CPN BASED COMPONENT ADAPTATION

Yoshiyuki Shinkawa
Department of Media Informatics, Ryukoku University
1-5 Seta Oe-cho Yokotani, Otsu, Shiga, Japan

Keywords: Component reuse, software engineering, colored Petri nets, formal methods.

Abstract: One of the major activities in component based software development is to identify the adaptable components to the given requirements. We usually compare requirement specifications with the component specifications, in order to evaluate the equality between them. However, there could be several differences between those specifications, e.g. granularity, expression forms, viewpoints, or the level of detail, which make the component evaluation difficult. In addition, recent object oriented approaches require many kinds of models to express software functionality, which make the comparison of the specification complicated. For rigorous component evaluation, it is desirable to use concise and simple expression forms of specifications, which can be used commonly between requirements and components. This paper presents a formal evaluation technique for component adaptation. In order to relieve the granularity difference, the concept of a virtual component is introduced, which is the reusable unit of this approach. A virtual component is a set of components that can acts as single component. In order to express requirements and components commonly and rigorously, algebraic specification and Colored Petri Nets (CPNs) are used. Algebraic specification provides the theoretical foundation of this technique, while CPNs help us to use it intuitively.

1 INTRODUCTION

One of the major tasks in Component Based Software Development (CBSD) is to search and identify the adaptable components to given requirements. For this purpose, we usually compare the specifications of requirements with those of components. There have been proposed many theoretical and practical approaches to the component evaluation, which include signature matching (Zaremski and Wing, 1993), specification matching (Zaremski and Wing, 1995), name matching (Michail and Notkin, 1999), specification-based browsing (Fischer, 1998), faceted classification of software (Prieto-Diaz and Freeman, 1987), and so on. When all the specifications are prepared in the same manner, the above methods work correctly, and help us to identify the adaptable components efficiently.

However, in many cases, there would be several kinds of *gaps* between the specifications of requirements and those of components, and the gaps make the comparison complicated. The typical gaps that we are often faced with are

1. A *semantic gap*, which is caused by the difference in viewpoints between requirements and components
2. A *syntactic gap*, which is caused by the difference in notational rules
3. A *structural gap*, which is caused by the difference in granularity

In addition to the above essential problems, there is another notational problem to express specifications. Since most specifications are expressed textually by formal or informal languages, it is not so easy to understand the functionality that they intend to show. As a result, the specification comparison becomes a difficult task. We need more visual notation tools as we use in analysis and design phase, e.g. UML, DFD, and IDEF.

This paper presents a formal approach to resolving the above mentioned gaps between requirements and components, along with a visual notation method for specifications. The paper is organized as follows. In section 2, several formal approaches to resolving the gaps are introduced. Section 3 shows how a textual notation of specifications is transformed into a visual notation using Colored Petri Net (CPN). Section 4 presents a process for identifying the adaptable components to the given requirements using visually expressed specifications.

2 RESOLVING GAPS BETWEEN REQUIREMENTS AND COMPONENTS

Before discussing component adaptability, we first define what a component is. The word *component* is often used vaguely in several contexts and situations. In CBSD, it usually means *a unit of reuse*, however there are different kinds of units depending on what programming environment we use. For example, while procedural languages regard a subroutine, a function, or a module as a unit of reuse, recent object oriented languages regard a class as a unit of reuse. The paper mainly focuses on the object oriented programming environment, since it is one of the most commonly used environments today, and *reuse* is one of the most critical issues in it.

Roughly speaking, object orientation provides us with two different ways for component reuse, that is, *class inheritance* and *method invocation*. As a result, there are two different units of reuse, namely a *class* and a *method*. A class is a template of an object that expresses encapsulated data and operations. The data is implemented as a set of instance variables, and the operations are implemented as a set of methods defined in the class.

When we consider reusing an existing class through inheritance, we have to decide whether each method in the class is reused, overridden, or ignored. In this decision, we evaluate the adaptability of each method to a given requirement, and therefore a method can be regarded as a unit of reuse even in the class inheritance. Therefore, we treat a method as a unit of reuse henceforth, and discuss how those methods are reused.

The adaptability of a method is evaluated by comparing its specification with a requirement specification. Those specifications have to be homogeneous between requirements and components for rigorous evaluation. However there could be several gaps between them.

2.1 Resolving a Semantic Gap

In order to create the specifications of requirements, we first model the requirements from various viewpoints. Since we focus on methods in classes for reuse, *function* and *behavior* are the most essential viewpoints for modeling. From a functional viewpoint, a method acts as a data transducer which transforms the arguments into the result value. On the other hand, a method acts as a finite state machine from a behavioral viewpoint, which responds to the stimuli.

The specifications of components are also created from either of the above two viewpoints. When a requirement specification is compared with a component specification for adaptability evaluation, both the specifications have to be prepared from the same viewpoint, unless we can not compare them easily. Consequently, all the requirement and component specifications involved in the adaptability evaluation must be prepared from the same viewpoint. If some of the specifications have been prepared from the different viewpoint, there arises a gap between requirements and components.

Since those viewpoints give semantics to requirements and components, we refer to such gap as a *semantic gap*. In order to resolve this gap, we first investigate the essential differences between *function* and *behavior*.

A functional aspect of requirements and components can be expressed by S-sorted functions in terms of many sorted Σ algebra (Shinkawa and Matsumoto, 2000). 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

1. $A = \{A_\sigma | \sigma \in S\}$ (a set of carriers) and
2. $F = \{f_A | f \in \Omega_{\sigma_1 \ldots \sigma_n, \sigma}\}$
   $f_A : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \to A_\sigma \ (\sigma_1, \ldots, \sigma_n, \sigma \in S)$.

When this f_A map an element $\langle u_1, \ldots, u_n \rangle$ in the domain of definition $A_{\sigma_1} \times \cdots A_{\sigma_n}$ to an element v in the co-domain A_σ, we denote

$$f_A(u_1, \ldots, u_n) = v$$

S-sorted function f_A is said to have arity $\sigma_1 \ldots \sigma_n$ and result sort σ. The series of arity and result sort is called the *rank* of the function.

If f_A is a partial function, that is, if some elements in the domain of definition $A_{\sigma_1} \times \cdots \times A_{\sigma_n}$ are not mapped into the co-domain A_σ, f_A is denoted by

$$f \ : \ \mathbb{D} \left(\subseteq A_{\sigma_1} \times \cdots \times A_{\sigma_n} \right) \longrightarrow A_\sigma$$

where $\mathbb{D}$ is a subset of the original domain of definition.

On the other hand, the behavioral aspect of a requirement is expressed in many ways, e.g., Finite State Machines (FSMs), Petri Nets, process algebra, and so on. The behavioral aspect of a requirement or a component expressed in the form of FSM is represented as a sextuplet

$\langle \Sigma, \Gamma, S, s_0, \delta, \omega \rangle$ where

Σ is the input alphabet
Γ is the output alphabet
S is a finite non empty set of states
s_0 is an initial state
δ is the state transition function $\delta : S\times\Sigma \rightarrow S$
ω is the output function $\omega : S \times \Sigma \rightarrow \Gamma$

In order to reveal the essential difference between *function* and *behavior*, we focus on the relationship of inputs and outputs. When a requirement is given as an S-sorted function, the inputs are transformed into the output uniquely according to the mapping rule from the domain to the co-domain.

On the other hand, when a requirement is given as behavior, the output is determined by the output function $\omega : S \times \Sigma \rightarrow \Gamma$, that is, the pair of a state and an input determines the output. In addition, the state is transferred according to the state transition function $\delta : S \times \Sigma \rightarrow S$. Therefore, the behavior is characterized by the function

$$\beta : S \times \Sigma \rightarrow S \times \Gamma$$

Since the input to a software component is a set of arguments, the output is a return value, and the state is implemented as a set of instance variables, the above function β is expressed as

$$\begin{aligned}\beta : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \times B_{\rho_1} \times \cdots \times B_{\rho_m} \\ \rightarrow B_{\rho_1} \times \cdots \times B_{\rho_m} \times A_\sigma\end{aligned}$$

where A_{σ_i}s and A_σ are the carriers of the arguments and the result sort, and B_{ρ_j}s are the carriers of the instance variables.

If we regard the direct product

$$C = B_{\rho_1} \times \cdots \times B_{\rho_m} \times A_\sigma$$

as a co-domain, β is an S-sorted function

$$\beta : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \times B_{\rho_1} \times \cdots \times B_{\rho_m} \rightarrow C$$

This β implies that the behavioral aspect of a component can be expressed as an S-sorted function. Similarly, the behavioral aspect of a requirement can also be expressed as an S-sorted function.

The above discussion leads us to create all the specifications in the form of S-sorted functions. We use the term *requirement function* as a function that expresses the requirement, and *component function* as a function that expresses the component.

2.2 Resolving a Syntactic Gap

As discussed above, all the requirements and the components can be treated as S-sorted functions. In order to evaluate component adaptability to given requirements, we first have to express those functions in a more concrete way. Roughly speaking, there are two contrastive approaches to express those functions, or more generally, objects or classes which include them inside. One is the *model-oriented* approach that puts emphasis on the structure of the object to be specified, while the other is the *property-oriented* approach that emphasizes the constraint on them (Fensel and Groenboom, 1995).

If a requirement specification and a component specification are created by the different approaches, we can not easily compare them for adaptability evaluation, and consequently all the related specifications to be compared must be prepared by the same approach, that is, by either model or property oriented approach. For component based software development, the property oriented approach is more applicable, since there might be no way to examine the structure of components.

In the property-oriented approach, the functional aspect of requirements or components can be expressed by such algebraic specification languages as OBJ (Goguen and Malcolm, 1996), CASL (Mosses, 1999), or Larch (Guttag and Horning, 1993). Even though the syntax and semantics are different among those languages, the basic structure of specifications is almost the same. The specifications of requirements or components written in those languages consist of two parts, often referred to as a *signature part* and a *axiom part*.

A signature part defines the data types and operation names used in the specification, while an axiom part describes the properties or the constraints that the above operations must follow.

The axiom part consists of axioms which are expressed as equations of terms. A term is a formula which can be derived from the signature of the specification. Terms are defined formally in the following way.

1. An S-sorted variable x is a term
2. if $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term

An equation is a formula in the form of $t = t'$, where t and t' are terms.

In addition to the functions defined in the signature part, we assume the well-known functions to be used, which include *conditional branch* function **if-then-else**, *projection* function **proj**, *tuple creation* function **tuple**, and so on. Those functions have the following inherent properties.

$$\mathbf{proj}(i, \vec{s}) = s_i$$
$$\mathbf{tuple}(s_1, \ldots, s_n) = \vec{s}$$
$$\forall i\ [\mathbf{proj}(i, \vec{s}) = s_i] \Longleftrightarrow \vec{s} = \mathbf{tuple}(s_1, \ldots, s_n)$$

The function **if-then-else** is often denoted as

$$\mathbf{if}\ \ t_1\ \ \mathbf{then}\ \ t_2\ \ \mathbf{else}\ \ t_2$$

where $\mathbf{type}(t_1) = \mathit{boolean}$, and $\mathbf{type}(t_2) =$

$\mathbf{type}(t_3)$. Since

$$\textbf{if-then-else}(t_1, t_2, t_3) = \begin{cases} t_2 & (t_1 = \mathit{true}) \\ t_3 & (t_1 = \mathit{false}) \end{cases}$$

and

$$f\big(\textbf{if-then-else}(t_1, t_2, t_3)\big) = \begin{cases} f(t_2) & (t_1 = \mathit{true}) \\ f(t_3) & (t_1 = \mathit{false}) \end{cases}$$

holds, the **if-then-else** function has the property of

$$f\big(\textbf{if-then-else}(t_1, t_2, t_3)\big) = \textbf{if-then-else}\big(t_1, f(t_2), f(t_3)\big)$$

As an example, let us consider a bank account class A with a method *getBalance*: $\mathrm{A} \rightarrow \mathit{int}$, and a method *debit*: $\mathrm{A} \times \mathit{int} \rightarrow \mathrm{A}$. One of the axioms that the methods must follow is

$$\mathit{getBalance}\big(\mathit{debit}(s, x)\big) = \textbf{if-then-else} \big(\mathit{getBalance}(s) - x \geq 0, \mathit{getBalance}(s) - x, \mathit{getBalance}(s)\big)$$

where $s \in \mathrm{A}$ and $x \in \mathit{int}$.

The correctness of an equation of terms is proved using the following inference rules, according to *equational calculus* (**?**), where t, t', t'' and t_i are terms.

1. Reflexivity $\dfrac{}{t = t}$ 2. Symmetry $\dfrac{t = t'}{t' = t}$

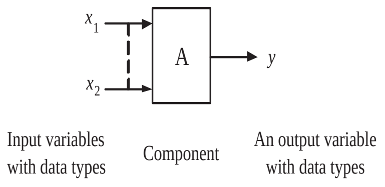
3. Transitivity $\dfrac{t = t' \quad t' = t''}{t = t''}$

4. Congruence $\dfrac{t_1 = t_1', \cdots, t_n = t_n'}{f(t_1, \cdots, t_n) = f(t_1', \ldots, t_n')}$

2.3 Resolving Structural Gap

Even though we resolve the semantic and syntactic gaps, there could be another gap between requirement and component specifications. This gap occurs when the granularity is different between them. Since the granularity defines the structure of specifications, we refer to such gap as a *structural gap*. In many cases, this gap is resolved using glue codes to combine multiple components.

A component, or a method in our case, receives a set of arguments to produce a return value. Therefore, it can be depicted as a data transducer shown in Figure 1. On the other hand, glue codes are placed between components to process the return values from components, or to provide arguments to components.



x_1
x_2
A
y
Input variables with data types
Component
An output variable with data types

Figure 1: A component illustration.

Even though the use of glue codes can enhance the components arbitrarily, it might reduce the productivity. In addition, the use of glue codes makes it difficult to evaluate the component adaptability to requirements, since glue codes can reconfigure component based systems intentionally.

In order to make the component adaptability evaluation precise, and make component usage simpler, we consider combining components without glue codes. Such combination of components is implemented by connecting the output (or the return value) of a component to the inputs (or the arguments) of other components. Since arguments and return values are associated with data types, such connection is allowed only when the data types of those outputs and inputs match. For example, we obtain the combination of components without glue shown in Figure 2, if data types match between outputs and inputs. In Figure 2, x_i and y_j on the arrows represent the variables for the arguments and the return values respectively. Assuming $\mathbf{type}(x)$ shows the data type of the variable x, $\mathbf{type}(x_6) = \mathbf{type}(y_1)$, $\mathbf{type}(x_7) = \mathbf{type}(y_2)$, $\mathbf{type}(x_8) = \mathbf{type}(y_3)$, and $\mathbf{type}(x_9) = \mathbf{type}(y_2)$ must hold.

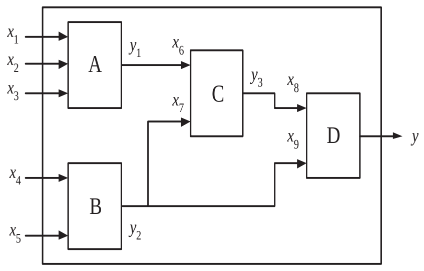


Figure 2: A Virtual Component.

The component combination in Figure 3 can be regarded as a single component that receives a set of arguments $\langle x_1, x_2, x_3, x_5, x_5 \rangle$ and returns the value y. We refer such combination of components as a *virtual component*. A virtual component represents a component based system without glue codes composed from a set of available components. Therefore, a set of all the possible virtual components shows the minimum adaptation capability of the available components. A virtual component is formally defined as follows.

1. A virtual component receives multiple inputs and yields a single output
2. Each input is associated with a data type and is connected to at least one component in the virtual component
3. Each input to a component in the virtual component is either an input to the virtual component or an output from another component
4. The output from the virtual component is associated with a data type, and is one of the output from a component in virtual component

A single component is also a virtual component from the above definition. A virtual component acts as a single component, and therefore can be regarded as a unit of reuse. Henceforth we use a virtual component including a single component as a unit of reuse.

3 CPN BASED SPECIFICATIONS

Algebraic specifications can express requirements and components accurately, however it seems difficult to understand them intuitively through text-based specifications. As a software development methodology, those specifications have to be expressed visually or graphically. UML is one of the most popular graphical specification tools, however it is not suitable to express the functionality of each method within a class. Colored Petri Net is an alternative to UML for expressing the functionality of a method within a class.

CPNs are one of the enhancements of Petri nets, and formally defined as follows (Jensen, 1997). *CPN=(S, P, T, A, N, C, G, E, I)* ,
where

1. S : a finite set of non-empty types, called color sets
2. P : a finite set of places
3. T : a finite set of transitions,
4. A : finite set of arcs $P \cap T = P \cap A = T \cap A = \emptyset$
5. N : node function $A \rightarrow P \times T \cup T \times P$
6. C : a color function $P \rightarrow S$
7. G : a guard function $T \rightarrow$ expression
8. E : an arc expression function $A \rightarrow$ expression
9. I : an initialization function : $P \rightarrow$ closed expression.

In order to express the above discussed formal specifications using CPN, we first define basic relationships between the elements of these two different notations. As shown in the previous section, a formal specification of a requirement or a component can be denoted by

$$S_p = \langle \Sigma, \{L_i\} \rangle$$

where $\Sigma = \langle S, \Omega \rangle$ is a signature which is composed of a set of sorts (data types) anf functions, and L_is are the equations of terms that organize the axiom part of the specification. A basic relationship between the elements of such a specification and those of a CPN is a s follows.

1. A function $f_i \in \Omega$ corresponds to a transition of a CPN.
2. A variable x_j of a function $y = f_i(x_1, \ldots, x_n)$ corresponds to an input place of the transition associated with the function f_i. A variable y corresponds to an output place of that transition.
3. A color represents a data type or a sort. If a place represents a variable $x_j \in A_{\sigma_j}$, the color that is associated with this place by the color function C_j corresponds to the sort or the carrier A_{σ_J}.
4. The *if-then-else* clause is implemented as a guard function on a transition.

The above relationships map the signature $\Sigma = \langle S, \Omega \rangle$ to the transitions, the places, and the color sets which compose the most basic elements in CPN. The next step is to depict the axiom part $\{L_i\}$ in the form of CPN.

Each L_i is denoted by an equation $t = t'$ where t and t' are terms. A term is in the form of either a variable or a function value $f(t_1, \ldots, t_n)$. A variable x is depicted as a place, while a function value $f(t_1, \ldots, t_n)$ is depicted by CPN iteratively as follows.

1. A function f is denoted by a transition with one output place and multiple input places
2. Each argument t_i is denoted by an input place to the above transition. If t_i is derived from another function f', that is, if $t_i = f'(t'_1, \ldots, t'_m)$, the place t_i is an output place of the transition f'.
3. repeat the above process until all the input places to the transitions become variables.

A CPN model constructed in the above way has multiple places with no inbound arcs and one place with no output arc. The former places represent the initial inputs or arguments to the term that the CPN model expresses, and the latter place represents the final output or return value of the term. We refer to the former places as *initial input places* and the latter place as *final output place*. The final output place must have the only one transition that provides tokens to it. This transition is called a *terminal transition* in this paper.

Any term except single variable is expressed as a CPN model of the above form. Such CPN model is referred to as a *functional CPN model* in this paper, since it represents data transformation that is performed by the original term.

For notational convenience, we make a small enhancement to CPN models shown as follows. **if-then-else** clause or function **if-then-else**(t1, t2, t3) is denoted as a rectangle with a triangle inside of it. ((a) in Figure 3) A projection function **proj** which extracts the ith element from a tuple, and tuple creation function **tuple** are denoted as figure (b) and figure (c) in Figure 3 respectively. To create multiple copies of a token, the distributor transition that is denoted by a double lined circle shown as figure (d) in Figure 3 is used. We can depict a CPN models for an equation of terms without those enhancements. For example, **if-then-else** transition can be substituted by a guard function, however those enhancements make CPN models simple, and reduce the size of the models.

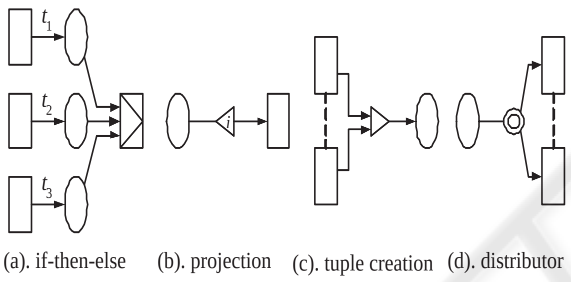


Figure 3: An enhancement to CPN models.

In order to express an equation of terms $t = t'$, we introduce a new type of a place denoted by a double lined ellipse, which is used as a common terminal place of t and t'. For example, the debit method of a bank account class A, which is expressed as a equation as discussed in the previous section

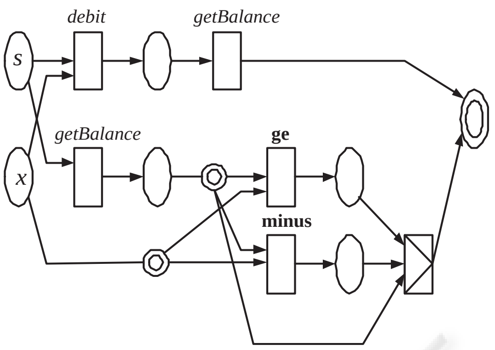
$$\begin{aligned}&getBalance\big(debit(s,x)\big) = \\ &\quad \textbf{if-then-else}\big(getBalance(s) \geq x, \\ &\qquad getBalance(s) - x, getBalance(s)\big)\end{aligned}$$

and the credit method

$$getBalance\big(credit(s,x)\big) = getBalance(s) + x$$

are described as CPN models shown in Figure 4.

As shown above, an equation of terms is represented by two functional CPN models with common initial input places and common single final output place. Each functional CPN model may include other functional CPN models inside. We refer to such inner functional CPN models as *subnets*.



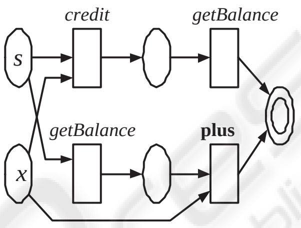


Figure 4: Example of CPN Models for Equations of Terms.

4 IDENTIFYING ADAPTABLE COMPONENTS

Once all the requirement specifications and the component specifications are transformed into CPN models, the next step is to identify an adaptable component or a virtual component to a requirement. When a component or a virtual component is adaptable to a requirement, the function that represents the requirement in a requirement specification can be substituted by the term that represents the component or the virtual component. If all the requirement functions in a requirement specification are substituted by the adaptable components or virtual components, the equations of terms in the requirement specification are transformed into the correct equations defined in the previous section.

In CPN models, a function is represented as a transition, and a term is represented as a subnet. Substitution of a requirement function by a component is represented as replacement of a transition that expresses the requirement function in the requirement CPN model. When the substitution is done by a virtual component, the transition is replaced by a subnet that represents the virtual component. The simplest form of CPN model that shows component adaptation is shown in figure (a) in Figure 5, if the subnet includes only component functions and well-known functions. In this case, the upper shaded transition f is implemented by the virtual component that is represented as the lower subnet. When the requirement

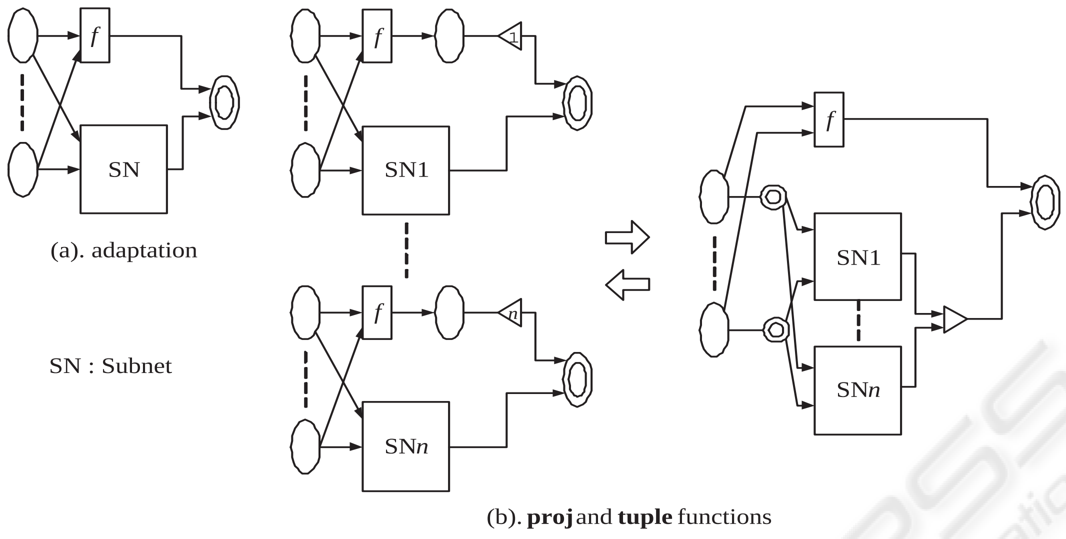


Figure 5: Component Adaptation Rule – 1.

function yields a tuple as a return value, the adaptation is represented by the figure (b) in Figure 5, from the properties of **proj** and **tuple** functions.

In addition, the property of **if-then-else** function introduced in the previous section is depicted as the CPN model shown in Figure 6.

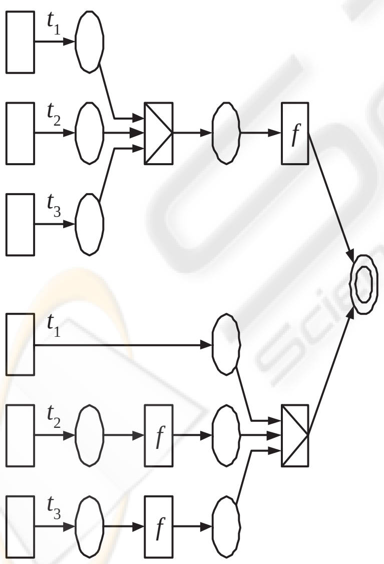


Figure 6: Component Adaptation Rule – 2.

However, most requirement CPN models have much more complicated forms. In order to transform the given requirement CPN model into the above simple form, we introduce the reduction rule of CPN models. The reduction rule is derived from the inference rule *congruence* in equational calculus discussed in the previous section, and shown in Figure 7.

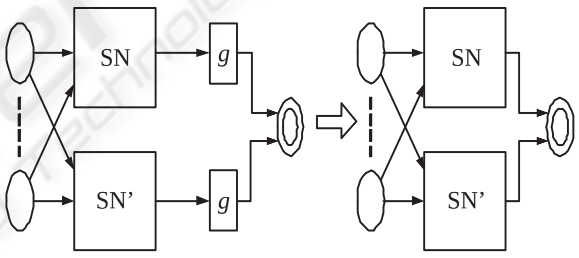


Figure 7: The Reduction Rule of CPN Models.

In order to show how the reduction process is used, let us consider the *transfer* method

$$(a).\ getBalance\Big(\textbf{proj}\big(1, transfer(s_1, s_2, x)\big)\Big)$$
$$= \textbf{if-then-else}\big(getBalance(s_1) - x \geq 0,$$
$$getBalance(s_1) - x, getBalance(s_1)\big)$$
$$(b).\ getBalance\Big(\textbf{proj}\big(2, transfer(s_1, s_2, x)\big)\Big)$$
$$= \textbf{if-then-else}\big(getBalance(s_1) - x \geq 0,$$
$$getBalance(s_2) + x, getBalance(s_2)\big)$$

The CPN models of those equations are denoted as shown in Figure 8. If we can use the methods getBalance, debit, and credit as components, that is, they are implemented correctly, the CPN models in Figure 8 are reduced into the simple adaptation forms by focusing on the subnets enclosed by the dotted lines,

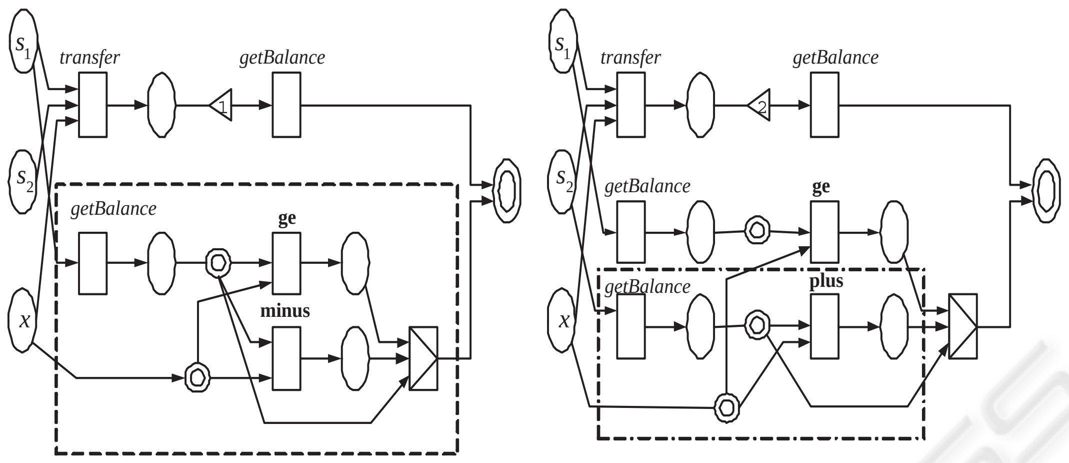


Figure 8: CPN Models of the Method transfer.

and we can obtain the adaptable virtual component

$$transfer(s_1, s_2, x) = \textbf{tuple}\Big(debit(s_1, x), \\ \textbf{if-then-else}\big(getBalance(s_1) - x \geq 0, \\ credit(s_2, x), s_2\big)\Big)$$

5 CONCLUSION

A formal approach to identifying a set of components which satisfies a single requirement was presented. In order to make component adaptability rigorous and simple, two different aspects of requirements and components, that is, function and behavior are consolidated into a functional aspect. Such a functional aspect is rigorously expressed in the form of algebraic specifications, which consist of a set of data type definitions and equations of terms. Those terms are composed of S-sorted functions which represent basic requirements and components. When a component satisfies a requirement function, the function that occurs in a requirement specification can be substituted by the component function. In order to enhance the reusability of components, the concept of a virtual component was introduced, which were composed of multiple components, and acts as a single component. Such a virtual component is expressed as a term and the above substitution is based on this term. As a software development methodology, we need more visible and intuitive way to accomplish the above substitution. A CPN model is one of the most suitable solutions, since a CPN model can express terms and equations accurately, and the validity of the substitution process can be implemented as a reduction process of a CPN model.

REFERENCES

Fensel, D. and Groenboom, R. (1995). Formal specification languages in knowledge and software engineering. In *The Knowledge Engineering Review. 10(4) pp.303-317*.

Fischer, B. (1998). Specification-based browsing of software component libraries. In *Proc. of Thirteenth International Conference on Automated Software Engineering pp.74–83*. ACM.

Goguen, J. A. and Malcolm, G. (1996). *Algebraic Semantics of Imperative Programs*. MIT Press.

Guttag, J. V. and Horning, J. J. (1993). *Larch: Languages and Tools for Formal Specification*. Springer.

Jensen, K. (1997). *Coloured Petri Nets Volume 1-3*. Springer.

Michail, A. and Notkin, D. (1999). Assessing software libraries by browsing similar classes, functions and relationships. In *Proc. of International Conference on Software Engineering, pp.463–472*.

Mosses, P. D. (1999). Casl: A guided tour of its design. In *Proc. Workshop on Abstract Datatypes pp.216–240*.

Prieto-Diaz, R. and Freeman, P. (1987). Classifying software for reuse. In *IEEE Software.* **4(1)** *pp.6–16*. IEEE.

Shinkawa, Y. and Matsumoto, M. J. (2000). Knowledge-based software composition using rough set theory. In *IEICE Trans. on Inf. and Syst, Vol.E83-D. No.4,pp.691-700*. IEICE.

Zaremski, T. M. and Wing, J. M. (1993). Signature matching: A key to reuse. In *Proc. of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering, pp182–190*. ACM.

Zaremski, T. M. and Wing, J. M. (1995). Specification matching of software components. In *ACM Trans. on Software Engineering and Methodology.* **6(4)**, *pp333–369*. ACM.