner classes are Ca_1, Ca_2 and Ca_3. The collaboration composition process actually consists of two mixin extension processes. The first process occurs inside all object classes contributing to the whole application class. In the Figure 2.1, this extension process occurs in constituent objects Ca_1, Ca_2 and Ca_3. In case of object Ca_2, the mixin definition for collaboration Co_4 should be defined as an abstract sub-class. At the same time, it also provides the implementation for role Co_{42} of object Ca_2 with respect to collaboration Co_4. If the super-class is instantiated as the concrete class $Ca_{2(3)}$ handling up to collaboration Co_3 layer, we then create a new concrete class $Ca_{2(4)}$ accommodating all roles Co_{12}, Co_{22}, Co_{32} and Co_{42} (i.e. Co_{42} can handle to the bottom layer). This process happens inside the layer and hence is named as *inner mixin.* It is necessary to note that this composition process only deals with each object class separately. On the other hand, the mixin layer deals with the whole suite of classes Ca_1, Ca_2 and Ca_3 together and the composition process happens outside individual class. Therefore, to distinguish between these two types of mixin, the mixin layer is also named as *outer mixin.*

By utilizing these two types of inner and outer mixin extensions, the layered design is mapped to implementation in a relatively straightforward manner.

2.3 Concurrent System Modeling - Concurrent Regular Expressions

In general, the formal models proposed for specification and analysis of concurrent systems can be categorized roughly into two groups: *algebra-based* and *transition-based.* The algebra-based models specify all possible behaviors of concurrent systems by means of expressions that consist of algebraic operators and primitive behaviors. Examples of such models are path expression and *concurrent regular expressions.* Examples of transition-based models are finite state machines and *Petri nets.* Transition-based models have the advantage that they are graphical in nature, while algebraic systems promote hierarchical description and verification. A good formal model hence should support both styles. Concurrent regular expressions can be easily converted into equivalent Petri nets which possesses many analysis, verification and simulation techniques for concurrent systems. Conversely, any Petri net can be converted to a concurrent regular expression providing further insights into its language power.

All the existing models can also be classified according to their inherent expressive power. For example, a finite state machine is inherently less expressive than a Petri net. However, the gain in expressive power comes at the expense of analyzability. Analysis questions such as reachability are more computationally expensive for Petri nets than for finite-state machine. A complex system may consist of many components requiring varying expressive power. There should be a formal description technique supporting models of different expressive powers under a common framework. An example of such a description technique for syntax specification is Chomsky hierarchy of models based on grammars. A similar hierarchy is required for formal description of concurrent systems. The model of concurrent regular expressions provides such a hierarchy. A regular expression is less powerful than a unit expression, which, in turn is less expressive than a concurrent regular expression.

2.3.1 Concurrent Regular Expressions

We use languages as the means for defining behaviors of a concurrent system. A language is defined over an alphabet and, therefore, two languages consisting of the same strings but defined over different alphabet sets will be considered different. For example, null languages defined over Σ_1 and Σ_2 are considered different. We will generally indicate the set over which the language is defined, but may omit it if clear from the context.

The following describes operators required for definition of concurrent regular expressions (CRE).

Choice, concatenation, Kleene closure

These are the usual regular expression operators. *Choice* denoted by "+" is defined as follows. Let L_1 and L_2 be two languages defined over Σ_1 and Σ_2. Then

$$L_1 + L_2 = L_1 \cup L_2 \text{ defined over } \Sigma_1 \cup \Sigma_2.$$

This operator is useful for modeling the choice that a process or an agent can make.

The *concatenation* of two languages (denoted by ".") is defined based on usual concatenation of two strings as

$$L_1.L_2 = \{x_1x_2 | x_1 \in L_1; x_2 \in L_2\}.$$

This operator is useful to capture the notion of a sequence of action followed by another sequence. The *Kleene closure* of a set A is defined as

$$A^* = \bigcup_{i=0}^{\infty} A^i \text{ where } A^i = A.A^{i-1} \text{ and } A^1 = A.$$

This operator is useful for modeling the situations in which some sequence can be repeated any number of times.

Interleaving

To define concurrent operations, it is especially useful to be able to specify the interleaving of two sequences. Consider for example the behavior of two independent vending machines VM_1 and VM_2. The behavior of VM_1 may be defined as $(coin.choc)^*$ and the behavior of VM_2 as $(coin.coffee)^*$. Then the behavior of the entire system would be an interleaving of VM_1 and VM_2. With this motivation, we define an operator called interleaving, denoted by "$||$". Interleaving is formally defined as follows: (ϵ denotes the hidden event in Σ)

- $\forall a \in \Sigma : a||\epsilon = \epsilon||a = \{a\}$
- $\forall a, b \in \Sigma; s, t \in \Sigma^* : a.s||b.t = a.(s||b.t) \cup b.(a.s||t)$

Thus, $ab||ac = \{abac, aabc, aacb, acab\}$. This definition can be extended to interleaving between two sets in a natural way, i.e.

$$A||B = \{w|\exists s \in A, t \in B, w \in s||t\}$$

For example, consider two sets A and B as follows: $A = \{ab\}$ and $B = \{ba\}$, then $A||B = \{abba, abab, baab, baba\}$.

Note that similar to $A||B$, we also get a set $A||A = \{abab, aabb\}$. We denote $A||A$ by $A^{(2)}$. The parentheses in the exponent is used to distinguish from the traditional use of the exponent for concatenation, i.e. $A^2 = A.A$.

Interleaving satisfies the following properties:

1. $A||B = B||A$ (Commutativity).
2. $A||(B||C) = (A||B)||C$ (Associativity).
3. $A||\epsilon = A$ (Identity of $||$).
4. $A||\emptyset = \emptyset$ where $\emptyset$ is the empty set (Zero of $||$).
5. $(A + B)||C = (A||C) + (B||C)$ (Distributivity over $+$).

This operator, however, does not increase the modeling power of concurrent regular expression.

Alpha closure

Consider the behavior of people arriving at a supermarket. We assume that the population of people is infinite. If each person CUST is defined as $(enter.buy.leave)$, then the behavior of the entire population is defined as interleaving of any number of people. With this motivation, we define an analogue of a Kleene closure for the interleaving operation, α-closure of a set A, as follows: $A^\alpha = \bigcup_{i=0}^{\infty} A^i$.

Then if $\#(a, w)$ means the number of occurrences of the symbol a in the string w, the interpretation of CUST$^\alpha$ is as follows:

$$\text{CUST}^{\alpha} = \{w \mid \forall s \in pref(w),\ \#(enter, s) \geq \#(buy, s) \geq \#(leave, s);\ \#(enter, w) = \#(buy, w) = \#(leave, w)\}$$ where $pref(w)$ is a set of all prefixes to w.

Note the difference between Kleene closure and α-closure. The language shown above can not be accepted by a finite-state machine. Therefore, the α-closure can not be expressed by ordinary regular expression operators.

Intuitively, the α-closure allows modeling the behavior of an unbounded number of identical independent agents. It satisfies the following properties:

1. $A^{\alpha^{\alpha}} = A^{\alpha}$ (Idempotence).
2. $(A^*)^{\alpha} = A^{\alpha}$ (Absorption of $*$).
3. $(A + B)^{\alpha} = A^{\alpha} || B^{\alpha}$.

Synchronous composition

To provide synchronization between multiple systems, we define a composition operator denoted by [ ]. Intuitively, this operator ensures that all events that belong to two sets occur simultaneously. For example, consider a vending machine VM described by the expression $(coin.choc)^*$. If a customer CUST wants a piece of chocolate he must insert a coin. Thus, the event *coin* is shared between VM and CUST. The complete system is represented by VM [ ] CUST, which requires that any shared event must belong to both VM and CUST. Formally,

$$A[\ ]B = \{w \mid w/\Sigma_A \in A; w/\Sigma_B \in B\}$$

where w/S denotes the restriction of the string w to the symbols in S. For example, $acab/\{a, b\} = aab$ and $acab/\{b, c\} = cb$. If $A = \{ab\}$ and $B = \{ba\}$, then $A[\ ]B = \emptyset$ as there exists no string satisfying the order imposed by both A and B. Consider another set $C = \{ac\}$. Then $A[\ ]C = \{abc, acb\}$.

Many properties of [ ] are the same as those of the intersection of two sets. Indeed, if both operands have the same alphabet, the [ ] is identical to intersection.

1. $A[\ ] = A$ (Idempotence).
2. $A[\ ]B = B[\ ]A$ (Communtativity).
3. $A[\ ](B[\ ]C) = (A[\ ]B)[\ ]C$ (Associativity).
4. $A[\ ]NULL = NULL$ where $NULL = (\Sigma_A, \emptyset)$ (Zero of [ ]).
5. $A[\ ]MAX = A$ where $MAX = (\Sigma_A, \Sigma_A^*)$ (Identity of [ ]).
6. $A[\ ](B + C) = (A[\ ]B) + (A[\ ]C)$ (Distributivity over $+$).

Renaming

In many applications, it is useful to rename the event symbols of a process. Some examples are:

- *Hiding*: We may want some events to be internal to a process. We can do so by means of renaming these event symbols to ϵ.
- *Partial observation*: We may want to model the situation in which two symbols a and b look identical to the environment. In such cases, we may rename both of these symbols with a common name such as c.
- *Similar processes*: many systems often have "similar" processes. Instead of defining each one of them individually, we may define a generic process which is then transformed to the required process by renaming operator.

Let L_1 be a language defined over Σ_1. Let σ represent a function from Σ_1 to $\Sigma_2 \cup \{\epsilon\}$, i.e. $\sigma : \Sigma_1 \rightarrow \Sigma_2 \cup \{\epsilon\}$. Then $\sigma(L_1)$ is a language defined over $\sigma(\Sigma_1)$ as follows:

$$\sigma(L_1) = \{\sigma(s) \mid \ s \in L_1\}.$$

A renaming operator labels every symbol a in the string by $\sigma(a)$.

Definition of CRE

A concurrent regular expression is any expression consisting of symbols from a finite set Σ and `+`, `.`, `*`, `[]`, `||`, α, $\sigma()$ and ϵ, with certain constraints as summarized by the following definition.

- Any a that belongs to Σ is a regular expression (RE). A special symbol called ϵ is also a regular expression. If A and B are REs, then so are $A.B$ (concatenation), $A + B$ (or), A^* (Kleene closure).
- A regular expression is also a *unit* expression. If A and B are unit expressions, then so are $A||B$ (interleaving) and A^α (indefinite interleaving closure).
- A unit expression is also a concurrent regular expression (CRE). If A and B are CREs then so are $A||B$, A`[ ]`B (synchronous composition), and $\sigma(A)$ (renaming).

The intuitive idea behind this definition is as follows. We assume that a system has multiple (possibly infinite) agents. Each agent is assumed to have a finite number of states and, therefore, can be modeled by a regular set. These agents can execute independently (`||` and α) and a *unit expression* models a group of agents (possibly infinite) which do not interact with each other. The world is assumed to contain a finite number of these units which either execute independently (`||`) or interact by means of synchronous composition (`[ ]`).

2.3.2 Modeling of Concurrent Systems

This section gives a simple example of the use of concurrent regular examples in modeling concurrent systems. That is the traditional producer-consumer problem. It concerns shared data. The producer generates items which are kept in buffer. The consumer then takes these items from the buffer and consumes them. The solution requires that the consumer wait if no item exists in the buffer. The problem can be specified in concurrent regular expressions as follows:

producer :: $(produce\ \ putitem)^*$,
consumer :: $(getitem\ \ consumer)^*$,
buffer :: $(putitem\ \ getitem)^\alpha$,
system :: $producer$ `[ ]` $buffer$ `[ ]` $consumer$.

The buffer process ensures that the number of *getitem* is always less than or equal to the number of *putitem*. Note that if α is replaced by $*$ in the description of the buffer, the system will allow at most one outstanding *putitem*.

2.3.3 Relationship with Petri Nets

Languages of Petri nets

Definition 1 *A* Petri net N *is defined as a five-tuple* (P, T, I, O, μ_0), *where*

- P *is a finite set of places.*
- T *is a finite set of transitions such that* $P \cap T = \emptyset$.
- $I : T \to P^\infty$ *is the* input *function, a mapping from the transition to the bag of places.*
- $O : T \to P^\infty$ *is the* output *function, a mapping from the transition to bag of places.*
- μ_0, *the initial net marking, is a function from the set of places to the set of non-negative integers* $\mathcal{N}$, $\mu_0 : P \to \mathcal{N}$.

Definition 2 *A transition* $t_j \in T$ *in a Petri net* $N = (P, T, I, O, \mu)$ *is* enabled *if for all* $p_i \in P$, $\mu(p_i) \geq \#(p_i, I(t_j))$, *where* $\#(p_i, I(t_j))$ *represents multiplicity of the place* p_i *in the bag* $I(t_j)$.

Definition 3 *The next-state function* $\delta : Z_+^n \times T \to Z_+^n$ *for a Petri net* $N = (P, T, I, O, \mu)$, $|P| = n$, *with transition* $t_j \in T$ *is defined iff* t_j *is enabled. The next state is equal to* μ', *where*

$$\forall p_i \in P : \quad \mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$$

We can extend this function to a sequence of transitions as follows:

$\delta(\mu, t_j o) = \delta(\delta(\mu, t_j), o)$.
$\delta(\mu, \lambda) = \mu$ where λ represents the null sequence.

To define the language of a Petri net, we associate a set of symbols called alphabet Σ with a Petri net by means of a labeling function, $\sigma : T \to \Sigma$. A sequence of transition firings can be represented as a string of labels. let $F \subseteq P$ designate a particular subset of places as *final* places and we call a configuration μ final if

$$\forall p_i \in P - F, \mu(p_i) = 0.$$

That is, all tokens are in final places in a final configuration. If a sequence of transition firings takes the Petri net from its initial configuration to a final configuration, the string formed by the sequence of labels of these transitions is said to be accepted by the Petri net. The set of all such strings is called the language of the Petri net.

Definition 4 *The* language L *of a Petri net $N = (P, T, I, O, \mu)$ with alphabet Σ, labeling function σ and the set of final places F is defined as*

$$L = \{\sigma(\beta) \in \Sigma^* | \ \beta \in T^*, \mu_f = \delta(\mu_0, \beta) : \ \forall p \in P - F, \ \ \mu_f(p) = 0\}$$

Note that the notion of final configurations is different from the traditional definition of Petri net languages which typically use a *finite* set of final configurations This definition of final configurations may result in infinite number of them.

Decomposed Petri Nets (DPNs)

In a concurrent process analysis, it is better to separate big process into several smaller threads in which there is no synchronization between actions. The synchronization only exists between those threads. That is also the case in analyzing a Petri net. A Petri net is partitioned into multiple *units* which share all the transitions of the Petri net. Each unit contains some of the places of the original Petri net. Intuitively, the decomposition is such that the tokens with in a unit need to synchronize only with tokens in other units. Each unit is a generalization of finite-state machine. Formally, a DPN D is a tuple (T, U), where

- T is a finite set of symbols called *transition alphabets*,
- U is a set of units $(U_1, U_2, ..., U_n)$, where each unit is a five-tuple, i.e. $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$, where
  - P_i is a finite set of *places*.
  - C_i is an initial *configuration* which is a function from the set of places to non-negative integers $\mathcal{N}$ and a special symbol "*", i.e., $C_i : P_i \to (\mathcal{N} \cup \{*\})$ (the symbol "*" represents an unbounded number of tokens. A place which has * tokens is called a **-place*).
  - Σ_i is a finite set of *transition* labels such that $\Sigma_i \subseteq T$.

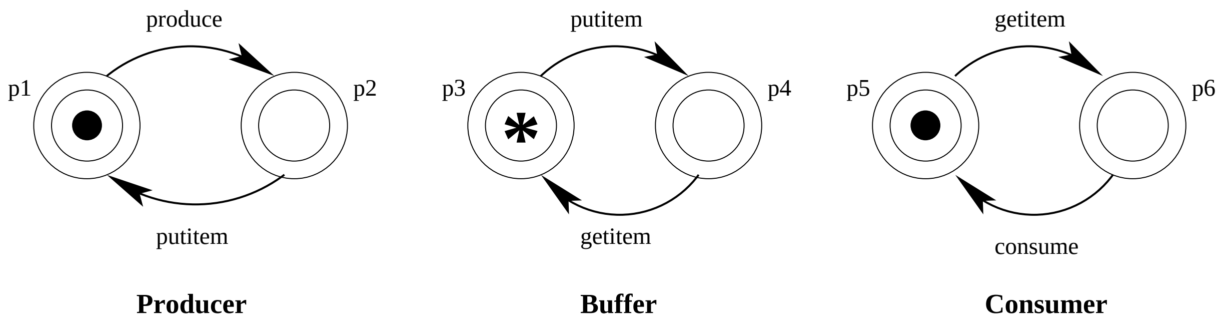

Figure 2.2: A DPN machine for producer-consumer problem.

- δ_i is a relation between $P_i \times \Sigma_i$ and P_i, i.e. $\delta_i \subseteq (P_i \times \Sigma_i) \times P_i$ (δ_i represents all transition arcs in the unit).
- F_i is a set of final places, $F_i \subseteq P_i$.

The configuration of a DPN can change when a transition is fired. A transition with label a is said to be *enabled* if for all units $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ such that $a \in \Sigma_i$, there exists a transition (p_k, a, p_j) with $C_i(p_k) \geq 1$. Informally, a transition a is enabled if all the units that have a transition labeled a, have at least one place with nonzero tokens and an outgoing edge labeled a. For example, in Figure 2.2, *getitem* is enabled only if both p_4 and p_5 have tokens. A transition may *fire* if it is enabled. The firing will result in a new marking C'_i for all participating units, and is defined by

$$C'_i(p_k) = C_i(p_k) - 1, \;\; C'_i(p_j) = C_i(p_j) + 1.$$

A *-place remains the same after addition or deletion of tokens.

As an example of a DPN machine, consider the producer consumer problem. The producer produces items which are kept in a buffer. The consumer takes these items from the buffer and consumes them. The solution requires that the consumer wait if no item exists in the buffer. The consumer can execute *getitem* only if there is a token in the place p_4. Note how the *-place is used to represent an unbounded number of buffers.

The definition of the language of a DPN is identical to that of a PN.

Relationship between CRE, Petri nets, Decomposed Petri nets

It has been proved that the expressive power of algebra-based CRE is equal to a Petri net, which in turn can be decomposed into a DPN. Those describes the same class of language, namely concurrent processes. The proof is based on the following theorems and lemmas.

1. Every Petri net can be decomposed, i.e. for every PN there exists a DPN such that they have the same language. Therefore, $PN \subseteq DPN$.
2. There exists an algorithm to derive a concurrent regular express that describes the set of strings accepted by a DPN. Hence, $DPN \subseteq CRE$.

3. There exists an algorithm to derive a Petri net that describes the set of strings described by a concurrent regular expression. Thus, $CRE \subseteq PN$.

From above arguments, we can conclude that those CRE, PN and DPN describes the same class of language. A system can be expressed in PN, DPN or CRE formalism and transformed to any other formalism. This transformation can be used for systems which are easier to specify in one formalism but easier to analyze in another.

By decomposing a PN into a DPN, the method has an advantage of separating concurrency and synchronization in PN. The resulting DPN and its equivalent CRE satisfy *modularity* properties and can be more easily used for specification of concurrent systems. The modularity of a system is considered as:

- A system is broken down into several units.
- Within each unit, there is no concurrency or synchronization. Hence, each unit executes in sequential style.
- Units execute concurrently in synchronous manner.

Chapter 3

Static Structure Modeling

3.1 Unifying Role Treatment

Inner mixin (or role) is treated as the primitive unit for system composition. That is because inner mixins are supplied as a whole indivisible units for any outer mixin (collaboration). In this model, an inner mixin is represented by a *mixin term*. The composition of two roles is also a mixin term but it is vertically composite. A collaboration is, on the other hand, simply a tuple of roles which is a horizontally composite term to distinguish from the former.

It is easy to see that class is a special kind of mixin. That argument can be justified if a mixin is created with the same structure as the class. In addition, its superclass is initialized with null type. The instantiated mixin and the initial class can then be regarded as equivalent. That argument helps to unify both mixin treatment and concrete OO class or type treatment. Hereafter, we only need to deal with mixins to represent both sets of mixins and classes. From that perspective, a layered system simply consists of several mixin layers. That system could instantiate to a concrete class by setting null type to all superclasses for top layer. That initializing step is simple and ignored in subsequent discussion about layered design [1].

Let $\mathcal{M}$ be the set of symbols representing a set of all mixin terms including ϵ. The symbol ϵ is a special one representing empty mixin. Note that as primitive in role-based design is role, the mixin term in $\mathcal{M}$ corresponds to role. We view a system as consisting of several classes. Each class encapsulates attributes and methods concerning with interactions with other classes to accomplish collaborations. This view is inherently object-oriented. The improvement over that OO view is our model reveals more about internal structure of each class rather than simply a black box. Our class structure is formed by composing several *class fragments*. Each class fragment is a role representing the intersection of a vertical class and a horizontal collaboration. Class fragments in our model are considered as primitive units. Each mixin term in $\mathcal{M}$ expresses a role of this kind, but not collaboration. In other words, $\mathcal{M}$ is a set of all inner mixins. Hereafter, the

[1]With that view, a system has one more mixin layer comparing with the original view. That mixin layer corresponds to the original top layer of concrete class/type which is now considered as a mixin layer.

mixin term is used to mean inner mixins. In case of outer mixins, they will be explicitly explained to avoid ambiguity.

From that perspective, a class is then formed by one or more mixin terms. Each mixin term represents the intersection of this class with a collaboration it takes part. In a typical collaborative system, its member classes does not participate in all collaborations. A class plays some role in a collaboration, but it does not in another. In order to unify our formal treatment of such a class with respect to all collaborations, in the latter case, we consider that class also contributes to the collaboration with a special empty role ϵ. Therefore, any class can join any collaboration. The importance lies in the role type it plays. If that role is empty, the class does not affect the collaboration by any means. The uniform treatment is shown in the following explanation. A role-based system consists of several collaborations. Let co be one of those collaborations. On the other hand, system contains a set of classes. And ca is one of them. Thus, the tuple (ca, co) represents the intersection part between class ca and collaboration co (i.e. class fragment). This tuple is then a mixin term in our model. It could be ϵ when class ca has no role in co.

In addition to $\mathcal{M}$, we assume the existence of an universal label set $\mathcal{L}$. This set contains all labels which are mapped one-to-one with roles in $\mathcal{M}$. A mixin term $m \in \mathcal{M}$ is uniquely associated with a label $l \in \mathcal{L}$. The mapping function is defined as: $\lambda : \mathcal{M} \rightarrow \mathcal{L}$ such as: $\forall m_1, m_2 \in \mathcal{M}$, $m_1 \neq m_2 : \lambda(m_1) \neq \lambda(m_2)$. In this sense, λ is called *role-labeling function*. This λ mapping function is used for tagging in composition and difference operators in Section 3.3. In some respect, the label represents the interface of the mixin while the mixin term is concerned with the actual implementation.

3.2 Basic Object-Oriented Type Theory

3.2.1 Evolution Relation in Object Types

Because our model is built on the assumption that inner mixins are basic units, inner mixins are equally treated as usual types in OO. Because mixin technique is built on top of object-oriented methodology, the basic operators will be related to object-oriented type theory. OO methodology is characterized by three features: encapsulation, inheritance and polymorphism. In dealing with type and class operation, only the first feature is retained. The last two features are dropped by a technique called *flattening*. In such a situation, there is no concept of class hierarchy in mixin domain. A class encapsulates all methods and variables, commonly called as *attributes*, which are directly reachable from the class. Let $\mathcal{C}$ be the universal set of such OO classes and types.

Definition 5 *Given two flattened classes c_1, c_2, from OO perspective, c_2 is said to be more evolved than c_1, denoted as $c_1 \sqsubseteq_R c_2$, if all attributes in c_1 are also encapsulated in c_2.*

And this relation is reflective, i.e. $\forall c \in \mathcal{C} : c \sqsubseteq_R c$. Certainly, the empty class ϵ is less evolved than any type as it contains no attribute, i.e. $\forall c \in \mathcal{C} : \epsilon \sqsubseteq_R c$. The reason of using subscript R will be explained in the next section.

3.2.2 Primitive Type Operators

In addition to evolution relation in object types, we need to define some basic composing or extracting operations between mixin terms (i.e. object types). Our ultimate formal model of role-based design relies on them as later discussed.

The basic operators are defined over two flattened classes. They are *difference* and *composition* type operators, denoted as $\ominus_R$ and $\oplus_R$ respectively. Operator $\ominus_R$ returns a new class having attributes in the first class but not in the second, provided the first class is more evolved than the second with respect to $\sqsubseteq_R$. On the other hand, given two classes whose attributes are disjoint, the semantic of $\oplus_R$ is to create a new class whose attributes are from the union set of attributes of two initial flattened classes.

For the primitive units, each unit (class fragment) corresponding to a role is expressed as a type in OO world. As role-based design has OO technology as its foundation, we cast those primitive units to simple classes or types in OO world. Therefore, the composing and subtracting operations between units are defined with respect to the OO domain.

Given two mixin terms m_1 and $m_2 \in \mathcal{M}$, the basic *composition* and *difference* operators between those are defined according to type theory in the object-oriented world.

Definition 6 *Let m_1, m_2 be two mixin terms, i.e. $m_1, m_2 \in \mathcal{M}$. The composition and difference operators between these two with respect to object type theory are defined as:*

- $\ominus_R : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$*, this difference operator returns the type difference between two flattened types (provided that $m_2 \sqsubseteq_R m_1$).*
- $\oplus_R : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$*, this composition operator returns the type formed by adding two flatten types together (given that m_1 and m_2 are disjoint in terms of attributes).*

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

3.3 Formal Static Structure Model of Collaboration-Based Design

This section gives the details of formal specification of a typical role-based system. Such a system contains some collaborations. In addition, from OO world, the system encapsulates some classes. Let $\mathcal{CA}$ and $\mathcal{CO}$ be universal sets of all class and collaboration labels. A typical system then has a set of classes $Ca \subseteq \mathcal{CA}$ and a set of collaborations $Co \subseteq \mathcal{CO}$. A function δ represents the set of roles available within the system at some specific class. As the order of collaborations is important to the whole system, we need to associate each collaboration with a set of collaborations it required for existence. This collaboration dependency mapping is realized by function ω.

From all the above arguments, the role-based system specification can be defined as in the following.

Definition 7 *The* static structure modeling *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.*
- *$\delta : \mathcal{CA} \times \mathcal{CO} \to \mathcal{M}$ is a role mapping of a class to a specific collaboration.*
- *$\omega : \mathcal{CO} \to 2^{\mathcal{CO}}$ is the dependency constraints between collaborations in the systems.*

The above δ function assigns a mixin term to a role of a class with respect to a collaboration. In addition, as in Section 3.1, there exists a mapping function λ associating a mixin term with its unique label.

Concerning with ω function, if a collaboration $o \in Co$ is independent of all other collaborations in Co, $\omega(o) = \emptyset$. Otherwise, a collaboration only depends on other collaborations existing in the system. That is, $\forall o \in Co : \ \omega(o) \subset Co$.

In essence, an empty collaboration is the same the an empty mixin ϵ because in term of OO type theory, they contain no attribute at all. So is an empty class. Thus we can assign ϵ to both collaboration or class which are empty. Since an empty collaboration or an empty class do not contribute anything to a system, the system properties are preserved if all empty collaborations and empty classes are extracted from such an above formal specification. This step is called *compacting. Horizontal compacting* step removes empty collaborations, while *vertical compacting* step cancels empty classes out. After a compacting process, the compact system is functionally equivalent to the initial.

Definition 8 *Let $s = (Ca, Co, \delta, \omega)$ be a role-based system specification. Specifications $s_H = (Ca_H, Co_H, \delta_H, \omega_H)$ and $s_V = (Ca_V, Co_V, \delta_V, \omega_V)$ are results after horizontal and vertical compacting processes on s.*

1. *In case of vertical compacting s_V,*
   - $Ca_V = Ca$,
   - $Co_V = \{o \mid (o \in Co) \wedge (\exists a \in Ca : \delta(a, o) \neq \epsilon)\}$,
   - $\delta_V = \{(a, o, r) \mid ((a, o, r) \in \omega) \wedge (o \in Co_H)\}$,
   - $\omega_V = \{(o, sc) \mid (o \in Co) \wedge (\forall c \in sc : (c \in \omega(o)) \wedge (\exists a \in Ca : \delta(a, c) \neq \epsilon))\}$.
2. *In case of horizontal compacting s_H,*
   - $Co_H = Co$,
   - $Ca_H = \{a \mid (a \in Ca) \wedge (\exists o \in Co : \delta(a, o) \neq \epsilon)\}$,
   - $\delta_H = \{(a, o, r) \mid ((a, o, r) \in \omega) \wedge (a \in Ca_V)\}$,
   - $\omega_H = \omega$.

The compacting step is used in optimizing system specifications. Hereafter, the specifications s in discussion have been already transformed into respective compacting forms. Let $\mathcal{S}_f$ be the universal set of all such formally defined s. In such a system, there is no redundant classes or collaborations.

3.4 Evolutionary Domain of Collaboration-Based Static Structure Specification

From the above formal definition, an evolution relationship $\sqsubseteq$ between static structure is defined to express how two systems are related to each other. In other words, this definition decides whether one system is more evolved than the other.

Definition 9 *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 if and only if all the followings 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 in terms of basic OO types. The fourth condition ensures that although each collaboration may evolve separately, the composition order between those collaborations in the former system are preserved in the latter. Composition dependency of the former are completely maintained in the latter.

It can be proved from the definitions above that the tuple $(\mathcal{S}_\int, \sqsubseteq)$ forms a lower semi-lattice. The proof is in Appendix A.1.

Lemma 10 *The tuple $(\mathcal{S}_\int, \sqsubseteq)$ forms a lower semi-lattice.*

After defining formal specification of static structure and evolution relation, i.e. domain $(\mathcal{S}_\int, \sqsubseteq)$, we need to define two operators for the complete evolutionary domain. They are $\ominus$ and $\oplus$ operators. Concerning with these operators, we need to define the tag set corresponding to *specification fragments* during evolution process. This is where the role-labeling function λ defined in Section 3.1 comes to play. We assigns the tag set A_S (specified in Chapter 2) to be exactly the product of $\mathcal{CA} \times \mathcal{CO} \times \mathcal{L}$ (defined in Section 3.3).

Comparing to the general definition in Chapter 2, the two operators have a slightly different forms as:

- $\ominus : \mathcal{S}_\int \times \mathcal{S}_\int \to 2^{\mathcal{CA} \times \mathcal{CO} \times \mathcal{L} \times \mathcal{M} \times 2^{\mathcal{CO}}}$.
- $\oplus : \mathcal{S}_\int \times\ 2^{\mathcal{CA} \times \mathcal{CO} \times \mathcal{L} \times \mathcal{M} \times 2^{\mathcal{CO}}} \to \mathcal{S}_\int$.

Note that the specification fragment is originally defined as a pair of label and specification. In this definition, the fragment is assigned to mixin term $\mathcal{M}$ and a set of required collaborations for the mixin instantiation.

Definition 11 *Let $s_1, s_2 \in \mathcal{S}_f$ and $s_1 \sqsubseteq s_2$. The difference between s_1 and s_2, i.e. $s_2 \ominus s_1 = \{(a, o, l, m, sc) \mid \forall a \in Ca_2, o \in Co_2 : m = \delta_2(a, o) \ominus_R \delta_1(a, o) \wedge l = \lambda(m) \wedge sc = \omega_2(o)\}$*

For the composition operator, let $s \in \mathcal{S}_f$, and $s = (Ca, Co, \delta, \omega)$. Given a set of mixin fragments $d = \{(ca, co, l, m, sc)\} \subset \mathcal{CA} \times \mathcal{CO} \times \mathcal{L} \times \mathcal{M} \times 2^{\mathcal{CO}}$. The composition $s \oplus d$ is defined by iteratively composing each member in d with s. In this composing process, there are two cases to consider. The first is more simple when this addition does not cause any new class and collaboration creation. On the contrary, the second case occurs when new class and collaboration is required. For simplicity, the following definition does not deal with a set of tuples (ca, co, l, m, sc). Instead, it defines the composition of s with a tuple only. In case of multiple pairs, we only need to iterate the same process until all members of the specification fragments set are done.

Definition 12 *Let $s \in \mathcal{S}_f$, and $s = (Ca, Co, \delta, \omega)$. The composition of s with specification set fragment $d = \{(ca, co, l, m, sc)\}$ is denoted as $s \oplus d = s' = (Ca', Co', \delta', \omega')$.*

1. *if $(ca \in Ca) \wedge (co \in Co) \wedge (\lambda(\delta(a, o)) = l)$ then*
    - $Ca' = Ca$,
    - $Co' = Co$,
    - $\delta'(x, y) = \delta(x, y) \qquad \forall x \in Ca, y \in Co \wedge ((x \neq ca) \vee (y \neq co))$,
    - $\delta'(a, o) = \delta(a, o) \oplus_R m$,
    - $\forall c \in Co, c \neq co : \omega'(c) = \omega(c)$,
    - $\omega'(co) = \omega(co) \cup sc$.
2. *if $(ca \notin Ca) \vee (co \notin Co)$ then*
    - $Ca' = Ca \cup \{ca\}$,
    - $Co' = Co \cup \{co\}$,
    - $\forall a \in Ca', o \in Co'$ *such that* $(a \neq ca) \vee (o \neq co) : \delta'(a, o) = \delta(a, o)$,
    - $\delta'(ca, co) = m$,
    - $\forall o \in Co', o \neq co : \omega'(o) = \omega(o)$,
    - $\omega'(co) = \omega(co) \cup sc$, *if* $co \in Co$,
    - $\omega'(co) = sc$, *otherwise.*

When inserting a new specification fragment (ca, co, l, m, sc) into the system, that fragment m only depends on existing collaborations. That is, $sc \subseteq Co$.

Theorem 13 *The set of $\mathcal{S}_f$ together with previously defined evolution relation $\sqsubseteq$, $\ominus$ and $\oplus$ operators forms a mixin evolutionary domain ($\mathcal{S}_f$, $\sqsubseteq$, $\ominus$, $\oplus$).*

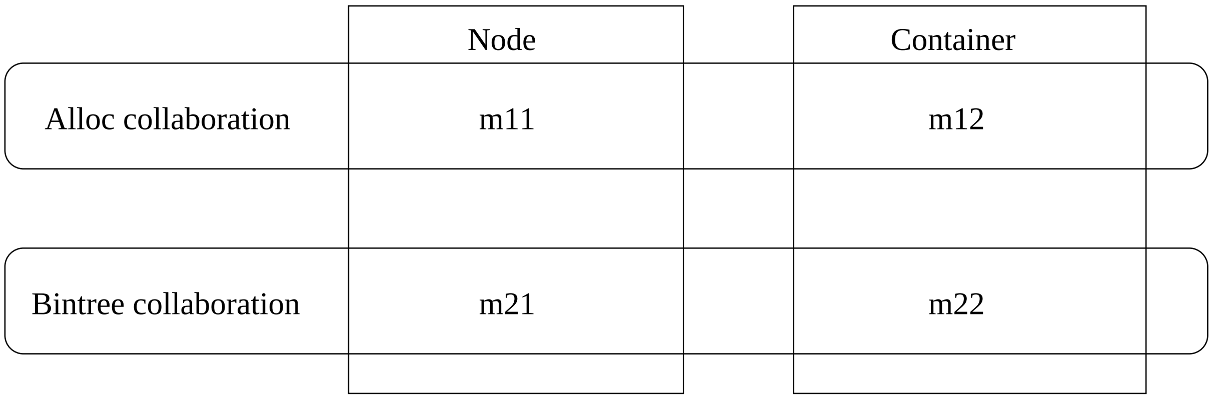

Figure 3.1: A simple binary tree data structure from the collaboration-based perspective.

This domain will be used as basis reference to analyze the evolution of role-based systems when new requirements come. The core of this evolution is a mapping between two evolutionary domains in specification and program sets. This mapping corresponds any specification (even a fragment) with a counterpart in program domain. The mapping is named *evolutionary development process*. This mixin specification domain is the common to role-based designs. However, their respective programs are different depending on the languages and underlying programming mechanism. This thesis selects two representative languages to deal with during software development process. They are C++ and Java [2]. The evolutionary program domain will be mapped with the specification evolutionary domain by an evolutionary development process. The evolutionary development process, namely $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{P}$, will be discussed in Chaper 5. Standing on the triangle of three formal definitions, namely specification domain, language-dependent program domain and specification-program mapping, software evolution can be performed in a consistent and systematic manner.

3.5 Some Examples

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. This data structure allocate the memory for a node into a container whose management scheme is like a binary tree. By analysis, this simple data structure consists of two collaborations: `Alloc` and `Bintree`. In addition, for each collaboration, there are only two actors: `Node` and `Container`. This system is shown in Figure 3.1. Note that the `Bintree` collaboration requires the existence of `Alloc` for its inserting, deleting and searching operations.

According to the Definition 7, this system $s = (Ca, Co, \delta, \omega)$ is formalized as follows:

[2]This thesis assumes the existence of evolutionary program domain independent of underlying languages. It is convincing that specification and program sets are very much in parallel. Let p_1, p_2 be implementations of s_1, s_2 respectively. If $s_1 \sqsubseteq_S s_2$, then p_2 is certainly more evolved than p_1, i.e. $p_1 \sqsubseteq_P p_2$. Formalization of program domain only gives insights into the semi-lattice structure of program set. It is left for future work.

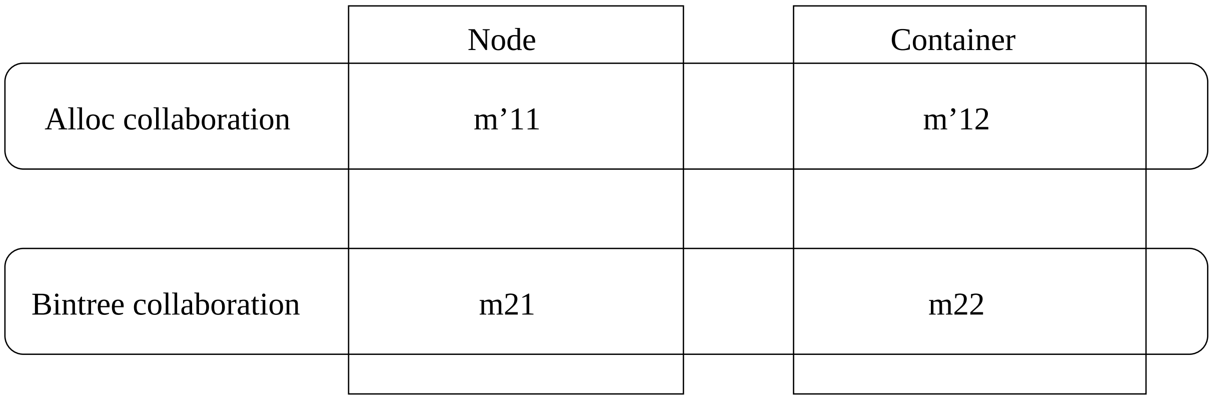

Figure 3.2: The same binary tree data structure but different implementation in Alloc layer: mixins m'_{11}, m'_{12} replace m_{11}, m_{12}.

- $Ca = \{Node, Cont\}$,
- $Co = \{Alloc, BTree\}$,
- $\delta = \{(Node, Alloc, m_{11}), (Cont, Alloc, m_{12}), (Node, BTree, m_{21}), (Cont, BTree, m_{22})\}$,
- $\omega = \{(BTree, \{Alloc\})\}$.

There is another implementation of the same data structure is shown in Figure 3.2. In this structure, mixins m'_{11}, m'_{12} inherit m_{11}, m_{12} respectively. Roles have evolved in this case.

This new system $s' = (Ca', Co', \delta', \omega')$ has:

- $Ca' = \{Node, Cont\}$,
- $Co' = \{Alloc, BTree\}$,
- $\delta' = \{(Node, Alloc, m'_{11}), (Cont, Alloc, m'_{12}), (Node, BTree, m_{21}), (Cont, BTree, m_{22})\}$,
- $\omega' = \{(BTree, \{Alloc\})\}$.

Furthermore, as the mixins inherit from their respective counter parts. We have:

- $m_{11} \sqsubseteq_R m'_{11}$ and $m'_{11} \ominus_R m_{11} = \Delta_{11}$.
- $m_{12} \sqsubseteq_R m'_{12}$ and $m'_{12} \ominus_R m_{12} = \Delta_{12}$.

By Definition 9, $s \sqsubseteq s'$. And their specification difference is expressed by $s' \ominus s = \{(Node, Alloc, \lambda(\Delta_{11}), \Delta_{11}, \emptyset), (Cont, Alloc, \lambda(\Delta_{12}), \Delta_{12}, \emptyset)\}$. As those class fragments Δs are in *Alloc* collaboration. Its dependency set is empty.

Adding back $\{(Node, Alloc, \lambda(\Delta_{11}), \Delta_{11}, \emptyset)\}$ to s, we do not change class and collaboration label sets Ca and Co, as well as ω. Instead, δ function becomes:

$$\delta = \{(Node, Alloc, m'_{11}), (Cont, Alloc, m_{12}), (Node, BTree, m_{21}), (Cont, BTree, m_{22})\}.$$

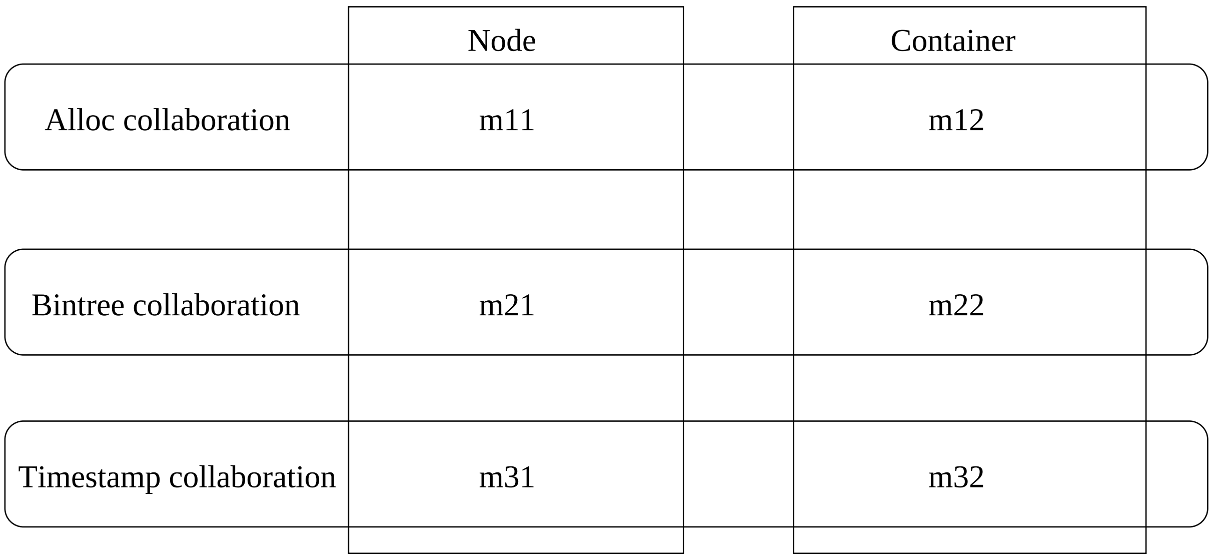

Figure 3.3: The data structure is extended with a Timestamp layer.

Adding again for Δ_{22} fragment, we get s'.

Suppose we have a pair of mixins defined as: m_{31} and m_{32}. They are parts of an implementation of `Timestamp` collaboration. Their associated labels could be expressed as $\lambda(m_{31}) = TStamp\_Node\_OtherInfo$ and $\lambda(m_{32}) = TStamp\_Cont\_OtherInfo$.

In addition, `Timestamp` layer requires the existence of both collaborations `Alloc, Bintree`. Hence, the dependency would be $sc = \{Alloc, BTree\}$.

Adding $\{(Node, TStamp, \lambda(m_{31}), m_{31}, sc)\}$ to s by pre-defined $\oplus$ would result in the creation of new collaboration $TStamp$. Inserting the mixin m_{32} then results in a new system $s'' = (Ca'', Co'', \delta'', \omega'')$ such as:

- $Ca'' = \{Node, Cont\}$,
- $Co'' = \{Alloc, BTree, TStamp\}$,
- $\delta'' = \{(Node, Alloc, m_{11}), (Cont, Alloc, m_{12}), (Node, BTree, m_{21}), (Cont, BTree, m_{22}), (Node, TStamp, m_{31}), (Cont, TStamp, m_{32})\}$,
- $\omega'' = \{(BTree, \{Alloc\}), (TStamp, \{Alloc, BTree\})\}$.

Chapter 4

Dynamic Behavior Modeling

4.1 Role-Based Dynamic Behavior as Composition of State Machines

The previous chapter is about the static structure of layered designs, namely how individual layers are structured from actor classes and how composite layer is constructed on those layers. In general, the static structure in that section deals with class diagram between actors in separate collaborations and in composite collaboration as a whole. The static structure does not reveal the actors' behavior during the run-time execution. The actors interact with each other to accomplish a goal by calling each other's method or sending/receiving messages. In traditional OO methodology, such behaviors are modeled by dynamic diagrams in OMT (Object Modeling Technique) or more recently UML (Unified Modeling Language). The core of those diagrams is state chart (or state-machine diagram) representing the change in an object state after some event has occurred.

We view a design as a set of classes. A collaboration consists of a set of class extensions (inner mixins) in a collaboration related to a common task. To make the dynamic model for a collaboration tractable, each actor class is described as a state machine, and hence, the mixin of an actor class in the lower layer extends an existing (base) state machine of all layers from top to the immediate above layer of the same actor. The extension is expressed by adding nodes, edges, and/or paths between states in the base machine. That extension is actually carried out by a graph merging process. The state diagram merging process can be viewed from two different perspectives. The first is a sequential composition of layers (via outer mixins), while the other emphasizes on parallel composition of extended actors (via inner mixins). This model in our opinion poses two apparent questions in role-based software development process. Firstly, for the software transparency, it is preferable to generate the source code directly from the inherited state diagrams for respective actors. This code generation process could be either manual or automatic. In fact, with the current status of tool support, there exists such a tool automatically generating codes from state charts. Most of such tools are related with the concept of domain specific languages (DSLs). Another question is about the consistency verification between this dynamic model and the static model discussed in previous chapter.

4.2 Sequential Collaboration Execution Modeling

As a first step, this section models the system in which collaborations do not interact to each other. Though they may execute concurrently, they are not under any effect from others. It is equivalent to a scenario in which collaborations run in sequential manner, one after the other's completion.

Like the approach taken in formalizing static structure, we first look at the primitive of our dynamic model. This primitive corresponds to dynamic state chart of an inner mixin (role). For a class, the roles are partitioned into layers according to collaborations. Each base or composed layer specifies interfaces, in terms of states, at which clients may attach extensions. The formal definition of interface is given below.

Definition 14 *A* sequential state machine *is a tuple $< S, \Sigma, \Delta, s_0, R, L >$, where:*

- *S is a set of states,*
- *Σ is the input alphabet,*
- *Δ is the output alphabet,*
- *$s_0 \in S$ is the initial state,*
- *$R \subseteq S \times PL(\Sigma) \times S$ is the transition relation (where $PL(\Sigma)$ denotes the set of propositional logic expressions over Σ),*
- *and $L : S \to 2^{\Delta}$ indicates which output symbols are true in each state.*

Definition 15 *A* base system *is a tuple $< M_1, ..., M_k >$ of state machines and a set of* interfaces. *We denote the elements of machine M_i as $< S_{Mi}, \Sigma_{Mi}, \Delta_{Mi}, s_{0_{Mi}}, R_{Mi}, L_{Mi} >$. An interface contains a sequence of pairs of states*

$$<< exit_1, reentry_1 >, ..., < exit_k, reentry_k >>.$$

Each $exit_i$ and $reentry_i$ is a state in machine M_i. State $exit_i$ is a state from which control can enter an extension machine, and $reentry_i$ is a state from which control returns to the base system. Interfaces also contain a set of properties and other information which are derived from the base system.

Definition 16 *An* extension *is a tuple $< E_1, ..., E_n >$ of state machines. Each E_i must induce a connected graph, must have a single initial state with in-degree zero, and must have a single state with out-degree zero. For each E_i, we refer to the initial state as in_i, and the state with out-degree zero as out_i. States in_i and out_i serve as place-holders for the states to which the collaboration will connect when composed with a based system. Neither of these states is in the domain of the labeling function L_i.*

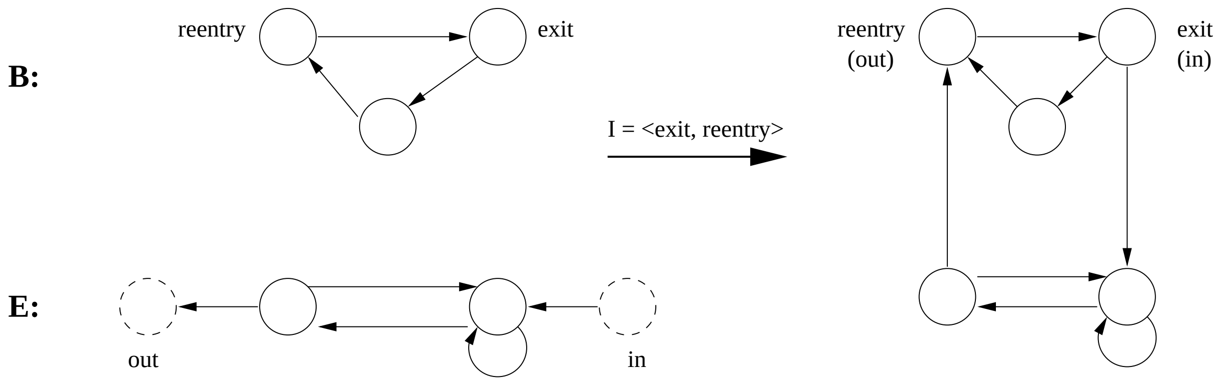

Figure 4.1: Composition of a base system B with an extension E via an interface.

Given a base system B, one of its interfaces I, and an extension E, we can form a new system by connecting the machines in E to those in B through the states in I in Figure 4.1. For simplicity, we assume that the number of state machines (actors) are the same in both B and E. To handle the case of role number mismatch, as discussed in static structure, we can uniform both collaborations as the same number of actors, while the actor in one collaboration not involving in another can be regarded as taking an empty role in that collaboration. Also the states in the constituent machines of base systems and extensions are distinct.

Definition 17 *Composing base system $B =< M_1, ..., M_k >$ and extension collaboration $E =< E_1, ..., E_k >$ via an interface $I =<< exit_1, reentry_1 >, ..., < exit_k, reentry_k >>$ yields a tuple $< C_1, ..., C_k >$ of state machines. Each state machine $C_i =< S_{Ci}, \Sigma_{Ci}, \Delta_{Ci}, s_{0_{Ci}}, R_{Ci}, L_{Ci} >$ is defined from $M_i =< S_{Mi}, \Sigma_{Mi}, \Delta_{Mi}, s_{0_{Mi}}, R_{Mi}, L_{Mi} >$ and its corresponding extension $E_i =< S_{Ei}, \Sigma_{Ei}, \Delta_{Ei}, s_{0_{Ei}}, R_{Ei}, L_{Ei} >$ as follows:*

- $S_{Ci} = S_{Mi} \cup E_{Mi} - \{in_i, out_i\}$
- $s_{0_{Ci}} = s_{0_{Mi}}$
- *R_{Ci} is formed by replacing all references to in_i and out_i in R_{Ei} with $exit_i$ and $reentry_i$ respectively, and unionizing it with R_{Mi}*
- *all other components are the union of the corresponding pieces from M_i and E_i.*

Definition 17 allows composed designs to serve as subsequent base systems by creating additional interfaces as necessary. This supports the notion of compound components that is fundamental in most definitions of component-based systems.

We will refer to the cross-product of $C_1, ..., C_k$ as the *global composed state machine*. In this perspective, this merging process is simply to connect two corresponding state charts M_i and E_i associated with the same actor i. The connecting point lies in the interface $I_i =< exit_i, reentry_i >$ of base system to be plugged with the client extension E_i with $< in_i, out_i >$. In general, the real work is about merging two graphs together with two matching pairs, namely $(exit_i, in_i)$ and $(reentry_i, out_i)$. However, the result of merging is not simply merging those four nodes into two and redirect all incoming/outgoing labeled edges to/from those new nodes.

Note that this simple state machine merging process assumes that collaboration threads execute in atomic fashion, i.e. there is no interrupt in the middle of collaboration execution sequence. One collaboration only starts after another has finished. In some respect, the collaborations are totally independent of each other. The event in one collaboration does not affect any others. And hence, the change of state in one layer is not observable from the other [1]. Such a system has no synchronized actions even though the layers actively execute in parallel. Therefore, all event sequences raised in the system, though they are initially in interleaving manner, can be reduced to an equivalent sequential event sequence in which all events relevant to a layer are concatenated in one consecutive sequence segment. The reduced form of event sequence is equivalent to the initial sequence in the sense that the behavior of each state machine does not change with respect to either sequences.

In reality, this assumption only true in some cases. For most of the cases, many collaborations between actors run in parallel. Because of possible interleaving actions due to concurrent collaboration execution, at any time during the execution sequence in base system, the actor can receive any call or message from the extension. That causes the base system to relinquish its control to the extension at the middle of its execution thread in the base machine. The proposed model is not strong enough to handle such a case. A more powerful model is required. That topic is discussed in detail in the next section.

4.3 Concurrent Collaboration Execution Modeling

4.3.1 Two views on Collaboration-based Designs

This section deals with the system consisting of many related collaborations. The relationship between layers are expressed by their dependency. One collaboration requires the existence of others. As one collaboration consists of possibly many actor state machines. When composing in vertical manner, a final actor state machine includes a base state machine and several extensions. The dependency between collaborations is actually caused by the dependency between base and extensions within some actor class. Some extension uses base or another extension in an upper layer for its associated operations. There are some synchronized actions. The change of state in the higher layer must be observed by the lower layer and this layer in turn will change its current state. In other words, a special event raised in one collaboration triggers the transitions of more than one base or extension state machines. Obviously, such a system is classified as a concurrent system with some synchronization in between. This type of system can not be modeled by the mechanism in Section 4.2. A better formal model to deal with such systems is required.

Looking at the Figure 4.2, we can see two different views on collaborative design. On the right is the view about parallel composition of actors. This is a traditional view of

[1]In fact, there is no need to do that.

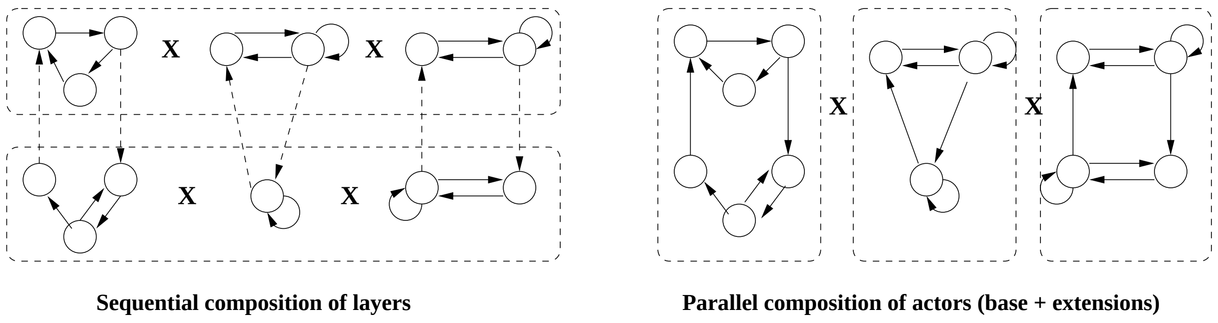
X
X
X
X
X
X
Sequential composition of layers
Parallel composition of actors (base + extensions)

Figure 4.2: Two views of a collaborative design: sequential composition of layers (left) versus parallel composition of extended actors (right)

OO approach, focusing on dominant dimension, i.e. class. Because of the encapsulation feature in OO world, this view does not care about the contents inside each actor class, but only about the interfaces for interaction between actors. Any class implementing this interface is quite capable to perform a role in this compound collaborative system. This type of software use is rather large-grained as a subtle variation in any collaboration protocol will lead to a change of the whole actor class. Clearly, this way is not desirable. Software developed in this way will certainly suffer in long term due to its weakness in maintainability, evolvability. In fact, this view is not what we rely on to formalize a model for layered designs. We want to know more about the internal structure of each actor. The second view considers the system as a sequential composition of collaboration layers. Those collaborations execute in a concurrent manner, quite possibly with synchronization between collaborations. If taking the actor class into the scope, we can see that within each class, there are a number of *class fragments* corresponding to the number of collaborations. Each fragment represents the role of that actor in a layer. Due to external behavior changes via events, messages, function calls etc raised by other actors, those fragments themselves change their "states"[2].

4.3.2 Concurrency and Synchronization in Dynamic Modeling

As a first impression about actor class's content, the state of each actor consists of several fragment states. Each class is then regarded as a composition of several fragments each of which behaves rather autonomously. During execution time, those fragments run concurrently in accord to their peers in other actors. In other words, modeling the dynamic behavior for each class now turns to modeling several fragments' behavior executed in a concurrent manner. Unlike the previous section assuming the total orthogonal between collaborations, our most general collaborative model has to take care of dependent constraints between layers. Those dependencies in layers express in terms of synchronizing between class fragments. Therefore, a required model for each class needs to handle both the concurrency and synchronization. Actually, only those two are most important

[2]Note that the term fragment state used here only involves with class attributes relevant to the role of the actor in a particular layer.

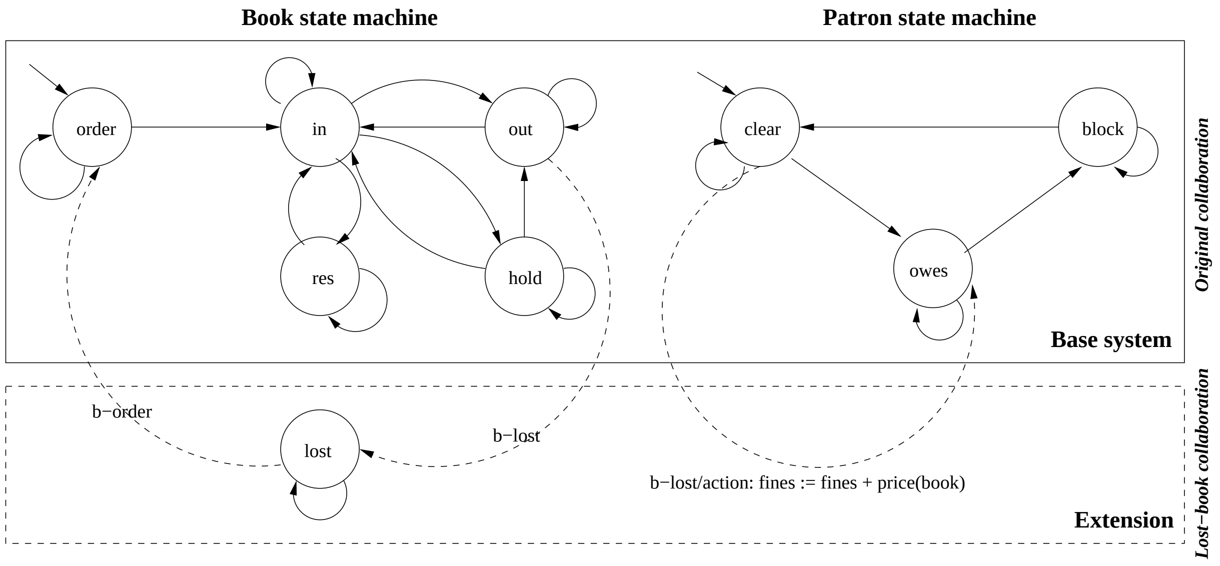

Figure 4.3: A library circulation system includes two actors book and patron. The dashed box encloses an extension handling lost book collaboration.

concerns. Intuitively, as in Chapter 2, Section 2.3, the language of this model type is similar to the one generated by a Petri net or algebraic expressions in Concurrent Regular Expression. The reason for choosing concurrent regular expressions is its explicitly comprehensible algebraic model. In addition, there is a proof of its equivalent language expressive power to Petri nets for concurrent systems. And Petri net is well-known for its abundant verification, analysis and simulation tools of concurrency and synchronization.

To illustrate our discussion to reach a powerful enough and comprehensible formal model, there is a simple example for such a composite collaborations in which there exists a dependency constraint. Figure 4.3 presents a layered design on a library circulation system. Books are in one of the states {*order, in, out, res(erve), hold*}; patrons are in one of the states {*clear, owes, block(ed)*} (corresponding to the level of fines on a patron's account). Labels on the transitions are omitted here, but support operations such as checking books out and putting books on hold. A later extension to the system adds facilities for handling lost books. This extension involves a new state and path in the book machine (to register a book as lost and possibly order it again) and a new path in the patron machine (to take into account the fines for losing books). The two machine extensions form a collaboration. Composing the Lost-Book collaboration with the original system through the dashed edges yields a new library system.

As an initial step, we only look at each actor class separately first. After formalizing the actor's dynamic behavior by state machine-based mechanism, we will proceed to model the whole system as a parallel composition of a number of actors.

As shown above, each actor class contains a number of fragments corresponding to roles of that actor in layers. Those fragments are inherently sequential but executes concurrently as in general, collaborations do not wait for each other's completion as in Section 4.2. However, between fragment execution threads are there a number of synchronous actions. Those actions only happen at some specific fragment states. Those states are “interesting” points in our model [3]. Their reaction to some specific external event will affect more than one state machine in base or extensions. Hereafter, all state machines in the base system or in the extension are commonly regarded as state machines for short. And the state of each class fragment, the state of a role are all commonly called *sub-state*. The behavior of the actor is better analyzed if the class is subdivided into a number of fragments along layer boundaries. In addition, an “interesting” state is duplicated to all state machines dependent on it. In the library example, considering the book actor, the *order* and *out* states are interesting states. They are duplicated to all concerning fragments. This step, in some aspects, is equivalent to the step that transforms a general Petri net (PN) to a decomposed Petri net (DPN) form. In fact, when we subdivide the class into fragments with some synchronized actions between them, the original PN for the class has been decomposed into DPN form in which each unit in DPN is the model for one class fragment [4]. This process is illustrated in Figure 4.4. In this figure, each fragment is inherently modeled by a state machine (or unit in DPN). The whole actor class is thus a DPN. It is easy to see that each class is modeled graphically by a DPN. Or that graphic-based DPN can be easily converted to an equivalent CRE for algebraic analysis purpose. The formal definition is quite similar to that of DPN as in Chapter 2, Section 2.3.

4.4 Concurrent Modeling with Decomposed Petri Net

In this section, we first try to model the dynamic behavior of each state machine separately. According to the previous section, each class consists of several state machines with respect to role fragments. State machines execute in concurrent (and possibly synchronized) manner. Referring to the definition of DPN in Chapter 2, Section 2.3.3, this model fits well to the language of a DPN. The state machine model is the basis where class and role-based system are formulated upon. Comparing with our approach in dealing with static structure mentioned previously in Chapter 3, it is very similar in the sense that the primitive elements in our model, regardless static or dynamic, are mapped to roles. Unlike static structure modeling, the dynamic behavior need to formalize that of each state machine. Whereas, in the former, mixin terms are presumably regarded as atomic. Their internal structures are not visible from outside. Furthermore, they are unified to classes or types in OO type theory.

After formalizing dynamic behavior of a typical state machine, we utilize that formalization to construct the compound state machines with greater granularity, namely classes

[3]This “interesting” point concept could be related to the hot-spot concept mentioned later in Chapter 5, Section 5.4.2. Further research are needed to clarify the relationship between these two concepts.

[4]Inside each class fragment (i.e. unit in DPN), there is no synchronization. The synchronization only exists between roles of the same actor. Our model has simplified role entities by assuming that there is no synchronization within a role. This assumption is generally true even in a more restrictive sense of no concurrency within a role. In reality, a role is often modeled separately by a regular state chart (i.e. a finite-state machine) whose power does not cover concurrency and synchronization.

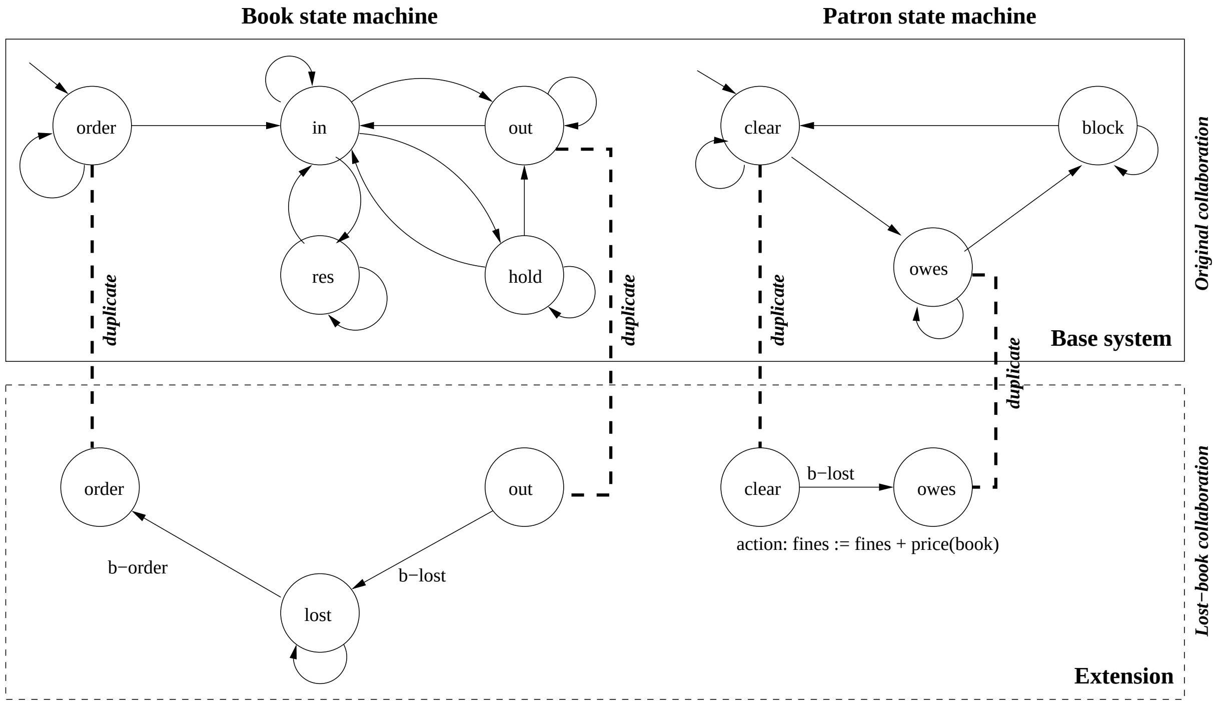

Figure 4.4: A library circulation system includes two actors book and patron. The dashed box encloses an extension handling lost book collaboration. Note that synchronized states are duplicated to all concerning layers.

and system as a whole. In the dynamic model of class, as the state machines are composed in some dependency ordering, the dependency mapping ω of formal static structure in Section 3.3 is applicable again. However, in dynamic diagram for classes, such kind of collaboration dependency is much more complicated and fine-grained if comparing with that of static case. Frequently, states in one collaboration are actually nodes of state chart in another. Such a state is called ' *"interesting"*. We also need a mapping on "interesting" states. This mapping takes a state in a machine as an input and maps it to a list of corresponding states in other layers which mean the same state after being duplicated in all concerning layers (referring to Figure 4.4). This is actually the enhancement from formal definition of DPN in Chapter 2.

4.4.1 A State Machine with DPN

Formal Model of a State Machine

In a layered design, each actor class's dynamic behavior is defined as a tuple (T, S, U, M), where

- T is a finite set of symbols called *transition alphabets*.
- S is a finite set of symbols called *state alphabets*.

- U is a ordered list of tuples each of which contains a state machine, namely U contains a set of units $(u_1, u_2, ..., u_n)$, where each unit is a 4-tuple, i.e. $u_i = (P_i, C_i, \Sigma_i, \Delta_i)$ [5].

- M is a mapping from an item in S to a subset of S, i.e. $M : S \to 2^S$. Note that M has to satisfy: $\forall s_1, s_2 \in S : s_2 \in M(s_1) \Rightarrow M(s_2) = M(s_1)$.

We may neglect initial configuration C_i in each unit if for each role, there is only sequential execution thread. The above definition is the most general case in which there is a possibility of concurrency inside each role. Comparing with the original definition of DPN, we note there are some substantial differences. Firstly, that is set of state symbols S to represent the alphabet of states (or places in DPN). Mapping M is used to represent the result of node duplication for a state in a layer. It groups nodes in different state machines together because they are all duplicated from the same node state in the topmost layer.

Each symbol in transition alphabet is a representation of an external event raised by other actor objects. On the other hand, the formal specification for each state machine is specified separately in units u_i. The formal dynamic behavior of an unit is shown below:

Definition 18 *Each unit u_i is a 4-tuple of $(P_i, C_i, \Sigma_i, \Delta_i)$ as in the following:*

- *P_i is a finite set of* places *or states; and $P_i \subseteq S$.*

- *C_i is an initial* configuration *which is a function from the set of places to non-negative integers and a special symbol "*", i.e. $C_i : P_i \to \mathcal{N} \cup \{*\}$ (the symbol "*", as previously defined, is unbounded number of tokens which is used for modeling concurrent and synchronized properties inside each role)* [6].

- *Σ_i is a finite set of* transition *labels, $\Sigma_i \subseteq T$.*

- *Δ_i is a relation between $P_i \times \Sigma_i$ and P_i, i.e. $\Delta_i \subseteq (P_i \times \Sigma_i) \times P_i$ (Δ_i represents all transition arcs in the unit).*

Let $\mathcal{U}$ be the universal set of all units defined as above definition. $\mathcal{U} = S \times (S \times \mathcal{N} \cup \{*\}) \times T \times ((S \times T) \times S)$. We can note the difference between this concurrent state machine by DPN with a sequential state machine defined in Definition 14. The main difference is the introduction of initial configuration C_i which is used to handle the concurrency. Transition label Σ_i and arc Δ_i combination is equal to the combination of input alphabet Σ, output alphabet Δ, transition R and output symbol L. P_i in this definition is similar to S in the former. Within an unit, there is no synchronization. Between several units, synchronization is expressed by transitions with the same name. As shown in Chapter 2, Section 2.3.3, that transition is enabled if the source states of that transition in all units have non-zero tokens. If it is enabled, it can fire to generate a new configuration C'_i such

[5]In a fully expanded form, the sets of state alphabets in units are disjoint, i.e. $i \neq j : P_i \cap P_j = \emptyset$.

[6]This term C_i could be dropped in case a role executes in sequential style. This configuration is only useful to deal with concurrency within role entity.

that the number of tokens in source state is decremented by 1, while at the other end of transition, that number is incremented by 1.

Given the above formal definition, we note that even though a full definition consists of 4-tuple, namely (T, S, U, M), the most important part lies in U. In general, a common context, T and S are simply representing the universal sets of transition ($\mathcal{T}$) and state ($\mathcal{S}$) labels respectively. Hence, we can uniform all specifications with the same pair of $(\mathcal{T}, \mathcal{S})$. In addition, we define a specification in a normal form if the mapping function M is an empty set. This normalizing process can be done by repeatedly reducing the cardinal of M. Each time, this process takes out all associated states by a single label and updates that label accordingly to the unit specifications. A general specification can be transformed into a normal form by replacing all the associated state labels in all units by the same state label. Certainly, after the transformation, the initial specification is completely equivalent to the normalized specification as they specify the same compound state machine. The normalized specification will have the form of $(\mathcal{T}, \mathcal{S},\ U', \emptyset) \equiv U'$. In subsequent discussion, we only consider the normal forms of classes, i.e. ignoring transition, state alphabets ($T = \mathcal{T}$ and $S = \mathcal{S}$), and setting mapping set M to empty set $\emptyset$. As a result, the initial 4-tuple is equivalent to the ordered list of units u_i's.

Before going into definition of evolution relationship between dynamic behaviors of classes, we first define the that relationship between units.

Definition 19 *An unit with* empty dynamic behavior *is a special 4-tuple of:* $(P, C, \Sigma, \Delta) = (\emptyset, \emptyset, \emptyset, \emptyset)$. *This is indeed the specification for dynamic behavior of an empty role. This special empty unit is represented by symbol* $\epsilon \in \mathcal{U}$.

Definition 20 *Let* u_1, u_2 *be two units according to the Definition 18 above, The unit* u_2 *is more evolved than* u_1*, i.e.* $u_1 \sqsubseteq_R u_2$*, if:*

- $P_1 \subseteq P_2$,
- $\forall p \in P_1 : C_1(p) \leq C_2(p)$, *with a presumption:* $\forall n \in \mathcal{N} : n <$ "$*''$,
- $\Sigma_1 \subseteq \Sigma_2$,
- $\Delta_1 \subseteq \Delta_2$.

Like Chapter 3, the subscript R is used in place of role since an unit defined here is a dynamic behavior of a role.

Primitive Operators on Units

In addition to evolution relation in units, we need to define some basic composing or extracting operations between these. Our ultimate formal model of role-based design relies on them as later discussed. Therefore, before going into the details of formal model for classes and role-based systems, there are some basic operators to be defined in advance. Together with definition of units, they serve as basic building blocks for subsequent class and system models.

The basic operators are defined over two units. They are *difference* and *composition* type operators, denoted as $\ominus_R$ and $\oplus_R$ respectively. The semantic of $\oplus_R$ is to create a new unit formed by the union of two state machines. On the other hand, $\ominus_R$ returns a new one having attributes (nodes, transition arcs) in the first unit but not in the second.

For the primitive operators, they are actually the operations of merging and subtracting two state charts.

Given two units u_1 and $u_2 \in \mathcal{U}$, the basic *composition* and *difference* operators between those should have the types:

- $\ominus_R : \mathcal{U} \times \mathcal{U} \to \mathcal{U}$.
- $\oplus_R : \mathcal{M} \times \mathcal{U} \to \mathcal{U}$.

Definition 21 *Let $u_1 = (P_1, C_1, \Sigma_1, \Delta_1)$ and $u_2 = (P_2, C_2, \Sigma_2, \Delta_2)$ be two units, i.e. $u_1, u_2 \in \mathcal{U}$. The composition and difference operators between these two are defined as:*

1. $u_1 \oplus_R u_2 = u_c = (P_c, C_c, \Sigma_c, \Delta_c)$
   - $P_c = P_1 \cup P_2$,
   - $\forall n \in P_1 \cap P_2 : C_c(n) = C_1(n) + C_2(n)$, *and a presumption: if* $(C_1(n) = *) \vee (C_2(n) = *)$, *then* $C_c(n) = *$,
   - $\forall n \in P_1 \setminus P_2 : C_c(n) = C_1(n)$,
   - $\forall n \in P_2 \setminus P_1 : C_c(n) = C_2(n)$,
   - $\Sigma_c = \Sigma_1 \cup \Sigma_2$,
   - $\Delta_c = \Delta_1 \cup \Delta_2$.
2. *if* $u_1 \sqsubseteq_R u_2$, $u_2 \ominus_R u_1 = u_d = (P_d, C_d, \Sigma_d, \Delta_d)$
   - $P_d = P_2$,
   - $\forall n \in P_1 \cap P_2 : C_d(n) = C_2(n) - C_1(n)$, *and a presumption: if* $(C_1(n) = *)$, *then* $C_c(n) = *$,
   - $\forall n \in P_2 \setminus P_1 : C_d(n) = C_2(n)$,
   - $\Sigma_d = \Sigma_2$,
   - $\Delta_d = \Delta_2$.

In the difference operator definition, if the number of tokens for the same node in both units are "*", we face an indeterminate state in the result. In that case, we assume that value is also "*", i.e. unbounded.

4.4.2 Dynamic Behavior Formalization of a Role-based System

After defining units and its primitive comparison relationship $\sqsubseteq_R$ and two basic operators $\oplus_R$ and $\ominus_R$, the formal specification of a role-based system is derived. Like static structure approach, a specification should include:

- a set of class labels Ca,
- a set of collaboration labels Co,
- a state machine mapping which corresponds a role of a class in a collaboration with a state chart, $\delta : \mathcal{CA} \times \mathcal{CO} \to \mathcal{U}$,
- dependency mapping between collaboration $\omega : \mathcal{CO} \to 2^{\mathcal{CO}}$.

For a general class definition, we may need a mapping M of state labels in several state machines of the same class. However, as shown in Section 4.4.1, we will duplicate that label to all corresponding places so that mapping M is an empty set $\emptyset$.

Definition 22 *The* dynamic behavior modeling *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.*
- *$\delta : \mathcal{CA} \times \mathcal{CO} \to \mathcal{U}$ is a role mapping of a class to a specific collaboration.*
- *$\omega : \mathcal{CO} \to 2^{\mathcal{CO}}$ is the dependency constraints between collaborations in the systems.*

The only difference between static and dynamic models are the substitution of unit set $\mathcal{U}$ in place of mixin set $\mathcal{M}$. In the same way, we need to define a *unit-labeling* function $\lambda : \mathcal{U} \to \mathcal{L}$ which returns the label associated with an unit. The reason to introduce λ function is the same for that of static structure modeling.

The set of role-based systems defined in the way above is denoted as $\mathcal{S}_\lceil$.

4.4.3 Evolutionary Domain of Role-based Dynamic Behavior Specification

Based on the normal form and evolution relationship between units, we derive the evolution relationship between systems in terms of dynamic behavior. The evolution relationship between elements in $\mathcal{S}_\lceil$ is defined as below:

Definition 23 *Let $s_1, s_2 \in \mathcal{S}_\lceil$ be two dynamic specifications, namely $s_1 = (Ca_1, Co_1, \delta_1, \omega_1)$ and $s_2 = (Ca_2, Co_2, \delta_2, \omega_2)$. s_2 is more evolved than s_1, denoted as $s_1 \sqsubseteq s_2$ if the followings are satisfied simultaneously:*

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

Similar to static structure formalization, the third condition confirms that all state machines (units) 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 order between those collaborations in the former system are preserved in the latter. Composition dependency of the former are completely maintained in the latter.

It can be proved from the definitions above that the tuple ($\mathcal{S}_\lceil$, $\sqsubseteq$) forms a lower semi-lattice. The proof is in Appendix A.2.

Lemma 24 *The tuple ($\mathcal{S}_\lceil$, $\sqsubseteq$) forms a lower semi-lattice.*

Now we go to the definition of composition and difference operators, namely $\oplus$ and $\ominus$, in the above semi-lattice structure. The basic operators in Section 4.4.1 are used in this circumstance.

Comparing to the general definition in Chapter 2, the two operators have a slightly different forms as:

- $\ominus : \mathcal{S}_\int \times \mathcal{S}_\int \rightarrow 2^{\mathcal{CA}\times\mathcal{CO}\times\mathcal{L}\times\mathcal{U}\times 2^{\mathcal{CO}}}$.
- $\oplus : \mathcal{S}_\int \times\ 2^{\mathcal{CA}\times\mathcal{CO}\times\mathcal{L}\times\mathcal{U}\times 2^{\mathcal{CO}}} \rightarrow S_s$.

Note that the specification fragment is originally defined as a pair of label and specification. In this definition, the fragment is assigned to mixin term $\mathcal{U}$ and a set of required collaborations for the mixin instantiation.

Definition 25 *Let $s_1, s_2 \in \mathcal{S}_\lceil$ and $s_1 \sqsubseteq s_2$. The difference between s_1 and s_2, i.e. $s_2 \ominus s_1$ $= \{(a,o,l,u,sc) \mid\ \forall a \in Ca_2, o \in Co_2 : u = \delta_2(a,o) \ominus_R \delta_1(a,o) \wedge l = \lambda(u) \wedge sc = \omega_2(o)\}$.*

For the composition operator, let $s \in \mathcal{S}_\lceil$, and $s = (Ca, Co, \delta, \omega)$. Given a set of units $d = \{(ca, co, l, u, sc)\} \subset \mathcal{CA} \times \mathcal{CO} \times \mathcal{L} \times \mathcal{U} \times 2^{\mathcal{CO}}$. The composition $s \oplus d$ is defined by iteratively composing each member in d with s. For simplicity, the following definition does not deal with a set of tuples (ca, co, l, u, sc). Instead, it defines the composition of s with a tuple only. In case of multiple pairs, we only need to iterate the same process until all members of the specification fragments set are done.

While inserting a specification fragment to a system, we may encounter two cases. The first is more simple when this addition does not cause any new class and collaboration creation. On the contrary, the second case occurs when new class and collaboration is required.

Definition 26 *Let $s \in \mathcal{S}_\lceil$, and $s = (Ca, Co, \delta, \omega)$. The composition of s with specification set fragment $d = \{(ca, co, l, u, sc)\}$ is denoted as $s \oplus d = s' = (Ca', Co', \delta', \omega')$.*

1. *if* $(ca \in Ca) \wedge (co \in Co) \wedge (\lambda(\delta(ca, co)) = l)$ *then*
   - $Ca' = Ca$,
   - $Co' = Co$,
   - $\delta'(x, y) = \delta(x, y) \quad \forall x \in Ca, y \in Co \wedge ((x \neq ca) \vee (y \neq co))$,
   - $\delta'(ca, co) = \delta(ca, co) \oplus_R u$,
   - $\forall c \in Co, c \neq co : \omega'(c) = \omega(c)$,
   - $\omega'(co) = \omega(co) \cup sc$.
2. *if* $(ca \notin Ca) \vee (co \notin Co)$ *then*
   - $Ca' = Ca \cup \{ca\}$,
   - $Co' = Co \cup \{co\}$,
   - $\forall a \in Ca', o \in Co'$ `such that` $(a \neq ca) \vee (o \neq co) : \delta'(a, o) = \delta(a, o)$,
   - $\delta'(ca, co) = u$,
   - $\forall o \in Co', o \neq co : \omega'(o) = \omega(o)$,
   - $\omega'(co) = \omega(co) \cup sc$, *if* $co \in Co$,
   - $\omega'(co) = sc$, *otherwise.*

An unit u only depends on existing layers in the system when (ca, co, l, u, sc) is added to the system. For the correct semantic of unit composition, we have, $sc \subseteq Co$.

The label of a state machine contains information about its class and collaboration interafaces. It is essential that the label is "compatible" with the interface at the position this fragment is plugged to. The naming function $\lambda : \mathcal{U} \rightarrow \mathcal{L}$ is already discussed in Section 4.4.2.

Theorem 27 *The set of $\mathcal{S}_\lceil$ together with above $\sqsubseteq$ evolution relation, $\ominus$ and $\oplus$ operators forms an evolutionary domain $(\mathcal{S}_\lceil, \sqsubseteq, \ominus, \oplus)$.*

This evolutionary domain of role-based dynamic behavior will serve as the basis of evolutionary development process acting over role-based designs.

4.5 Some Examples

This section focuses on ideas of formalization and evolution of system dynamic behavior so far. The example in this section is about a simple library system. Firstly, it handles only very basic operations of a typical library, namely borrowing/returning a book from/to a library. In such a library, we consider only two main actors: book and patron. Initially, the system does not support holding book. Later, this new feature is added to the system. The behavior of both classes are shown in Figure 4.5.

This base system has two state machines, namely $u_{11} = (P_{11}, C_{11}, \Sigma_{11}, \Delta_{11})$ and $u_{12} = (P_{12}, C_{12}, \Sigma_{12}, \Delta_{12})$ for book and patron respectively. u_{11} does not initially support holding book.

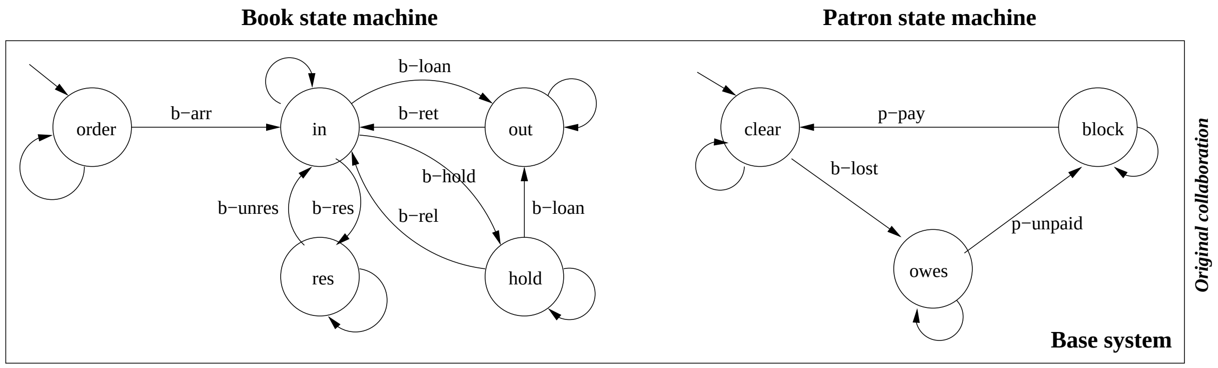

Figure 4.5: The state machines for basic library system.

- $P_{11} = \{order, in, out, res\}$,
- $C_{11} = \{(order, 1), (in, 0), (out, 0), (res, 0)\}$,
- $\Sigma_{11} = \{b\_arr, b\_loan, b\_ret, b\_res, b\_unres\}$,
- $\Delta_{11} = \{(order, b\_arr, in), (in, b\_loan, out), (out, b\_ret, in), (in, b\_res, res), (res, b\_unres, in)\}$.

In case of u_{12}, we have:

- $P_{12} = \{clear, block, owes\}$,
- $C_{12} = \{(clear, 1), (block, 0), (owes, 0)\}$,
- $\Sigma_{12} = \{b\_lost, b\_pay, b\_unpaid\}$,
- $\Delta_{12} = \{(clear, b\_lost, owes), (owes, b\_unpaid, block), (block, b\_pay, clear)\}$.

The place labels are obvious. For transition labels, there is a need to explain: $b\_arr, b\_loan, b\_ret, b\_res, b\_unres$ correspond to (book) arrival, loan, return, reserved and unreserved respectively. On the other hand, b_lost, b_pay, b_unpaid are labels of (book) lost, payment for the fine on time and overdue.

The initial system is specified as: $s = \{Ca, Co, \delta, \omega\}$ in which:

- $Ca = \{Book, Patron\}$,
- $Co = \{Base\}$,
- $\delta = \{(Book, Base, u_{11}), (Patron, Base, u_{12})\}$,
- $\omega = \emptyset$.

If the base system handles book-holding support, the state machine of this functionality can be expressed as: $u = (P, C, \Sigma, \Delta)$ where

- $P = \{in, hold\}$,

- $C = \{(in, 0), (hold, 0)\}$,
- $\Sigma = \{b\_hold, b\_rel\}$,
- $\Delta = \{(in, b\_hold, hold), (hold, b\_rel, in)\}$.

Transitions b_hold, b_rel represent the events of holding and releasing book. The label $l = \lambda(u)$ associating with state machine u would be $Base\_Book\_OtherInfo$. This unit does not require the existence of any other collaboration, namely $sc = \emptyset$. Adding the specification fragment $\{(Book, Base, l, u, sc)\}$ to s would results in $s' = s \oplus \{(Book, Base, l, u, sc)\} = (Ca', Co', \delta', \omega')$ where:

- $Ca' = Ca = \{Book, Patron\}$,
- $Co' = Co = \{Base\}$,
- $\delta' = \{(Book, Base, u'_{11}), (Patron, Base, u_{12})\}$,
- $\omega' = \omega = \emptyset$.

Here, $u'_{11} = u_{11} \oplus_R u = (P'_{11}, C'_{11}, \Sigma'_{11}, \Delta'_{11})$. By Definition 21, we should have:

- $P'_{11} = \{order, in, out, res, hold\}$,
- $C'_{11} = \{(order, 1), (in, 0), (out, 0), (res, 0), (hold, 0)\}$,
- $\Sigma'_{11} = \{b\_arr, b\_loan, b\_ret, b\_res, b\_unres, b\_hold, b\_rel\}$,
- $\Delta'_{11} = \{(order, b\_arr, in), (in, b\_loan, out), (out, b\_ret, in), (in, b\_res, res),$
  $(res, b\_unres, in), (in, b\_hold, hold), (hold, b\_rel, in)\}$.

Certainly, $u_{11} \sqsubseteq_R u'_{11}$ and hence, $s \sqsubseteq s'$.

Now, our system is extended with capability of charging user a fine in case he/she lost a book. In addition, once the book loss is notified, our system will automatically order to publisher a new copy of the book. This new extended system is shown in Figure 4.6.

This system $s'' = (Ca'', Co'', \delta'', \omega'')$ is formed by adding above s' with a specification fragment set of two elements $\{(Book, Extension, l_1, e_1, sc_1), (Patrol, Extension, l_2, e_2, sc_2)\}$ where e_1, e_2 corresponds to the extended behavior of $Book$ and $Patrol$ in $Extension$ collaboration respectively. We should have:

- $l_1 = \lambda(e_1) = Ext\_Book\_OtherInfo$,
- $l_2 = \lambda(e_2) = Ext\_Patrol\_OtherInfo$,
- $sc_1 = sc_2 = \{Base\}$.

e_1 and e_2 are extensions of base state machines, and formally expressed as:

1. $e_1 = (P_1, C_1, \Sigma_1, \Delta_1)$

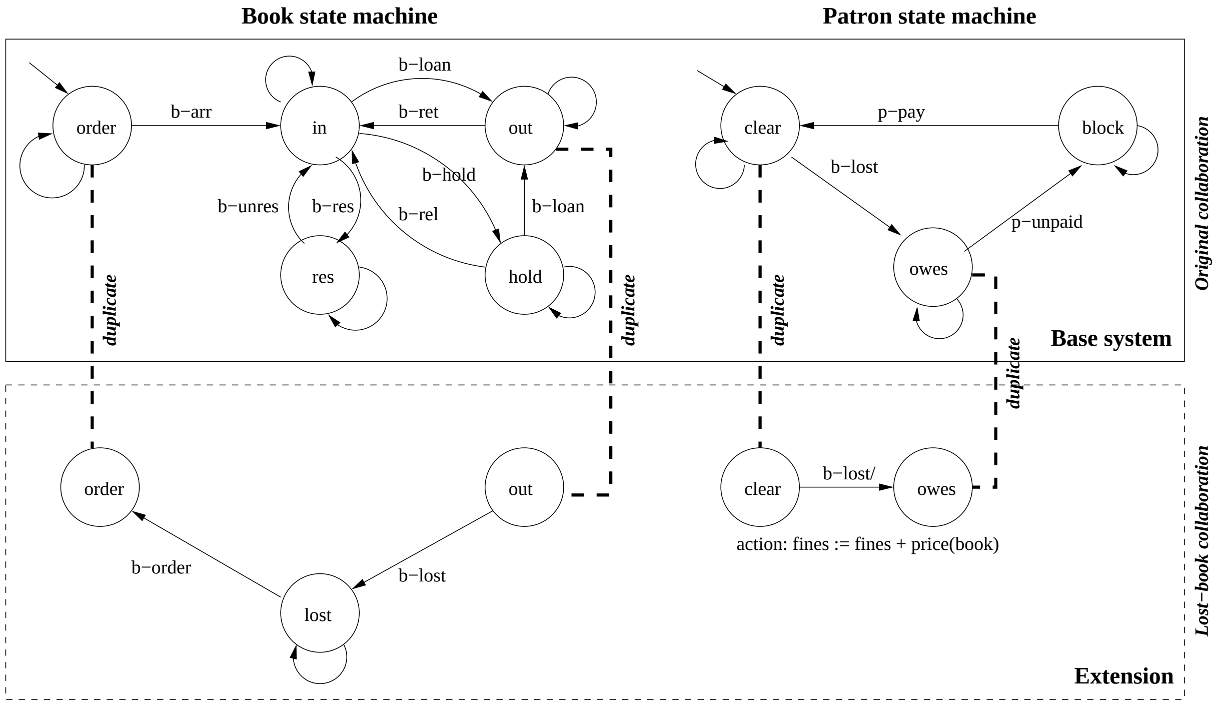

Figure 4.6: The state machines of extended library system.

- $P_1 = \{order, lost, out\}$,
- $C_1 = \{(order, 0), (lost, 0), (out, 0)\}$,
- $\Sigma_1 = \{b\_lost, b\_order\}$,
- $\Delta_1 = \{(out, b\_lost, lost), (lost, b\_order, order)\}$.

2. $e_2 = (P_2, C_2, \Sigma_2, \Delta_2)$

- $P_2 = \{clear, owes\}$,
- $C_2 = \{(clear, 0), (owes, 0)\}$,
- $\Sigma_2 = \{b\_lost\}$,
- $\Delta_2 = \{(clear, b\_lost, owes)\}$.

Note that states *order*, *out* in *Book* state machine are duplicated, while for *Patron* case, states *clear* and *owes* is copied. Transition label *b_lost* are copied in both actors. New transition label *b_order* shows the book-ordering event raised after notifying a book loss.

An important point is on *Patrol* class. We can see that state machine for this class (base and extension) share the same transition arc *b_lost* from state *clear* to *owes*. It seems the transition in those two units are the same but in fact, they are different in the action activated on transition fire. In the base layer, the action simply records that the user owes to our library a book, while in the extension, his/her fine record is updated by the price of that book. If composing two state machines together, we need to compose two actions in the resulting compound state chart.

Adding extensions e_1 and e_2 to s' results in $s'' = (Ca'', Co'', \delta'', \omega'')$. Certainly, $s' \sqsubseteq s'' = s' \oplus \{(Book, Extension, l_1, e_1, sc_1), (Patrol, Extension, l_2, e_2, sc_2)\}$. After some simple steps, we can derive the specification for s'',

- $Ca'' = \{Book, Patron\}$,
- $Co'' = \{Base, Ext\}$,
- $\delta'' = \{(Book, Base, u'_{11}), (Patron, Base, u_{12}), (Book, Ext, e_1), (Patron, Ext, e_2)\}$,
- $\omega'' = \{(Ext, \{Base\})\}$.

Chapter 5

Evolutionary Development Process in Layered Designs

5.1 Program Development Process

In the previous chapters, we have established a formal foundation for the specification domain of role-based systems.

The development process maps mixins in classes and collaborations in specification correspondingly to the fragments in program. Each mixin is initially mapped with a class fragment whose class is set with the mixin's class name. This mapping is very transparent from specification to implementation. The evolution relationship in mixin specification also reflects in the respective class fragment implementing that mixin. If a mixin is more evolved than the other, then its class fragment in program domain is also more evolved than the other's counterpart with respect to OO type evolution. This is a rationale behind skipping the formalization part of evolutionary program domain and claiming the structure similarity between specification and program domains.

As the program fragments corresponding to mixins are completed, the fragments sharing same class name are composed. This class fragments composing step relies on two basic factors. Firstly, programming language should support separate mixin implementation. In fact, this property is guaranteed by many OO programming languages. The second factor is involved with composing program fragments. The composition mechanism is different between languages. Some OO languages utilize inheritance feature for mixins composition. C++ is among these. On the other hand, some AOP-flavored languages deal with composition via a compositor. This compositor parses directives written in high-level script to compose modules written in some typical OO language. Notable languages with this mechanism are Hyper/J and AspectJ which compose layers written in traditional Java.

The specification of role-based systems includes both static and dynamic behavior specification discussed in Chapters 3, 4 respectively. Of course, the set of specification $\mathcal{S}$ should involve both $\mathcal{S}_\int$ and $\mathcal{S}_\lceil$. On the other hand, let $\mathcal{P}$ be the set of role-based programs. Due to its transparency its the specification domain, intuitively, the structure of element in

$\mathcal{P}$ is similar to that in $\mathcal{S}$. As discussed in Chapter 2, Section 2.1.3, an evolutionary development process, mapping $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{P}$ has to satisfy 4 basic properties, namely:

- Realizability: any *valid* specification is realized by at least a program. It turns to the question of what a valid specification is and how to verify the validity of a specification.
- Monotonicity: This property is ensured due to the monotonic co-variation of mixin specification and class fragment.
- Incrementality: The role-based approach even gives simpler effort in evolving role-based specification, i.e. $\mathcal{F}(S \oplus_S \Delta S) = \mathcal{F}(S) \oplus_P \mathcal{F}'(\Delta S)$. This is clearly simpler compared with the general form defined in Section 2.1.3, $\mathcal{F}(S \oplus_S \Delta S) = \mathcal{F}(S) \oplus_P \mathcal{G}(\mathcal{F}'(\Delta S))$. This is due to the transparency between specification and implementation domains.
- Locatability: Due to the above property, the *locator* $\mathcal{L}$ is greatly simplified. The effort to maintain specification - program mapping is very simple.

From the above observation, transparency property has helped a lot during software development and evolution. In addition, simple composition (extraction) keeps the effort to integrate the change in program with the existing system small. Overall, these two properties eases software development and evolution work.

5.2 Formal Evolutionary Development Process

Based on previous formal specification of role-based designs, this section presents a formal process description in transforming specification to program. As the target domain is dependent on its underlying facility including programming languages, it is better to express the target in a pseudo-language. Later, those constructs of this pseudo-language are mapped to specific constructs if we implement programs in the respective programming language.

As a first step toward a complete development process description, this section tries to deal with formal static structure specification. In case of dynamic behavior, this section does not present the way mapping the dynamic behavior of a role[1] to the pseudo-code due to inherent difficulty in dealing with dynamic behavior and the lack of investigation in that area.

With those restrictions in mind, the formal mapping deals with static structure specification only. Therefore, the mapped programs are only represented as class skeletons whose contents are later determined by dynamic behavior specification.

[1]Possibly by a DPN unit or an equivalent language construct such as CRE mentioned in Chapter 4

$$\begin{array}{rcl}
P & = & \mathit{defn}^{*}\ s^{*} \\
\mathit{defn} & = & \mathit{outermixin} \mid \mathit{innermixin} \\
 & & \mid \textbf{mixin}\ m\ =\ m\ \textbf{compose}\ m \\
 & & \mid \textbf{interface}\ i\ \textbf{extends}\ i^{*}\ \{\mathit{meth}^{*}\} \\
\mathit{outermixin} & = & \textbf{mixin}\ m\ \textbf{extends}\ m\ \{\ \mathit{innermixin}^{*}\ \} \\
\mathit{innermixin} & = & \textbf{mixin}\ m\ \textbf{extends}\ m :: m\ \textbf{implements}\ i^{*}\ \{\ \mathit{field}^{*}\ \mathit{method}^{*}\ \} \\
 & & \mid \mathit{outermixin} \\
\mathit{field} & = & t\ \mathit{fd} \\
\mathit{method} & = & t\ \mathit{md}\ (\mathit{arg}^{*})\ \{\ \mathit{body}\ \} \\
\mathit{arg} & = & t\ \mathit{var} \\
\mathit{body} & = & s^{*} \mid \textbf{abstract} \\
s & = & \texttt{a statement} \\
\mathit{var} & = & \texttt{a variable name or } \textbf{this} \\
m & = & \texttt{a class name, a mixin name or Object} \\
i & = & \texttt{interface name or Empty} \\
\mathit{fd} & = & \texttt{a field name} \\
\mathit{md} & = & \texttt{a method name} \\
t & = & m \mid i
\end{array}$$

Figure 5.1: Pseudo-language basic syntax: A statement is treated as a primitive syntactic unit because the details of statement types are not relevant to subsequent discussion in this section.

5.2.1 Mixin-Supporting Pseudo-Language: A Basic Syntax

On the program side, a minimum pseudo-code is assumed to implement necessary constructs for a role-based program. The pseudo-code used here is actually very similar to the syntax extensions for MixedJava. As a result, the terminology is quite Java-flavored. Another note is, since this pseudo language inherently supports mixin, during real implementation state, a language not facilitating mixin like Java requires a little workaround if this formal development process is still utilized. That case will be explained later in Section 5.3.2.

In the Figure 5.1, a basic syntax notation for the language is displayed. Note that the mixin declaration is responsible for both mixin and concrete class declarations. In addition, statements are treated as primitive units and their detailed syntactic notations are skipped due to its irrelevance to subsequent discussion. The subsequent formal development process only deals with mixin, class or interface skeletons. It is not concerned with the internal details of methods in those entities. Statement syntactic details are only needed in case of generating codes from dynamic behavior model which, as mentioned before, is left for future work.

A program in this language consists of a sequence of declarations followed by a sequence of statements. A declaration could be either an outer mixin, an inner mixin, a mixin composition or an interface definition. An interface, as expected, could extend several interfaces. At the same time, its content are only method declarations. An outer mixin extends[2] another outer mixin. Within this outer mixin, there is a list of associated inner mixins. Each inner mixin could be defined by a regular mixin class implementing fields and methods. The notation of `m::m` shows a fact that this inner mixin extends another intentionally mapped inner mixin belonging to an outer mixin referable from the initial inner mixin scope. Or even an inner mixin could be an outer mixin itself.

A field is declared with a type and a name. Similarly, a method signature is associated by a name, several arguments with types and a body. That body could be either concrete or abstract.

5.2.2 Linear Ordering of Mixin Layers

According to Chapter 3, a collaborative static structure s is formalized as a tuple of (Ca, Co, δ, ω). This section presents a framework mapping a static structure to a partially complete mixin-based pseudo-code program.

Let consider the $\omega : \mathcal{CO} \to 2^{CO}$ element. It specifies the ordering dependency between collaborations. This framework only deals with acyclic dependency collaborations. In case of cyclic dependency, a subtle technique, later mentioned in Section 5.4, is utilized to break the cycle. However, that technique is only applicable in some specific cases requiring internal structures of collaborations and roles. As a consequence, at this stage, that technique is skipped.

If we represent the dependencies between collaborations by a directed graph in which each collaboration is a node, 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 lemmas involving with dependency acyclic graph.

Lemma 28 *In an acyclic directed graph, there is at least one node with zero in-degree.*

The proof is simple and based on the finite set of nodes. That node of zero in-degree is obviously the collaboration in which there is no layer depending on it.

Lemma 29 *Given an acyclic directed graph, if one or more nodes are taken away from the graph together with their inward directed edges, the resulting subgraph is still acyclic.*

The proof is clear. If a subgraph is cyclic then its parent graph is certainly cyclic.

The formal evolutionary development process is defined as: $\mathcal{F} : \mathcal{S} \to \mathcal{P}$. It definition is defined as following:

Definition 30 *Let $s \in \mathcal{S}$ be a role-based specification, $s = (Ca, Co, \delta, \omega)$, the mapping of s to program domain is defined as: $\mathcal{F}(s) = F_o(Ca, \{o\}, \delta/o, \omega/o) \lhd \mathcal{F}(s/o)$ where o is a layer with zero in-degree.*

[2]This terminology use is due to Java. Actually, I would prefer to use the term "inherit" of C++ to express meaning of this relation. That is because most of cases, an outer mixin uses the facility provided by the layer above to achieve its goal, while that upper layer does not care about the existence of the initial outer mixin lying below itself, it does not need "extension".

In this definition, s/o represents the a new system after deleting collaboration o. That system is like the one created after vertical compacting s whose roles in o are set to empty. $\delta_o = \delta/o$ is a the mapping of roles in collaboration o to respective mixin terms. Similarly, $\omega_o = \omega/o$ is a set of collaborations o depends on. The operational semantic of $\lhd$ operator is to initialize the second parameter as the super-mixin of the first parameter.

Of course, the dependency subgraph resulted from s/o is acyclic due to Lemma 29. It should contain at least one zero in-degree node because of Lemma 28. That node is selected for recursive call up to the top layer. During this recursive call, the specification s gets simpler and simpler.

F_o is a fragment mapping for outer mixins.

Definition 31 *The mapping definition F_o for outer mixin is defined as: $F_o(Ca, \{o\}, \delta_o, \omega_o) = F_i(\{a\}, \{o\}, \delta_o, \omega_o) \prec F_i(Ca/a, \{o\}, \delta_o, \omega_o)$.*

In the definition, a is a class label, i.e. $a \in Ca$. Moreover, Ca/a is a set of class label after removing a from Ca. F_i maps inner mixins to program domain. If $Ca = \emptyset$, then $F_i(Ca, \{o\}, \delta_o, \omega_o) = \epsilon$ - empty mixin. The operational semantic of operator $\prec$ is to append the first parameter to the head of list represented by second parameter.

Definition 32 *Mapping F_i is defined as: $F_i(\{a\}, \{o\}, \delta_o, \omega_o) = \delta_o(a, o)[\texttt{SuperLayer::}a]$.*

Keyword `SuperLayer` is used to specify the layer that will be set as super-mixin of current outer mixin. `SuperLayer::a` simply means the mixin whose name is `a` in that layer. The above definition of inner mixin mapping results in a mixin m (equal to $\delta_o(a, o)$) whose name is set to `a`. Furthermore, its parameter is `SuperLayer::a`.

5.2.3 From Formal Mixin Mappings to Pseudo-Language

This section shows how the formal definition of system, outer-mixin and inner-mixin mapping in Definitions 30, 31, 32 really mean to pseudo-language. This section reverses the approach dealt in the previous section, namely starts from lowest level and then goes bottom-up.

For F_i mapping, a role specification is mapped with a mixin term whose super-role is parameterized. In pseudo-code, $F_i(\{a\}, \{o\}, \delta_o, \omega_o)$ should be represented as:

```
mixin a extends SuperLayer::a [implement i]{
  Interface i specifies the protocol this role performs within the layer.
  Internal content of this mixin is mixin term $m = \delta_o(a, o) = \delta(a, o)$.
  Sometimes, this inner mixin could be an outer mixin itself.
}
```

In case of F_o mapping, a layer specification is mapped to a list of inner mixins, enclosed by an outer mixin declaration to limit the scope of those mixins. In pseudo-code, $F_o(Ca, \{o\}, \delta_o, \omega_o)$ can be expressed as:

```
mixin o extends <Parameter> {
  In general, outer mixin does not need to implement any interface.
  Inner mixin list.
```

```
    mixin FstRole extends SuperLayer::FstRole [implement $i_1$]{
      etc
    }

    mixin SndRole extends SuperLayer::SndRole [implement $i_2$]{
      etc
    }

    etc

    mixin LastRole extends SuperLayer::LastRole [implement $i_l$]{
      etc
    }
}
```

With $\mathcal{F}$ mapping, it only order outer mixin in a consistent manner and then provides a mechanism to instantiate the super-mixin at each layer from top to bottom.

```
mixin $o_1$ extends <Parameter> {
  Inner mixin list declaration.
}

mixin $o_2$ extends <Parameter> {
  Inner mixin list declaration.
}

etc.

mixin $o_n$ extends <Parameter> {
  Inner mixin list declaration.
}
```

o_n is the first zero in-degree node in the whole dependency directed graph. After removing o_n, the subgraph results in o_{n-1} as its next zero in-degree candidate. This process is iterated until o_1 .

The final system is constructed by composing outer mixins in the following way.

```
mixin $o_{21}$ = $o_2$ compose $o_1$
mixin $o_{321}$ = $o_3$ compose $o_{21}$
etc.
mixin $o_{n..1}$ = $o_n$ compose $o_{(n-1)..1}$
```

The `compose` operator initializes the second parameter as the super-mixin of the first.

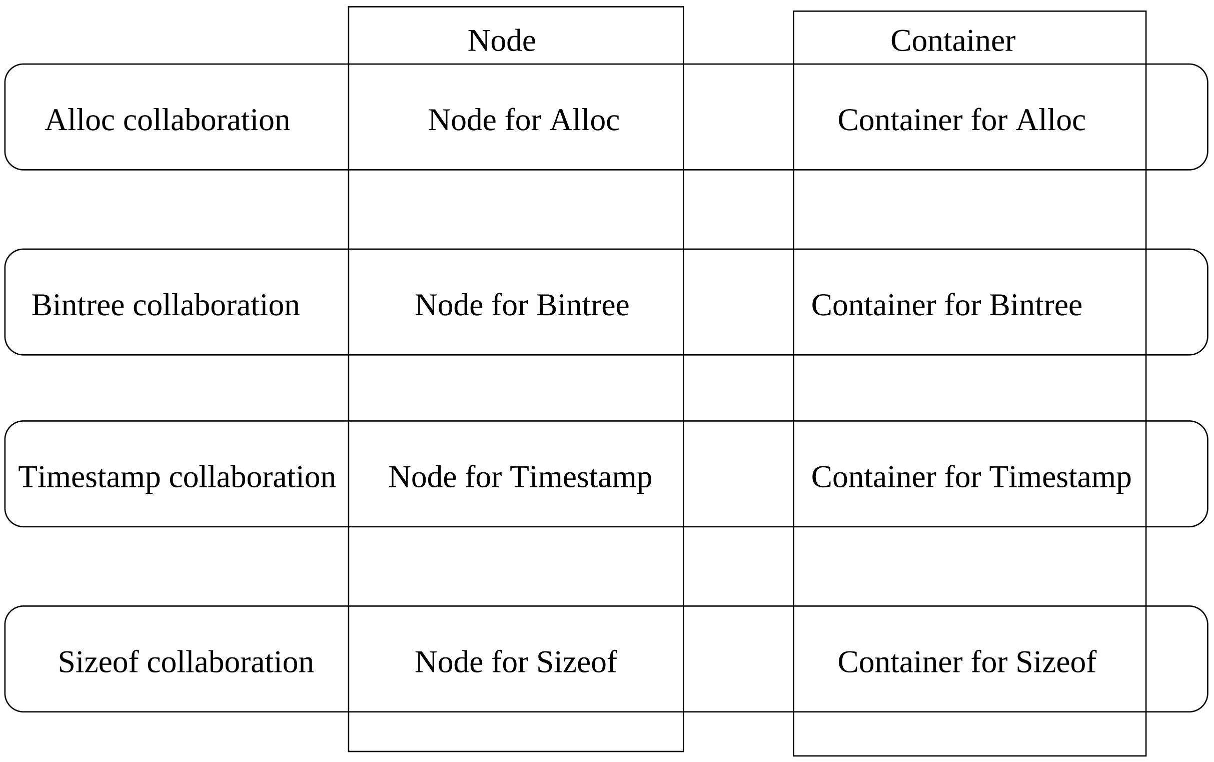

Figure 5.2: Complex data structure design and implementation based on mixin layers.

5.3 Implementing Layered Designs

The example used for illustrating of implementing layered designs shown in Figure 5.2 is taken from [32]. The example is about a data structure design. In this simplified example, there are two classes participating in some collaborations of the design. They are: `Node` class and `Container` class. All data nodes in this data structure are instances of node class, while there is only one instance of container class per data structure. The target data structure consists of four different collaborations: `BinTree`, `Alloc`, `TimeStamp` and `SizeOf`. `BinTree` captures the functionality of a binary tree. `Alloc` is in charge of memory allocation. `TimeStamp` is responsible for maintaining timestamps for data structure and element updates. `SizeOf` simply keeps track of the data structure size. This section will show a revised mixin layer definition mechanism in C++ from the original version in [32]. After that, the paper will present a different way in building mixin layers with Java. The layer composition mechanisms in C++ and Hyper/J are also presented as we go through the detailed implementation.

5.3.1 Mixin Layer Mechanism with C++ - A Template Composition Approach

C++ programming language provides facility in defining mixins. That mechanism is accomplished via template. In this approach, each mixin layer corresponding to a collaboration is implemented as a C++ template class whose super-class is parameterized. Each role inside that collaboration is respectively mapped to a nested class in that mixin layer. This inner mixin also inherits from the corresponding role in super-class of outer mixin. The name of an inner mixin is hard-coded into source. So does the name of its super-class. For correct mapping between inner mixin of lower layer with its super-class of layers above, the respective names must be specified correctly. Otherwise, if the name mismatches, the lower role could not instantiate because its parent is not found. In general, for the convenience, all inner mixins associated with the same actor are assigned with the name of that actor. The composition at a layer occurs by instantiating its super-class with a concrete class of all layers above it. This process results in a new composite concrete class capable of handling all collaborations down to this layer. This new concrete class will be then fed to the immediate below mixin layer as its super-class. This process continues in that manner until the bottom layer is reached. The composition mechanism is characterized by inheritance. This approach composes classes in C++ source code level.

To make `Node` and `Container` classes instantiable, a concrete class for the root is needed. The top layer is the `Alloc` collaboration requiring some type instantiation for its Node class member. The simplified C++ source code from top to bottom layer in this data structure layered design is a revised version having some enhancements compared with the original version taken from [32].

```
template <class InstantiationType> class ALLOC{
  typedef InstantiationType EleType;
public:
  class Node {
    EleType m_tElement;
    // The actual stored data
  public:
    ...// methods definition
  };

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

template <class SC> class BINTREE: public SC{
// SC stands for SuperCollab.
public:
  typedef typename SC::EleType EleType;
  // type passing mechanism between layers.
  // This basic type is visible in this layer.
  typedef typename SC::Node SN;
  typedef typename SC::Container ST;
  // SN stands for SuperNode.
  // ST stands for SuperContainer.

  class Node:  public SN {
  // Mixin name is hard-coded into definition.
  // Upper layer must consist of the class Node.
  // Otherwise, error - superclass not found.
    Node* m_pPLink;// parent link
    Node* m_pLLink;// left link
    Node* m_pRLink;// right link
    // Node data members
  public:
    ...// node interface
  };

  class Container:  public ST {
    Node* m_pHeader;// point to head of tree.
  public:
    virtual Node* NodeAlloc();
    void Insert(EleType e) {...};
    void Delete(EleType e) {...};
    Node* Search(EleType e) {...};
    ...
  };
};

template <class SC> class TIMESTAMP: public SC{
public:
  typedef typename SC::EleType EleType;
  typedef typename SC::Node SN;
  typedef typename SC::Container ST;

  class Node:  public SN {
  // Similar to class Node in BINTREE layer.
    time_t m_nCreationTime, m_nUpdateTime;
    // Node data members
  public:
    bool MoreRecent(time_t t) {...};
    ...// node interface
  };

  class Container:  public ST {
    time_t m_nUpdateTime;
  public:
    virtual Node* NodeAlloc();
    bool SearchNewer(EleType e, time_t t) {...};
    // This method uses a call to ST::Search(e)
    void Insert(EleType e) {...};
    // This method uses a call to ST::Insert(e)
    ...
  };
};
```

```
template <class SC> class SIZEOF: public SC{
public:
  typedef typename SC::EleType EleType;
  typedef typename SC::Node SN;
  typedef typename SC::Container ST;

  class Node:  public SN {
  // etc.
  };

  class Container:  public ST {
    int m_nCount;
  public:
    virtual Node* NodeAlloc();
    Container():m_nCount(0), ST() {};
    void Insert(EleType e) {...};
    // This method uses a call to ST::Insert(e)
    ...
  };
};
```

The target data structure is formed by composing mixin layers. A binary storing integers and maintaining time information and size is defined a type `ABTS` standing for above four collaborations, namely `Alloc`, `BinTree`, `TimeStamp`, `SizeOf`.

```
typedef SIZEOF< TIMESTAMP< BINTREE< ALLOC< int > > > > ABTS;
```

The `Node` and `Container` classes in this data structure can be retrieved from `ABTS` as:

- `typedef ABTS::Container ABTS_Container;`
- `typedef ABTS::Node ABTS_Node;`

The usual operations on binary tree structure with timestamp and counter management can be achieved via collaborations of some objects from above two classes. As each collaboration is encapsulated in separate module, we have created a group of components. By plugging those components together in a consistent manner, namely preserving dependency constraints between layers, we can produce a family of products. Furthermore, any change in a collaborative protocol is localized to the component implementing that collaboration. Because the interface between layers are considered very stable, as long as new changes in the component implementation compromise with that interface, new component can replace its old version in any existing products without a need to alter the implementation of other components.

5.3.2 Mixin Layer Mechanism in Java and Hyper/J - A MDSOC-Based Approach

Java does not have the facility to support mixin directly. Thus, mixin layer implementation by Java is much different from C++ approach. The implementation mechanism

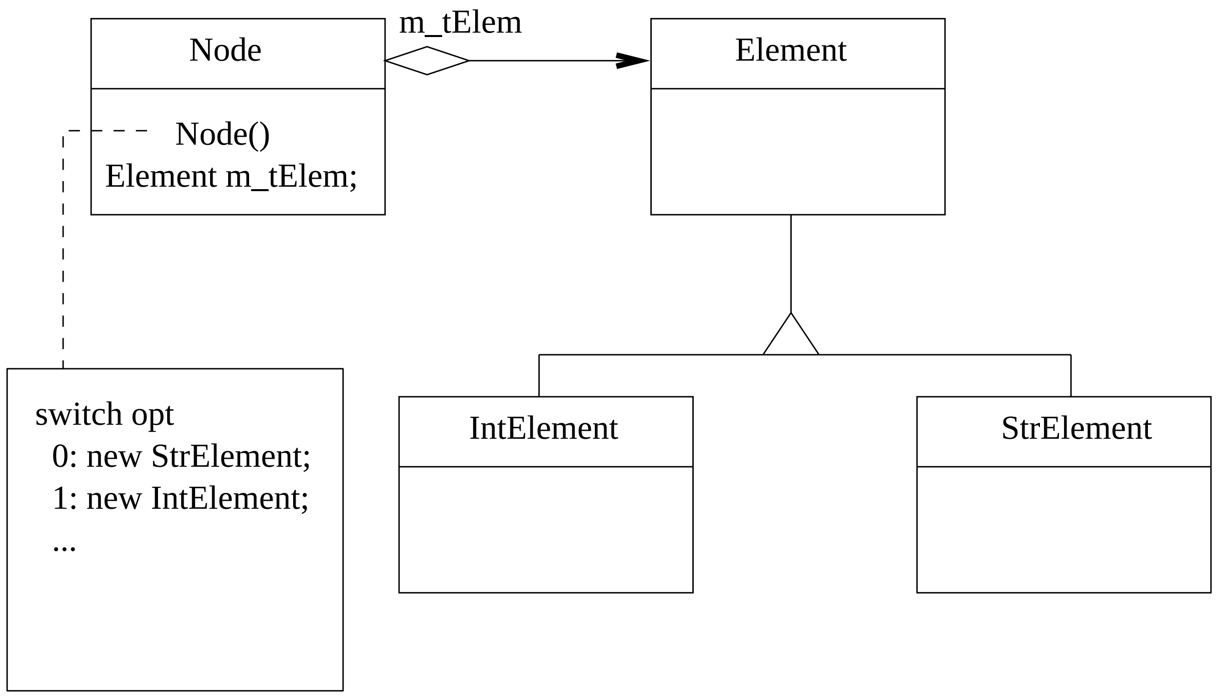

Figure 5.3: Builder pattern to instantiate different concrete Element classes by Node class constructor depending on option.

of collaborations in this section is not based on mixin. In fact, this section presents a workaround method to implement mixin layer in case the underlying language does not support mixin. This workaround is rooted at MDSOC model. As previously claimed, collaboration is a special case of concern. Hence, mixin layer for a collaboration can be equally considered as the representation of a concern in MDSOC. In Java, each concern can be encapsulated in a separate module. Hyper/J then acts as a compositor to synthesize those concerns. Characteristically, this section shows another way to simulate mixin layers in implementing collaborative designs.

In this example, as above, every collaboration requires only two classes, namely `Node` and `Container`. In Java-based implementation, the outer mixin is not explicitly programmed as the case of C++. Instead, we wrap each outer mixin into a separate Java package. Each package has two class definitions. In addition, each package is mapped by Hyper/J script to a separate feature. Because of four collaborations to be implemented, there are four features to be specified in Hyper/J project file. To deal with type instantiation at the top layer, namely `EleType` for class `Node`, the Builder design pattern is used as illustrated in Figure 5.3 [11].

The top layer, besides `Node` and `Container` classes, has another class called `OpenDS` (Open Data Structure) for main program. Its job is to request the container member to "process" a node. The meaning of this call is different in layers. In `Alloc` layer, it asks the container to allocate memory for a node. On the other hand, in `BinTree` layer, container will insert a value to the tree. However, these `process` methods in containers at different layers will be merged into a composite `process` method by a composition rule in Hyper/J - `mergeByName`. This composite `process` method calls all member `process` methods in sequence.

```
public class OpenDS{
  public static void main ( String[] args ){
    m_nOption = args[0];
    m_cCont = new Container();
    Element e;      ...
    try{ m_cCont.Process(e);
    }
    catch ( Exception e ){...};
  }; // END main
    ...
  private static Container m_cCont;
  private static String m_nOption;
};

// Alloc collaboration definition
package collab.Alloc;
public class Container{
// Container role definition
  public Node AllocNode(int n, String s){
    Node res = new Node(n, s);
    return res;
  };
  void Process(Element e){
    // do something appropriate for the
    // element e in this layer
    ...
  };
}; // END class Container

package collab.Alloc;
public class Node{
// Node role definition
  public Node(int n, String s){
  // create m_tElem according to option s.
  };
  ...
  private Element m_tElem;
}; // END class Node
```

The Java code for the other three layers, namely `BinTree`, `TimeStamp` and `SizeOf`, are quite similar to that of `Alloc` layer. The source code of those layers are skipped here.

Once a layer implementation is completed. We can compile inner mixins (`Container` and `Node` classes) separately into Java's `.class` files. Hyper/J compositor then comes in to deal with those `.class` files. It looks for its three component files in the project.

```
// Hyperspace specification file:  OpenDS.hs
// Instructing Hyper/J to compose all Java
// classes in provided packages.
hyperspace CollabHyperspace
  composable class collab.Element.*;
  composable class collab.Alloc.*;
  composable class collab.BinTree.*;
  composable class collab.TimeStamp.*;
  composable class collab.SizeOf.*;

// Concern mapping file:  concerns.cm
// It specifies the feature (collaboration)
// corresponding to each package.
package collab.Element :  Feature.Alloc
package collab.Alloc :  Feature.Alloc
package collab.BinTree :  Feature.BinTree
package collab.TimeStamp :  Feature.TimeStamp
package collab.SizeOf :  Feature.SizeOf

// Hypermodule specification file:  ABTS.hm
// It instructs Hyper/J to merge classes,
// methods with each other by name.
hypermodule ABTS
  hyperslices:
    Feature.Alloc,
    Feature.BinTree,
    Feature.TimeStamp,
    Feature.SizeOf;
  relationships:
    mergeByName;
    override action Feature.BinTree.Node.Value
      with action Feature.Alloc.Node.Value;
end hypermodule;
```

Denote `A.B` for mixin `B` in layer `A`; and `B::C` for method/data member `C` in class `B`. The `override` instruction in the above hypermodule is to override any `Value` function call of `Node` class defined in `BinTree` layer by that in `Alloc` layer. `BinTree` layer is not aware of the contents inside each node because that value is associated with `m_tElem` data member in `Alloc`. `BinTree` will have to refer to `Alloc::Value` to get the right value of node for binary search, insertion or deletion. To make each package a separate compilation, the reference to `Value` function call in `BinTree` is initially directed to a dummy `Value` method in `BinTree.Node` mixin. That dummy method is eventually overriden by `Alloc.Node::Value` function (as written in hypermodule file `ABTS.hm`). People may wonder another way of not implementing `BinTree.Node::Value()` but leaving it as an abstract method. Later we can map that abstract method with the method `Alloc.Node::Value()` to complete the implementation. This does not work simply because that abstract method would give `BinTree.Node` as an abstract class. That means we can not create any `BinTree.Node` instance within the context of this `BinTree` layer. But for a binary tree insertion, a `Node` creation is needed for any `Element` value while doing the appropriate binary tree insertion. Thus, `Node` instance, even with respect to `BinTree` layer only, must be concrete.

```
package collab.BinTree;
public class Node{
// Node role in BinTree collaboration.
  ...
  Element Value(){
  //dummy implementation, overridden by
  //Node::Value() in Alloc collab.
    return null;
  };
  ...
}; // END class Node
```

There is another important note on dependency between layers. `BinTree` layer and `TimeStamp` layer are both working on value insertion operation for a `Element` value. `BinTree` layer only deals with usual binary tree insertion, deletion and search. For searching operation, it provides a method whose signature is `''Node FindNode(Element e)''` defined within `BinTree` layer's context. On the other hand, `TimeStamp` layer needs to update the timestamp associated with a node when its value is inserted into a data structure. Its insertion does not care about the way node is inserted and stored, i.e. whether the higher layer deals with those values according to a binary tree or a linked list management scheme. In this case, that higher layer is binary tree. The main goal of node insertion in this `TimeStamp` layer is to update the current timestamp to a node which has been just inserted by the layer above. To do that, it needs to get a pointer to the place where that value was inserted. That pointer can be retrieved through `BinTree.Container::Search()`. Through the pointer, `TimeStamp.Container` can update the current timestamp of system to the appropriate node. This relation between these two `BinTree` and `TimeStamp` layers identifies a dependency from the latter on the former. To deal with this dependency, an abstract method `''Node FindNode(Element e)''` is declared in `TimeStamp.Container` class definition. By declaring in such manner, `TimeStamp` package can be separately compiled but the `Container` class is not instantiable for its abstract method. That method will be merged with the above method of the same signature in `BinTree.Container` during Hyper/J composition. That is called "declarative complete"[34].

```
package collab.TimeStamp;
import java.util.*;
import collab.Element.*;

public abstract class Container{
```

```
// class fragment for TimeStamp collaboration.
  ...  // methods definition.
  void InsertNode(Element e){
  // Upper layer will insert the element.
  // That InsertNode() runs before this method.
  // This layer only updates time of that node.
    Calendar c = Calendar.getInstance();
    Date d = c.getTime();// get update time
    Node n = FindNode(e);
    // retrieve the pointer to that node.
    n.UpdateTime(d);
  };

  abstract Node FindNode(Element e);
};

package collab.BinTree;
import java.io.*;
import collab.Element.*;

public class Container{
// class fragment for BinTree collaboration.
  Node FindNode(Element e){...};
  etc.// other methods and data declaration.
};
```

5.4 Discussion

Previous sections are about the design of collaborative design. The implementation mapping from designs to different languages are also presented. Two mixin layer mechanisms are mentioned, namely real mixin layer (in C++) and MDSOC-based layer (in Java). This section devotes for the strength and weakness as those two are compared with each other. After that, some typical issues arisen during programming mixin layers are discussed.

5.4.1 Comparing Two Methods

Problems in Mixin Layer Approach

As commented in Section 5.3.1, corresponding inner mixins between layers, in general, should be the same name. Nevertheless, in some inevitable cases requiring adaptation, we can redirect a mixin to inherit a super-class with a different name via `typename` facility in C++ like the following:
`typedef typename SC::DiffContainer ST;`
This declaration will direct `Container` role to inherit attributes from `DiffContainer` actor of a layer above it. However, all name mappings must be declared right at the

source code. This is not a problem in Hyper/J as we can connect mixins with different names together by using `equate` directive in hypermodule file.

In the former approach, because all inner mixins are written with their corresponding outer mixin, we face the tightly coupled problem between inner mixins and outer mixin. As a single role evolves while other roles in the same collaboration is not changed, this type of coding is not appropriate for such an evolution. In such a case, we have to change the source code of inner mixin which is the same as of outer mixin. And a recompilation of a new inner mixin activates the recompilation of the whole outer mixin and inner mixins of other roles.

Because the mixin composition happens at the source level in C++, a new product built from available correctly-compiled collaborations always requires recompiling. That is not true for Java and Hyper/J because the later operates on byte-code level (`.class` files).

The above disadvantage of the former approach is largely due to the limitation of C++ template static binding. In C++, when a template is used, its source must be included. Mixin layer composition always operates on template source.

As pointed out in [32], because composition is not commutative, there are some semantical dependency orders between layers. That means the data structure formed by `BINTREE< ALLOC< > >` is not the same as that of `ALLOC < BINTREE< > >`. To enforce a correct composing order, low-level codings, such as assertion or supplementary data members, are essential in outer mixin class definition. Additional member variables in outer mixins for composing assertion do not properly reflect the encapsulation of mixins. That is because a collaboration is a separate code module and it is intended for reuse in other contexts. Hence, some additional variables like `P_NoSizeof dummy1`, `P_NoTimestamp dummy2`...[32] as introduced in asserted version of `BINTREE` outer mixin definition are not appropriate. If we utilize this `BINTREE` collaboration in different application without `SIZEOF` and `TIMESTAMP` collaborations, such dummy variables are not relevant in that circumstance. By introducing those variables into class definition, more dependencies are added between collaborations. A layer should be supplied as a component with a well-designed interface and semantic, and the ordering scheme to plug those components together must follow their interfaces and ordering dependencies. Therefore, in our view, composing order should be specified at higher level, outside of mixin definition.

Mixin layer approach therefore requires a concise way to specify layer constraints in addition to the mixin layer definition itself. At the moment, ordering collaborations are still done manually. In MDSOC-based approach, a concern-mapping layer written in Java also suffers the same kind of dependency constraints. However, Hyper/J comes to help in dealing with specifying those dependencies between features, although the specification is not very explicit. It is desirable to construct a pre-processor for mixin layer dependency specification which is somewhat similar to Hyper/J. Given a set of mixin layers, this pre-processor can automatically detect the dependency between layer based on their interfaces and give the correct composition order. How an interface is represented is still not clear though. As later mentioned in Section 5.4.2, the characteristics of layer dependency are *hot-spots*. Some layers supply concrete definitions for hot-spots, while others use those hot-spots without concerning about their actual implementations. To tackle this type of problem, a layer interface should specify clearly the hot-spots supplied and required by its member mixin classes. This would be a topic of future work.

Setting aside its drawback due to language (i.e. C++) capability, mixin layer also suffers another setback because of its theoretical nature. This weakness is independent of the underlying languages. As mixin layer composition in C++ utilizes linear inheritance at its basis, there could be a problem if there exists a cyclic dependency between layers. At the first thought, there is no way to order such layers into a linear composition sequence. Dealing with such a problem requires a subtle technique to rearrange layers. The core of this technique is to break a *selected* link between two layers in the cycle. After breaking the link, dependent layer `A` with respect to that link will be on top of the other layer `B` according to usual composition scheme because the link is not considered. That link will be implicitly implemented as methods of layer `A` depending `B` will be relegated downward to `B`. Because some methods can not be relegated in such a way, not all dependencies can be solved in this technique. That is why only special links satisfying some properties can be selected as candidates to be broken.

On the other hand, cyclic dependency does not pose serious problem to mechanism in Hyper/J. The only concept in this mechanism is group matching without any consciousness about order between layers. A layer calling a method without knowing about its implementation should declare that method abstract. Another layer will defines concretely that method. At the same time, the second layer will declare a different method as abstract which is in fact implemented by the first method.

Problems in MDSOC-based Layer Approach

Comparing with Java, because of its generic programming support with template construct, initial type instantiation for collaboration implemented by C++-based mechanism is uniformly treated for any primitive type or concrete class. Hence, the type parameter like `SIZEOF < TIMESTAMP < BINTREE < ALLOC <t> > > >` is very simple for `t` whether it is a primitive type (int, string) or any concrete class. This advantage is due to the language capability of C++. Currently, Java does not have template construct, it treats primitive types and reference types differently. Therefore, type instantiation in Java-based program is more complicated (via abstract class `Element`).

The characteristic of mixin layer mechanism in C++ is based on real mixin. There exist some "stable" relations between layers. A role in a layer can call methods provided by its superclass in layer above. Hence, the interfaces between layers are in general well-defined and stable. Moreover, as corresponding methods in different layers will be eventually replaced by a respective method at the bottom layer, control logic inside this method can be very complicated as new collaborations are composed. This inheritance-based composition mechanism helps defining very complicated logic dealing all layers at once. In fact, this mechanism allows interleaving corresponding methods into a single composite method which represents all initial methods if they are ever called.

On the contrary, Hyper/J approach has no idea about relations between layers. Role in one layer can not call a method of the same actor in other layers. The characteristic of this approach is biased toward the so-called *flattening* technique. In that technique, there is only *encapsulation* feature in OO world. There is no traditional features of OO technology like *inheritance* and *polymorphism*. Hyper/J flattens all classes of roles in layers. It then matches classes from the same actor together via grouping the corresponding attributes (i.e. methods and data). Calls to abstract methods are eventually replaced by those calls to concrete counterparts (eg. `FinNode` method in `TimeStamp` is abstract and eventually mapped to `FindNode` method in `BinTree` layer). Concrete methods are merged into one composite method executing its element methods in a schedule according to layers composition order (eg. composite `Process` in main program is defined as a sequential execution of `Process` methods in layers `Alloc`, `BinTree`, `TimeStamp` and `SizeOf`). Due to their unawareness of the other layers, control logic inside the composite methods are generally simple as they are either merge all or select one method from several member methods. Methods can not be interleaved in this approach. A constituent method either runs as a whole or does not run at all (when it is overridden). This is due to the fact that at the moment, MDSOC model considers a method as atomic [34] [3]. Comparing with real mixin layer mechanism in C++, this approach is much less powerful. In addition, most of the time, role in a layer requires the existence of some method in other layer, it has to declare the method in the calling layer as well (like the case of abstract `TimeStamp.Container::FindNode` above).

An example to illustrate problems in Java and Hyper/J approach is about dependency between `BinTree` and `SizeOf` layers. The constraint is related with `Insert()` method in these layers. As expected, `BinTree` layer inserts a value in the binary as usual whether that value is already in the tree or not. However, with the addition of `SizeOf` collaboration, some extra cares are needed. If the value is not in the tree, node counter variable `m_nCount` in `SizeOf.Container` increases by 1. Otherwise, it stays the same as it was. As the order of execution, the method `Insert()` in `BinTree` layer will be executed ahead of that in `SizeOf` layer. As the first method is done, the value is in the tree regardless of its initial availability before composite `Insert` call or not. Hence, `SizeOf` layer loses the track of that value in order to update `m_nCount` properly. In the following, a workaround is proposed to this problem. A supplementary variable is added to both layers to keep track of the existence of a value before `Insert` call of `BinTree` is made. That variable is passed to `SizeOf` layer for later proper update. This variable is redundant and imposes extra synchronization between two layers. It leads to bad design and possible flaws as system evolves.

```
package collab.SizeOf;

public abstract class Container{
// class fragment for SizeOf collaboration.
  ...// other methods
  void InsertNode(Element e){
```

[3]In our opinion, to overcome this setback, MDSOC model must provide a way to refer features from using side to other supplying sides. In mixin layer, layers are referred to each other via super-class.

```
    if(m_bAlreadyThere == false){
    // node is not there before
      m_nCount++;
    }
    else{
    }
  };
  private boolean m_bAlreadyThere;
  // synchronization point with BinTree layer.
  private int m_nCount;
};
package collab.BinTree;
public class Container{
  ...// other declarations.
  void InsertNode(Element e){
    m_bAlreadyThere = false;
    // assume this value is not in tree yet.
    // update m_bAlreadyThere accordingly.
    ...
  };
  private boolean m_bAlreadyThere;
  // synchronization point with SizeOf layer.
};
```

On the other hand, in C++ case, the real mixin implementation is simple as it reuses many of calls in upper layers. The above job can be easily achieved in real mixin layer as the following.

```
template <class SC> class SIZEOF: public SC{
public:
  ...// other definitions.
  class Container:  public ST{
    ...// other definitions.
    void Insert(EleType e){
      Node *pn = (Node *) ST::Search(e);
      ST::Insert(e);
      if (pn == NULL) m_nCount++;
      return;
    };
  };
};
```

5.4.2 Programming Issues Within Mixin Layer Context

It is important to note that though mixin layer programming is based on OO technology, programming in mixin layers are more difficult and much different from traditional OO programming. In the traditional OO approach, we only need to consider one-dimensional horizontal interface formed between classes. In the mixin layer implementation paradigm, there are two dimensions of concerns. The first is between classes as usual OO. The second is between layers and hence vertical. Handling calls in horizontal direction is the same as basic OO programming. The points to be discussed further are related with vertical direction, namely layer-crossing calls. Due to space limitation, those points are not shown in details in this paper.

With the implementation of the small example about composite data structure, we have observed some frequent function calls between roles in different layers. Each of call type requires a different technique.

- Memory management calls for objects: allocating and deallocating memory of some roles and objects. These calls in general has to be dynamically bound (virtual functions and virtual destructors in C++).

- Up-calls in the same actor: a role uses some methods in the same actor of upper layer. In C++, it is simple since calls are directly accessible via inheritance.

- Down-calls in the same actor: a role relegates a call to lower layer to deal with. This type of calls is usually accomplished via virtual calls.

- Skew up- and down-calls: a role in a layer issues a call to another actor in different layers. This is usually done via two-step function call. The skew call is assigned to a horizontal call from source role to target role. Then a vertical (up or down) call is directed to the appropriate layer.

Layer dependency specification is another point to be discussed. A layer may depend on several layers. In our previous data structure example, `BINTREE`, `TIMESTAMP` and `SIZEOF` layers are all depending on `ALLOC` layer. The link between these dependencies is the instatiation type `EleType` defined in `ALLOC` layer. At the same time, `TIMESTAMP` and `SIZEOF` layers are dependants of `BINTREE` for their uses of `ST::Insert` call in their respective function calls. Between layers, those are *hot-spots* in which layers synchronize with each other. Hot-spots are either data member, type parameter (`EleType`) or method (Insert). Identifying hot-spots properly and including them in layer component interfaces could be very useful in verifying the consistency of a collaboration-based design.

Chapter 6

Future Work

6.1 Dynamic Behavior Formalization

This thesis has provided some preliminary attempts on dynamic behavior modeling of role-based designs. So far most of the work has focused on static structure which is, in general, regarded as easier than the counterpart in dynamic behavior. Furthermore, static structure only reveals visible interfaces of roles from outside. The real contents inside class fragments are constructed from role dynamic behavior.

Formal model of dynamic behavior is so important in verifying the consistency of a design. In addition, that model is used as a basis of a mapping framework of evolutionary development process between specification domain to program domain written in some selected programming languages.

As presented in Chapter 4, the dynamic behavior could be represented by Decomposed Petri Net (DPN). However, this is just a preliminary attempt. Another way is to use concurrent regular expressions (CRE). Both ways need time to validate their respective power and efficiency.

6.2 Evolutionary Development Process in Selected Programming Languages

So far, this research has mentioned about mapping between static structure model to some representative languages such as C++, Java. However, this mapping only deals with the external interface, not with the contents inside of each role. To do that, this formal mapping, from arguments of previous section, requires formal model of dynamic behavior.

Constructing the formal model for dynamic behavior is interesting enough. Based on such a model, a framework of code generator would be even more appreciated. Such a framework is inspired by research about *domain-specific languages.* The general rule of such a code generator is to encapsulate state charts of roles in a high level language for its high abstraction and simplicity. Then a generator will produce codes associated with transitions and nodes in composite state charts. The codes are written in some regular programming languages such as Java or C++.

6.3 Formalization and Evolution of AOSD Paradigm

We claim that collaboration-based approach is a special case in AOSD paradigm. Our research so far has dealt with only static structure of role-based designs. It is preferable to extend the work to more advanced model, namely MDSOC or AOP. In addition to static structure analysis, the work on dynamic behavior evolution of AOSD paradigm are much essential.

As AOSD is a promising approach to build software with low cost and high evolvability. A formal specification and representation of AOSD is essential to understand the static and dynamic behavior of a system constructed under the approach. However, it is necessary to find the common between current two major trends in AOSD. One approach would start with MDSOC and subsequently extend toward AOP. This is my favorite approach since MDSOC treats concerns as primitive unit by hyperslice encapsulation. It focuses much more on interconnection between hyperslices rather than details inside each hyperslice. This trend is rather in favor of large-grained modules. In my opinion, the formalization of static structure specification is the first to be dealt with. That will utilize much experience from MDSOC. Later step in tackling dynamic behavior will require the help of AOP.

A comprehensive formal specification of concern (MDSOC) and aspect (AOP) is needed for specification domain of such systems. Based on this specification, the complete system model of static structure and dynamic behavior is specified. This process is similar to that having been applied in role-based designs. The details are following:

- primitive units in concerns and aspects.
- concern and aspect formal representations.
- formal tag set representation.
- dependency order and semantic constraints between concerns and aspects;
- the changes in static structure, dynamic behaviors due to evolution process: composition and difference operators on concerns and aspects.

In Hyper/J, primitive units contributing for each concern are methods, data members. However, in AspectJ, it is possible to control execution flow at an arbitrarily nested depth. In AOP, experiments have shown that it often leads to more compact code but sometimes the program semantic is altered in an unexpected way.

6.4 Evolutionary Development Process in AOSD Paradigm

Evolutionary development process serves as a bridge between specification and program domains. Even though we want to formalize both specification and program sets as evolutionary domains whose structures are similar, the effect of the transformation mapping $\mathcal{F}$ is not negligible.

Some analysis on software developed by Java and Hyper/J, and possibly AspectJ are to be considered in reference to general evolutionary development framework.

- mapping $\mathcal{F}$ definition: how various units in specification domain are mapped to program fragments in Java.
- how Hyper/J (or AspectJ) integrates those program fragments in Java together in a consistent way.
- run time concern update: dynamic behavior of concern-composing system.

Appendix A

Semi-lattice Structure

A.1 Static Structure Modeling

This section gives the proof of semi-lattice structure of $(\mathcal{S}_{\int}, \sqsubseteq)$ where $\mathcal{S}_{\int}$ is a set of static structure specification defined in Chapter 3, Section 3.3, whereas $\sqsubseteq$ is the evolution relation in static specification set $\mathcal{S}_{\int}$ as shown in Definition 9.

Let $s_1 = (Ca_1, Co_1, \delta_1, \omega_1)$, $s_1 = (Ca_2, Co_2, \delta_2, \omega_2)$, $s_1 = (Ca_3, Co_3, \delta_3, \omega_3)$.

1. Order relation $\sqsubseteq$ in $\mathcal{S}_{\int}$:

   Reflective property: In term of OO type theory, we have: $\forall$ `type` $c : c \sqsubseteq_R c$. Therefore, $\forall a \in Ca_1,\ \ o \in Co_1 : \delta_1(a, o) \sqsubseteq_R \delta_1(a, o)$.
   In addition, $Ca_1 \subseteq Ca_1$, $Co_1 \subseteq Co_1$ and $\forall o \in Co_1$, $\omega_1(o) \subseteq \omega_1(o)$. According to Definition 9, $s_1 \sqsubseteq s_1$. Hence, the $\sqsubseteq$ relation is reflective. $\Box$

   Asymmetric property: Suppose $s_1 \sqsubseteq s_2$ and $s_2 \sqsubseteq s_1$. From Definition 9, we have: $Ca_1 \subseteq Ca_2$ and $Ca_2 \subseteq Ca_1$. Hence, $Ca_1 = Ca_2 = Ca$. Similarly, $Co_1 = Co_2 = Co$.

   For a class $a \in Ca$, a collaboration $o \in Co$, as $s_1 \sqsubseteq s_2$, $\delta_1(a, o) \sqsubseteq_R \delta_2(a, o)$. Because $s_2 \sqsubseteq s_1$, by similar step, $\delta_2(a, o) \sqsubseteq_R \delta_1(a, o)$.

   In term of OO type, the above two relations lead to the equality: $\forall a \in Ca,\ \ o \in Co : \delta_1(a, o) = \delta_2(a, o)$. That is, $\delta_1 = \delta_2$.

   It is easy to derive that $\forall o \in Co : \omega_1(o) = \omega_2(o)$, i.e. $\omega_1 = \omega_2$.

   Comparing s_1 and s_2, we have: $Ca_1 = Ca_2$, $Co_1 = Co_2$, $\delta_1 = \delta_2$ and $\omega_1 = \omega_2 \Rightarrow s_1 = s_2$. The $\sqsubseteq$ relation is asymmetric. $\Box$.

   Transitive property:
   Suppose $s_1 \sqsubseteq s_2 \sqsubseteq s_3$, we could derive:

   - $Ca_1 \subseteq Ca_2 \subseteq Ca_3 \Rightarrow Ca_1 \subseteq Ca_3$,
   - $Co_1 \subseteq Co_2 \subseteq Co_3 \Rightarrow Co_1 \subseteq Co_3$,
   - Consider $a \in Ca_1$, $o \in Co$, we have: $\delta_1(a, o) \sqsubseteq_R \delta_2(a, o)$ and $\delta_2(a, o) \sqsubseteq_R \delta_3(a, o)$. For OO type theory, we have: $\forall a \in Ca_1,\ \ o \in Co_1 : \delta_1(a, o) \sqsubseteq_R \delta_3(a, o)$.

- $\forall o \in Co_1, \omega_1(o) \subseteq \omega_2(o)$ and of course, $\omega_2(o) \subseteq \omega_3(o)$. As a result, $\forall o \in Co_1 : \omega_1(o) \subseteq \omega_3(o)$.

According to Definition 9, $s_1 \sqsubseteq s_3$. The $\sqsubseteq$ relation is transitive. $\square$

2. The existence of *greatest lower bound*:

   To prove this part, we assume some basic properties of OO types. In the set of OO types, the evolution relation between type is an order relation. For any two types t_1, t_2, the greatest supertype to both is the intersection of those types, denoted as $t_1 \sqcap_R t_2$.

   Applying that argument to static structure specification set, given two specifciations s_1 and s_2, the greatest lower bound of s_1 and s_2, $s_1 \sqcap s_2$, is the tuple (Ca, Co, δ, ω), where:

   - $Ca = Ca_1 \cap Ca_2$,
   - $Co = Co_1 \cap Co_2$,
   - $\forall a \in Ca, \;\; o \in Co : \delta(a, o) = \delta_1(a, o) \sqcap_R \delta_2(a, o)$.
   - $\forall o \in Co, \, \omega(o) = \{x \;\mid\; x \in \omega_1(o) \wedge x \in \omega_2(o) \wedge x \in Co\}$.

   We can prove easily that $(s_1 \sqcap s_2) \sqsubseteq s_1$ and $(s_1 \sqcap s_2) \sqsubseteq s_2$. Furthermore, for any $s \sqsubseteq s_1$ and $s \sqsubseteq s_2$, $s \sqsubseteq (s_1 \sqcap s_2)$. The intersection of s_1, s_2 exists. Indeed, it is the greatest "super-specification" of both s_1 and s_2. $\square$

3. The existence of $\perp$ specification:

   This special specification $\perp = (\emptyset, \emptyset, \emptyset, \emptyset) = \epsilon$. It is quite easy to prove that $\forall s \in \mathcal{S}_{\int}$, $\epsilon \sqsubseteq s$. $\square$

A.2 Dynamic Behavior Modeling

This section proves the semi-lattice structure of $(\mathcal{S}_{\lceil}, \sqsubseteq)$ where $\mathcal{S}_{\lceil}$ is a set of all dynamic behavior specification defined in Chapter 4, Section 4.4.2, while $\sqsubseteq$ is the evolution relation in the specification set $\mathcal{S}_{\lceil}$ as shown in Definition 23.

The proof of semi-lattice structure for dynamic behavior is very much identical to that of previous Section A.1.

1. Order relation $\sqsubseteq$ in $\mathcal{S}_{\lceil}$:

   As the relation $\sqsubseteq$ is defined on the assumption of relation $\sqsubseteq_R$ in OO types, and relation $\subseteq$ in general set. The latter two relations are reflective, asymmetric, transitive. It is obvious that our evolution relation $\sqsubseteq$ is also reflective, asymmetric and transitive. $\square$

2. The existence of *greatest lower bound*:

   To prove this part, we assume some basic properties of decomposed Petri nets. In the set of DPN, the evolution relation between DPN is an order relation. For any two DPNs u_1, u_2, the greatest subgraph to both is the intersection of those graphs, denoted as $u_1 \sqcap_R u_2$. The intersection of two graphs contains the common nodes and arcs in both graphs. In case of DPN, we need to deal with configuration $C : P \to \mathcal{N} \cup \{*\}$. The configuration at a node in the intersection is set to the minimum value of configurations for that node in both former graphs.

   Applying that argument to dynamic behavior specification set, given two specifciations s_1 and s_2, the greatest lower bound of s_1 and s_2, $s_1 \sqcap s_2$, is the tuple (Ca, Co, δ, ω), where:

   - $Ca = Ca_1 \cap Ca_2$,
   - $Co = Co_1 \cap Co_2$,
   - $\forall a \in Ca,\ \ o \in Co : \delta(a,o) = \delta_1(a,o) \sqcap_R \delta_2(a,o)$.
   - $\forall o \in Co$, $\omega(o) = \{x \ \mid \ x \in \omega_1(o) \wedge x \in \omega_2(o) \wedge x \in Co\}$.

   We can prove easily that $(s_1 \sqcap s_2) \sqsubseteq s_1$ and $(s_1 \sqcap s_2) \sqsubseteq s_2$. Furthermore, for any $s \sqsubseteq s_1$ and $s \sqsubseteq s_2$, $s \sqsubseteq (s_1 \sqcap s_2)$. The intersection of s_1, s_2 exists. Indeed, it is the greatest "subgraph" of both s_1 and s_2. $\square$

3. The existence of $\perp$ specification:

   This special specification $\perp = (\emptyset, \emptyset, \emptyset, \emptyset) = \epsilon$. It is quite easy to prove that $\forall s \in \mathcal{S}_\lceil$, $\epsilon \sqsubseteq s$. $\square$

Appendix B

Open Data Structure Implementations

This appendix gives the detailed implementation of mixin layers. The codes are written in C++ and Java. As told in previous chapters, the illustrated example is about an open data structure encapsulating several collaborations. Within this data structure, there is only two classes, namely `Node` and `Container`.

B.1 C++ Template Composition

The source files written in C++ for the data structure are in the followings:

- C++ template file definition, `OpenDS.h`
- Main program, `Collab4.cpp`
- Makefile for compiling the project, `Makefile`

```
// --------------------------------
// Template definition file OpenDS.h
// --------------------------------

#ifndef __OPENDS_H__
#define __OPENDS_H__

#include <sys/time.h>
#include <time.h>
#include <stdio.h>

#define TRUE 1
#define FALSE 0

template <class InstantiationType> class ALLOC{
  typedef InstantiationType EleType;

 public:
  class Node{
    EleType m_tElement; //The actual stored data.
  public:
    Node(){
      // Constructor can not be virtual, so for dynamically bound constructor
      // of Node, refer to virtual Container::NodeAlloc()
      m_tElement = 0;
    };

    /* This workaround for memory deallocation not dynamically bound */
    virtual ~Node(){// virtual destructor to delete composite Node.
    };
    /* End of workaround */

    void SetValue(EleType e){
      m_tElement = e;
    };

    EleType GetValue(){
      return m_tElement;
    };

  };

  class Container{
  public:
    /* This workaround for memory allocation not dynamically bound */
    virtual void* NodeAlloc(){
      Node *p = new Node;
      printf("ALLOC:: nodesize = %d\n", sizeof(*p));
      return p;
    };
    /* End of workaround */
  };
};

template <class SuperCollab> class BINTREE: public SuperCollab{
 public:
  typedef typename SuperCollab::EleType EleType;
  typedef typename SuperCollab::Node SuperNode;
  typedef typename SuperCollab::Container SuperContainer;

  class Node: public SuperCollab::Node{
    Node *m_pPLink; //Node pointer data members.
    Node *m_pRLink;
    Node *m_pLLink;
  public:
    Node(): SuperNode::Node(){
      // Constructor can not be virtual, so for dynamically bound constructor
      // of Node, refer to virtual Container::NodeAlloc()
      m_pPLink = NULL;
      m_pRLink = NULL;
      m_pLLink = NULL;
    };

    /* This workaround for memory deallocation not dynamically bound */
    virtual ~Node(){// virtual destructor to delete composite Node.
    };
    /* End of workaround */

    Node *ParentLink(){
      return m_pPLink;
    };

    Node *RightLink(){
      return m_pRLink;
    };

    Node *LeftLink(){
      return m_pLLink;
    };

    void SetParentLink(Node *p){
      m_pPLink = p;
    };

    void SetRightLink(Node *p){
      m_pRLink = p;
    };

    void SetLeftLink(Node *p){
      m_pLLink = p;
    };

    void SetValue(EleType e){
```

```
    SuperNode::SetValue(e);
  };

};

class Container: public SuperCollab::Container{
  Node* m_pHeader; //Container data member
public:
  Container(){
    m_pHeader = NULL;
  };

  ~Container(){
    if (m_pHeader == NULL){
      printf("Empty Container...\n");
      return;
    }

    printf("Non-empty Container destructor called...\n");
    PruneBranch(m_pHeader->RightLink());
    PruneBranch(m_pHeader->LeftLink());
    delete m_pHeader;
    m_pHeader = NULL;
  };

  void Insert(EleType e){
    if (m_pHeader == NULL){
      //m_pHeader = new Node;// not dynamically bound, requires workaround.
      m_pHeader = (Node *) NodeAlloc();
      m_pHeader->SetValue(e);
      m_pHeader->SetLeftLink(NULL);
      m_pHeader->SetRightLink(NULL);
      m_pHeader->SetParentLink(m_pHeader);
      return;
    }

    Node *cp = m_pHeader;
    Node *ptemp;
    bool totheleft;
    while (TRUE){
      if(cp->GetValue() < e){//Insert to the right branch
        ptemp = cp->RightLink();
        if (ptemp == NULL){
          totheleft = FALSE;
          break;
        }
        cp = ptemp;
      }
      else if(cp->GetValue() > e){//Insert to the left branch
        ptemp = cp->LeftLink();
        if (ptemp == NULL){
          totheleft = TRUE;
        break;
      }
      cp = ptemp;
    }
    else{//it is there
      return;
    }
  }

  //up to here: ptemp = NULL.

  // ptemp = new Node; // not dynamically bound, require workaround.
  ptemp = (Node *) NodeAlloc();
  ptemp->SetValue(e);
  ptemp->SetLeftLink(NULL);
  ptemp->SetRightLink(NULL);
  ptemp->SetParentLink(cp);
  if (totheleft)
    cp->SetLeftLink(ptemp);
  else
    cp->SetRightLink(ptemp);

  return;
};

void Delete(EleType e){// delete a node with value e in current BINTREE layer.
  Node *pn = Search(e);
  if (pn == NULL)// nothing to delete
    return;

  Node *pcs;// pointer to closest smaller value,
            // i.e. the rightmost leaf node in the left branch.
  pcs = GetClosestSmallerValue(pn);
  if (pcs == NULL){// There is no left branch under node pointed by pn.
    pn->ParentLink()->SetRightLink(pn->RightLink());
    delete pn;// virtual Node destructor.
    pn = NULL;
  }
  else{// There exist a left branch under node pointed by pn.
    pn->SetValue(pcs->GetValue());
    if(pn->LeftLink() == pcs){// left branch under pn having 1 leaf only
      pn->SetLeftLink(NULL);
    }
    else{// Left branch having more than 1 leaf.
      pcs->ParentLink()->SetRightLink(NULL);
    }
    delete pcs;// workaround by virtual Node destructor.
    pcs = NULL;
  }

  return;
};
```

```
    Node* Search(EleType e){
      if (m_pHeader == NULL)
        return NULL;

      Node *cp = m_pHeader;
      while (cp != NULL){
        if(cp->GetValue() < e)
          cp = cp->RightLink();
        else if(cp->GetValue() > e)
          cp = cp->LeftLink();
        else{//it is there
          return cp;
        }
      }
      return NULL;//not found
    };

    /* This workaround for memory allocation not dynamically bound */
    virtual void* NodeAlloc(){
      Node *p = new Node;
      printf("BINTREE:: nodesize = %d\n", sizeof(*p));
      return p;
    };
    /* End of workaround */

  protected:
    virtual void PruneBranch(Node *pn){
      if (pn == NULL)
        return;
      PruneBranch(pn->RightLink());
      PruneBranch(pn->LeftLink());
      delete pn; // workaround with virtual Node destructor.
    }

    Node* GetClosestSmallerValue(Node *pn){
      Node *pl = pn->LeftLink();
      if (pl == NULL)// There is no left branch
        return NULL;

      Node *cp = pl;
      while (cp->RightLink() != NULL)
        cp = cp->RightLink();

      return cp;
    };

  };
};

template <class SuperCollab> class TIMESTAMP: public SuperCollab{
 public:
```

```
typedef typename SuperCollab::EleType EleType;
typedef typename SuperCollab::Node SuperNode;
typedef typename SuperCollab::Container SuperContainer;

class Node: public SuperCollab::Node{
  time_t m_nCreationTime;
  time_t m_nUpdateTime; //Node data members
public:
  Node(): SuperNode(){
    // Constructor can not be virtual, so for dynamically bound constructor
    // of Node, refer to virtual Container::NodeAlloc()
    m_nCreationTime = 0;
    m_nUpdateTime = 0;
  };

  /* This workaround for memory deallocation not dynamically bound */
  virtual ~Node(){// virtual destructor to delete composite Node.
  };
  /* End of workaround */

  bool MoreRecent(time_t t){
    printf("Checking for timestamp at Node...\n");
    if (t < m_nUpdateTime)
      return TRUE;
    else
      return FALSE;
  };

  void SetUpdateTime(time_t t){
    m_nUpdateTime = t;
  };

  void SetCreationTime(time_t t){
    m_nCreationTime = t;
  };

};

class Container: public SuperCollab::Container{
  time_t m_nUpdateTime; //Container data member
public:
  bool SearchNewer(EleType e, time_t t){
    Node *pn = (Node *)SuperContainer::Search(e);
    if (pn == NULL)
      return FALSE;
    return pn->MoreRecent(t);
  };

  void Insert(EleType e){// update timestamp.
    Node *pn = (Node *) SuperContainer::Search(e);
    SuperContainer::Insert(e);
```

```
      time_t curtime;
      char *ptime;
      time(&curtime);
      ptime = ctime(&curtime);
      printf("%s\n", ptime);

      if (pn != NULL){//the value is already in the tree.
        pn->SetUpdateTime(curtime);
      }
      else{ //newly inserted node.
        pn = (Node *) SuperContainer::Search(e);
        // of course now pn != NULL
        pn->SetCreationTime(curtime);
        pn->SetUpdateTime(curtime);
      }

    };

    /* This workaround for memory allocation not dynamically bound */
    virtual void* NodeAlloc(){
      Node *p = new Node;
      printf("TIMESTAMP:: nodesize = %d\n", sizeof(*p));
      return p;
    };
    /* End of workaround */

  };

};

template <class SuperCollab> class SIZEOF: public SuperCollab{
 public:

  typedef typename SuperCollab::EleType EleType;
  typedef typename SuperCollab::Node SuperNode;
  typedef typename SuperCollab::Container SuperContainer;

  class Node: public SuperCollab::Node{ //This class not needed in this layer.
  public:
    Node(): SuperNode(){
      // Constructor can not be virtual, so for dynamically bound constructor
      // of Node, refer to virtual Container::NodeAlloc()
    };

    /* This workaround for memory deallocation not dynamically bound */
    virtual ~Node(){// virtual destructor to delete composite Node.
    };
    /* End of workaround */
  };

  class Container: public SuperCollab::Container{
    int m_nCount; //Container data member
  public:
    Container():m_nCount(0), SuperCollab::Container(){//Constructor
    };

    void Insert(EleType e){
      Node *pn = (Node *) SuperContainer::Search(e);
      SuperContainer::Insert(e);
      if (pn == NULL) m_nCount++;
      return;
    };

    void Delete(EleType e){
      Node *pn = (Node *) SuperContainer::Search(e);
      if (pn != NULL){
        SuperContainer::Delete(e);
        m_nCount--;
      }
      return;
    }

    int Size(){
      return m_nCount;
    };

    /* This workaround for memory allocation not dynamically bound */
    virtual void* NodeAlloc(){
      Node *p = new Node;
      printf("SIZEOF:: nodesize = %d\n", sizeof(*p));
      return p;
    };
    /* End of workaround */

  };

};

#endif
```

```
// --------------------------------
// Main program file Collab4.cpp
// --------------------------------

#ifndef __OPENDS_H__
#include "OpenDS.h"
#endif

#include <stdio.h>
#include <unistd.h>

typedef SIZEOF< TIMESTAMP< BINTREE< ALLOC< int > > > > Collab4Int;
typedef Collab4Int::Container Collab4Container;
typedef Collab4Int::Node Collab4Node;

main(int argc, char *argv[]){
  Collab4Container cont;
  Collab4Node node;

  int i = 3;

  cont.Insert(3);
  sleep(1);
  cont.Insert(-1);
  sleep(1);
  cont.Insert(-3);
  sleep(3);
  cont.Insert(-1);
  sleep(2);

  bool recent = cont.SearchNewer(i, 20000000000);
  if (recent == TRUE)
    printf("Current node in container is newer!\n");
  else
    printf("Current node in container is older!\n");

  printf("OpenDS size: %d\n", cont.Size());

  cont.Delete(-1);
  printf("After deleting, OpenDS size: %d\n", cont.Size());
}
```

```
// --------------------------------
// Makefile of the C++-based project
// --------------------------------

OBJS = Collab4.o
OBJS_DBG = Collab4_dbg.o
CC = g++
DEBUG_FLAG = -g # empty now, but assign -g for debugging

Collab4 : ${OBJS}
$(CC) -o Collab4 ${OBJS}
clean:
rm *~
Collab4.o : Collab4.cpp OpenDS.h
$(CC) -c Collab4.cpp

Collab4_dbg : ${OBJS_DBG}
$(CC) $(DEBUG_FLAG) -o $@ ${OBJS_DBG}
Collab4_dbg.o : Collab4.cpp OpenDS.h
$(CC) $(DEBUG_FLAG) -c -o $@ Collab4.cpp
```

B.2 Java and Hyper/J

The source files written in Java and Hyper/J for the data structure are in the followings:

- Element abstract class file, `Element.java`
- Integral element class file, `IntElement.java`
- String element class file, `StrElement.java`
- Node class in Alloc collaboration, `/Alloc/Node.java`
- Container class in Alloc collaboration, `/Alloc/Container.java`
- Main testing program, `/Alloc/OpenDS.java`
- Node class in BinTree collaboration, `/BinTree/Node.java`
- Container class in BinTree collaboration, `/BinTree/Container.java`
- Node class in TimeStamp collaboration, `/TimeStamp/Node.java`
- Container class in TimeStamp collaboration, `/TimeStamp/Container.java`
- Node class in SizeOf collaboration, `/SizeOf/Node.java`
- Container class in SizeOf collaboration, `/SizeOf/Container.java`
- Concern mapping files, `/Alloc/concerns.cm`, `/BinTree/concerns.cm`, `/TimeStamp/concerns.cm` and `/SizeOf/concerns.cm`.
- Hyperspace file, `OpenDS.hs`
- Hypermodule to create a AB data structure, `AB.hm`
- Hypermodule to create a ABT data structure, `ABT.hm`
- Hypermodule to create a ABS data structure, `ABS.hm`
- Hypermodule to create a ABTS data structure, `ABTS.hm`
- Project makefile, `Makefile`

```
// -----------------------------------------
// Element class file definition Element.java
// -----------------------------------------

package collab.Element;

public abstract class Element{
    public abstract int Compare(Element m);
    public abstract String ToString();
};   // END class Element
```

```
// -----------------------------------------
// Integral element class file IntElement.java
// -----------------------------------------

package collab.Element;

public class IntElement extends Element{
    int m_nValue;
    public IntElement(int n){
        System.out.println("Instantiating Element with IntElement...");
        m_nValue = n;
    };

    int Value(){return m_nValue;};

    public String ToString(){
        String temp = new String();
        return temp.valueOf(m_nValue);
    }

    public int Compare(Element m){
        if(m instanceof IntElement){
            IntElement temp = (IntElement) m;
            if (m_nValue < temp.Value())
                return -1;
            else if (m_nValue == temp.Value())
                return 0;
            else
                return 1;
        }
        else
            return -2; // incomparable
    };
};   // END class IntElement
```

```
// ----------------------------------------
// String element class file StrElement.java
// ----------------------------------------

package collab.Element;

public class StrElement extends Element{
    String m_sValue;
    public StrElement(int n){
        System.out.println("Instantiating Element with StrElement...");
        m_sValue = new String() + n;
    };

    String Value(){return m_sValue;};

    public String ToString(){return m_sValue;};

    public int Compare(Element m){
        if(m instanceof StrElement){
            StrElement temp = (StrElement) m;
            if (m_sValue.compareTo(temp.Value()) < 0)
                return -1;
            else if (m_sValue.compareTo(temp.Value()) == 0)
                return 0;
            else
                return 1;
        }
        else
            return -2; // incomparable
    };
};   // END class StrElement
```

```
// -------------------------------------------------
// Node class in Alloc collaboration - file Node.java
// -------------------------------------------------

package collab.Alloc;

import collab.Element.*;

public class Node{

    public Node(int n, String s){
        if (s.compareTo("0") == 0)
            m_tElem = new StrElement(n);
        else
            m_tElem = new IntElement(n);
    };

    Element Value(){
        return m_tElem;
    };

    void SetValue(Element e){
        m_tElem = e;
    };

    private Element m_tElem;
}; // END class Node
```

```
// ---------------------------------------------------------------
// Container class in Alloc collaboration - file Container.java
// ---------------------------------------------------------------

package collab.Alloc;

import collab.Element.*;
import java.io.*;

public class Container{

    public Node AllocNode(int n, String s){
        Node res = new Node(n, s);
        return res;
    };

    void Process(Element e){
        System.out.println("Allocating layer is active.");
    }
}; // END class Container
```

```
// ------------------------------------------------------
// OpenDS class for main testing program - file OpenDS.java
// ------------------------------------------------------

package collab.Alloc;

import java.io.*;
import java.util.Random;
import collab.Element.*;

public class OpenDS{

    public static void main ( String[] args ){
        if(args.length < 1){
            System.out.println("Usage: java collab.Alloc.OpenDS opt [loopcount]");
            return;
        }
        m_nOption = args[0];

        int loopcount;

        if (args.length < 2)
            loopcount = 3;
        else
            loopcount = Integer.valueOf(args[1]).intValue();

        m_cCont = new Container();

        for ( int i = 1; i <= loopcount; i++ ){
               Node anode = build();
               Element elem = anode.Value();

               System.out.println("Created node with value of " + elem.ToString());

               try{
                   m_cCont.Process(elem);
                   Thread.sleep(2000);
               }
               catch ( Exception e ){
                   System.out.println ( "*** Exception in inserting: "
                       + e.getMessage());
               };
        };   // END loop, for i

    };   // END main

    public static Node build(){
        Random rn = new Random();
        int num = rn.nextInt();
        num = (num >= 0) ? num % 10 : (-num) % 10;

        return m_cCont.AllocNode(num, m_nOption);
    };   // END build

    private static Container m_cCont;
    private static String m_nOption;
};   // END class OpenDS
```

```
// ---------------------------------------------------
// Node class in BinTree collaboration - file Node.java
// ---------------------------------------------------

package collab.BinTree;

import collab.Element.*;

public class Node{ // class fragment for BinTree-related collaboration.

    void SetLeftLink(Node n){
        m_pLLink = n;
    };

    Node LeftLink(){
        return m_pLLink;
    };

    void SetRightLink(Node n){
        m_pRLink = n;
    };

    Node RightLink(){
        return m_pRLink;
    };

    void SetParentLink(Node n){
        m_pPLink = n;
    };

    Node ParentLink(){
        return m_pPLink;
    };

    Element Value(){
    //dummy implementation, overridden by Node::Value() in Alloc collab.
        return null;
    };

    void SetValue(Element e){
    //dummy implementation, overidden by Node::SetValue() in Alloc collab.
        return;
    };

    private Node m_pLLink, m_pRLink, m_pPLink;
}; // END class Node
```

```
// -------------------------------------------------------------
// Container class in BinTree collaboration - file Container.java
// -------------------------------------------------------------

package collab.BinTree;

import java.io.*;
import collab.Element.*;

public class Container{ // class fragment for BinTree-related collaboration.
    Container(){
        m_bAlreadyThere = false;
        m_pHeader = null;
    };

    void Process(Element e){
        System.out.println("BinTree layer is active.");
        InsertNode(e);
    };

    Node FindNode(Element e){
    // return a pointer to that node in tree, if not found: null.
        if(m_pHeader == null)
            return null;

        Node cp = m_pHeader;
        Element he;
        int comp;

        while (cp != null){
            he = cp.Value();
            comp = e.Compare(he);

            if (comp == 1)// search value is greater.
                cp = cp.RightLink();
            else if (comp == -1)// search value is smaller.
                cp = cp.LeftLink();
            else// it is here
                return cp;
        }
        return null;// not found.
    };

    void DeleteNode(Element e){
        Node p = FindNode(e);
        if (p == null)// nothing to delete
            return;

        Node pcs;//pointer to the closest smaller value,
                 //i.e. the rightmost leaf node in the left branch of p.
        pcs = GetClosestSmallerValue(p);
        if (pcs == null){// there is no left branch under node pointed by p.
```

```
            p.ParentLink().SetRightLink(p.RightLink());
        }
        else{// there exists a left branch under node p.
            p = pcs;
            if (p.LeftLink() == pcs)// one leaf only in left branch
                p.SetLeftLink(null);
            else
                pcs.ParentLink().SetRightLink(null);

        }
        return;
    };

    void InsertNode(Element e){
        m_bAlreadyThere = false;// assume that this node is not in tree yet.

        if(m_pHeader == null){
            System.out.println("Init tree.");
            m_pHeader = new Node();
            m_pHeader.SetValue(e);
            m_pHeader.SetLeftLink(null);
            m_pHeader.SetRightLink(null);
            m_pHeader.SetParentLink(m_pHeader);
            // top tree node points to itself as its parent
        }
        else{
            Node cp = m_pHeader;
            Node ptemp;
            boolean totheleft;

            Element he;
            int comp;

            while (true){
                he = cp.Value();
                comp = e.Compare(he);

                if(comp == 1){// value to insert is greater.
                    System.out.println("---> Inserting to right branch");
                    ptemp = cp.RightLink();
                    if (ptemp == null){
                        totheleft = false;
                        break;
                    }
                    cp = ptemp;
                }
                else if(comp == -1){// value to insert is smaller.
                    System.out.println("---> Inserting to left branch");
                    ptemp = cp.LeftLink();
                    if (ptemp == null){
                        totheleft = true;
                        break;
                }
                cp = ptemp;
            }
            else{// it is there.
                m_bAlreadyThere = true;
                return;
            }
        }

        // up to here ptemp = null
        ptemp = new Node();
        ptemp.SetValue(e);
        ptemp.SetLeftLink(null);
        ptemp.SetRightLink(null);
        ptemp.SetParentLink(cp);
        if (totheleft)
            cp.SetLeftLink(ptemp);
        else
            cp.SetRightLink(ptemp);

        return;
    }
};

Node GetClosestSmallerValue(Node p){
    Node pl = p.LeftLink();
    if (pl == null)// there is no left branch.
        return null;

    Node cp = pl;
    while (cp.RightLink() != null)
        cp = cp.RightLink();

    return cp;
};

private boolean m_bAlreadyThere;// synchronization point with SizeOf layer.
private Node m_pHeader = null;
}; // END class Container
```

```
// ---------------------------------------------------
// Node class in TimeStamp collaboration - file Node.java
// ---------------------------------------------------

package collab.TimeStamp;

import java.util.*;
import java.io.*;

public class Node{ // class fragment for TimeStamp-related collaboration.
    public Node(int n, String s){ // option s is not used.
        Calendar c = Calendar.getInstance();
        m_dUpdatetime = m_dTimestamp = c.getTime();
        System.out.println(n + " is created at " + m_dTimestamp.toString());
    };

    public void UpdateTime(Date d){
        m_dUpdatetime = d;
        System.out.println("Node is updated at " + m_dUpdatetime.toString());
    };

    private Date m_dTimestamp;
    private Date m_dUpdatetime;
}; // END class Node
```

```
// ------------------------------------------------------------
// Container class in TimeStamp collaboration - file Container.java
// ------------------------------------------------------------

package collab.TimeStamp;

import java.io.*;
import java.util.*;
import collab.Element.*;

public abstract class Container{
// class fragment for TimeStamp-related collaboration.
    Container(){
        Calendar c = Calendar.getInstance();
        m_dTimestamp = c.getTime();
    };

    void Process(Element e){
        System.out.println("TimeStamp layer is active:
            Container created at " + m_dTimestamp.toString());
    };

    void InsertNode(Element e){
        // Upper layer will insert the element into tree via InsertNode().
        // That InsertNode() will run before this InsertNode().
        // This layer only set the update time for that node in the tree.
        Calendar c = Calendar.getInstance();
        Date d = c.getTime();// get update time

        Node n = FindNode(e);// retrieve the pointer to that node in tree.
        n.UpdateTime(d);
    };

    abstract Node FindNode(Element e);

    private Date m_dTimestamp;
}; // END class Container
```

```
// ---------------------------------------------------
// Node class in SizeOf collaboration - file Node.java
// ---------------------------------------------------

package collab.SizeOf;

import java.io.*;

public class Node{ // class fragment for SizeOf-related collaboration.
    public Node(int n, String s){ // option s is not used.
        System.out.println(n + " is created at SizeOf layer");
    };
}; // END class Node
```

```
// ------------------------------------------------------------
// Container class in SizeOf collaboration - file Container.java
// ------------------------------------------------------------

package collab.SizeOf;

import java.io.*;
import collab.Element.*;

public abstract class Container{// class fragment for SizeOf-related collaboration.
    Container(){
        m_nCount = 0;
        m_bAlreadyThere = false;
    };

    void Process(Element e){
        System.out.println("SizeOf layer is active: Size = " + m_nCount);
    };

    void InsertNode(Element e){
        if(m_bAlreadyThere == false){// node is not there before
            m_nCount++;
        }
        else{
            System.out.println("Node is there");
        }
    };

    private boolean m_bAlreadyThere;// synchronization point with BinTree layer.
    private int m_nCount;
}; // END class Container
```

```
// ------------------------------------------------------------
// Concern mapping files in various collaborations - concerns.cm
// ------------------------------------------------------------

// Concern mapping file of Element package - /Element/concerns.cm
package collab.Element : Feature.Alloc

// Concern mapping file of Alloc package - /Alloc/concerns.cm
package collab.Alloc : Feature.Alloc

// Concern mapping file of BinTree package - /BinTree/concerns.cm
package collab.BinTree : Feature.BinTree

// Concern mapping file of TimeStamp package - /TimeStamp/concerns.cm
package collab.TimeStamp : Feature.TimeStamp

// Concern mapping file of SizeOf package - /SizeOf/concerns.cm
package collab.SizeOf : Feature.SizeOf
```

```
// ------------------------------------------------------------------
// Hyperspace file specifying which packages are composable - OpenDS.hs
// ------------------------------------------------------------------

hyperspace CollabHyperspace
        composable class collab.Element.*;
        composable class collab.Alloc.*;
        composable class collab.BinTree.*;
        composable class collab.TimeStamp.*;
        composable class collab.SizeOf.*;
```

```
// -------------------------------------------------------
// Hypermodule file to compose 2 layers A and B - AB.hm
// -------------------------------------------------------

hypermodule AllocBinTree
    hyperslices:
        Feature.Alloc,
        Feature.BinTree;

    relationships:
        mergeByName;

        override action Feature.BinTree.Node.Value with \
                 action Feature.Alloc.Node.Value;
        override action Feature.BinTree.Node.SetValue with \
                 action Feature.Alloc.Node.SetValue;
end hypermodule;
```

```
// ------------------------------------------------------
// Hypermodule file to compose 3 layers A, B and T - ABT.hm
// ------------------------------------------------------

hypermodule AllocBinTreeTimeStamp
    hyperslices:
        Feature.Alloc,
        Feature.BinTree,
        Feature.TimeStamp;

    relationships:
        mergeByName;

        override action Feature.BinTree.Node.Value with \
                 action Feature.Alloc.Node.Value;
        override action Feature.BinTree.Node.SetValue with \
                 action Feature.Alloc.Node.SetValue;
end hypermodule;
```

```
// -----------------------------------------------------------
// Hypermodule file to compose 3 layers A, B and S - ABS.hm
// -----------------------------------------------------------

hypermodule AllocBinTreeSizeOf
    hyperslices:
        Feature.Alloc,
        Feature.BinTree,
        Feature.SizeOf;

    relationships:
        mergeByName;

        override action Feature.BinTree.Node.Value with \
                 action Feature.Alloc.Node.Value;
        override action Feature.BinTree.Node.SetValue with \
                 action Feature.Alloc.Node.SetValue;
end hypermodule;
```

```
// ---------------------------------------------------------
// Hypermodule file to compose 4 layers A, B, T and S - ABTS.hm
// ---------------------------------------------------------

hypermodule AllocBinTreeTimeStampSizeOf
    hyperslices:
        Feature.Alloc,
        Feature.BinTree,
        Feature.TimeStamp,
        Feature.SizeOf;
    relationships:
        mergeByName;

        override action Feature.BinTree.Node.Value with \
                 action Feature.Alloc.Node.Value;
        override action Feature.BinTree.Node.SetValue with \
                 action Feature.Alloc.Node.SetValue;
end hypermodule;
```

```
// ---------------------------------------
// Makefile of the Java and Hyper/J project
// ---------------------------------------

# Environment declaration
JC = javac # Java compiler
JI = java  # Java interpreter
HYPERJ = com.ibm.hyperj.hyperj
HS_FLAG = -hyperspace
CM_FLAG = -concerns
HM_FLAG = -hypermodules
OUTPUT_FLAG = -output
VERBOSE_FLAG = -verbose
DEBUG_FLAG = -Xdebug # empty now, but assign -Xdebug for debugging
CURRENT = .
CUR_DIR = $(CURRENT)
ELEMENT_DIR = $(CUR_DIR)/Element
ALLOC_DIR = $(CUR_DIR)/Alloc
BINTREE_DIR = $(CUR_DIR)/BinTree
TIMESTAMP_DIR = $(CUR_DIR)/TimeStamp
SIZEOF_DIR = $(CUR_DIR)/SizeOf
VPATH = .:$(ELEMENT_DIR):$(ALLOC_DIR):$(BINTREE_DIR):$(TIMESTAMP_DIR):$(SIZEOF_DIR)

# Pattern rule
%.class : %.java
$(JC) $<

# No need ALLOC_DIR below since they are already handled by pattern rule.
Node.class : $(BINTREE_DIR)/Node.class \
             $(TIMESTAMP_DIR)/Node.class \
             $(SIZEOF_DIR)/Node.class

Container.class : $(BINTREE_DIR)/Container.class \
                  $(TIMESTAMP_DIR)/Container.class \
                  $(SIZEOF_DIR)/Container.class

# Configuration
ABTS: cleartarget Node.class Container.class Element.class IntElement.class \
      StrElement.class OpenDS.class
$(JI)  $(HYPERJ) \
$(HS_FLAG) OpenDS.hs \
$(CM_FLAG) $(ELEMENT_DIR)/concerns.cm \
$(ALLOC_DIR)/concerns.cm \
$(BINTREE_DIR)/concerns.cm \
$(TIMESTAMP_DIR)/concerns.cm \
$(SIZEOF_DIR)/concerns.cm \
$(HM_FLAG) ABTS.hm \
$(VERBOSE_FLAG)
cp AllocBinTreeTimeStampSizeOf/collab/Alloc/*.class Alloc
echo "To test, type the command: java collab.Alloc.OpenDS opt [loopcount]"

ABT: cleartarget Node.class Container.class Element.class IntElement.class \
     StrElement.class OpenDS.class
$(JI)  $(HYPERJ) \
$(HS_FLAG) OpenDS.hs \
$(CM_FLAG) $(ELEMENT_DIR)/concerns.cm \
$(ALLOC_DIR)/concerns.cm \
$(BINTREE_DIR)/concerns.cm \
$(TIMESTAMP_DIR)/concerns.cm \
$(HM_FLAG) ABT.hm \
$(VERBOSE_FLAG)
cp AllocBinTreeTimeStamp/collab/Alloc/*.class Alloc
echo "To test, type the command: java collab.Alloc.OpenDS opt [loopcount]"

ABS: cleartarget Node.class Container.class Element.class IntElement.class \
     StrElement.class OpenDS.class
$(JI)  $(HYPERJ) \
$(HS_FLAG) OpenDS.hs \
$(CM_FLAG) $(ELEMENT_DIR)/concerns.cm \
$(ALLOC_DIR)/concerns.cm \
$(BINTREE_DIR)/concerns.cm \
$(SIZEOF_DIR)/concerns.cm \
$(HM_FLAG) ABS.hm \
$(VERBOSE_FLAG)
cp AllocBinTreeSizeOf/collab/Alloc/*.class Alloc
echo "To test, type the command: java collab.Alloc.OpenDS opt [loopcount]"

AB : cleartarget Node.class Container.class Element.class IntElement.class \
     StrElement.class OpenDS.class
$(JI)  $(HYPERJ) \
$(HS_FLAG) OpenDS.hs \
$(CM_FLAG) $(ELEMENT_DIR)/concerns.cm \
$(ALLOC_DIR)/concerns.cm \
$(BINTREE_DIR)/concerns.cm \
$(HM_FLAG) AB.hm \
$(VERBOSE_FLAG)
cp AllocBinTree/collab/Alloc/*.class Alloc
echo "To test, type the command: java collab.Alloc.OpenDS opt [loopcount]"

cleartarget:
rm -f $(ALLOC_DIR)/*.class
clean:
rm -f *~
rm -f $(ELEMENT_DIR)/*~
rm -f $(ALLOC_DIR)/*~
rm -f $(BINTREE_DIR)/*~
rm -f $(TIMESTAMP_DIR)/*~
rm -f $(SIZEOF_DIR)/*~
clearall:
rm -f $(ELEMENT_DIR)/*.class
rm -f $(ALLOC_DIR)/*.class
rm -f $(BINTREE_DIR)/*.class
rm -f $(TIMESTAMP_DIR)/*.class
rm -f $(SIZEOF_DIR)/*.class
```

Bibliography

[1] Martin Abadi and Luca Cardelli. *A theory of objects.* Springer, 1996.

[2] Maher Awad, Juha Kuusela, and Jurgen Ziegler. *Object-Oriented Technology for Real-Time Systems.* Prentice Hall, 1996.

[3] Jan Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach.* Addison-Wesley, 2000.

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

[5] B. MacDonald D. Batory, C. Johnson and D. v. Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proc. International Conference on Software Reuse*, July 2000.

[6] Patrick Donohoe, editor. *Software Product-Lines, Experience and Research Directions.* Kluwer Academic Publishers, 2000.

[7] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. Symposium on the Foundations of Software Engineering*, September 2001.

[8] K. Fisler, S. Krishnamurthi, and D. Batory. Verifying component-based collaboration designs. In *Proc. ICSE Workshop on Component-Based Software Engineering*, May 2001.

[9] Martin Fowler. *Refactoring: Improving the Designs of Existing Code.* Addison-Wesley Inc., 1999.

[10] J. Lamping et al. G. Kiczales. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming - ECOOP'97*, pages 220–242. Springer, 1997.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing, 1995.

[12] V.K. Garg and M.T. Ragunath. Concurrent regular expressions and their relationship to Petri nets. *Theoretical Computer Sciences*, 96:285–304, 1992.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification.* Addison-Wesley, 2nd edition, 2000. Free download from Sun Microsystems.

[14] Paul Harmon and Mark Watson. *Understanding UML The Developer's Guide with a Web-Based Application in Java.* Morgan Kaufmann, 1998.

[15] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *ACM OOPSLA*, pages 303 – 311, 1993.

[16] Michael G. Hinchey and Jonathan P. Bowen. *Applications of Formal Methods.* Prentice Hall, 1995.

[17] I. M. Holland. Specifying reusable components using contracts. In *Proc. European Conference on Object-Oriented Programming - ECOOP'92*, pages 287–308. Springer-Verlag, 1992.

[18] Software Engineering Institute. *A Framework for Software Product-Line Practice.* Carnegie Mellon University, 2nd edition, 1999.

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

[20] Leslie Lamport. *LaTeX- A Document Preparation System.* Addison-Wesley Publishing Co., 1986.

[21] Karl Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method.* PWS Publishing Co., 1996.

[22] S. Krishnamurthi M. Flatt and M. Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages*, 1998.

[23] Jeff Magee and Jeff Kramer. *Concurrency - State Models and Java Program.* John Wiley & Sons, 1999.

[24] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1997.

[25] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus.* Addison-Wesley, 1989.

[26] 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.

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

[28] G.E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming.* Cambridge University Press, 1988.

[29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice Hall, 1991.

[30] James Rumbaughm, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language - Reference Manual.* Addision-Wesley, 1999.

[31] Mary Shaw and David Garlan. *Software Architecture - Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

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

[33] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[34] P. Tarr and H. Ossher. *Hyper/J(TM) User and Installation Manual.* IBM Research, IBM Corp., 2000.

[35] 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.

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

[37] L. Toduka and D. Batory. Automating three modes of evolution for object-oriented software architectures. In *5th Conference on Object-Oriented Technologies, (COOTS'99)*, May 1999.

[38] L. Toduka and D. Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, October 1999.

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

[40] 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.

[41] David M. Weiss and Chi Tau R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.

[42] Glynn Winskel. *The Formal Semantics of Programming Languages, An Introduction.* The MIT Press, 1993.

Acknowledgement

This thesis has been prepared according to LaTeX version 2.09 [20] and compiled with LaTeX2e. This software makes the typesetting for my scientific report much easier and more beautiful. It solves many difficult problems in rearranging text and mathematical symbols as I usually faces in MS-Word. The illustrated example is tested with gcc 2.96 and Java 2 Standard Edition 1.4.0 Beta 3 on RedHat Linux 7.1 operating system.

This thesis is my work in two-year time. It can not be completed with the help and inspiration of so many people. First of all, I want to thank all my colleagues at Foundations of Software Lab - JAIST. Without their excellent cooperation and facility maintenance, I could not finalize this work as it is. In particular, I want to thank Prof. Katayama Takuya and Associate Prof. Gondow Katsuhiko, who are my direct supervisors on this research. Prof. Katayama Takuya has inspired my research and given many hints for my research direction. In addition, the LAN and computing facility served during the thesis compilation are very well managed and constantly upgraded to the best equipment available by Associate Aoki Toshiaki and many others. Sometimes, their continuous facility upgrade causes some slight discomfort as my papers are due. However, after the service is completed, everything goes more smoothly than before and well in order. All in all, they have done an excellent job.

I have been received many advices from Okazaki Mitsutaka and Kawashima Hayato. They are really interesting guys who showed the way to adapt with a typical Japanese living style, especially research life at JAIST which is usually boring to many foreigners.

I must admit that my research has been pushed ahead with the exceptional computing infrastructure at JAIST. I have never been to such a working environment which is so modern, environment-friendly like this. The working condition in JAIST is truly top class of the world. For this reason, I need to thank all staff members of JAIST who are working continuously for the best working condition to all students.

Furthermore, I could not forget the help of Vietnamese group of researchers at JAIST, headed by Prof. Ho Tu Bao. Without his effort and guidance, many Vietnamese researchers in here including me could not have been this successful.

Finally, above all, my enthusiasm would not be very high without the faith in me, encouragement and support of my family for so long. Neither do my achievements so far. This thesis is dedicated to my parents who gives me so much.

References in each chapter:
Chapter 1: [29, 24, 30, 34, 35, 19, 10].
Chapter 2: [23, 31, 21, 22, 4, 15, 32, 39, 19, 40, 27, 12].
Chapter 3: [16, 1].
Chapter 4: [16, 8, 7, 5].
Chapter 5: [11, 33, 13, 9, 22, 4, 32, 34, 35, 26].
Chapter 6: [21, 15, 35, 10, 36].
Related materials: [42, 14, 2, 41, 18, 3, 6, 28, 25, 9, 17, 37, 38].