Chapter 4

Software Composition from Available Components

Today's business operations and/or business processes depend a lot on information systems. These information systems, especially large-scale information systems, have been developed in such a way as :

1. Analyzing domain problems and users' requirements.
2. Building models of the above analysis results.
3. Decomposing the models into elements (functional decomposition), then transforming these elements into software modules or components.

As far as domain problems are stable, the system that was built in the above way is adaptable to the domain problems, in other words, it satisfies or meets the requirements of the domain, since we can verify the adaptability step by step from the requirements to the final software system. A system of this kind is often called a *monolithic system*, which means the system is a solution only for the problems, and any part of it may not be reused for other problems.

However, we are faced with several difficulties caused by its stiff structure when enhancing and/or changing a monolithic system for adapting it to such new requirements as improved business processes, restructured organizations, new business areas and new technologies [75].

In order to overcome the problems caused by the stiff structure of monolithic systems, many efforts have been made to "componentize" the systems, that is, to build the systems by reusable or replaceable *building blocks*, often referred to as *software components* or *components* in short.

Such style of system developments is called Component Based Software Development (CBSD), and it shifts the development emphasis from programming to composing software systems.

A large number of the above building blocks or components recently become available to the general public, and can be bought by anyone. Such kinds of components are referred to as *Commercial-Off The Shelf* (COTS) components or *shrinkwrapped* software.

Another approach to this problem is package based development, which utilize software packages that cover some of significant business operations in an enterprise. So-called ERP (Enterprise Resource Planning) is one kind of these packages.

These methods indeed improve the stiff structure of monolithic systems. However, it is hard to know whether components or packages meet the requirements, since they are not the result of functional decomposition from the requirements. In addition, there is no systematic and formal way to construct software systems which are adaptable to the requirements by combining the selected components.

There have been many theoretical and practical researches on software composition [40], [53]. They include frameworks, architectures, composition models, composition languages, tools, and so on [30], [52].

They are effective in *system oriented* applications, such as graphical user interfaces, communication control, database management systems and so forth. However, when applying them to *business oriented* applications, such as enterprise back-office applications, we are faced with at least three difficulties.

The first is complexity of requirements and components, in functionality and interfaces, because of the variety of business operations. The second is the difference in terminology and concepts between requirements and components, since requirements are elicited from a business realm, while the components are provided by a software realm, and those realms are too different from each other. Those two difficulties make component mining difficult.

The last is that we have to deal with the behavior of an enterprise, which might last long time, e.g. more than a month. There are few methodologies suitable to composing software systems which are adaptable to the behavioral aspects of an enterprise.

In this chapter, we discuss a systematic and formal way to construct software systems from CPN models obtained in the previous sections, using available components. There are four major activities in CBSD, that is, *component qualification*, *component adaptation*, *assembling components to systems* and *software evolution* [37].

This thesis deals with the above second and third activities, since they are the core activities in software construction, and the succeeding activities of enterprise modeling.

We assume all the components are qualified in some ways, and do not deal with such software evolutions as software maintenance or enhancement.

The activity "component adaptation" is that for selecting adaptable components to the model from various sources, while the activity "assembling components into systems" is that for combining the components in appropriate ways. We refer to the former as *component mining*, and the latter as *software synthesis*. The software synthesis is defined as a part of *software composition*.

4.1 A Common Notation between Requirements and Components

The first step of software composition is to select the adaptable software components from various sources, e.g. COTS (Commercial Off-The-Shelf Software), software packages, legacy codes and so on. It is one of the major problems in component selection that there usually are notational differences between requirements and components. Those differences often make it difficult to evaluate adaptability of components to requirements.

Therefore, we should express both requirements and components in a common notation, that is common syntax and semantics. Algebraic specification is one of the most suitable notations for component adaptability evaluation, since it has rigorous concepts and definition of equivalency between models expressed by it.

We use Σ algebra, which is the most basic form of algebraic specification, to express requirements and components, since the thesis does not assume any specific functional notations, e.g. *terms* or *equations*. Σ algebra provides an interpretation for the signature $\Sigma = (S, \Omega)$, where S is a set of sorts, and Ω is an $S^* \times S$ sorted set of operation names. S^* is the set of finite sequences of elements of S. A Σ algebra is an algebra (A, F), where

$A = \{a_\sigma | \sigma \in S\}$ (a set of carriers) and

$F = \{f_A | f \in \Omega_{\sigma_1 \ldots \sigma_n, \sigma}\} \quad f_A : A_{\sigma_1} \times A_{\sigma_n} \rightarrow A_\sigma$.

S-sorted function f_A is said to have arity $\sigma_1 \ldots \sigma_n$ and result sort σ.

When requirements and components are expressed in Σ algebra, adaptability of the components to the requirements can be evaluated by Σ homomorphism, if both the requirements and the components are expressed in Σ algebra [64], [69].

Σ homomorphism $\eta = \{\eta_\sigma | \sigma \in S, \eta_\sigma : A_\sigma \rightarrow B_\sigma\}$ is a family of functions such that

$$\forall f \in \Omega_{\sigma_1 \ldots \sigma_n, \sigma}, \forall a_i \in A_{\sigma_i}$$
$$\eta_\sigma(f_A(a_1, \ldots, a_n)) = f_B(\eta_{\sigma_1}(a_1), \ldots, \eta_{\sigma_n}(a_n))$$
$$(n \geq 0)$$

where $\mathcal{A}$=(A, F) and $\mathcal{B}$=(B, G) are Σ algebras. f_A and f_B are the functions in F and G respectively, or elements of A and B if $n = 0$.

In component adaptability evaluation, the above algebra $\mathcal{A}$ represents require-

ments, while the above $\mathcal{B}$ represents components. We discuss this in the following sections.

4.1.1 Transforming Requirements Model into an Algebraic Form

After expressing requirements in CPN, the next step is to transform them into Σ algebra. Arc functions in CPN models could become S-sorted functions in Σ algebra. However, they are often defined in a complicated way to compose the model with fewer arcs. Therefore those functions might be reduced to more simplified ones in order to compose Σ algebra. This simplification will be achieved by composing an S-sorted function from a pair of one output arc function and several input arc functions relating to it [24]. We can derive S-sorted functions and carriers uniquely from a given CPN by this method.

In order to construct Σ algebra from a CPN model $CPN = (S, P, T, A, N, C, G, E, I)$, we first focus on one transition "t" with input places "$p_1, \ldots, p_m$", input arc functions [1] "$f_1, \ldots, f_m$", output places "$p'_1, \ldots, p'_{m'}$", and output arc functions "$f'_1, \ldots, f'_{m'}$".

Each f_i transforms a token $x_i \in C(p_i)$ in input place p_i into a token or tokens $f_i(x_i)$. Similarly, each f'_j produces a token or tokens according to the input tokens to the transition produced by the input arc functions, hence a token or tokens produced by f'_j can be denoted by $f'_j(f_1(x_1), \ldots, f_m(x_m))$.

Since those f_i and f'_j are kinds of operations over the color sets, we can regard the color sets as carriers in Σ algebra. If each $f_i(x_i)$ and $f'_j(f_1(x_1), \ldots, f_m(x_m))$ represent single tokens, that is, $f_i(x_i) \in C(p_i)$ and $f'_j(f_1(x_1), \ldots, f_m(x_m)) \in C(p'_j)$, we can define the carriers as $A_{\sigma_i} = C(p_i)$ and $A_\sigma = C(p'_j)$ respectively. S-sorted function g_j with the arity $\sigma_1 \ldots \sigma_m$ and the result sort σ is derived from them by defining $g_j(y_1, \ldots, y_m) = f'_j(f(x_1), \ldots, f_m(x_m))$, where $y_i = f_i(x_i) \in A_{\sigma_i}$ $(A_{\sigma_i} = C(p_i))$ and $g_i \in A_\sigma = C(p'_j)$.

However, f_i and f'_j could be a multi-set over $C(p_i)$ and $C(p'_j)$ respectively, therefore in this case, f_i and f'_j are in the form of:

$f_i(x_i) = n_{i1}`y_1 + n_{i2}`y_2 + \cdots$

$f'_j(f_1(x_1), \ldots, f_m(x_m)) = n_{j1}`z_1 + n_{j2}`z_2 + \cdots$

where $y_k \in C(p_i)$ and $z_k \in C(p'_j)$. In such case, the carriers should be regarded as $(C(p_i))^{n_{i1}} \times (C(p_i))^{n_{i2}} \times \cdots$ and $(C(p'_j))^{n_{j1}} \times (C(p'_j))^{n_{j2}} \times \cdots$. The term "$n_{i1}`y_1$" means that n_{i1} tokens which have the value y_1 are yielded by the function.

In addition, since a multi-set for each f_i and f'_j could be different according to the value of x_i, we must define multiple carriers for each f_i and f'_j in this case. For example, if an arc expression function is in the form of

[1] We refer to arc expression functions on input arcs to a transition as *input arc functions*, and those on output arcs from a transition as *output arc functions*.

$E(a) = if\ (conditions\ for\ x_i)\ then \ldots$
$else \ldots$

there could be two different multi-sets, and hence two different carriers would be generated.

From the above discussion, multiple S-sorted functions with different arities and result sorts could be derived from those f_i $(i = 1, \ldots, m)$ and f'_j. Let $\{A^t_\sigma\}$ be a set of carriers identified in transition t, and $\{g^t\}$ be a set of S-sorted functions identified in t. By identifying those sets of carriers and functions all over the transitions in the CPN model, we can construct Σ algebra $\mathcal{A}$=(A, F), where

$$A = \bigcup_{t \in T}\{A^t_\sigma\}, F = \bigcup_{t \in T}\{g^t\}.$$

For example, by applying this procedure to the transition t_1 in Figure 3.5 (Order Entry) and the related places and arcs, we get a Σ algebra (A, F), where

$A = \{A_{\sigma_1}, A_{\sigma_2}, A_{\sigma_3}, A_{\sigma_4}, A_{\sigma_5}, A_\sigma, A_{\sigma'}\}$
$F = \{g_1, g_2\}$
and
$A_{\sigma_1} = C_1, A_{\sigma_2} = C_3, A_{\sigma_3} = C_4, A_{\sigma_4} = C_5,$
$A_{\sigma_5} = C_6, A_\sigma = D_2, A_{\sigma'} = D_3$
$(g_1)_{\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5,\sigma}(x_1, x_2.x_3, x_4, x_5) = E(t_1 \rightarrow p_1)$
$(g_2)_{\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5,\sigma'}(x_1, x_2.x_3, x_4, x_5) = E(t_1 \rightarrow p_3)$

Function g_1 and g_2 have the arity (a sequence of sorts) $\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$. They have the result sort σ and σ' respectively.

4.1.2 Transforming Components into Algebraic Form

Each component basically transforms a set of inputs to an output or outputs. Therefore it seems possible to construct Σ algebra from components by identifying all the data types used as inputs and outputs, in addition to all the transformation rules performed by them.

However, each component could be designed in different granularity from each other. Some components could perform very complex functionality manipulating many kinds of data types with various transformation rules, while the others could do only simple functionality.

From adaptability evaluation viewpoint, it is desirable that each component is made in similar granularity. Therefore, we first decompose components into more primitive ones, in order to adjust them to the similar granularity levels.

CPN can be used for this purpose. A component is expressed in CPN with only one transition and several input/output places. Each input/output place represents a data type manipulated by the component. An output arc function expresses a primitive data transformation rule or rules performed by the component.

We can obtain Σ algebra from each CPN corresponding to a component in the same way we used for requirements. Let a Σ algebra obtained from a component C_i be $\mathcal{B}_i$=(B_i, G_i), where B_i is a set of sorts and G_i is a set of operations or functions. By combining those Σ algebras, that is, constructing Σ algebra $\mathcal{B}$=(B, G) as $B = \cup B_i$ and $G = \cup G_i$, we can obtain the Σ algebra representing a set of components which are available to build software systems.

Appendix B.5 shows an example of transformation of components into Σ algebra.

4.2 Mining Adaptable Components by Coordinating Conceptual Differences

Once both requirements and components are expressed in the form of Σ algebra, adaptability of the components to the requirements can be evaluated by Σ homomorphism between those algebras [26].

Σ homomorphism is defined on the premise that two algebras have a common signature $\Sigma = (S, \Omega)$, or in other words, there must be one-to-one correspondence between functions reside in those two algebras, along with the related sorts. However, in many cases, sorts in requirements are different from those in components, and it makes difficult to identify the common signature between them.

The difference in sorts between requirements and components mainly causes from the fact that the components are not the result of functional decomposition of the requirements, but the solutions for some generic problems which might be different from our specific problems.

For example, the sort "product name" in our example in Figure 3.5 and Table 3.6 might be implemented as a set of sorts "material name" and "part name" in some components. The other example is that sorts in components might be more computational ones like "integer" or "string".

In addition, there is another difficulty in identifying correspondence of functions between those two algebras. The problem is that each function could include unnecessary sorts in its arity. Those unnecessary sorts could be introduced due to unoptimized requirements or components from the viewpoint of functions.

In order to resolve the above problems, we introduce RST [57], [58]. In the following sections, we discuss how RST resolves the problems for selecting adaptable components.

4.2.1 Simplifying S-Sorted Functions by RST

Each S-sorted function, which is identified in Section 4.1 from the requirements CPN model, could have unnecessary carriers, or in other words, unnecessary sorts could be included in its arity.

The major reason for those unnecessary carriers is that the requirements are often built based on incomplete knowledge on the enterprise. Therefore, we need to eliminate those unnecessary carriers from each S-sorted function before adaptable components selection.

Let an S-sorted function in Σ algebra of the requirements be $g_{\sigma_1 \ldots \sigma_n, \sigma}$, and the involved carriers be $A_{\sigma_1}, \ldots, A_{\sigma_n}, A_\sigma$.

An S-sorted function can be regarded as a decision table, if all involved carriers are countable [2].

The decision table of such an S-sorted function is in the form of Table 4.1.

Table 4.1: A decision table of an S-sorted function

N	σ_1	σ_2	$\ldots$	σ_n	σ
1	v_{11}	v_{12}	$\ldots$	v_{1n}	v_1
2	v_{21}	v_{22}	$\ldots$	v_{2n}	v_2
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
x	v_{x1}	v_{x2}	$\ldots$	v_{xn}	v_x
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$

Each number in the column N is an index of the decision table. Each row represents data transformation of the S-sorted function $g_{\sigma_1 \ldots \sigma_n, \sigma}$, that is

$g_{\sigma_1 \ldots \sigma_n, \sigma}(v_{x1}, \ldots, v_{xn}) = v_x$

By RST we can find the unnecessary columns which have no meanings to determine the data transformation.

For any $x \in N$, let X_i and Y be $X_i = [x]_{\sigma_i}$ and $Y = [x]_\sigma$ respectively. X_i and Y are equivalent classes in N which satisfy

$\forall x' \in X_i$, $v_{x'i} = v_{xi}$ and $\forall x' \in Y$, $v_{x'} = v_x$.

If for all $x \in N$, the family $F = \{X_1, X_2, \ldots\}$ satisfies the two conditions:

$\cap F \subseteq Y$ and $\cap\{F - \{X_i\}\} \subseteq Y$,

X_i is called dispensable and the decision table shows the same data transformation even though we eliminate the column σ_i [57].

In this case, $g_{\sigma_1 \ldots \sigma_n, \sigma}$ and $g_{\sigma_1 \ldots \sigma_{i-1} \sigma_{i+1} \ldots \sigma_n, \sigma}$ are equivalent, that is :

$\forall a, b \in A_{\sigma_i}$

[2]This assumption is realistic for back-office applications.

$[g(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n) = g(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)]$
where A_{σ_i} is the corresponding carrier to the sort σ_i.

By eliminating a dispensable σ_i one after another until there is no dispensable attribute, we get the simplest decision table which is equivalent to the original one. This simplest one is called *reduct* of the original one.

The tables like Table 4.1 could be huge in size, however by introducing *subdomains* of the carriers in an S-sorted function, we can reduce them to practical size. A subdomain of a carrier A_{σ_i} is a class in terms of RST, if we regard A_{σ_i} as the universe. For example, suppose there are such subdomains in the carriers of the S-sorted function g_1 which we derived in Section 4.1.1 as:

$A_{\sigma_1} = O_1 \cup O_2 \cup O_3$
$A_{\sigma_2} = P_1 \cup P_2$
$A_{\sigma_3} = Q_1 \cup Q_2$
$A_{\sigma} = W_1 \cup W_2 \cup W_3$

where O_i means a subdomain consisting of similar orders, e.g. those from the same region. P_i, Q_i and W_i are also subdomains defined based on such similarity as the above. Assuming that those subdomains determine particular transformation rules of g_1, we can make a decision table by those subdomains instead of individual values, and can reduce the size of the table. For example, if we have the decision table like Table 4.2, we will identify the sort σ_3 to be *dispensable*, by the set calculus mentioned above, and conclude $(g_1)_{\sigma_1\sigma_2\sigma_3,\sigma} = (g_1)_{\sigma_1\sigma_2,\sigma}$.

Table 4.2: A decision table of a sample S-sorted function

N	σ_1	σ_2	σ_3	σ
1	O_1	P_1	Q_1	W_1
2	O_1	P_2	Q_2	W_2
3	O_2	P_2	Q_1	W_3
4	O_3	P_1	Q_1	W_3
5	O_1	P_1	Q_2	W_1
6	O_2	P_2	Q_2	W_3
7	O_1	P_2	Q_1	W_2
8	O_3	P_1	Q_2	W_3

A reduct represents the essential part of the original S-sorted function, and therefore we use reducts instead of the original functions for components adaptability evaluation.

4.2.2 Sorts Adjustment and Adaptable Components Mining

First, we focus on one reduct of an S-sorted function derived from the requirements. Let ϕ be a reduct with arity $\sigma_1 \ldots \sigma_m$ and result sort σ, which have the corresponding carriers $A_{\sigma_1}, \ldots, A_{\sigma_m}$ and A_σ. Each carrier can be regarded as representing a set of resources or information in the enterprise, and hence corresponds to a class in the universe U discussed in the previous section. We refer to this class as the *home class* of the carrier.

Let $K = (U, \mathrm{R})$ be the knowledge base which can make all the home classes for the reducts in the requirements. Assuming that the home classes are $X_i \subseteq U$ and $X \subseteq U$ for A_{σ_i} and A_σ respectively.

A_{σ_i} and A_σ are derived by a function I_{m_A}, which means

$A_{\sigma_i} = I_{m_A}(X_i)$ and $A_\sigma = I_{m_A}(X)$

Similarly, we can express a carrier $B_{\sigma'_i}$ and $B_{\sigma'}$ in the components as $B_{\sigma'_i} = Im_B(Y_i)$ and $B_{\sigma'} = Im_B(Y)$, where Y_i and Y are classes of U classified by the knowledge base $K' = (U, \mathrm{R}')$ in the components.

The common sorts between the requirements and the components can be derived as follows.

1. For each A_{σ_i} and A_σ, let X_i and X be the home classes in U, that is,

   $A_{\sigma_i} = I_{m_A}(X_i)$ and $A_\sigma = I_{m_A}(X)$.

   Calculate the upper approximations of X_i and X by the knowledge base $K' = (U, \mathrm{R}')$. These approximations are denoted by

   $\overline{R'}X_i = \{Y_{i1}, \ldots, Y_{ip_i}\}$, $\overline{R'}X = \{Y_1, \ldots, Y_p\}$

   where $R' = IND(\mathrm{R}')$.

2. Derive the new carriers with the new sorts as the following.

   $A_{\rho_{ij}} = I_{m_A}(X_i \cap Y_{ij}) \subseteq A_{\sigma_i}$

   $B_{\rho_{ij}} = I_{m_B}(X_i \cap Y_{ij}) \subseteq B_{\sigma'_{ij}} = I_{m_B}(Y_{ij})$

   $A_{\rho_k} = I_{m_A}(X \cap Y_k) \subseteq A_\sigma$

   $B_{\rho_k} = I_{m_B}(X \cap Y_k) \subseteq B_{\sigma'_k} = I_{m_B}(Y_k)$

   where $j = 1, \ldots, p_i$ and $k = 1, \ldots, p$ respectively.

Figure 4.1 shows this intuitively.

We can define a set of new S-sorted functions $\{\tilde{\phi}\}$ with arity $\rho_{1i_1} \ldots \rho_{mi_m}$ and result sort ρ_k, where $i_1 = 1, \ldots, p_1, i_2 = \ldots i_m = 1, \ldots, p_m$, and $k = 1, \ldots, p$. $\tilde{\phi}$ can be defined as $\tilde{\phi}(x_1, \ldots, x_m) = \phi(x_1, \ldots, x_m)$ for $x_i \in A_{\rho_{ij}}$ (a *restriction* of ϕ to $A_{\rho_{ij}}$).

Those $\{\tilde{\phi}\}$, $\{A_{\rho_{ij}}\}$ and $\{A_{\rho_k}\}$ compose a Σ algebra $\mathcal{A}$=$(A, \tilde{\Phi})$.

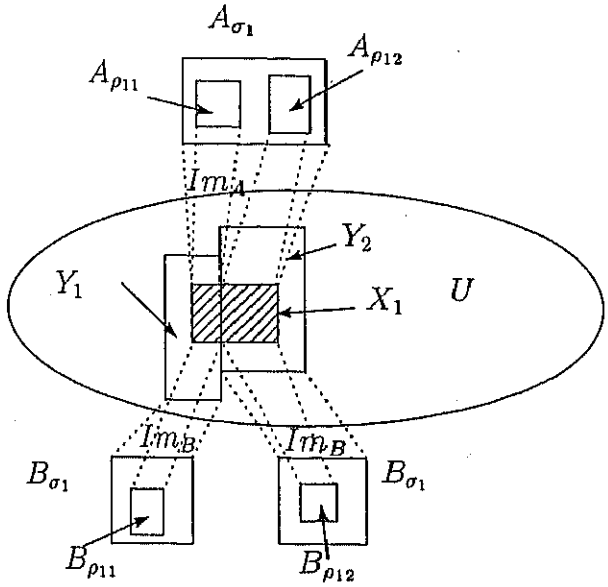

Figure 4.1: Sorts adjustment

On the other hand, if there is an S-sorted function ψ with arity $\sigma'_{1i_1} \ldots \sigma'_{mi_m}$ and result sort σ'_k, we can define a set of new S-sorted function $\{\tilde{\psi}\}$ with arity $\rho_{1i_1} \ldots \rho_{mi_m}$ and result sort ρ_k, in the similar way to $\{\tilde{\phi}\}$.

There could be multiple sets of $\{\tilde{\psi}\}$ with the carriers $\{B_{\rho_{ij}}\}$ and $\{B_{\rho_k}\}$, and those are the candidates for the corresponding functions for $\{\tilde{\phi}\}$.

Those $\{\tilde{\psi}\}$, $\{B_{\rho_{ij}}\}$ and $\{B_{\rho_k}\}$ compose a Σ algebra $\mathcal{B}$=$(B, \tilde{\Psi})$.

Therefore, we can determine the equivalency between Σ algebras $\mathcal{A}$=$(A, \tilde{\Phi})$ and Σ algebras $\mathcal{B}$=$(B, \tilde{\Psi})$ by examining the existence of Σ homomorphism between $\mathcal{A}$ and $\mathcal{B}$.

If we can identify equivalent Σ algebra $\mathcal{B}$=$(B, \tilde{\Psi})$ to Σ algebra $\mathcal{A}$=$(A, \tilde{\Phi})$, the original function g of $\{\tilde{\phi}\}$ has the equivalent functions $\{\psi\}$ with arity $\sigma'_{1i_1} \ldots \sigma'_{mi_m}$ and result sort σ'_k.

In order to show a simple example, we use a reduct of S-sorted function g_1 which was introduced in Section 4.1.1. We denote the reduct by ϕ. The reduct has the arity $\sigma_1\sigma_2$ and the result sort σ which correspond to "*order number*", "*product number*" and "*shipment information*" respectively. The discussion is based on the following assumptions.

1. We already identified two S-sorted functions or their reducts ψ_1, ψ_2 from components, which have such arities and result sorts as $\sigma'_1\sigma'_{21}, \sigma'_3$ and $\sigma'_1\sigma'_{22}, \sigma'_3$ respectively.

2. The home classes of σ'_1 and σ' are identical to those of σ_1 and σ. The home classes of σ'_{21} and σ'_{22} represent a set of "*raw materials*" and "*parts*" respectively.

Let the home classes of σ_2, σ'_{21} and σ'_{22} be X_2, Y_{21} and Y_{22}, which correspond to carriers A_{σ_2}, $B_{\sigma'_{21}}$ and $B_{\sigma'_{22}}$ respectively. If those home classes satisfy:

$X_2 \subseteq Y_{21} \cup Y_{22}$ and $X_2 \cap Y_{2i} \neq \emptyset$ $(i = 1, 2)$,

then we can introduce the two new sorts ρ_{21} and ρ_{22} which correspond to the home classes $X_2 \cap Y_{21}$ and $X_2 \cap Y_{22}$. By using those sorts, we can define the pairs of S-sorted functions with the common sorts as:

$(\tilde{\phi}_1)_{\sigma_1\rho_{21}\sigma_2,\sigma} \longleftrightarrow (\tilde{\psi}_1)_{\sigma_1\rho_{21}\sigma_2,\sigma}$

$(\tilde{\phi}_2)_{\sigma_1\rho_{22}\sigma_2,\sigma} \longleftrightarrow (\tilde{\psi}_2)_{\sigma_1\rho_{22}\sigma_2,\sigma}$,

where $\tilde{\phi}_1$ and $\tilde{\psi}_1$ can be interpreted as the S-sorted functions to deal with "*raw materials*", while $\tilde{\phi}_2$ and $\tilde{\psi}_2$ are those to deal with "*parts*".

By examining all S-sorted functions within the requirements model in the above way, we can determine whether the components are functionally adaptable to the requirements. The S-sorted functions in the requirements model, which do not have the corresponding functions in the components, must be developed to construct the adaptable software system.

The sorts adjustment technique will help us to identify the adaptable components to the requirements, which have the different sorts from each other. Therefore, the range of component searching becomes much wider than that by trials and errors performed by human designers.

However, this technique assumes the existence of the common universe, in other words, it is based on the semantics of requirements and components. Therefore, it can not be applied to the components for the different domains from the requirements, even though they have the same computational structure.

By following the procedure discussed in this section, we will identify the adaptable components formally, and the scope of component searching is widened by the sort adjustment technique.

4.3 Evaluating Behavior of Requirements

After selecting all the adaptable components, we must define how those components are executed in order to realize the requirements as a software system. In other words, we must define the appropriate component to be executed under each possible condition.

As stated in Chapter 4, execution of the components corresponds to execution of arc expression functions caused by transition firing, which is determined by marking of color tokens. Therefore, by describing the relationship among transition firing, execution of the arc functions and marking of color tokens, we can define how the components are executed. This relationship represents the control structure or the behavior of the CPN model.

In the following sections, we discuss how the control structure is extracted from the CPN model and how it is optimized by RST.

4.3.1 Transforming CPN Models to Decision Tables

It is possible to represent the behavior of a CPN model by specifying a firing rule for each transition independently, if each rule shows:

1. Marking of color tokens in the input places to the transition, which enables the transition to fire.

2. Output arc functions to be executed.

3. Marking of color tokens in the output places from the transition as the result of the firing.

These items can be expressed in a decision table for each transition, which is constructed as follows.

Let "t" be a transition with input places "$p_1, \ldots, p_m$", input arc functions "$f_1, \ldots, f_m$", output places "$p'_1, \ldots, p'_{m'}$", and output arc functions "$f'_1, \ldots, f'_{m'}$". The conditions in the decision tables represent markings of color tokens in the input places which enable the transition to fire. There are three decisions in the table. The first is the output function to be executed. The second is the marking of color tokens produced by the function. The third is the output place into which the color tokens put.

Those decision tables can express the behavior of the CPN model exactly, however there is a problem to use the decision tables in this form to determine the components to be executed. Since there could be multiple components corresponding to one output arc function as stated in Section 4.1, we need another information to find the appropriate component out of them. Therefore, it seems more convenient to include the components instead of the output arc functions in the decision tables.

In Section 4.2, we revealed the relationship between the requirements and the components as follows:

1. From each output function f'_j of transition t, one or multiple S-sorted functions $\{g^j_k\}$ are derived. If the reduct ϕ^j_k of g^j_k has an algebraic equivalent S-sorted function ψ^j_k by sorts adjustment, there is an adaptable components to g^j_k.

2. Sorts adjustment is achieved based on the knowledge about the requirements and the components. The carriers of each S-sorted function in the requirements and the components have their home classes in the common

universe U which represents all the instances of resources and information in the enterprise. Sort adjusted S-sorted functions of ϕ_k^j and ψ_k^j are denoted by $\tilde{\phi}_k^j$ and $\tilde{\psi}_k^j$ respectively.

Therefore, we can construct the new decision tables more suitable for our purpose by substituting those $\{\psi_k^j\}$ for output functions, the home classes of the arities for the input place markings and the home classes of the result sorts for the output place markings.

The new decision tables are in the form of Table 4.3. Each number in column N represents the rule number. D_{xj} in column $C(p_j)$ represents the home class which determines the carrier of S-sorted function in the components. This carrier is corresponding to the color set $C(p_j)$[3]. D_{xj} could be blank if the sort corresponds to the color set $C(p_j)$ is eliminated by simplification of S-sorted function described in Section 4.2.1. ψ_x in column Ψ denotes an S-sorted function of the components to be executed when the conditions in the left columns are satisfied. D_x in column n represents the home class corresponds to the output of ψ_x. π_x in column Π represents the output place relevant to S-sorted function ψ_x.

By those decision tables, we can describe the behavior of the CPN model, along with the corresponding components execution.

Table 4.3: A decision table for a transition

N	$C(p_1)$	$C(p_2)$	...	$C(p_m)$	Ψ	n	Π
1	D_{11}	D_{12}	...	D_{1m}	ψ_1	D_1	π_1
2	D_{21}	D_{22}	...	D_{2m}	ψ_2	D_2	π_2
⋮	⋮	⋮		⋮	⋮	⋮	⋮
x	D_{x1}	D_{x2}	...	D_{xm}	ψ_x	D_x	π_x
⋮	⋮	⋮		⋮	⋮	⋮	⋮

For example, we would build such decision table as Table 4.4 from the transition t_4 (*Evaluation*) in the sample application.

The table implies such control structure as:

1. If both the "*Inventory Check*" and "*Credit Check*" succeed, then pass the control to the "*Shipping*" and the "*Billing*". This condition is represented by D_{41} in the column $C(p_4)$ and D_{51} in the column $C(p_5)$, where D_{41} and D_{51} are the subdomains in $C(p_4)$ and $C(p_5)$, which make the "*Inventory Check*" and the "*Credit Check*" successful.

[3]Evidently $D_{xj} \subseteq C(p_j) \subseteq U$

Table 4.4: A decision table for a sample transition

N	$C(p_4)$	$C(p_5)$	Ψ	n	Π
1	D_{41}	D_{51}	ψ_1	D_1	p_8
2	D_{41}	D_{51}	ψ_2	D_2	p_9
3	D_{41}	D_{52}	ψ_3	D_3	p_7
4	D_{42}	D_{51}	ψ_4	D_4	p_6
5	D_{42}	D_{52}	ψ_2	D_3	p_7

2. If the "*Credit Check*" fails, then pass the control to the "*Rejection*". The condition is represented by D_{52} in the column $C(p_5)$.

3. If the "*Inventory Check*" fails and the "*Credit Check*" is succeeds, then pass the control to the "*Manufacturing*". The condition is represented by D_{42} in the column $C(p_4)$ and D_{51} in the column $C(p_5)$.

The total size of those tables from a CPN model depends on the number of transitions and S-sorted functions derived from them, since each row of the table corresponds to an S-sorted function which is to be implemented by a component. Therefore, the total size of them is approximately the number of the components to be used in the system, and would be manageable size.

4.3.2 Reducing Decision Tables for Optimization

Since the decision tables discussed in the previous section include the preconditions under which the components are to be performed, we can develop a software system that implements the requirements from those decision tables.

However, they often include the redundant entries, or in other words, the unnecessary entries, which lead us to develop unnecessary codes. This redundancy is mainly caused by duplication of knowledge among domain-experts, who provide us with the same requirements in the different forms. In order to eliminate the duplication or redundancy in the decision tables, we can use RST in the similar way we discussed in Section 4.2.1.

For each transition in the CPN model, there is a decision table in the form of Table 4.3. For any $x \in N$, let X_i and Y be $X_i = [x]_{C(p_i)}$ and $Y = [x]_{\Psi,n,\Pi}$ respectively. X_i and Y are equivalent classes in N which satisfy

$\forall x' \in X_i\ [D_{x'i} = D_{xi}]$ and

$\forall x' \in Y\ [\psi_{x'} = \psi_x, D_{x'} = D_x$ and $p_{x'} = p_x]$.

As described in Section 3.1.1, if for all $x \in N$, the family $F = \{X_1, X_2, \dots\}$ satisfies the two conditions:

$\cap F \subseteq Y$ and $\cap\{F - \{X_i\}\} \subseteq Y$,
X_i is dispensable and we can eliminate the column $C(p_i)$ from the decision table.

After reducing all the tables, we get the decision tables without redundancy, that is, the optimized decision tables.

The methods mentioned in this section are so formal that we can obtain homogeneous systems from quality and performance viewpoints independently to the skills and backgrounds of designers.

4.4 Composing Software Systems Using Selected Components

The final step of CBSD in our approach is to implement the optimized control structure as a software system. There are several options to implement the control structure, by which the selected components are appropriately executed. We need an infrastructure to launch the appropriate software components according to the conditions depicted in the decision tables, which were obtained in the previous section.

There are many commercially available software products for this purpose. They include CORBA (Common Object Request Broker Architecture) [71] based brokering software products which utilize the standardized component invocation mechanism defined by OMG (Object Management Group), workflow management systems [16], [46] which assist business process automation and have the capability of performing components in predefined sequences with conditions, and message brokers [9] which provide us with reliable messaging mechanism for inter-components communication and are often referred to as Message Oriented Middleware (MOM). Those software products are often called *glue codes*.

While CORBA based brokering software and MOM require us to make program codes which implement the component invocation rules, workflow management systems have some interfaces for defining such component invocation rules. Besides, workflow management systems provide additional functionality for business processes, such as event notification and work assignment to organization staffs. Therefore, workflow management systems would be better solutions for implementing the control structure of enterprise models.

Those software products are recently used widely in various large-scale industrial applications such as banking and manufacturing applications, and they have proved to be reliable solutions for mission-critical applications providing good performance.

Those solutions require us to construct systems in different styles, e.g. in the different ways for state controls, activity tracking, error and/or exception handling

and specification syntax. However the essential part of the developments is the same, that is, interpreting the relationship among the conditions, the components to be executed and the results of the execution of the components. In addition, we need to convert this interpretation into appropriate forms which the selected infrastructure requires.

The glue codes must be able to identify the current marking or its representation in the software system in order to launch the appropriate components. A marking of the CPN model can be regarded as a state of the model and that of the corresponding software system.

From the decision table, we can determine the component to be executed under each condition which is represented as a state by the software system, however we need a mechanism by which the state of the system changes correctly according to the requirements model. The software infrastructures mentioned above in this chapter usually provide us with such mechanisms, therefore we only need to interpret and convert the decision tables in the above way.

The procedure discussed in this chapter provides a systematic way to compose software systems by combining the identified adaptable components, from the behavioral viewpoint of enterprise models. Decision tables will help us to find and express the control structure of an enterprise model, which is the basis of software composition.